Praca powinna być cytowana jako:
Borkowski, J., 2006. Sterowanie w programach równoległych oparte na spójnych stanach aplikacji. Rozprawa doktorska. Polsko-Japońska Wyższa Szkoła Technik Komputerowych.

Polsko-Japońska Wyższa Szkoła Technik Komputerowych

Sterowanie w programach równoległych oparte na spójnych stanach aplikacji

Praca doktorska

zrealizowana pod kierunkiem promotora dr hab inż. Marka Tudruja
w Polsko-Japońskiej Wyższej Szkole Technik Komputerowych w Warszawie

Janusz Borkowski

14 grudnia 2006

Streszczenie

Praca uzasadnia tezę, że sterowanie oparte o analizę stanów globalnych aplikacji dla wybranych klas problemów umożliwia poprawną i wygodną strukturalizację sterowania w programach, przy uzyskaniu nie gorszej, a w wielu wypadkach lepszej, wydajności obliczeniowej, niż przy użyciu klasycznych metod sterowania, polegających na specyfikacji sterowania i obliczeń przemieszanych z komunikacją poprzez przesyłanie komunikatów.

Przegląd stosowanych metod sterowania i synchronizacji w programach równoległych i ich usystematyzowanie pozwoliły określić pożądane cechy tych metod. Uzyskane zestawienie doprowadziło do wniosku, że korzystne byłoby opracowanie mechanizmu sterowania wykonaniem programów równoległych, który działa w środowiskach rozproszonych, a zarazem daje programiście wygodę charakterystyczną dla metod bazujących na wspólnych zasobach, pozwala na określanie sposobu działania stosowanych w programie prymitywów sterujących oraz jest niezależny od mechanizmów przekazywania danych. Wymienione postulaty spełnia metoda sterowania, w której decyzje sterujące podejmowane są na podstawie analizy stanu globalnego aplikacji.

W pracy przedstawione są podstawy teoretyczne określania globalnych spójnych stanów oraz sposoby (modalności) wartościowania predykatów określonych na stanach globalnych. Szczególną uwagę zwrócono na stany spójne konstruowane w oparciu o znaczniki czasu rzeczywistego w systemach z częściowo zsynchronizowanymi zegarami lokalnymi, zwane silnie spójnymi stanami globalnymi (*SCGS*). Zdecydowano wykorzystać monitorowanie na bieżąco SCGS i wartościowanie predykatów globalnych w modalności *Instantly* w proponowanej metodzie sterowania wykonaniem programów.

Znany z literatury algorytm wykrywania SCGS został poprawiony. Opracowano nowe algorytmy wykrywania SCGS, pozwalające wykrywać więcej/szybciej SCGS. Algorytmy te wykorzystują: znajomość maksymalnego czasu transmisji komunikatu do monitora, znajomość minimalnego czasu trwania stanów lokalnych procesów, znajomość maksymalnego błędu synchronizacji zegarów lokalnych pomiędzy poszczególnymi parami procesów. Standardowe podejście, w którym pojedynczy centralny monitor zbiera informacje o stanach lokalnych procesów, konstruuje globalne stany spójne i wartościuje na nich predykaty, zostało rozszerzone do formy zhierarchizowanej. Umożliwiło to rozdzielenie obliczeń związanych z konstrukcją SCGS, oraz realizację hierarchicznego sterowania, charakteryzującego się lepszą efektywnością i skalowalnością.

Proponowana metoda sterowania zakłada, że spełnienie zdefiniowanych predykatów globalnych ma wpływać na zachowanie procesów. Rozważono możliwe sposoby reakcji procesów na na spełnienie predykatów. Wybrano asynchroniczną reakcję na przybycie sygnału oznaczającego spełnienie predykatu. Sygnał aktywuje skojarzony z nim kod. Na czas wykonania tego kodu główne obliczenia w procesie są wstrzymywane. Alternatywnie, sygnał sterujący może spowodować porzucenie bieżących obliczeń i przejście do dalszej części kodu programu. Wprowadzony mechanizm regionów w kodzie programu reguluje czy i kiedy proces powinien zareagować na przybycie sygnału sterującego.

Za pomocą opracowanego symulatora środowiska obliczeniowego eksperymentalnie sprawdzono charakterystyki opracowanych algorytmów wykrywania SCGS oraz realizowanego za ich pomocą sterowania. Charakterystyki te zostały określone przy użyciu wprowadzonych miar efektywności.

Praktyczna weryfikacja proponowanej metody sterowania zrealizowana została za pomocą systemu PS-GRADE. PS-GRADE to system graficznego projektowania programów równoległych, w którym do podstawowego mechanizmu przekazywania komunikatów dodano sterowanie za pomocą predykatów globalnych. Realizacja asynchronicznej reakcji na sygnały sterujące poprawiła wydajność systemu PS-GRADE względem wersji posługującej się samym przekazywaniem komunikatów, pozwalając unikać oczekiwania na przybycie komunikatu.

Wyniki testów efektywności proponowanej metody sterowania zrealizowane z użyciem wybranych aplikacji, przeprowadzone w systemie PS-GRADE oraz przy użyciu symulatora pokazały, że postawiona a wstępie teza jest prawdziwa.

Spis treści

Spis rysunków

List of Algorithms

Wstęp

Synchronizacja i sterowanie w programach równoległych są zagadnieniami badanymi od lat. Od momentu wprowadzenia wieloprogramowości w komputerach, nastepnie wspomagając rozwój architektur równoległych i programowania równoległego, oraz systemów rozproszonych i masywnie równoległych, ta problematyka była nieustannie rozwijana. Opracowano wiele metod specyfikacji sterowania w środowiskach równoległych i rozproszonych. Część z nich stała się powszechnie stosowana, inne znalazły swoją rolę w pewnych niszowych zastosowaniach, jeszcze inne nie zdobyły szerszego uznania. Analizując przekrojowo znane metody sterowania w programach równoległych doszliśmy do wniosku, że pewne obszary pozostały słabo zbadane. W obszarach tych mieszczą się pewne niestosowane do tej pory metody sterowania o interesujących charakterystykach.

Dla programisty częstokroć wygodniejsze w użyciu są metody sterowania odwołujące się do abstrakcji wspólnych, scentralizowanych zasobów, w przeciwieństwie do metod w pełni rozproszonych, jak np. przekazywanie komunikatów. W obecnych czasach wydaje się jednak konieczne, aby sposób sterowania nadawał się do wykorzystania także w środowiskach rozproszonych. Wygoda użycia metod sterowania zależy w dużej mierze od oferowanego poziomu abstrakcji - właściwy (najwygodniejszy) poziom można osiągnąć przez określanie sposobu działania stosowanych w programie prymitywów sterujących. Za istotne uznaliśmy oddzielenie mechanizmu sterowania od mechanizmu przekazywania danych, uwypuklając potencjalną niezależność obu i koncentrując się na rozbudowanym sterowaniu.

W pracy przedstawiamy metodę sterowania wykonaniem programów równoległych/rozproszonych opracowaną według przedstawionych założeń i spełniającą wyżej wymienione postulaty. Metoda ta opiera się na bieżącej obserwacji i analizie stanów globalnych aplikacji oraz na przekazywaniu procesom aplikacyjnym decyzji sterujących podejmowanych na podstawie tychże analiz. Zalety proponowanej metody z punktu widzenia programisty wynikają w dużej mierze z przyjętych założeń. W pracy przeprowadzamy szeroką analizę realizowalności i efektywności proponowanej metody sterowania, prezentujemy szczegóły jej implementacji i rozwiązania szerokiej gamy napotykanych problemów, oraz symulacyjne i rzeczywiste wyniki uzyskane przy zastosowaniu proponowanej metody sterowania.

Rozdział 1

Wprowadzenie

Implementacja dowolnych algorytmów w formie programu wymaga od programisty określenia, kiedy (w jakiej sytuacji) mają się wykonać poszczególne fragmenty kodu programu. Innymi słowy konieczne jest wyspecyfikowanie przepływu sterowania. Decyzje sterujące, takie jak wybór dalszej ścieżki w kodzie, czy wstrzymanie programu, zależą od bieżącego stanu obliczeń. Pojedynczy proces w sposób naturalny zna tylko swój stan lokalny. Tymczasem decyzje sterujące w programie równoległym muszą brać pod uwagę więcej niż tylko izolowany stan pojedynczego procesu. Ten zasadniczy problem wymusił powstanie nowych języków programowania, ich rozszerzeń oraz bibliotek programistycznych, które dają do dyspozycji metody wymiany informacji pomiędzy procesami. Stosując te metody, programista ma możliwość zapisu decyzji sterujących, które uwzględniają stan obliczeń w wielu procesach, tak jak tego wymaga dany implementowany algorytm równoległy. W tym rozdziale zajmiemy się najpierw przeglądem wspomnianych metod. Następnie przedstawimy pojęcie stanu globalnego aplikacji i związane z nim pojęcie predykatów globalnych. W końcowej części rozdziału zaproponujemy zastosowanie stanów i predykatów globalnych bezpośrednio do specyfikacji sterowania w programach równoległych.

1.1 Analiza stosowanych metod specyfikacji sterowania w programach równoległych

W tym rozdziale (i w kolejnych) używać będziemy pojęć *sterowanie* i *synchronizacja*. Poniżej podajemy ich definicje:

Sterowanie w programie jest to określenie porządku wykonywania instrukcji w tym programie

Synchronizacja jest to uzależnienie szeroko rozumianego stanu procesu od stanu innych (zwykle współbieżnych) procesów.

Jeśli sterowanie w programach równoległych zależy od stanu innych procesów, to sterowanie wymaga lub korzysta z synchronizacji. Synchronizacja jest jednym z wielu elementów sterowania w programach równoległych. W tej pracy stanowi ona centrum zainteresowania.

Na przestrzeni lat opracowano wiele metod pozwalających na określenie sterowania w programach równoległych. Poniżej przedstawiamy przegląd tych metod wraz z odniesieniami do literatury.

1.1.1 Podstawowe rodzaje synchronizacji w programach równoległych

1.1.1.1 Synchronizacja dostępu do wspólnych zasobów

Jest to historycznie najstarszy wariant synchronizacji. Polega on na określeniu reguł dostępu (i ich przestrzeganiu) do globalnie i bezpośrednio dostępnego procesom zasobu. Procedura uzyskania

dostępu do zasobu jest operacją synchronizacji. Ta koncepcja wywodzi się z systemów scentralizowanych, gdzie o wspólne zasoby łatwo. Istnieją też rozwinięcia odpowiednie dla systemów rozproszonych, bazujące na tym, że pewien zasób (zasoby) są udostępniane globalnie. Najczęstsze zastosowanie synchronizacji dostępu do wspólnych zasobów sprowadza się do kontroli dostępu do sekcji krytycznej. Wymagana jest przy tym realizacja dwóch procedur:

- WE: bezpieczne wejście do sekcji krytycznej (oczekiwanie na zwolnienie zasobu)
- WY: wyjście z sekcji krytycznej (pozostawienie zasobu w stanie wolnym)

W systemie może istnieć wiele zasobów używanych do koordynacji procesów: Z1, Z2, ... W relacji do tych zasobów proces w każdej chwili może znajdować się w jednym z trzech stanów:

- stan 1. Proces działa niezależnie od zasobu Z,
- stan 2. Proces czeka, aż zasób Z pozwoli mu na dalsze działanie,
- stan 3. Proces działa za pozwoleniem Z.

Wykonanie sekcji krytycznej jest reprezentowane przez stan 3.

To, czy i kiedy zasób da zezwolenie na dalsze działanie ubiegającemu się o to procesowi, zależy od stanu wewnętrznego tego zasobu. Procesy mają możliwości zmiany tego stanu, zwykle za pomocą predefiniowanych procedur, dzięki czemu pośrednio oddziaływują na siebie. Istotne, że zasób nie jest aktywny i samodzielnie nie wykonuje żadnych akcji.

Istnieje kilka realizacji omawianej idei.

Arbiter pamięci, operacje typu test&set Metoda ta wymaga, aby procesy dysponowały wspólną pamięcią i bazuje na fakcie, że żądania skierowane do tej samej komórki pamięci są wykonywane szeregowo. Realizacja procedur WE i WY korzysta z operacji odczytu/zapisu uzgodnionych komórek pamięci [41, 96]. Nie jest oferowany żaden mechanizm usprawniający oczekiwanie na zwolnienie zasobu, zatem wykorzystanie tej metody wymaga, aby procesy w stanie 2 cyklicznie sprawdzały, czy mogą już przejść do stanu 3. Program użytkownika musi zawierać kod realizujący procedury WE i WY (polegając na działaniu arbitra), jak też kod określający kiedy dana procedura ma być użyta (reguły aplikacji).

Instrukcje typu test&set wykorzystuje się w systemach z pamięcią wspólną jako narzędzie do implementacji bardziej abstrakcyjnych metod (np. semafory). Jedynie w kodzie jąder systemów operacyjnych oraz bibliotek systemowych (np. realizacja wątków) można spotkać bezpośrednie zastosowanie instrukcji test&set oraz pokrewnych (np. test&test&set) [99].

Semafory Semafor używany jest do sterowania dostępem do dzielonego zasobu oraz sam jest rodzajem zasobu dzielonego [43]. Semafor to zmienna całkowita związana ze zbiorem wstrzymanych procesów, na której zdefiniowano niepodzielne operacje P() i V(), zapewniające techniczną realizację procedur odpowiednio WE i WY. Programista używa semafora właśnie za pośrednictwem tych operacji. Semafor wnosi istotne usprawnienie do metody oczekiwania na zwolnienie zasobu. Proces oczekując w stanie 2 jest zawieszony, nie zużywa czasu procesora. Podniesienie semafora (V()) powoduje automatyczne wznowienie procesu oczekującego na tym semaforze. Określenie kiedy należy użyć procedur semafora (reguły sterowania aplikacją) jest zadaniem programisty i polega na wstawieniu odpowiedniego kodu w określone miejsca programu.

Semafory stanowią część specyfikacji POSIX (Portable Operating System Interface, definicja przenośnego interfejsu systemu operacyjnego opracowana przez IEEE) i są obecne we wszystkich wcieleniach UNIXa oraz innych systemów zgodnych z POSIX. Aktualnie znajdują istotne zastosowanie w systemach wielowątkowych, najczęściej w formie uproszczonej, stosującej operacje lock/unlock.

Pamięć synchronizująca To rozwiązanie wymaga specjalnego wspomagania sprzętowego. Komórki wspólnej pamięć posiadają dodatkowo znaczniki zawartości (pusty/pełny). Operacje zapis/odczyt mogą uwzględniać stan zawartości, np. wstrzymując proces próbujący odczytać pustą komórkę pamięci do czasu, aż ta komórka się zapełni. W ten sposób uzyskujemy realizację procedur WE i WY. Poza samą synchronizacją mechanizm ten pozwala oczywicie na przekazywanie danych.

Z uwagi na konieczność użycia niestandardowego sprzętu jest to rzadko spotykane rozwiązanie. Stanowi podstawowy mechanizm synchronizacji w eksperymentalnej konstrukcji JUMP-1 [114] oraz w niektórych maszynach data flow [59].

Monitory Monitor [30] jest zasobem dzielonym, dostępnym za pomocą określonego przez programistę zbioru procedur wykonywanych w sposób wyłączny (nie można uruchomić dwóch instancji procedur jednocześnie). Monitor może zawierać (określone przez programistę) dane wewnętrzne, w szczególności kolejki zawieszonych procesów, na których operują te procedury.

Monitory są wygodniejsze w użyciu niż poprzednio wymienione prymitywy dzięki połączeniu realizacji procedur WE i WY z opisem zasad działania aplikacji. Program użytkownika posługuje się monitorem wywołując zdefiniowane procedury monitora. Umieszczony tam kod określa reguły synchronizacji dla danej aplikacji i nie miesza się z właściwym kodem implementowanego algorytmu w poszczególnych procesach.

Monitory doczekały się wielu realizacji, czy to jako rozszerzenia istniejących języków programowania, czy też jako element języków nowo zaprojektowanych. Obecnie wszystkie te realizacje zostały daleko zdystansowane w sensie rozpowszechnienia przez język Java. W Javie monitory są standardowym elementem, przeznaczonym to koordynowania współbieżnych wątków, także będacych integralną częcią Javy.

Zdalny dostęp do pamięci RDMA W modelu tym jeden z komunikujących się procesów pozwala na czasowy dostęp do określonego fragmentu swej pamięci innym procesom [84, 8]. Inny proces może wtedy wpisać dane do udostępnionego fragmentu pamięci. W procesie udostępniającym zamiast punktu kontrolnego mamy pewien przedział - fragment kodu, w którym zdalny dostęp do pamięci jest otwarty. Stosuje się tam też instrukcje pozwalające określić, czy dokonano zdalnego dostępu. W procesie realizującym zdalny dostęp używa się instrukcji potwierdzających uzyskanie dostępu (wyłącznego w przypadku operacji zapisu). Operacje zdalnego dostępu do pamięci koncentrują się na transferze danych, dodatkowe mechanizmy pozwalają jednak na synchronizacje komunikujących się procesów w celu np. niedopuszczenia do nadpisania jeszcze nie skonsumowanych danych.

RDMA jest podstawowym mechanizmem komunikacji w komputerach Hitachi SR2201 [45] i SR8000, ma tam realizację sprzętową. Choć istnieją możliwości jego bezpośredniego użycia [111], to tak niskopoziomowe realizacje RDMA służą zwykle do implementacji wygodniejszych mechanizmów, np. MPI. Metodyka zdalnego dostępu do pamięci jest częścią standardu MPI-2 [90]. Ten fakt, oraz konstrukcje nowych typów kart sieciowych umożliwiających sprzętową realizację RDMA [81], pozwalają sądzić, że RDMA będzie się rozpowszechniał.

Rozproszona pamięć dzielona (DSM) DSM polega na przedstawieniu programiście pamięci rozproszonej, dostępnej bezpośrednio tylko lokalnie, jako globalnie jednolitej pamięci dzielonej o bezpośrednim dostępie. Do tej klasy rozwiązań należą tak programowe, jaki i sprzętowe realizacje [37]. Realizacje te muszą zapewnić implementację mechanizmów synchronizacji dostępu do pamięci dzielonej i dlatego DSM wchodzi w zakres naszej uwagi. Mechanizmy te zwykle umożliwiają blokowanie procesów do czasu uzyskania wymaganego rodzaju dostępu (wyłączny / niewyłączny) do określonego obszaru pamięci dzielonej.

Wspólna przestrzeń danych Ten pomysł umożliwia procesom wymianę danych oraz synchronizację poprzez operacje na wirtualnej wspólnej przestrzeni danych. Procesy mogą wstawiać,

odczytywać i usuwać zapisy w tej przestrzeni. Wyszukanie żądanych danych przy operacji odczytu następuje przy pomocy wzorca, np. ilość składowych elementów istniejącego zapisu oraz typy składowych muszą być zgodne z podanym wzorcem. Brak pasującego zapisu może powodować wstrzymanie procesu do czasu pojawienia się żądanych danych - to podstawowa operacja synchronizująca. Trzeba zauważyć, że możliwości wykorzystania wzorców oraz przekazywania danych w zapisach stawiają to podejście wysoko pod względem poziomu abstrakcji.

Bezpośrednie odbicie tej idei wspólnej przestrzeni danych znajdziemy w języku Linda [31]. Innym przykładem jest biblioteka PVM wersja 3.4 [52]. Oferuje ona podobną jak Linda koncepcję, choć tylko jako uzupełnienie zasadniczego mechanizmu przekazywania komunikatów.

1.1.1.2 Synchronizacja wykonania fragmentów programów

Myślą przewodnią jest tu uzgadnianie tempa wykonania procesów, przy czym procesy czynią to bezpośrednio między sobą bez udziału dodatkowych elementów. W kod programów wstawione są jawnie instrukcje sterujące. Instrukcje te mogą być ze sobą powiązane wg rozmaitych reguł, np. wykonanie instrukcji X procesu P1 związane jest z wykonaniem instrukcji Y procesu P2 regułą *jednocześnie*. Koordynacja polega na uzgodnieniu czasu, w którym sterowanie procesów znajdzie się w powiązanych instrukcjach sterujących. I w tym przypadku istnieje kilka realizacji tego pomysłu, poniżej omówionych.

Przekazywanie komunikatów Polega ono na bezpośredniej komunikacji pomiędzy procesami. Choć przekazywanie komunikatów kładzie nacisk na aspekt wymiany danych i w tym świetle jest zwykłe omawiane, my zwrócimy uwagę na oferowane tu możliwości sterowania przebiegiem wykonania programu. Wspomniane wyżej instrukcje sterujące reprezentowane są tu przez: wysłanie, odebranie, wysłanie i odebranie jednoczesne komunikatu. Reguły powiązań, podane niżej, muszą zapewniać, że choć jedena instrukcja to wysłanie, a druga odebranie komunikatu. Klasyfikacja reguł wg liczby instrukcji nadawczych i odbiorczych wygląda tak:

a) jeden nadawca i jeden odbiorca (point to point)
b) jeden nadawca i wielu odbiorców (broadcast, scatter)
c) jeden odbiorca i wielu nadawców (gather)
d) wszyscy do wszystkich (pełna wymiana, allToAll)

Bezpośrednia komunikacja pomiędzy procesami wymaga specyfikacji adresowania procesów - każdy nadawca musi podać, dla którego procesu przeznaczony jest wysyłany komunikat. Warianty b) i d) wymagają wprowadzenia pojęcia grupy procesów, aby określić które procesy mają uczestniczyć w wymianie komunikatów. Generalną zasadą jest, że dla powiązanych punktów proces zawierający punkt odbiorczy musi zaczekać aż proces nadawca wyśle komunikat i ten dotrze do punktu przeznaczenia. Rozmaite warianty interakcji pomiędzy nadawca i odbiorcą omówione są w [89]. Najważniejsze z nich to

- Komunikacja asynchroniczna, w której nadanie komunikatu może nastąpić dowolnie wcześniej niż odbiorca zadeklaruje gotowość odbioru.
- Komunikacja synchroniczna (rendez-vous), w której nadanie i odbiór komunikatu muszą nastąpić jednocześnie. W praktyce oznacza to, że jeden z partnerów (pierwszy gotowy) musi zaczekać na gotowość drugiego.

Z punktu widzenia tworzenia programów niezwykle kłopotliwa jest konieczność wyboru rodzaju i położenia instrukcji sterujących w kodzie programu oraz dokładnego określenia powiązań pomiędzy nimi. Celem jest odzwierciedlenie wymaganych przez aplikacje reguły synchronizacji oraz wymiany danych. Z drugiej strony prostota (brak dodatkowych elementów pośredniczących), efektywność implementacji oraz łatwość realizacji tej idei stanowią o jej dużej popularności.

Obecnie przekazywanie komunikatów stanowi jeden z głównych stosowanych paradygmatów programowania równoległego. Rozwinęło się razem z konstrukcjami komputerów współbieżnych z

pamięcią rozproszoną oraz wraz z rosnącym wykorzystanie sieci komputerów. Historycznie Communicating Sequential Processes [60] było jedną z pierwszych teoretycznych propozycji przekazywania komunikatów i nadal przywoływane jest jako wzorcowy model. Warto tu wspomnieć o języku Occam, który stanowi dość wierną praktyczną realizację CSP. Znane z Unix BSD gniazdka (socket), obecne w niemal każdym nowoczesnym systemie operacyjnym, są uznanym standardem w programowaniu przesyłania danych w aplikacjach systemowo-sieciowych. Biblioteki dostarczające wygodnych funkcji przekazywania komunikatów: PVM [53] i MPI mają silną pozycję w praktyce tworzenia aplikacji równoległych, dla sieci stacji roboczych i komputerów masywnie równoległych.

Synchronizacja grupowa Synchronizacja grupowa polega na interakcji pomiędzy procesami w celu osiągnięcia pewnego wspólnego stanu. W operacji synchronizującej muszą wziąć udział wszystkie procesy z danej grupy, ich role są symetryczne.

Najpowszechniej stosowanym mechanizm synchronizacji grupowej jest bariera. Procesy uczestniczące w operacji bariery mają zdefiniowane punkty kontrolne, po jednym na proces, a łącząca te punkty reguła jest następująca: po dojściu do punktu kontrolnego poczekaj, aż wszystkie pozostałe procesy uczestniczące w operacji bariery także dojdą do swych punktów kontrolnych. Bierne oczekiwanie na proces najpóźniej docierający do punktu bariery bywa przyczyną niezadowalającej wydajności programów często stosujących barierę.

Bariera jest powszechnie używana w maszynach z pamięcią dzieloną aby zapewnić, że dane obliczenie zakończyło się przed możliwym wykorzystaniem wyników tegoż obliczenia. Kompilatory zrównoleglające dla takich komputerów potrafią automatycznie wstawiać instrukcje bariery do produkowanego kodu, aby zapewnić jego poprawność.

Opracowano bardziej złożone warianty bariery jak np. bariera rozmyta [56], używająca zamiast punktu pojęcia przedziału. Procesy nie mogą przejść poza przedział bariery dopóki wszystkie nie wejdą w swoje przedziały. Bariera topologiczna natomiast uwzględnia logiczne położenie procesów względem siebie, wymusza oczekiwania tylko na procesy sąsiadujace z danym [106]. Warianty te nie są jednak rozpowszechnione.

Model programowania Bulk-Synchronous Parallel (BSP) stosuje barierę jako podstawową metodę synchronizacji [110]. Stosując ten model obliczenia dzieli się na fazy (supersteps), na koniec każdej z nich wykonując barierę. Wymiana danych pomiędzy procesami dokonuję się po wykonaniu obliczeń w każdej fazie. Bariera pozwala zapewnić, że wszystkie procesy zakończyły lokalne obliczenia w danej fazie przed etapem wymiany danych.

Biblioteki MPI i PVM także udostępniają operacje bariery, choć nie ma ona tam podstawowego znaczenia.

Widząc istotną rolę bariery producenci sprzętu postarali się w kilku przypadkach o jej sprzętową realizację (np. Cray T3E, Fujitsu AP1000, Hitachi SR2201).

Sterowanie przepływem komunikatów (aktywne komunikaty) Ten rodzaj sterowania różni się od omówionego wyżej przekazywania komunikatów sposobem reakcji na przyjmowanie komunikatów przez procesy. W przypadku przekazywania komunikatów proces odbiorca wykonywał swój kod aż dotarł do punktu odbioru komunikatu. Tutaj odbiorca jest bierny, samodzielnie nie wykonuje obliczeń. Dopiero przybycie komunikatu uaktywnia procedurę (metodę) skojarzoną z danym typem przybyłego komunikatu. Po zakończeniu zainicjowanej procedury zwykle do nadawcy komunikatu odsyłany jest rezultat wykonanych obliczeń. Zatem istotną cechę sterowania przepływem komunikatów jest połączenie operacji odbioru komunikatu z aktywacją skojarzonego z nim kawałka kodu.

UNIX oraz Windows oferują mechanizm Remote Procedure Call (RPC), jest to realizacja zdalnego wywołania procedur na poziomie systemu operacyjnego. Znajduje ona zastosowanie w realizacji wszelkiego rodzaju rozproszonych aplikacji. Praktycznie mamy tu jednak do czynienia raczej ze zdalnym wywołaniem usług niż z metodą synchronizacji jako taką. Podobnie przedstawia się znane z Javy Remote Method Invocation. Active Messages z naszego punktu widzenia stanowią niskopoziomową realizację mechanizmu sterowania przepływem komunikatów, dlatego tu o nich wspominamy, choć powstały z myślą nie o sterowaniu, lecz o transparentnym dołączaniu

odebranych danych do trwających obliczeń [121]. Koncepcję Active Message na wyższym poziomie abstrakcji przypomina mechanizm *message handlers* w PVM wersja 3.4. Język Dagger [57] oferuje rozbudowane możliwości sterowania przepływem komunikatów, oraz promuje drobnoziarnisty model współbieżności.

1.1.1.3 Sterowanie przepływem danych (dataflow)

W modelu tym mamy do dyspozycji skrajnie drobnoziarnistą współbieżność. Wykonanie pojedynczej instrukcji uzależnione jest jedynie od dostępności jej argumentów. Prowadzi to do specyficznego modelu obliczeń. Zamiast licznika rozkazów mamy zbiór instrukcji gotowych do wykonania, realizowalnych w dowolnej kolejności. Efektywna realizacja tego podejścia wymaga specjalizowanych architektur komputerowych (praktycznie nie istniejących obecnie na rynku) lub transformacji w stronę modelu makro-dataflow.

1.1.1.4 Sterowanie makro przepływem danych (makro-dataflow)

Aplikacja współbieżna, jeśli zostanie jawnie podzielona na składowe sekwencyjne procesy, może zostać przedstawiona jako graf skierowany, gdzie węzłami są wyróżnione procesy sekwencyjne, krawędziami natomiast zależności danych pomiędzy nimi. Krawędź od procesu P1 do P2 oznacza, iż proces P2 potrzebuje danych, które wyprodukuje proces P1. Dane będa dostępne, gdy proces P1 się zakończy. Jeśli do procesu P2 prowadzą krawędzie od kilku procesów, P2 musi zaczekać, aż wszystkie te procesy się zakończą.

Przykładem, raczej instruktażowym niż praktycznym, użycia makro dataflow jest graficzny język HeNCE [12]. Idea polega na wizualnym tworzeniu grafu, gdzie węzły reprezentujące obliczenia sekwencyjne łączone są scieżkami reprezentującymi przepływy danych. W powszechnym zastosowaniu makro dataflow jest postacią wyprowadzoną przez kompilator z programu zapisanego w inny sposób. Przykładem może być [97] oraz [94]. Z modelem makro dataflow silnie powiązane są badania nt. schedulingu oraz wielowątkowości [108, 120].

1.1.2 Możliwe klasyfikacje metod synchronizacji

Powyższe omówienie metod realizacji synchronizacji stanowi pewne podejście do próby usystematyzowania ogólnych założeń, na których oparto realizację poszczególnych metod. W dalszym ciągu zaproponujemy inne sposoby klasyfikacji metod synchronizacji w programach równoległych. Taka systematyka pozwoli lepiej ujrzeć w jakim kierunku posuwają się badania w dziedzinie sterowania w programach równoległych, które strefy pozostają najmniej rozeznane i w jakiej relacji do metod już stosowanych będzie zaproponowana w dalszej części pracy nowa metoda sterowania.

1.1.2.1 Sposób współpracy pomiędzy procesami dla potrzeb synchronizacji

Skoro synchronizacja może wymagać znajomości stanów innych procesów, stany te (informacja o nich) muszą być dostępne. Sposób, w jaki potrzebna informacja jest przekazywana, stanowi obecne kryterium podziału.

Procesy korzystają z usług koordynatora Monitor jest wzorcowym przykładem koordynatora. Proces korzystający z monitora nie wie nic o innych procesach, interakcjami międzyprocesowymi zarządza monitor, jest on niezbędnym pośrednikiem. Cała wiedza i funkcjonalność wymagane do realizacji sterowania zaszyte są w monitorze. Semafory także przystają do tej kategorii, jako pośrednicy.

Procesy prowadzą wzajemną bezpośrednią interakcję Najlepszym przykładem jest tu przekazywanie komunikatów. Procesy muszą znać się nawzajem (nadawca, odbiorca). Proces może otrzymać informację od innego procesu jedynie wtedy, gdy ten wyślę ją do tego konkretnego odbiorcy. W przypadku komunikacji grupowej wystarczyć może znajomość nazwy grupy procesów.

Inne przykłady wzajemnych interakcji to RDMA, zdalne wywołanie procedur oraz sterowanie przepływem komunikatów.

Procesy obserwują się wzajemnie Podstawową zasadą jest tu bierna obserwacja - proces nie może ingerować w tok wykonania innego procesu, np. nie może wydać polecenia jego wstrzymania czy wznowienia. Może natomiast obserwować stany innych procesów i sam na tej podstawie podejmować decyzje sterujące swym działaniem.

Bariera jest dobrym przykładem. Proces wykonując operacje bariery staje się widoczny dla innych procesów i sam staje się procesem obserwującym. Niestety zakres widocznej informacji jest ustalony i bardzo niewielki: można stwierdzić tylko czy wszystkie procesy już osiągnęły zadany punkt w kodzie programu.

Przestrzeń krotek w Lindzie może być przedstawiona jako tablica ogłoszeń, przy użyciu której procesy publikują informacje o swym stanie. W ten sposób Linda jest realizacją idei wzajemnej obserwacji procesów. Zakres widocznej informacji odpowiada informacji wpisanej do publikowanych krotek i jest pod kontrolą programisty.

1.1.2.2 Stopień powiązania procesów

Przez stopień powiązania procesów rozumiemy

- jak wiele muszą one wiedzieć o sobie nawzajem
- jak bardzo zachowanie danego procesu (np. jego awaria) może wpływać na zachowanie pozostałych procesów

Ścisłe powiązanie Procesy wiedzą nawzajem o swym istnieniu. Wiedzą też kiedy i czego oczekiwać od innych procesów w celu przeprowadzenia operacji sterującej wykonaniem aplikacji, co i kiedy zasygnalizować każdemu innemu procesowi. Konieczne jest więc pojęcie identyfikatora procesu, każdy proces powinien znać identyfikatory procesów z którymi współpracuje.

Przekazywanie komunikatów realizuje model ścisłego powiązania. Programowanie z użyciem tego modelu wymaga, aby scenariusz interakcji pomiędzy poszczególnymi procesami był wpisany w kod aplikacji. Podobnie jest w przypadku RDMA.

Asymetryczne powiązanie Jest to model zbliżony do powiązań ścisłych, lecz w tym przypadku tylko jeden z pary koordynujących się procesów zna identyfikator tego drugiego. Rozwiązania to odpowiada sterowaniu przepływem komunikatów i jest zastosowane np. w języku Dagger oraz Unix RPC. Jeden proces występuje w biernej roli przyjmującego nadesłane komunikaty. Nie ma potrzeby określania oczekiwanego nadawcy ani jawnego programowania stanu oczekiwania na zgłoszenie ani procedury odbioru komunikatu.

Powiązanie w ramach grupy Synchronizacja opiera się na pojęciu grupy procesów. Każda operacja synchronizacji bierze pod uwagę i wpływa na grupę procesów. Procesy daję się identyfikować pod względem przynależności do grupy.

Grupy mogą być ustalane statycznie. Ma to miejsce w przypadku modeli makro dataflow i dataflow, tam kompilator tworząc graf zależności procesów definiuje zarazem grupy - bezpośredni przodkowie oraz następcy danego procesu tworzą grupy. PVM i MPI pozwalają kreować grupy dynamicznie, ich tworzenie jest pod kontrolą programisty.

Powiązania anonimowe Procesy nie znają się wzajemnie ani nie wiedzą ile ich jest. Każdy proces wie co i kiedy powinien zrobić dla innych i czego ma oczekiwać w celu przeprowadzenia operacji sterującej wykonaniem aplikacji.

Przestrzeń krotek w Lindzie jest dobrym reprezentantem tej zasady. Proces oczekuje na pojawienie się określonej danej nie wiedząc kto ją utworzy, sam publikuje nie wiedząc kto skorzysta z opublikowanych danych.

Semafory i monitory także każą traktować procesy bezimiennie. Operacja wznowienia nie pozwala na wybranie konkretnego procesu. Z tego powodu na programistę spada zadanie umieszczenia w procesach kodu sprawdzającego, czy wznowiony został właściwy proces (o ile tego typu niepewność istnieje w danej sytuacji).

1.1.2.3 Możliwości wyrażania złożonych schematów synchronizacji

Chcielibymy określić, jak wygodne jest użycie poszczególnych metod synchronizacji. Przez wygodę rozumiemy łatwość i zwartość wyrażenia pożądanych warunków sterujących w kodzie aplikacji oraz przejrzystość kodu aplikacji korzystającego z operacji synchronizujących.

Proste operacje podstawowe Do tej kategorii należą prymitywy synchronizujące posiadające bardzo prostą i niemodyfikowalną semantykę. Kod aplikacji musi zawierać fragmenty realizujące wymagany schemat sterowania w oparciu o proste prymitywy, fragmenty te są przeplecione z właściwym kodem realizowanego algorytmu obliczeniowego.

Przekazywanie komunikatów ilustruje omawianą sytuację, daje ono bardzo proste elementy budulcowe, użycie tych elementów musi być wsparte dodatkowym kodem organizującym bardziej złożone wzorce synchronizacji. Tekst programu poprzeplatany jest procedurami przekazywania komunikatów otoczonymi kodem wspomagającym sterowanie. Dodatkową trudność stanowi konieczność ustalenia konkretnych par nadawca-odbiorca, przy czym należy ustalić nie tylko procesy, ale też konkretne instancje instrukcji komunikacji wstawionych w kod programu.

Semafory udostępniają skromną wbudowaną zasadę synchronizacji. One też wymagają dopisywania własnego kodu organizującego bardziej złożone strategie sterowania. Procesy traktowane są anonimowo, zatem jeśli istnieje potrzeba precyzyjnego ich rozróżniania, programista musi opracować metodę na to pozwalającą.

Możliwość dostosowywania operacji synchronizujących do potrzeb Ta kategoria zawiera metody sterowania, w których rodzaj i sposób działania operacji sterujących można zdefiniować. Po zdefiniowaniu, programista ma do dyspozycji prymitywy sterujące dokładnie dopasowane do konkretnych potrzeb. Stosując je w programie nie musi już dopisywać dodatkowego kodu realizującego wymagany schemat sterowania.

Monitory zawierają w swym kodzie wymagany schemat synchronizacji, procesy jedynie korzystają z przygotowanych operacji o dowolnej złożoności służących realizacji wymaganego schematu sterowania. Monitor jest centralnym zarządcą, zna globalny stan aplikacji w zakresie potrzebnym do realizowanych operacji synchronizacji. Innym przykładem jest system ACLT oparty się na Lindzie [39]. Klasyczne operacje na przestrzeni krotek poszerzono o możliwość definiowania reakcji na wstawienie określonej krotki. Zdefiniowana reakcja jest wyzwalana automatycznie w momencie, gdy oczekiwana krotka pojawia się w przestrzeni. Reakcje mogą być dowolnie złożone, mogą wyzwalać kolejne reakcje. Obowiązuje zasada wszystko albo nic - albo zostanie wykonana cała treść reakcji, albo gdy choć jeden fragment się nie powiedzie, całość jest odwoływana (np. w kodzie reakcji wczytanie pewnej krotki zawiodło, bo brak takowej w przestrzeni). Taki system pozwala budować złożone procedury sterujące.

1.1.2.4 Podział metod synchronizacji wg ich związku z komunikacją danych używanych w obliczeniach

Przesyłanie danych oraz koordynacja dostępu do nich to istotne problemy w programowaniu współbieżnym. Różne metody synchronizacji są w różnym stopniu związane z przekazywaniem danych. Część metod koncentruje się na przysyłaniu danych, ale też znamy metody sterowania pomijające w ogóle ten aspekt. Chcielibyśmy tu wprowadzić rozróżnienie pomiędzy danymi przetwarzanymi przez aplikację, np. macierz reprezentująca rozwiązywany układ równań, od danych służących sterowaniu, np. numer ostatnio zakończonej iteracji. Jest oczywiste, że każdy nietrywialny schemat sterowania musi posługiwać się pewnymi strukturami danych zawierającymi niezbędne parametry kontrolne czy wartości reprezentujace bieżący stan obliczeń. Przesyłanie tego typu danych

traktujemy jako integralną część mechanizmu sterowania, nie stanowi ono właściwego mechanizmu przesyłania danych. Zdajemy sobie sprawę, że podane rozróżnienie bazujące na znaczeniu transferowanych informacji nie jest ścisłe. Intuicyjnie widać jednak, że na potrzeby sterowania powinno wystarczyć przesyłanie pojedynczych wartości, podczas gdy przesyłanie danych aplikacji może wymagać transferów dużych objętości danych.

Ścisły związek Należące tu metody synchronizacji są w gruncie rzeczy metodami przekazywania danych obliczeniowych. Możliwości sterowania wynikają ze sposobu realizacji przekazywania danych. Synchroniczne przesłanie komunikatu (rendez-vous) wymaga, aby nadanie i odbiór nastąpiły jednocześnie, przez co strony komunikujące się zarazem synchronizują swój bieg. W przypadku asynchronicznego wysyłania wymagane jest tylko, aby odbiorca czekał na przybycie komunikatu. W obu sytuacjach zasadniczym celem jest jednak transfer informacji zawartej w komunikacie. Metody zdalnego dostępu do pamięci idą dalej, umożliwiając zdalny dostęp do danych bez angażowania i informowania strony udostępniającej swą pamięć. Gdy potrzebna jest dalej idąca koordynacja, musi być ona dodatkowo wyspecyfikowana. Podobnie jest w przypadku rozproszonej pamięci dzielonej - globalny dostęp do danych stanowi sedno tego rozwiązania, koordynacja tego dostępu jest wtórnym dodatkiem, koniecznym do zapewnienia poprawności działania programu. Paradygmaty dataflow i makro dataflow, zajmują się ściśle przepływami danych. Dostępność danych w naturalny sposób, wg grafu zależności, zarządza przepływem sterowania w aplikacji.

Częściowy związek W metodach tu należących przekazywanie danych może odgrywać znaczącą rolę. Oferowane są jednak pewne dodatkowe możliwości sterowania, które można stosować niezależnie od transmisji danych. Dla przykładu - sterowanie przepływem komunikatów polega na dostarczeniu argumentów, czyli przekazaniu danych, oraz na uruchomieniu określonej metody. Wspomniane dodatkowe możliwości sterowania są widoczne w wymienianym już systemie Dagger [57]. Dagger daje programiście narzędzia służące do czasowego zablokowania możliwości uruchomienia wskazanych metod - są to operacje służące już wyłącznie do sterowania.

Brak związku Należące tu rozwiązania koncentrują się wyłącznie na sterowaniu, a nie na przekazywaniu danych obliczeniowych. Służą innym celom niż wymiana informacji albo są uzupełniane osobnym mechanizmem pozwalającym przekazywać dane. Zaliczamy tu semafory i barierę. Brak w nich jakiejkolwiek funkcjonalności mogącej służyć przekazywaniu danych dla obliczeń. Także monitory służą do synchronizacji oddzielonej od przekazywania danych obliczeniowych. Wprawdzie, istnieje możliwość użycia monitora jako centrum wymiany danych, wprowadzanych jako parametry procedur, wyjmowanych jako wartości funkcji. Potraktujemy jednak monitory zgodnie z rozróżnieniem zaprezentowanym wyżej, traktując dane wymieniane za pośrednictwem monitora jako dane służące sterowaniu i obsługiwane przez mechanizm sterowania, a nie przez mechanizm przesyłania danych.

1.1.2.5 Zastosowanie synchronizacji w systemach scentralizowanych i rozproszonych

Metody synchronizacji oparte o dzielone zasoby w naturalny sposób wykorzystywane są w systemach scentralizowanych. Przykładami są tu semafory, monitory, wspólna przestrzeń danych. Istnieją wyjątki od tej reguły, np. realizacje semaforów w systemie rozproszonym [100]. Można spotkać też implementacje języka Linda, języka posługującego się globalną przestrzenią krotek, na komputery z pamięcią rozproszoną, lub inne realizacje pamięci globalnej w systemach rozproszonych. Te i inne podobne wysiłki mają na celu danie do dyspozycji programiście wygodnej abstrakcji znanej z systemów scentralizowanych, a normalnie nie dostępnej w przypadku systemów rozproszonych.

Metody sterowania oparte na bezpośredniej interakcji procesów dobrze są dostosowane do funkcjonowania w systemach rozproszonych. Przekazywanie komunikatów jest tu flagowym przykładem. W klasycznych systemach rozproszonych wręcz brak innych możliwości interakcji mię-

dzyprocesowej, niż wymiana komunikatów. Brak pośredniczących centralnych zasobów pozwala na dobrą skalowalność i poprawia wydajność.

1.1.3 Wnioski

Na przestrzeni lat konstrukcja systemów równoległych, dla których opracowywano metody sterowania, zmieniła się zasadniczo. Obecnie wiodącą rolę odgrywają systemy dysponujące setkami i tysiącami procesorów oraz systemy komputerów połączone siecią ogólnego przeznaczenia. Co za tym idzie, systemy te posiadają pamięć rozproszoną, ewentualnie pamięć współdzieloną rozproszoną. Tradycyjne metody synchronizacji oparte na współdzielonych skupionych zasobach stały się nieadekwatne do nowych potrzeb. Wiodącą rolę zaczęła odgrywać wymiana rozproszonych danych. Przekazywanie komunikatów oraz zdalny dostęp do pamięci stały się wiodącym paradygmatem w praktyce programowania równoległego. Pozwalają one tworzyć przenośne i efektywne programy. Jednak użycie przekazywania komunikatów jest stosunkowo trudne, oferuje niski poziom abstrakcji operacji synchronizujących i nie daje żadnego systemowego wspomagania pomocnego w realizacji sterowania w aplikacji równoległej. Wygoda użycia metod posługujących się wspólnymi zasobami, skontrastowana z trudnością stosowania przekazywanie komunikatów czy RDMA, dała motywację do prób przeniesienia tych metod do środowisk rozproszonych. Mnogość takich prób, szczególnie chodzi tu o realizacje rozproszonej pamięci dzielonej wsparte rozmaitymi (brak tu standardu) metodami synchronizacji [67], wskazuje, że problem jest trudny a dotychczasowe wyniki nie są zadowalające [98]. Monitory nie zostały zapomniane, a wręcz przeciwnie - znalazły szerokie zastosowanie dzięki ich włączeniu do języka Java. Daje to w sumie świadectwo, że metody sterowania opierające się o dzielone zasoby są preferowane z punktu widzenia wygody ich użycia. Z przedstawionej wyżej systematyki można wnioskować, że pozwalają też one na słabsze powiązanie procesów. Kłopoty wynikłe z konieczności silnego powiązania (wzajemnej znajomości) procesów w przypadku metod typu przekazywanie komunikatów dostrzeżono dawno. Środkiem zaradczym jest zwykle zastosowanie pewnego rodzaju dzielonych zasobów - globalnych usług katalogujących procesy - pozwalających procesom na wzajemne znalezienie się, np. usługa portmap w UNIX, mechanizm mailbox w PVM.

W czasach dominacji systemów scentralizowanych wyraźnie widać było ewolucję metod sterowania w programach równoległych w kierunku umożliwienia dostosowania dostępnych dla programisty operacji do jego potrzeb. Proste operacje semafory zostały wyparte przez strukturalnie zorganizowane monitory. Monitory nadal znajdują uznanie w wysokopoziomowych językach programowania, podczas gdy operacje semaforowe oraz typu test&set praktycznie stosuje się tylko w programowaniu systemowym. Obecnie, w dobie dominacji metod opartych na przekazywaniu komunikatów, brak jest efektywnych rozwiązań synchronizacji przystosowanych do działania w systemach rozproszonych, a pozwalających na definiowanie działania operacji synchronizujących, później stosowanych w kodzie aplikacji.

Konieczność częstego przesyłania danych w środowiskach z pamięcią rozproszoną spowodowała zepchnięcie synchronizacji na drugi plan przez operacje przekazywania danych. Sądzimy jednak, że osobna specyfikacja problemu sterowania, niezależnie od transmisji danych w programie, pozwoli na lepsze ustrukturalizowanie programów równoległych oraz może dać programiście wygodniejszy i efektywny mechanizm kontroli wykonania programu.

Będziemy dążyć zatem do opracowania mechanizmu sterowania wykonaniem programów równoległych, który działa w środowiskach rozproszonych, a zarazem daje programiście wygodę charakterystyczną dla metod bazujących na wspólnych zasobach, pozwala na określanie sposobu działania stosowanych w programie prymitywów sterujących oraz jest niezależny od mechanizmów przekazywania danych.

Zaobserwujmy, że decyzje sterujące w procesach są podejmowane na podstawie analizy stanu lokalnego procesu, np. wartości danej zmiennej. W programach równoległych uzyskanie skoordynowanego działania procesów wymaga, aby wpierw do stanu lokalnego danego procesu wprowadzić informacje o stanie innych procesów (np. odbierając od nich komunikaty), tak aby decyzje sterujące mogły (pośrednio) uzwględnić te stany. Istniejące realizacje tego podejścia pozostawiają na programiście cały ciężar opracowania aparatu wymiany danych o stanach procesów - kto komu,

kiedy i co musi przekazać. Dopiero po uzyskaniu dostępu do potrzebnych informacji programista może umieścić w programie kod na ich podstawie podejmujący decyzje sterujące.

Alteratywnym rozwiązaniem jest delegowanie podejmowania decyzji sterujących do zewnętrznych obiektów znających stosowny wycinek stanu całości aplikacji - przykładem może być semafor (wstrzymać, czy nie) lub monitor (czy wstrzymać, który proces uaktywnić). Interakcje procesów z takimi bytami przekazują im potrzebną informację. Jednak byty te pozostają bierne (proces musi sam zwrócić się do nich o decyzję) oraz ich rola ogranicza się praktycznie do wstrzymywania/wznawiania działania procesów.

Syntezą obydwu podejść może być podejmowanie decyzji sterujących na podstawie analizy stanu globalnego aplikacji. Stan globalny uzyskujemy przez odpowiednie połączenie stanów lokalnych procesów wchodzących w skład aplikacji równoległej. Stan taki można udostępniać zainteresowanym procesom, aby na jego podstawie podejmowały one decyzje sterujące. Można też wydzielić osobny byt analizujący na bieżąco stany globalne aplikacji i aktywnie przekazujący procesom podejmowane przez siebie decyzje sterujące. Obie wersje pomysłu można zrealizować wprowadzając w środowisko gotową infrastrukturę wyznaczania i analizy stanów globalnych aplikacji, co znakomicie uprości i ustrukturalizuje implementację sterowania w programach równoległych. W niniejszej pracy postanowiliśmy przebadać właśnie takie rozwiązanie.

Poniżej prezentujemy potrzebną wiedzę dotyczącą wyznaczania stanów globalnych oraz ich analizy, oraz stosowany w dalszej części pracy aparat formalny.

1.2 Stany spójne w systemach równoległych i rozproszonych

1.2.1 Ogólna teoria globalnych stanów spójnych

Od tej pory zajmiemy się systemami równoległymi, w których brak wspólnych zasobów. Procesy mają dostęp do lokalnej pamięci, wszelka komunikacja między procesami możliwa jest dzięki wymianie komunikatów. Ta klasa systemów jest niezwykle popularna. Zaliczają się tu zarówno klastry komputerów połączonych siecią ogólnego przeznaczenia, zespoły komputerów komunikujących się przez internet, jak także systemy równoległe stosujące dedykowane szybkie sieci. Statystyki wskazują, że maszyny równoległe z pamięcią rozproszoną stanowią coraz większy odsetek wśród komputerów wysokiej wydaności na świecie [2]. Pprzy braku wspólnych zasobów, metody opracowane dla omawianej klasy systemów, można stosować także w systemach scentralizowanych.

Definicja systemów, kórymi bedziemy się zajmować w dalszej części tego rozdziału jest następująca:

- system równoległy $\mathcal{SR}$ składa się z N sekwencyjnych procesów $P_1..P_N$,
- procesy mogą porozumiewać się ze sobą każdy z każdym dzięki wymianie komunikatów poprzez jednokierunkowe asynchroniczne kanały komunikacyjne, graf procesów połączonych kanałami jest silnie spójny,
- komunikaty mogą zostać odebrane w kolejności innej niż kolejność ich nadania, innymi słowy czasy przesłania komunikatów są różne i nieprzewidywalne
- sieć oraz procesy działają niezawodnie - komunikaty nie giną, procesy wykonują swój kod bezbłędnie.

Interesować nas będzie przebieg obliczeń w takim systemie. Na poziomie pojedynczego procesu obliczenie jest reprezentowane przez sekwencję zdarzeń e^0, e^1, ... Zdarzeń tych są trzy typy:

- wysłanie komunikatu, *send(m)* oznacza zdarzenie wysłania komunikatu m,
- odebranie komunikatu, *recv(m)* oznacza zdarzenie odebrania komunikatu m,
- zdarzenie wewnętrze procesu, zmieniające jego stan niezależnie od komunikacji.

Proces P_i znajduje się w stanie lokalnym s_i^k pomiędzy wystąpieniem zdarzeń e_i^k a e_i^{k+1}. Formalnie będziemy utożsamiać stan s_i^k ze skończoną sekwencją zdarzeń $e_i^0 \dots e_i^k$ z procesu P_i (taka sekwencja bywa zwana odcięciem, ang. *cut*). Zdarzenie e_i^0 oznacza utworzenie procesu P_i, zatem stan s_i^0 to stan początkowy tego procesu. W dalszym ciągu, gdy użycie indeksów górnych nie będzie konieczne, będziemy je pomijać.

Stan kanału k_{ij} prowadzącego od procesu P_i do P_j to zbiór komunikatów, które P_i nadał, a których jeszcze P_j nie odebrał. Formalnie dla procesów P_i i P_j w stanach s_i^k i s_j^l: $k_{ij}^{kl} = \{m : send(m) \in s_i^k \wedge recv(m) \notin s_j^l\}$.

Stan globalny S systemu $\mathcal{SR}$ to zbiór stanów lokalnych wszystkich procesów oraz stanów wszystkich kanałów. Formalnie:

$$Sp = \{s_i : i = 1..N\},\ Sk = \{k_{ij}^{kl} : s_i^k \in Sp \wedge s_j^l \in Sp\},\ S = Sp \cup Sk$$

Rozwój sieci komputerowych oraz spowodowany tym rozwój systemów rozproszonych, spowodował wzmożone zainteresowanie problematyką określenia globalnego chwilowego stanu obliczeń. Przy braku wspólnej pamięci oraz wspólnego zegara zadanie to nie jest proste. Stan globalny składa się ze stanów lokalnych procesów oraz stanów kanałów. Informacja o składowych musi zostać zgromadzona w jednym miejscu, do jej przesłania trzeba użyć komunikatów. Jednak, skoro procesy nie mają wspólnego zegara, a czasy przesłania komunikatów są różne i nieprzewidywalne, to zgromadzenie informacji pochodzących z jednej i tej samej chwili czasu rzeczywistego jest niewykonalne. Dlatego, do ustalenia wzajemnej kolejności zdarzeń w systemie, co za tym idzie też kolejności stanów lokalnych, trzeba posłużyć się innymi kryteriami niż referencja do zegara czasu rzeczywistego. Zdarzenia w ramach pojedynczego procesu są uporządkowane liniowo według kolejności ich wystąpienia. Wzajemne uporządkowanie zdarzeń pochodzących z różnych procesów wymaga użycia dodatkowych kryterii. Dalsze rozumowanie przeprowadzimy stosując następującą, najpowszechniej stosowaną regułę porządkującą: pewne jest, że zdarzenie $send(m)_i$ w procesie P_i musiało nastąpić przed zdarzeniem $recv(m)_j$ w procesie P_j. Na tej podstawie można zbudować relację częściowo porządkującą zdarzenia w systemie $\mathcal{SR}$ na podstawie ich zależności przyczynowo-skutkowej. Powiemy, że $e_i^k \overset{ps}{\rightarrow} e_j^l$ (czyli, że zdarzenie e_i poprzedza zdarzenie e_j), gdy zachodzi jeden z następujących warunków:

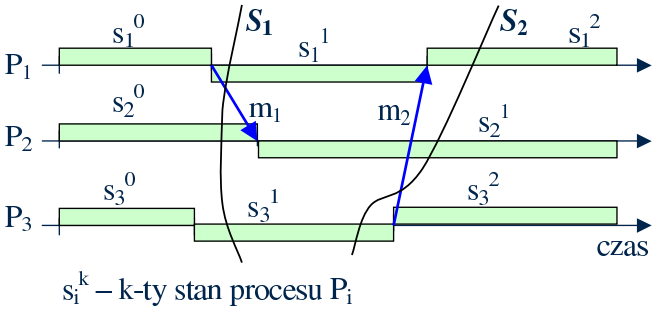
- $i = j$ oraz $k < l$ (e_i wystąpiło przed e_j)
- $e_i^k = send(m)$ oraz $e_j^l = recv(m)$ (zdarzenie e_i^k jest wysłaniem, a e_j^l odebraniem tego samego komunikatu, zdarzenia te są bezpośrednio zależne)
- $\exists e_m : e_i^k \overset{ps}{\rightarrow} e_m \wedge e_m \overset{ps}{\rightarrow} e_j^l$ (domknięcie przechodnie relacji bezpośredniej zależności)

Zdarzenia, dla których $e_i \overset{ps}{\nrightarrow} e_j \wedge e_j \overset{ps}{\nrightarrow} e_i$, gdzie $\overset{ps}{\nrightarrow}$ jest relacją przeciwną do $\overset{ps}{\rightarrow}$, nazywamy zdarzeniami współbieżnymi i oznaczamy je $e_i \parallel e_j$. Możemy teraz określić, co rozumiemy przez obliczenie równoległe:

Obliczenie równoległe O w systemie $\mathcal{SR}$ to zbiór zdarzeń z częściowo porządkującą je relacją $\overset{ps}{\rightarrow}$.

Definicja zależności przyczynowo-skutkowej dla stanów lokalnych procesów opiera się na wprowadzonej powyżej relacji na zdarzeniach (użycie tego samego symbolu nie powinno wprowadzić niejednoznaczności):

$$s_i^k \overset{ps}{\rightarrow} s_j^l \Leftrightarrow e_i^k \overset{ps}{\rightarrow} e_j^l.$$



Rysunek 1.1: Przykład globalnych stanów: spójnego (S1) i niespójnego (S2)

Stany, dla których $s_i \overset{ps}{\nrightarrow} s_j \wedge s_j \overset{ps}{\nrightarrow} s_i$, gdzie $\overset{ps}{\nrightarrow}$ jest relacją przeciwną do $\overset{ps}{\rightarrow}$, nazywamy stanami współbieżnymi i oznaczamy je $s_i \parallel^{ps} s_j$. Wprowadzone relacje $\overset{ps}{\rightarrow}$ są porządkami częściowymi na zbiorach odpowiednio zdarzeń oraz stanów lokalnych procesów $P_1 \dots P_N$. Relacje te stają się porządkami liniowymi po zawężeniu ich do, odpowiednio, zdarzeń oraz stanów pojedynczego procesu.

Teraz możemy zdefiniować pojęcie spójnego stanu globalnego. Intuicyjnie jest to stan, w którym procesy mogą znaleźć się jednocześnie - składowe stany lokalne procesów są wzajemnie współbieżne, czyli nie są związane relacją przyczyny i skutku. Odwołując się do definicji stanu lokalnego, jako skończonej sekwencji zdarzeń, możemy też powiedzieć, że globalny stan spójny to stan, w którego składowych stanach lokalnych dla każdego zdarzenia $recv(m)$ występuje odpowiadające mu zdarzenie $send(m)$. Na rysunku 1.1 widzimy, że zbiór stanów lokalnych $\{s_1^1, s_2^0, s_3^1\}$ wyznacza stan spójny S_1, natomiast zbiór $\{s_1^2, s_2^1, s_3^1\}$ stan niespójny S_2 - S_2 zawiera zdarzenie $recv(m_2)$, ale nie zawiera $send(m_2)$. Zbiór globalnych stanów spójnych będziemy oznaczać jako CGS (Consistent Global States). Ponieważ spójność stanów globalnych nie zależy w żaden sposób od stanu kanałów komunikacyjnych, oraz stan tychże kanałów jest wtórny względem stanu procesów (można go wyznaczyć rozpatrując te zdarzenia $send(m)$, dla których nie nastąpiły jeszcze odpowiadające im zdarzenia $recv(m)$), w dalszej części pracy stan kanałów będzie pomijany. Zatem formalnie:

$$S \in CGS \Leftrightarrow \forall_{s_i, s_j \in S} s_i \parallel^{ps} s_j.$$

Z każdym obliczeniem O w systemie $\mathcal{SR}$ związany jest wyznaczony przez nie zbiór $CGS(O)$.

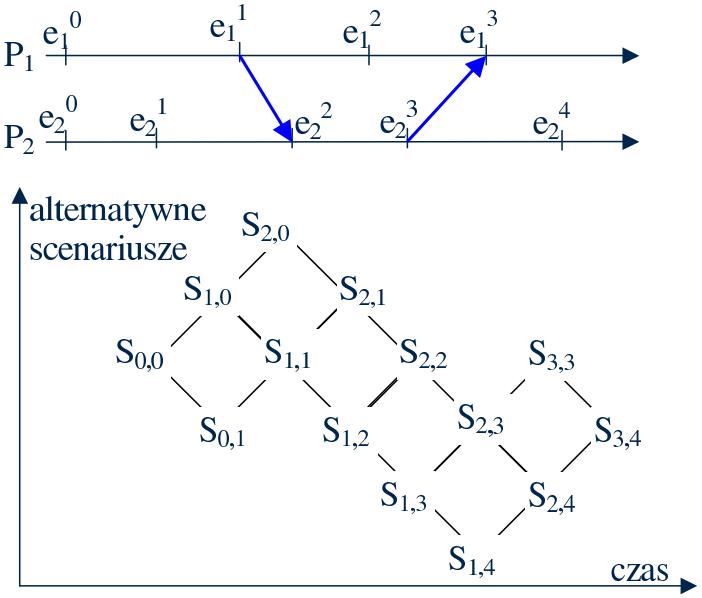
Zajmiemy się teraz wzajemnymi relacjami pomiędzy globalnymi stanami spójnymi. W naturalny sposób można określić następujący porządek częściowy na zbiorze CGS:

$$S_k \preceq S_l \equiv \forall_{i=1..N} \exists_{r,s>0} : \; s_i^r \in S_k \wedge \left(s_i^r \in S_l \vee s_i^{r+s} \in S_l \right),$$

czyli S_l składa się z tych samych stanów lokalnych, co S_k, lub stanów późniejszych. Intuicyjnie, porządek ten określa, który spójny stan globalny był wcześniejszy, który późniejszy oraz wystąpienia których stanów wzajemnie się wykluczają. Zbiór CGS wraz z określoną na nim relacją $\preceq$ oznaczamy jako $CGS^{\preceq}$. Stanowi on kratę o N osiach. Weźmy teraz dwa spójne stany S_k i S_l takie, że różnią się one tylko stanem lokalnym jednego dowolnie ustalonego procesu P_r - w S_l w procesie P_r nastąpiło o jedno zdarzenie więcej. Powiemy wtedy, że S_l jest *bezpośrednio osiągalny* z S_k. Oznaczmy tę relację przez $\twoheadrightarrow$. Formalnie

$$S_k \twoheadrightarrow S_l \Leftrightarrow \exists_{1 \leq r \leq N} \left(s_r^t \in S_k \wedge s_r^{t+1} \in S_l \wedge (\forall_{i=1..N, i \neq r} \; s_i \in S_k \Leftrightarrow s_i \in S_l) \right).$$

Przechodnie domknięcie relacji bezpośredniej osiągalności tworzy relację *osiągalności* pomiędzy spójnymi stanami globalnymi oznaczaną symbolem $\hookrightarrow$. Zapis $S_l \hookrightarrow S_k$ oznacza, że istnieje taka skończona sekwencja zdarzeń długości $d \geq 0$, której wykonanie doprowadzi system ze stanu S_k do



Rysunek 1.2: Przykład obliczenia oraz odpowiadającej mu kraty $CGS^{\hookrightarrow}$

stanu S_l poprzez d pośrednich globalnych stanów spójnych. Relacja ta jest porządkiem częściowym w CGS. Zbiór CGS wraz z określoną na nim relacją $\hookrightarrow$ oznaczamy jako $CGS^{\hookrightarrow}$. Stanowi on kratę izomorficzną z $CGS^{\preceq}$. Wynika to z faktu, że miminalny przyrost w sensie $\preceq$ w $CGS^{\preceq}$ odpowiada zawsze wystąpieniu jednego zdarzenia.

Przykład obliczenia z udziałem dwóch procesów oraz odpowiadającej mu kraty widać na rysunku 1.2. Użyte tam oznaczenie $S_{i,j}$ oznacza stan globalny składający się ze stanów lokalnych s_1^i oraz s_2^j.

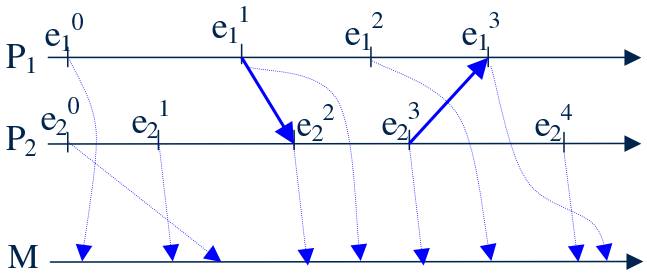
Ogólność przedstawionej teorii polega na tym, że relację $\stackrel{ps}{\rightarrow}$ można zastąpić innym porządkiem częściowym, który zawężony do pojedynczego procesu staje się porządkiem liniowym. Omówione własności pozostaną w mocy [112]. Pełniejsze omówienie teorii stanów spójnych, obejmujące też materiał przedstawiony dalej w punktach 1.2.2 i 1.2.3 można znaleźć m.i. w [9, 105].

1.2.2 Zależności między obliczeniem, jego obserwacjami oraz wyznaczonym zbiorem CGS

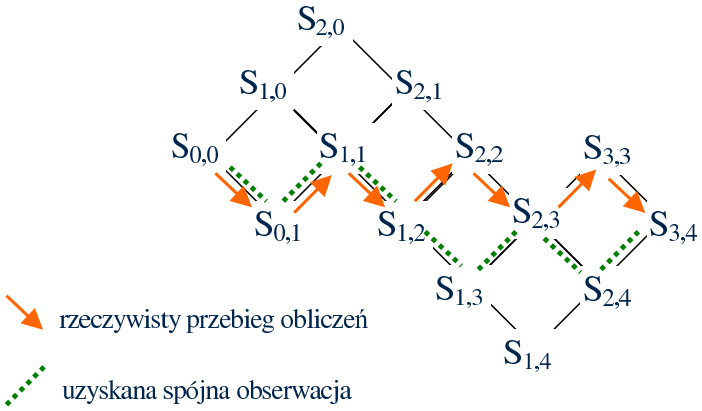
Dla systemów posiadających wspólną pamięć lub globalny zegar można zagwarantować, że obserwacja obliczenia jest w pełni zgodna z rzeczywistym przebiegiem obliczeń. Celem przedstawionej w punkcie 1.2.1 teorii jest umożliwienie prowadzenia obserwacji obliczeń w systemie $\mathcal{SR}$. Tutaj takiej zgodności nie można uzyskać.

Weźmy pod uwagę obliczenie przedstawione na rys. 1.2. Faktyczna kolejność zdarzeń jest w nim następująca: $e_1^0/e_2^0, e_2^1, e_1^1, e_2^2, e_1^2, e_2^3, e_1^3, e_2^4$. Wprowadźmy do systemu dodatkowy proces, obserwatora monitorującego przebieg obliczeń. Każdy proces aplikacyjny jest zobowiązany do powiadomienia obserwatora o każdym wystąpieniu zdarzenia poprzez wysłanie komunikatu do obserwatora z odpowiednią informacją. Na rysunku 1.3 widzimy możliwy scenariusz takiego powiadamiania. Przebieg obliczeń widziany przez proces monitorujący (monitor M) odpowiada następującej sekwencji zdarzeń : $e_1^0, e_2^1, e_2^0, e_2^2, e_1^1, e_2^3, e_1^2, e_2^4, e_1^3$. Jest to *obserwacja.* Jednak ta obserwacja nie może odpowiadać żadnemu obliczeniu, gdyż np. zdarzenie e_2^1 występuje w niej przed e_2^0. Jeśli obserwator dokona permutacji zaobserwowanej sekwencji zdarzeń tak, aby była ona zgodna z relacją przyczynowo-skutkową występującą między zdarzeniami w obliczeniu, może uzyskać np. sekwencję $e_1^0, e_2^0, e_2^1, e_1^1, e_2^2, e_2^3, e_1^2, e_2^4, e_1^3$. Tak uzyskana sekwencja jest spójną obserwacją.

Obserwacja spójna w systemie $\mathcal{SR}$ obliczenia równoległego O jest to taka sekwencję zdarzeń



Rysunek 1.3: Obserwacja obliczenia usyskana przez monitor (M)



Rysunek 1.4: Ścieżki odpowiadające uzyskanej spójnej obserwacji oraz rzeczywistej sekwencji zdarzeń.

> seq, że $e \in O \Leftrightarrow e \in seq$, oraz kolejność zdarzeń w seq jest zgodna z częściowym porządkiem $\overset{ps}{\rightarrow}$generowanym przez O.

Innymi słowy obserwacja powstaje przez rozszerzenie porządku częściowego $\overset{ps}{\rightarrow}$ do porządku liniowego.

Mając do dyspozycji informacje o zależności przyczynowo-skutkowej zdarzeń obserwator może przekształcić uzyskaną obserwację w obserwację spójną. Praktyczna strona tego zagadnienia opisana jest w punkcie 1.2.3. Obserwacja spójna odpowiada ścieżce od stanu początkowego $S_{0,0}$ do stanu końcowego $S_{3,4}$ w grafie reprezentującym kratę $CGS^{\preceq}$, taką ścieżkę będziemy nazywali *ścieżką wykonania*, odpowiednią ilustrację widzimy na rysunku 1.4. Ścieżek takich może być wiele. Jedna z nich odpowiada rzeczywistemu przebiegowi obliczeń. Nie istnieje jednak żadna metoda pozwalająca na odróżnienie tej ścieżki od pozostałych. W rezultacie obserwator może uzyskać nawet komplet alternatywnych sekwencji zdarzeń, czyli wszystkie możliwe spójne obserwacje danego obliczenia, lecz nie wie, która z tych sekwencji faktycznie miała miejsce. Jest to fundamentalny fakt charakteryzujący możliwość dokonywania obserwacji obliczeń w systemach zgodnych z definicją systemu $\mathcal{SR}$. Zauważmy jeszcze, że stany należace do ścieżki wykonania można uporządkować liniowo relacją $\preceq$ (oraz relacją $\hookrightarrow$, co wynika z izomorfizmu tych relacji). Natomiast stany nieporównywalne w sensie $\preceq$ nie mogą należeć do tej samej ścieżki wykonania.

1.2.3 Wyznaczanie globalnych stanów spójnych

Podstawy teoretyczne zaprezentowane w punkcie 1.2.1 mogą być wykorzystane w praktyce. Pierwszym krokiem w tym kierunku była praca Lamporta wprowadzająca pojęcie zegara logicznego [79]. Zegar logiczny stanowią zmienne lokalne procesów przyjmujące wartości naturalne, po jednej zmiennej na proces. ZL_i oznacza zegar logiczny procesu P_i. Proces dołącza bieżącą wartość swego

zegara do każdego wysyłanego komunikatu. Wartość zegara zmienia się w momencie wystąpienia zdarzenia według następujących reguł:

$ZL = ZL + 1$ w przypadku wystąpienia zdarzenia wewnętrznego lub zdarzenia *send*,

$ZL = max(ZL, ZL(m)) + 1$ w przypadku zdarzenia $recv(m)$, gdzie $ZL(m)$ oznacza wartość zegara przypisaną do odebranego komunikatu (timestamp).

Przez $ZL(e_i)$ oznaczmy wartość zegara P_i w momencie wystąpienia e_i. Gwarantowane jest, że

$$e_i \xrightarrow{ps} e_j \Rightarrow ZL(e_i) < ZL(e_j)$$

Jeśli zatem obserwator przesortuje otrzymane informacje o zdarzeniach rosnąco według $ZL(e)$, otrzyma obserwację spójną. Sortowanie, a zatem analizę obserwacji, można jednak wykonać dopiero po zakończeniu obliczeń, gdy zebrano już raporty o wszystkich zdarzeniach. Analiza w trakcie obliczeń wymagałaby utworzenia prefiksu końcowej posortowanej pełnej sekwencji zdarzeń, co jest niewykonalne. Załóżmy, że najwyższą wartość zegara w zbiorze zdarzeń znanych obserwatorowi przed końcem obliczeń ma e_i oraz $ZL(e_i) = K$. Posortowanie tego zbioru utworzy poprawny prefiks, o ile w przyszłości nie nadejdzie komunikat o pewnym zdarzeniu e_j, takim że $ZL(e_j) < K$. Takiej pewności nie ma. Opóźnienia w transmisji są nieprzewidywalne, a na podstawie już otrzymanych informacji nie można wywnioskować o (nie)istnieniu opóźnionego komunikatu. Innym mankamentem zegara logicznego Lamporta jest to, że nie można go użyć do sprawdzenia współbieżności zdarzeń: $\neg ZL(e_i) = ZL(e_j) \Rightarrow e_i \parallel e_j$. Zatem jest on niewystarczający do konstrukcji $CGS^{\preceq}$.

W pracy późniejszej Lamport i Chandy zaproponowali sposób określenia globalnego stanu spójnego w trakcie obliczeń [71]. Ich rozproszony algorytm (algorytm migawki) stosuje specjalne komunikaty kontrolne, których odbiór nakazuje procesom zapamiętanie ich bieżącego lokalnego stanu, względnie stanu kanału komunikacyjnego, którym nadszedł dany komunikat kontrolny. Zbiór zapamiętanych stanów w poszczególnych procesach jest spójnym stanem globalnym S, zwanym *migawką* (ang. snapshot). S jest osiągalny w sensie relacji $\hookrightarrow$ ze stanu, w którym zainicjowano algorytm migawki, a z S osiągalny jest stan, w którym algorytm ten się zakończył. Jednak nie ma gwarancji, że S faktycznie miał miejsce w trakcie obliczeń. Dla przykładu przyjmijmy, że algorytm migawki zainicjowano w stanie $S_{1,2}$, a zakończono w stanie $S_{3,4}$ według rysunku 1.4. Poszukiwanym stanem jest $S_{2,3}$, lecz rezultatem wykonania algorytmu (stanem globalnym obliczonym przez algorytm migawki) może być dowolny stan pomiędzy $S_{1,2}$ a $S_{3,4}$, np. $S_{1,3}$. Wówczas algorytm migawki nie wykryje, że obliczenie przeszło przez $S_{2,3}$. Możliwa jest też odwrotna pomyłka, jeśli przyjmiemy, że poszukiwanym stanem jest $S_{1,3}$ - rezultat wykonania algorytmu będzie sugerował, że ten stan zaistniał, co nie jest zgodne z rzeczywistym przebiegiem obliczeń. Z tego powodu migawki nadają się tylko do wykrywania stanów stabilnych. *Stan stabilny* to stan taki, że system raz osiągnąwszy go, pozostaje w nim. Przykładami stanów stabilnych mogą być blokada (deadlock) oraz zakończenie obliczeń. System nigdy tych stanów spontanicznie nie opuści. Oczywiście, po wykryciu blokady można podjąć specjalną akcję usuwającą blokadę i pozwalającą na kontynuowanie obliczeń. Jest to jednak interwencja zewnętrzna w stosunku do samych obliczeń.

Wprowadzenie logicznych zegarów wektorowych [47, 87] dało wreszcie narzędzie umożliwiające obserwację obliczeń w trakcie ich trwania (on-line) oraz pozwalające na konstrukcję wszystkich spójnych obserwacji danego obliczenia (czyli konstrukcję kraty $CGS^{\preceq}$) . Logiczny zegar wektorowy (ZW) ma postać wektora liczb naturalnych rozmiaru N, którego lokalna kopia znajduje się w każdym procesie. Przyjmijmy następujące oznaczenia:

$ZW_i[j]$ oznacza bieżącą wartość j-tej składowej wektora w procesie P_i,

$ZW(e)$ oznacza wartość zegara wektorowego przypisaną zdarzeniu e.

$ZW(e_i)[j]$ oznacza wartość j-tej składowej wektora w procesie P_i przypisaną wystąpieniu zdarzenia e_i,

$ZW(m)[j]$ oznacza wartość j-tej składowej wektora przypisanemu zdarzeniu, o którym informacja zawarta jest w komunikacie m,

$ZW(s)$ oznacza wartość zegara wektorowego przypisaną zdarzeniu rozpoczynającemu stan s.

Reguły zmiany wartości ZW są następujące:

- $ZW_i[i] = ZW_i[i] + 1$ po wystąpieniu zdarzenia wewnętrznego lub zdarzenia *send* w P_i, zdarzeniu temu przypisuje się nową wartość zegara,
- $\left.\begin{array}{l} ZW_i[i] = ZW_i[i] + 1 \\ \forall_{j=1..N, j\neq i} ZW_i[j] = \max(ZW_i[j], ZW(m)[j]) \end{array}\right\}$ po wystąpieniu zdarzenia $recv(m)$ w P_i, zdarzeniu temu przypisuje się nową wartość zegara

Intuicyjnie, zegar wektorowy pozwala określić, że najpóźniejsze zdarzenie w P_j, o którym wie P_i to $e_j^{ZW_i[j]}$. Zdarzenie $e_j^{ZW_i[j]}$ jest zarazem najpóźniejszym zdarzeniem w P_j poprzedzającym w sensie $\stackrel{ps}{\rightarrow}$ zdarzenie $e_i^{ZW_i[i]}$ w P_i.

Zdefiniujemy relację $<$ porządkującą częściowo zbiór wartości zegarów wektorowych:

$$ZW_i \quad < \quad ZW_j \Leftrightarrow ZW_i \neq ZW_j \wedge \forall_{k=1..N} ZW_i[k] \leq ZW_j[k]$$

Posługując się nią można określić współbieżność zdarzeń ($\neg$ oznacza negację) :

$$e_i \quad \|^{ps} \quad e_j \Leftrightarrow \neg ZW(e_i) < ZW(e_j) \wedge \neg ZW(e_j) < ZW(e_i)$$

a co za tym idzie współbieżność stanów lokalnych procesów i na tej podstawie wykrywać globalne stany spójne. Z kolei uzyskany zbiór CGS można uporządkować relacją $\hookrightarrow$ uzykując pełną kratę $CGS^{\hookrightarrow}$. Jeśli przez $S_i(k)$ oznaczymy stan procesu P_k, wchodzący w skład stanu globalnego S_i, to

$$S_i \hookrightarrow S_j \Leftrightarrow \forall_{k=1..N} ZW(S_i(k)) < ZW(S_j(k)) \vee ZW(S_i(k)) = ZW(S_j(k))$$

Inkrementalna budowa $CGS^{\preceq}$ w trakcie obliczeń jest wykonalna, gdyż zegar wektorowy pozwala stwierdzić istnienie zdarzeń wcześniejszych w sensie $\stackrel{ps}{\rightarrow}$ niż zdarzenia znane już obserwatorowi. Jeśli takie zdarzenia istnieją, obserwator musi zaczekać na przybycie informacji o nich. Szczegółowe omówienie tematyki zegarów logicznych i ich użycia do konstrukcji globalnych stanów spójnych można znaleźć w [9, 105]. Konstrukcja $CGS^{\preceq}$ w kontekście ewaluacji predykatów określonych na stanach globalnych omówiona jest w kolejnym punkcie.

1.3 Predykaty określone na stanach spójnych

Obserwacja obliczenia równoległego ma zwykle na celu weryfikację pewnych własności tego obliczenia. Na przykład może nas interesować, czy w procesach P_2 i P_3 jednocześnie wartość zmiennej `temperatura` była większe niż 200, albo czy zdarzyło się, że dwa procesy jednocześnie były w sekcji krytycznej. Warunki logiczne określone na stanach globalnych nazywane są *predykatami globalnymi*. Ich wartościowanie ma sens tylko w spójnych stanach globalnych. Choć powyższe przykładowe predykaty są intuicyjnie dobrze sformułowane i zrozumiałe, to kwestia sprawdzenia czy są one spełnione w danym obliczeniu w systemie $\mathcal{SR}$ wymaga dodatkowych rozważań. Przyjrzyjmy się dokładniej jednemu z podanych przykładów - weźmy predykat globalny o postaci $\varphi = {}_2 > 200 \wedge {}_3 > 200$. Załóżmy, że temperatura z naszego przykładu może rosnąć i maleć w trakcie obliczeń, czyli że φ może wiele razy stawać się prawdziwe na pewien czas. Powiemy, że φ jest predykatem niestabilnym. Omówiliśmy już wcześniej w punkcie 1.2.3 algorytm Lamporta i Chandy pozwalający uzyskać migawkę stanu systemu. Z powodów tam opisanych, algorytm ten może zarówno pominąć stany, w których φ było prawdziwe, jak też wyznaczyć stan nie mający miejsca w rzeczywistym przebiegu obliczeń, a w którym φ jest spełnione. Przydatność migawki polega na możliwości wykrywania predykatów stabilnych, czyli takich, które gdy raz staną się prawdziwe, pozostają prawdziwe. Formalnie:

$$\varphi(S_i) \text{ jest predykatem stabilnym} \equiv \forall_{S_j \in CGS}\, \varphi(S_i) \wedge S_i \hookrightarrow S_j \Rightarrow \varphi(S_j).$$

Jeżeli przyjmiemy, że w rozważanym przykładzie temperatura może tylko rosnąć, φ stanie się predykatem stabilnym. Periodycznie uruchamiana migawka zostanie w końcu uruchomiona w stanie $S_i : \varphi(S_i)$ i skonstruuje stan, w którym φ jest spełnione.

Inne podejście do zagadnienia weryfikacji, czy dany predykat niestabilny był spełniony w danym obliczeniu zapoczątkowali Marzullo i Neiger [85]. Zauważyli oni, że skoro nawet konstruując kratę $CGS^{\preceq}$, czyli używając całej dostępnej o obliczeniu informacji, nie jesteśmy w stanie określić stanów przez które obliczenie rzeczywiście przeszło (patrz 1.2.2), to pytanie, czy predykat φ został spełniony jest po prostu źle postawione. Zaproponowali oni dwie interpretacje warunków spełnienia predykatu (tzw. *modalności* predykatów), a mianowice:

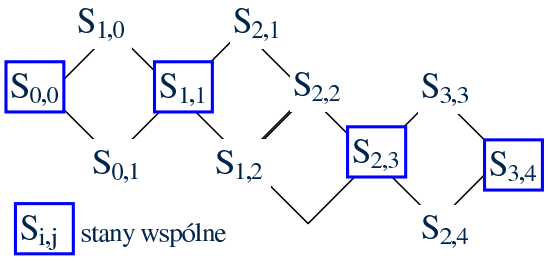
Possibly oznaczane Poss(φ): Poss(φ) jest prawdziwe, gdy φ mógł być spełniony w obliczeniu, tzn. istnieje ścieżka wykonania zawierająca taki stan S, że $\varphi(S)$ jest prawdziwe.

Definitely oznaczane Def(φ): Def(φ) jest prawdziwe, gdy φ na pewno był spełniony w obliczeniu, tzn. każda ścieżka wykonania zawiera taki stan S, że $\varphi(S)$ jest prawdziwe.

Algorytmy wykrywania Poss(φ) i Def(φ) korzystają z mechanizmu zegarów wektorowych i budują kratę $CGS^{\hookrightarrow}$. Poss(φ) jest wykrywane w momencie znalezienia pierwszego stanu spójnego, w którym φ jest prawdziwe, natomiast do wykrycia Def(φ) konieczne jest sprawdzenie, że każda ścieżka zawiera taki stan. Algorytmy te muszą przejrzeć wszystkie globalne stany spójne w pesymistycznym przypadku. Ponieważ moc zbioru CGS w pesymistycznym przypadku (brak komunikacji między procesami) wynosi E^N, gdzie E jest maksymalną liczbą zdarzeń w pojedynczym procesie, koszt wykrycia Poss(φ) i Def(φ) wynosi $\Omega(E^N)$. Z tego powodu wykrywanie Poss(φ) i Def(φ) jest niezwykle czasochłonne, a często wręcz niemożliwe. Nic dziwnego, że redukcja tego kosztu była motywem szeregu dalszych prac. Zwrócono uwagę, że spełnienie pewnych zawężonych klas predykatów globalnych można sprawdzić łatwiej. Garg i Waldecker [49, 51] podali scentralizowane algorytmy sprawdzania spełnienia Poss(φ) i Def(φ) w koszcie $O(N^2E)$ dla φ w postaci koniunkcji predykatów lokalnych. Rozproszony algorytm detekcji Poss(φ) o tym samym koszcie dla także predykatów w postaci koniunkcji predykatów lokalnych można znaleźć w [65]. Koszt $O(NE)$ uzyskał Minas dla predykatów będących koniunkcją dwóch predykatów lokalnych [92]. Dalsze badania nad wybranymi klasami predykatów zaowocowały wnioskiem, że problem weryfikacji spełnienia predykatu w postaci klauzulowej normalnej k-CNF, k>1, jest NP-zupełny [93] (postać k-CNF to taka, w której predykat składa się z koniukcji alternatyw, z których każda jest alternatywą k zmiennych logicznych lub ich negacji).

Osobnym kierunkiem prac mających na celu zmniejszenie kosztu monitorowania obliczeń jest redukcja ruchu w sieci powodowanego przez system monitorujący. Jednym z najistotniejszych czynników są tu znaczniki czasu wektorowego, doczepiane do każdego komunikatu aplikacyjnego. Znacznik taki ma rozmiar N, zatem stanowi istotny narzut w systemach o większej liczbie procesorów. Unikanie przesyłania częsci wektorów już znanych odbiorcy pozwala zredukować ten rozmiar w średnim przypadku kosztem użycia dodatkowej pamięci w każdym procesie oraz rezygnacji z natychmiastowej dostępności części informacji [109]. Odtworzenie pełnej relacji zależności przyczynowo-skutkowej jest kosztowne i raczej wymaga przetwarzania off-line [105]. Metodę użycia znaczników w stałym rozmiarze, niezależnym od liczby procesów w systemie, opracowali Baldoni i Melideo [11]. Także tu jednak część informacji o relacjach przyczynowo-skutkowych przestaje być łatwo dostępna, a jej odtworzenie wymaga czasu i dodatkowych obliczeń.

Wykładniczy koszt sprawdzania predykatów w modalnościach Definitely i Possibly wynika z konieczności przejrzenia potencjalnie wszystkich globalnych stanów spójnych. Przy tym, nawet wykrywając Poss(φ) lub nie wykrywając Def(φ) nie mamy pewności, czy predykat φ był rzeczywiście spełniony w danym obliczeniu, czy też nie. Niezwykle kuszące w tej sytuacji byłoby branie pod uwagę tylko faktycznej ścieżki wykonania obliczenia, a nie całego zbioru CGS. Długość takiej ścieżki jest liniowa względem liczby zdarzeń, do tego analiza stanów faktycznie zaistniałych pozwoliłaby wnioskować o rzeczywistym przebiegu obliczeń. Do tych założeń zbliża się propozycja opisana w [36]. Zdefiniowana tam modalność *Currently* oznacza badanie wartości predykatu na



Rysunek 1.5: Przykład stanów wspólnych.

bieżących stanach globalnych. Aby to było możliwe, procesy kooperują z obserwatorem. Przed każdą taką zmianą lokalnego stanu, która może uczynić badany predykat globalny φ fałszywym, pytają o pozwolenie. Do czasu jego uzyskania zawieszają obliczenia. Pozwolenie jest wydawane, gdy obserwator uwzględni już dotychczas uzyskane informacje. Dodatkowe komunikaty pozwalają lepiej uporządkować obserwatorowi zdarzenia i tym samym zmniejszyć liczbę analizowanych stanów. Niestety, algorytm wymaga „świadomego" udziału procesów w wykrywaniu predykatu, oraz może blokować ich działanie. Blokowanie jest szczególnie niekorzystne, gdyż spowalnia i potencjalnie zmienia przebieg obliczeń. Odmienny pomysł na analizę rzeczywistego przebiegu obliczeń i niski koszt tej analizy opublikowali Fromentin i Raynal [48]. Zauważyli oni, że w kracie $CGS^{\preceq}$ mogą istnieć stany należące do każdej ścieżki wykonania. Są to *stany wspólne* (common states), patrz rysunek 1.5. Wykrycie predykatu φ w stanie wspólnym gwarantuje więc, że obliczenie faktycznie spełniło ten predykat, do tego mamy pewność w którym rzeczywiście zaistniałym stanie predykat ten został spełniony. Wprowadzona modalność *Properly* (Prop(φ)) oznacza określenie i sprawdzanie predykatów globalnych tylko na stanach wspólnych. Autorzy opracowali wielomianiowy algorytm detekcji stanów wspólnych pozwalający efektywnie wykrywać Prop(φ). Zasadniczą wadą zaproponowanego podejścia jest trudność zagwarantowania istnienia stanów wspólnych, a w szczególności ich istnienia w momentach, w których wykrycie φ byłoby pożądane (np. na rys. 1.4 stanów wspólnych brak, poza stanami początkowym i końcowym). Podane częściowe rozwiązanie tego problemu opiera się na użyciu omawianych wcześniej predykatów globalnych, będących koniunkcją predykatów lokalnych, jest zatem bardzo ograniczone.

Ogólny i nieprecyzyjny charakter warunków odpowiadajacych spełnieniu Poss(φ) lub Def(φ) dał motywację do poszukiwań modalności pozwalających znacznie dokładniej określić sytuację, w której spełniony zostaje dany predykat. Kshemkalyani zaproponował drobnoziarnistą klasyfikację modalności, opartą o analizę wzajemnej relacji stanów lokalnych pary procesów [78]. Wyróżnił on aż 40 wzajemnie wykluczających się relacji, z których każda odpowiada pewnemu szczególnemu następstwu zdarzeń ograniczających badaną parę stanów. Tego rodzaju modalności można stosować także dla predykatów globalnych określonych na więcej niż dwóch procesach, lecz pod warunkiem, że użyte predykaty mają postać koniunkcji predykatów lokalnych.

Wykrywanie spełnienia predykatów globalnych w systemach, w których uporządkowanie zdarzeń możemy opierać jedynie na ich zależności przyczynowo skutkowej, pozostaje zagadnieniem trudnym. Wykładniczy koszt wykrycia spełnienia predykatu postaci ogólnej, mimo opracowania pewnych technik przyspieszających [113], utrudnia monitorowanie on-line obliczeń z udziałem większej liczby procesorów. Niedogodnością jest także brak jednoznacznej odpowiedniości pomiędzy wykrytym spełnieniem predykatu (przy zastosowaniu danej modalności), a konkretnym rzeczywistym stanem systemu.

1.3.1 Zastosowanie predykatów globalnych - przegląd literatury

Prace nad możliwościami określenia stanu systemu rozproszonego oraz sprawdzenia, czy pewne predykaty zostały spełnione w obliczeniu, w dużej mierze motywowane były potrzebami praktycznymi. Powszechność systemów rozproszonych spowodowała zwrócenie uwagi na tworzenie i testowanie aplikacji rozproszonych. Techniki używane do uruchamiania programów sekwencyjnych

okazały się mało przydatne. Powszechnie stosowane weryfikacje (assertions), warunkowe zatrzymania programu (breakpoints), czy po prostu odczyty aktualnych wartości zmiennych, nie dały się zastosować w zwykły sposób w systemach rozproszonych. Teoria spójnych stanów globalnych oraz predykatów globalnych umożliwiły adaptację tradycyjnych technik testowania i uruchamiania do rozproszonych środowisk obliczeniowych [92, 51, 50].

Drugim często wymienianym zastosowaniem testowania wartości predykatów globalnych jest monitorowanie jako takie. Systemy przemysłowe, linie technologiczne składają się z zestawu urządzeń oraz czujników połączonych siecią komunikacyjną. Niezbędne jest ciągłe monitorowanie pracy takich systemów w celu wykrywania stanów oznaczających niepoprawną pracę lub awarię. Monitor badający predykaty weryfikujące sprawność systemu jest wtedy nie tymczasowym dodatkiem, ale integralną częścią tegoż systemu [33].

Prace Marzullo i innych [85, 36], które zapoczątkowały badania dotyczące wykrywania predykatów globalnych, miały na celu opracowanie metodyki pozwalającej na monitorowanie i sterowanie aplikacji rozproszonych. Jednak wątek sterowania nie doczekał się szerszej kontynuacji, pozostawiając monitorowanie na pierwszym planie.

Podejmowanie decyzji sterujących wykonaniem aplikacji na podstawie rezultatów ewaluacji predykatów globalnych, jest koncepcją wykraczającą poza omówione powyżej tradycyjne zastosowania predykatów globalnych. Określenia *sterowanie* (control) w powiązaniu z użyciem predykatów globalnych jako jedni z pierwszych użyli Raynal i Helary [101]. W ich ujęciu problem sprowadzał się jedynie do obliczenia wartości funkcji, nazywanej funkcją sterującą, określonej na spójnym stanie globalnym. Funkcja musiała być monotonicznie rosnąca, aby zapewnić, że gdy badanym warunkiem jest $F(S) > T$, to procedura jego wykrywania spełnia własności żywotności (ang. liveness) i zapewniania (ang. safety). Praktycznie więc był to czyste monitorowanie predykatów globalnych, który teoretycznie i w bliżej nie sprecyzowany sposób mogłoby być użyte do sterowania.

System Meta wykorzystujący sterowanie oparte o predykaty globalne opisali Marzullo i Wood [86]. Wykorzystali oni koncepcję systemów reaktywnych. *System reaktywny* składa się z dwóch komponentów: środowiska i programu kontrolnego. Program kontrolny odczytuje (jest informowany) stan środowiska i może wpływać na środowisko przez aktywację tzw. mechanizmów wykonawczych, które środowisko udostępnia. Stan środowiska dostępny jest dzięki zestawowi tzw. sensorów, czyli funkcji pozwalających programowi kontrolnemu na odczyt określonych parametrów. Reakcja programu kontrolnego na zmiany parametrów środowiska jest automatyczna i natychmiastowa. Programowanie systemów reaktywnych zwane jest programowaniem reaktywnym [72]. Brak tutaj pojęcia danych wejściowych oraz wyników obliczeń, celem jest ciągła kontrola parametrów środowiska. Bardzo często chodzi tu zarazem o programowanie systemów czasu rzeczywistego, gdyż systemy reaktywne zwykle są właśnie systemami czasu rzeczywistego. Istnieją specjalizowane języki służące do tworzenie programów reaktywnych np. Esterel [13] czy Lustre [58]. Powszechnie używane w programowaniu reaktywnym pojęcie rozgłaszanego sygnału służy do aktywacji wspomnianych mechanizmów wykonawczych. Procesy, składające się na współbieżny program reaktywny, reagują na otrzymane sygnały, oraz w odpowiedzi emitują nowe sygnały. Emisja (propagacja) sygnału jest natychmiastowa. Pozwala to na przedstawienie programiście synchronicznego modelu środowiska równoległego. Założenia systemu Meta są nieco inne. System ten został zaprojektowany jako warstwa nadrzędna, spajająca zestaw istniejących aplikacji (pełniących rolę środowiska) w jedną aplikację rozproszoną. Aplikacje podrzędne musiały zostać wyposażone przez programistę w zestaw funkcji pełniących role sensorów oraz mechanizmów wykonawczych. Do ich kodu głównego trzeba było dołożyć wywołania procedur określających momenty, w których możliwy jest odczyt sensorów oraz uruchomienie mechanizmów wykonawczych. Program kontrolny Meta miał postać warunkowo wykonywanych poleceń w postaci $\varphi \rightarrow A_i.akcja$, interpretowanych następująco: gdy predykat globalny φ jest spełniony, uruchom funkcję $akcja$ w aplikacji podrzędnej A_i. Program taki był dołączany do aplikacji podrzędnych, najlepiej tak, aby wyliczenie predykatu oraz uruchomienie akcji w jak największym stopniu odbywały się lokalnie (predykaty często odnosiły się do stanu jednego lub tylko paru procesów). Predykaty wartościowane były na stanach spójnych należących do $CGS^{\hookrightarrow}$, czyli otrzymanych na podstawie relacji przyczyny i skutku. Globalne stany spójne były dostępne dla wszystkich aplikacji podrzędnych dzięki wykorzy-

staniu rozgłoszeń globalnie uporządkowanych liniowo zgodnie z relacją przyczynowości do realizacji komunikacji. Zaimplementowano przeglądanie tylko pojedynczej obserwacji obliczenia (ścieżki wykonania), a nie całej kraty spójnych stanów globalnych. Tak uproszczona semantyka pozwoliła na osiągnięcie odpowiednio małego kosztu ewaluacji predykatów globalnych i okazała się wystarczająca dla przypadków rozważanych przez autorów sytemu Meta. Istotnym szczegółem jest to, że monitorowane własności, w oparciu o które konstruowano predykaty globalne, rzadko zmieniały swe wartości, a zatem obserwowane stany globalne miały stosunkowo długi okres trwania. Bardzo niewygodna w użyciu postać programu kontrolnego systemu Meta, mocno ograniczona semantyka wykrywania spełnienia predykatów globalnych oraz pewne problemy z uzyskiwaniem rzeczywiście spójnych stanów (z uwagi na komunikacje aplikacji podrzędnych niekontrolowaną przez Meta), silnie ograniczyły zastosowanie tego systemu. Postać programu kontrolnego próbowano ulepszyć tworząc deklaratywny język Lomita, temat ten nie został jednak daleko rozwinięty.

Tradycyjne zastosowanie predykatów globalnych do wykrywania błędów przy uruchamianiu programów rozproszonych uzyskało nową postać za sprawą pracy Tarafdara i Garga [115]. Mając dany zapis obliczenia równoległego w postaci zbioru zdarzeń, wraz z relacją przyczyny i skutku częściowo porządkującą ten zbiór, autorzy zastanawiali się nad możliwościami rozszerzenia istniejącej relacji przyczyny i skutku tak, aby w obliczeniu odpowiadającym relacji rozszerzonej stale spełniony był pewien predykat globalny, nie spełniony w obliczeniu oryginalnym. Nie spełnienie predykatu odpowiadało wystąpieniu błędu w programie. Uzupełnienie programu o dodatkową synchronizację, reprezentowaną jako rozszerzenie relacji przyczynowo-skutkowej, w sposób prowadzący do spełnienia danego predykatu, oznaczało wyeliminowanie błędu i zostało nazwane mianem *kontroli predykatowej* (predicate control). Podstawowym problemem było tu znalezienie odpowiedniego rozszerzenia oryginalnej relacji przyczyny i skutku. Dla relacji wyznaczanej po zakończeniu obliczeń na podstawie zarejestrowanego śladu wykonania, problem ten dla predykatów w ogólnej postaci jest NP-zupełny. Autorzy podali algorytm wielomianowy dla predykatów w postaci alternatywy predykatów lokalnych oraz algorytm pozwalający zastosować kontrolę predykatową on-line (dla trwających obliczeń) zapewniającą spełnienie predykatu w postaci alternatywy predykatów lokalnych przy pewnych dodatkowych założeniach. Jako zasadniczy cel stosowania kontroli predykatowej widziano skrócenie cyklu testowania i uruchamiania aplikacji rozproszonych. Próbna modyfikacja sterowania/synchronizacji w aplikacji, osiągnięta przez użycie sterowania predykatami, powinna pozwolić na szybkie ustalenie zakresu niezbędnych korekt w kodzie programu.

Podejście zbliżone do idei wykorzystania predykatów globalnych do sterowania przedstawione jest w pracy [116]. Jej autor proponuje rozszerzenie klasycznych instrukcji sterujący (if, while, ..) tak, aby zakresem ich działania była cała aplikacja równoległa (lub określony podzbiór jej procesów). W tym celu wprowadzone są widoczne globalnie meta-zmienne, których wartości mogą być testowane przez globalne instrukcje sterujące. Meta-zmienne są dostępne w każdym procesie ich używającym dzięki replikacji. Konflikty dostępu i problem zgodności kopii nie występują dzięki zastosowaniu restrykcyjnego protokołu dostępu i modyfikacji. Poprzez wartości meta-zmiennych możliwe jest wyrażanie globalnego stanu aplikacji, testowanego przez globalne instrukcje sterujące. Znaczącym ograniczeniem jest tu oczekiwany koszt komunikacji indukowanej przez protokół replikacji meta-zmiennych (proponowana jest specjalna architektura sprzętowa), a także wspomiany restrykcyjny protokół modyfikacji wartości meta-zmiennych, wymagający podziału programu na fazy. W każdej fazie wartość danej meta-zmiennej może zmienić tylko jeden proces, wartość ta staje się widoczna dopiero w następujących fazach.

Koncepcja wprowadzenia instrukcji synchronizacji do systemu graficznego projektowania programów równoległych GRADE została przedstawiona w [119]. Instrukcje synchronizacji, w postaci nowego typu węzłów grafu, miały być używane w specjalnie zaproponowanym diagramie synchronizacji międzyprocesowej, pozwalając na definiowanie faz wykonania aplikacji równoległej. Fazy aplikacji graficznie przedstawione na diagramie łączone były za pośrednictwem węzłów reprezentujących instrukcje synchronizacji. Instrukcje synchronizacji brały pod uwagę parametry dostarczane jednorazowo przez procesy realizujące obliczenia poprzedzającej fazy. Procesy musiały zaczekać, aż instrukcja synchronizacji, na podstawie dostarczonych parametrów, zezwoli na przejście do kolejnej fazy obliczeń.

System monitorowania na bieżąco programów równoległych, umożliwiający sterowanie wyko-

naniem monitorowanego programu zaproponowano w pracy [55]. Ogólna zasada działania jest zbliżona do tej zaproponowanej w systemie Meta: procesy zostają wyposażone w sensory oraz funkcje wykonawcze. Sensory przekazują wybrane elementy stanu procesów do monitora, funkcje wykonawcze są aktywowane zdarzeniami (komunikatami) pochodzącymi od monitorów. System Falcon pozwala wizualnie przedstawić bieżący stan aplikacji równoległej np. obciążenie poszczególnych procesów, podobnie jak inne systemy monitorujące. Tu jednak użytkownik może wpływać na przebieg wykonania programu, ręcznie wpisując stosowne komendy, lub dzięki odpowiedniemu oprogramowaniu reakcji. Falcon został zaprojektowany do sterowania programami obliczeniowymi w systemach o architekturze SMP. Autorzy nie posługują się pojęciami stanów spójnych i predykatów globalnych. Problemy związane z potencjalnym brakiem spójności obserwowanych stanów aplikacji rozpatrują w kategoriach wizualizacji - uzyskiwane diagramy mogą być błędne. Proponowanym rozwiązaniem jest sortowanie zdarzeń przez monitor według reguł określonych dla konkretnej aplikacji, lub reguł systemowych gdy monitorowanie odbywa się na poziomie systemu. Reguły te określają dopuszczalną globalną kolejność zdarzeń. Zdarzenie, które nie spełnia reguł jest wstrzymywane. Opisy i przykłady tyczą się monitorowania na poziomie systemu (operacje na semaforach binarnych, tworzenie i usuwanie wątków, itp.) np. komunikat o wykonaniu dowolnej operacji przez wątek, którego utworzenie jeszcze nie zostało zarejestrowane przez monitor, będzie wstrzymany do czasu nadejścia wiadomości o utworzeniu wątku. Takie rozwiązanie wymaga osobnego podejścia do każdej aplikacji oraz umożliwia tylko częściowo właściwe uporządkowanie zdarzeń.

Symetryczne i rozproszone wyznaczanie predykatów globalnych, w modalnościach wykorzystujących częściowo zsynchronizowane zegary czasu rzeczywistego, zastosowano w pracy Mayo [88]. Pominięto tam etap osobnego wyznaczania globalnych stanów spójnych. Predykaty są wartościowane w sposób gwarantujący wykorzystanie danych odpowiadających pewnego rodzaju spójności stanu globalnego (patrz punkt 2.2). Opisane przez Mayo podejście nie jest ogólne - dla każdego predykatu stosuje się osobny specjalizowany algorytm. Podano kilka przykładów predykatów wraz z odpowiednimi algorytmami, przy czym skoncentrowano się na predykatach stabilnych. Potrzeba użycia innych predykatów oznacza konieczność opracowania właściwego algorytmu do jego wartościowania i wprowadzenie tego algorytmu w kod wszystkich procesów.

Wśród dotychczasowych opracowań brak jest rozwiązań pozwalających w sposób ogólny stosować dowolne predykaty globalne (w szczególności niestabilne), definiowane przez programistę w sposób oddzielny od podstawowego kodu procesów aplikacyjnych i wartościowane na bieżąco, jako metodę sterowania wykonaniem programów równoległych. Nierozwiązane dostatecznie pozostają zarówno sposoby wartościowania predykatów w odpowiednich modalnościach na stanach globalnych, jak też sposoby reagowania na spełnienie tychże predykatów. Niniejsza praca ma na celu wypełnienie tej luki.

1.4 Teza i cele pracy - możliwość udoskonalenia sterowania w oparciu o analizę stanów spójnych aplikacji

Jak przedstawiliśmy, predykaty globalne są pojęciem ściśle związanym z systemami rozproszonymi. Nie stanowią naprędce sformułowanego rozszerzenia koncepcji znanych z systemów scentralizowanych, są zbudowane na solidnych podstawach teoretycznych popartych wieloma praktycznymi aplikacjami. Nie tylko dobrze nadają się, ale wręcz zostały stworzone do zastosowania w systemach rozproszonych.

Jednocześnie predykaty globalne są ściśle związane z bytem wspólnym dla wszystkich procesów - ze stanami globalnymi. Wspominaliśmy, że kierunki rozwoju metod sterowania w programach równoległych sugerują większą wygodę użycia metod wykorzystujących współdzieloną informację. Wspólne zasoby umożliwiające współdzielenie informacji można zastąpić samą informacją - stanem systemu. Chcąc wykorzystać ten stan na potrzeby sterowania, wystarczy oczywiście posłużyć się pewną jego abstrakcją, zawierającą tylko te dane, które mają być wykorzystane do podejmowania decyzji sterujących wykonaniem programu.

Stan systemu oraz weryfikacja spełnienia predykatów globalnych nie spełniają roli przekazywania danych. Skonstruowany w oparciu o nie mechanizm sterowania mógłby być zupełnie niezależny od mechanizmu przesyłania danych. Odpowiada to postulowanemu oddzieleniu tych mechanizmów, co powinno zaowocować lepszą strukturalizacją programu.

W oczywisty sposób treść predykatu nie stanowi części kodu programu, którego wykonanie ma być monitorowane. Predykaty są zewnętrzne w stosunku do programu. Nic nie stoi na przeszkodzie, aby je definiować oddzielnie, gwarantując separację właściwego kodu aplikacyjnego od kodu sterującego wykonaniem programu. Taka struktura pozwoli na lepsze rozumienie kodu programu oraz łatwiejszą i niezależną modyfikację części aplikacyjnej i sterującej. Jednocześnie treść predykatów może wyrażać pewne złożone warunki, których spełnienie powinno wywołać określone reakcje. Uzyskalibyśmy w ten sposób możliwość uprzedniego i osobnego definiowania tego, co mechanizm sterujący ma robić, aby jego użycie w programie było możliwie najwygodniejsze i nie wymagało wprowadzania dodatkowego kodu pomocniczego w treść procesów aplikacyjnych.

Skoro predykaty globalne znalazły szerokie zastosowanie do weryfikacji poprawności działania programów rozproszonych, to mechanizm sterujący je wykorzystujący mógłby sam być użyty (bez dodatkowych narzędzi) do przeprowadzania takich weryfikacji. Dodatkowo, zamiast organizować w programie sterowanie stosując inne metody, by następnie weryfikować poprawność tak uzyskanego sterowania za pomocą predykatów (co jest zwykłą procedurą - napisz program, potem użyj monitora i programu uruchomieniowego), widzimy możliwość otrzymania sterowania poprawnego od razu, przez konstrukcję, gdy będzie ono bezpośrednio oparte o predykaty globalne, wyrażające warunki, które program ma spełniać.

Powyższe obserwacje są silną motywacją do opracowania metody sterowania w programach równoległych bazującej na ewaluacji predykatów określonych na stanach globalnych programu.

Po konfrontacji znanych metod sterowania w programach równoległych, omówionych w punkcie 1.1, z możliwościami wynikającymi z umiejętności oceny stanu globalnego aplikacji i sprawdzenia predykatów globalnych, można określić następujące cele tej pracy:

- opracowanie metody sterowania wykonaniem programu, rozłącznej od specyfikacji transmisji danych, która pozwoli na lepsze ustrukturalizowanie programów równoległych,
- opracowanie metody sterowania, która da programiście wygodny i efektywny mechanizm kontroli wykonania programu,
- opracowana metoda sterowania powinna działać w środowiskach rozproszonych, a zarazem dawać programiście wygodę charakterystyczną dla metod scentralizowanych,
- wyodrębnienie kodu specyfikującego sterowanie na poziomie aplikacji z całości kodu procesów,
- określanie sposobu działania stosowanych w programie prymitywów sterujących,
- eksperymentalna weryfikacja zaprojektowanej metody sterowania.

Postulaty te mogą zostać spełnione, gdy sterowanie w programie oprzemy na analizie predykatów globalnych.

Tezą pracy jest wykazanie, że sterowanie oparte o analizę stanów globalnych aplikacji dla wybranych klas problemów umożliwia poprawną i wygodną strukturalizację sterowania w programach, przy uzyskaniu nie gorszej, a w wielu wypadkach lepszej, wydajności obliczeniowej, niż przy użyciu klasycznych metod sterowania, polegających na specyfikacji sterowania i obliczeń przemieszanych z komunikacją poprzez przesyłanie komunikatów.

Dla rozwiązania zadania określonego przez tezę niniejszej rozprawy opracowane będą następujące problemy częściowe:

- Wybór odpowiedniej dla sterowania modalności predykatów oraz wybór sposobu reakcji na spełnienie predykatu w systemie.
- Opracowanie efektywnych i skalowalnych algorytmów wyznaczania globalnych stanów spójnych.
- Opracowanie symulatora środowiska obliczeniowego ze sterowaniem przez predykaty.
- Symulacyjna ewaluacja efektywności opracowanych algorytmów.
- Opracowanie rzeczywistego systemu programowania i wykonania programów, pozwalającego na sterowanie przez predykaty.
- Symulacyjna i praktyczna ocena efektywności sterowania przez predykaty.

Rozdział 2

Silnie spójne stany programu

Rozdział ten opisuje własności oraz metody detekcji globalnych stanów spójnych, otrzymanych przy użyciu częściowo zsynchronizowanych zegarów czasu rzeczywistego. Znaczniki czasu przypisane zdarzeniom odzwierciedlają wskazania tychże zegarów, przez co uzyskujemy inny, niż w przypadku zegara logicznego, porządek zdarzeń, oraz uporządkowany zbiór stanów spójnych o innych niż $CGS^{\preceq}$ własnościach.

Terminu „silnie spóje stany globalne" użył Stoller w odniesieniu do odmiany globalnych stanów spójnych opartych o znaczniki czasu rzeczywistego [112]. Nie chcąc wprowadzać nowej nazwy dla rzeczy wcześniej już nazwanej, pozostaniemy przy określeniu silnie spójne stany globalne. Zaznaczamy jednak, że inni autorzy, np. Wu [123], używają tego pojęcia w innym znaczeniu i przez silnie spójne stany rozumieją spójne stany globalne oparte o relacje przyczyny i skutku, bez nieodebranych komunikatów (wszystkie kanały komunikacyjne są puste). Aby pozostać w zgodzie z terminologią Wu musielibyśmy wprowadzić zupełnie nowy termin na oznaczenie stanów spójnych opartych o znaczniki czasu rzeczywistego (Wu nie zajmuje się nimi i nie proponuje żadnej nazwy) i pominęlibyśmy istniejącą w literaturze terminologię Stollera. Ponadto pojęcie silnie spójnych stanów globalnych według Wu nie jest szeroko rozpowszechnione i wielu autorów go nie wprowadza np. [9, 50].

2.1 Stany silnie spójne i predykaty na nich oparte

2.1.1 Systemy częściowo zsynchronizowane

Cechy systemu równoległego $\mathcal{SR}$ zdefiniowane w punkcie 1.2.1 są bardzo ogólne. W szczególności procesy w systemie $\mathcal{SR}$ nie dysponują globalnym zegarem, toteż nie mogą znakować zdarzeń wartościami czasu rzeczywistego w sensowny sposób. Jednak w praktyce budowy i eksploatacji systemów równoległych powszechnie spotyka się rozwiązania, w których procesy dysponują zegarami lokalnymi oraz zegary te są zsynchronizowane ze sobą z pewną tolerancją błędu. Uwzględnienie tego faktu w modelu opisujacym system równoległy prowadzi do pojęcia systemu równoległego częściowo zsynchronizowanego. Zastosowanie przybliżonego czasu rzeczywistego, dostępnego w takim systemie, powoduje duże zmiany w możliwościach obserwacji obliczeń.

Proponujemy wydzielenie trzech kategorii systemów równoległych częściowo zsynchronizowanych. Ich cechy wspólne są następujące:

- system składa się z N sekwencyjnych procesów $P_1..P_N$,
- procesy dysponują lokalnymi zegarami czasu rzeczywistego, zr_i oznacza bieżące wskazanie zegara P_i,
- zegary lokalne są zsynchronizowane ze sobą z dokładnością ε: $\forall_{i,j=1..N}|zr_i - zr_j| \leq \varepsilon$

Tablica 2.1: Zestawienie ważniejszych symboli stosowanych w tekście. Indeksy mogą być pomijane, jeśli nie sa istotne.

symbol	znaczenie
P_i	proces aplikacyjny numer i
N	liczba procesów aplikacyjnych
e_i^k	zdarzenie numer k w procesie P_i, górny indeks może być pominięty
zr_i	bieżąca wartość wskazywana przez zegar czasu rzeczywistego procesu P_i
$zr(e_i)$	czas przypisany wystąpieniu zdarzenia e_i, jest to czas wskazywany przez zegar czasu rzeczywistego procesu P_i w momencie wystąpienia zdarzenia e_i
$\underline{zr}(e), \overline{zr}(e)$	zwykle: najlepsze możliwe oszacowanie czasu, odpowiednio z dołu i z góry, w którym nastąpiło zdarzenie e według zegara wzorcowego. Pełna definicja patrz 2.1.3.
s_i^k	stan lokalny numer k procesu P_i
$\underline{s}, \overline{s}$	zdarzenia odpowiednio rozpoczynające i kończące stan lokalny s
S_i	spójny (w rozdziale 2: silnie spójny) stan globalny numer i
$\underline{zr}(S), \overline{zr}(S)$	odpowiednio początek i koniec gwarantowanego okresu trwania stanu silnie spójnego S według zegara wzorcowego
ε_i	maksymalna różnica wskazań pomiędzy zegarem czasu rzeczywistego procesu P_i a zegarem wzorcowym

- procesy mogą porozumiewać się ze sobą każdy z każdym dzięki wymianie komunikatów poprzez jednokierunkowe asynchroniczne kanały komunikacyjne, graf procesów połączonych kanałami jest silnie spójny,
- komunikaty mogą zostać odebrane w kolejności innej niż kolejność ich nadania, innymi słowy czasy przesłania komunikatów są różne i nieprzewidywalne,
- kanały łączące procesy aplikacyjne z monitorem są typu FIFO (ma to znaczenie przy monitorowaniu on-line, gdy monitor na bieżąco przetwarza otrzymane informacje o stanach procesów),
- sieć oraz procesy działają niezawodnie - komunikaty nie giną, procesy wykonują swój kod bezbłędnie,
- obliczenie w pojedynczym procesie reprezentowane jest przez sekwencję zdarzeń e^0,e^1,...; $zr(e_i)$ oznacza wskazanie zegara P_i w momencie wystąpienia zdarzenia e_i.

Proces P_i znajduje się w stanie lokalnym s_i^k pomiędzy wystąpieniem zdarzeń e_i^k a e_i^{k+1}. Formalnie będziemy utożsamiać stan s_i^k ze skończoną sekwencją zdarzeń $e_i^0 \ldots e_i^k$ z procesu P_i (taka sekwencja bywa zwana odcięciem, ang. *cut*). Zdarzenie e_i^0 oznacza utworzenie procesu P_i, zatem stan s_i^0 to stan początkowy tego procesu. W dalszym ciągu, gdy użycie indeksów górnych nie będzie konieczne, będziemy je pomijać.

Poniżej wymieniamy proponowane kategorie systemów równoległych częściowo zsynchronizowanych oraz ich cechy charakterystyczne:

$\mathcal{SRO}$ Czas przesłania komunikatu jest ograniczony z góry przez znaną wartość L_{max} oraz z dołu przez L_{min}. Wymiana komunikatów zawierających wskazania zegarów lokalnych służy do synchronizacji tych zegarów z dokładnością przy $\varepsilon \geq L_{max} - L_{min}$.

$\mathcal{SRD}$ Czas przesłania komunikatu jest skończony, ale nieograniczony. System posiada dodatkowy podsystemem synchronizacji zegarów niezależny od sieci komunikacyjnej. Od własności podsystemu synchronizacji zależy dokładność synchronizacji ε.

$\mathcal{SRP}$ Czas przesłania komunikatu jest skończony lecz nieograniczony, czas ten jest z prawdopodobieństwem γ mniejszy niż znana wartość L_{max}. Wymiana komunikatów zawierających wskazania zegarów lokalnych służy do probabilistycznej synchronizacji tych zegarów z dokładnością $\varepsilon = L_{max}$: $\forall_{i,j=1..N} |zr_i - zr_j| \leq L_{max}$. System pozostaje w stanie częściowo zsynchronizowanym tylko z określonym prawdopodobieństwem, zależnym od częstości wymiany komunikatów synchronizujących oraz błędu pełzania (drift) zegarów lokalnych [34].

W praktyce można spotkać systemy z każdej z proponowanych kategorii, w kolejnym punkcie zaprezentujemy przykłady. Dla wygody wprowadźmy jeszcze oznaczenie dla wszystkich systemów równoległych częściowo zsynchronizowanych: $\mathcal{SRCS} = \mathcal{SRO} \cup \mathcal{SRD} \cup \mathcal{SRP}$.

2.1.2 Przegląd metod synchronizacji zegarów lokalnych

Zegary lokalne najprościej zsynchronizować ze znaną precyzją w systemach klasy $\mathcal{SRO}$. Przez $t(m)$ oznaczamy wskazanie zegara nadawcy komunikatu m w momencie wysyłania tego komunikatu. Mamy gwarancję, że $t(m)$ dla komunikatu m odebranego przez P_i od P_j, pochodzi z chwili należącej do przedziału $\langle zr_i - L_{max},\ zr_i - L_{min} \rangle$ według zegara P_i.

- Jeśli $t(m) < zr_i - L_{max}$, to zr_i spieszy się względem zr_j być może więcej niż $L_{max} - L_{min}$.
- Jeśli $t(m) > zr_i - L_{min}$, to zr_i późni się względem zr_j być może więcej niż $L_{max} - L_{min}$.

Trywialny algorytm synchronizacji zegarów może polegać na przestawianiu zegara P_i według reguły: $zr_i := t(m) + L_{min} + \frac{L_{max} - L_{min}}{2}$, co da gwarancję, że $|zr_i - zr_j| < L_{max}$. Taka korekta musi być dokonywana dostatecznie często, aby kompensować pełzanie zegarów.

Do implementacji systemów $\mathcal{SRO}$ wygodnie jest użyć sieci gwarantującej jakość usług (Quality of Service, QoS). Sieć taka, po zadeklarowaniu przez proces zapotrzebowania na określoną przepustowość danego kanału, gwarantuje dostarczenie wymaganej przepustowości oraz w jej ramach określa maksymalny czas przesyłu komunikatu. Badania w zakresie QoS są obecnie szeroko prowadzone, z powodu wymagań stawianych sieciom przez coraz powszechniejsze transmisje multimedialne [4]. Sieci oparte na deterministycznych protokołach dostępu łatwo mogą implementować wymagania QoS, np. szyna IEEE1394 (FireWire) [66], lub sieć typu TokenRing oparta na łączach bezprzewodowych [80]. Z użyciem dodatkowych protokołów i nowoczesnych urządzeń sieciowych je implementujących, QoS można też uzyskać w sieciach klasy Ethernet [77]. Apte et al. zademonstrowali, że w klastrach obliczeniowych opartych o sieć Myrinet można użyć globalnego schedulingu komunikacji pozbawionego konfliktów wynikających z prób jednoczesnego użycia tego samego kanału transmisyjnego i w rezultacie otrzymać ograniczony z góry maksymalny czas transmisji komunikatów [7].

Systemy $\mathcal{SRD}$ pozwalają zwykle osiągnąć najlepszą jakość synchronizacji zegarów z uwagi na zastosowanie specjalizowanych rozwiązań. Od dawna znane są propozycje rozwiązań sprzętowych realizujące synchronizację zegarów w pojedynczej lokalizacji, np. [107]. Bardziej interesujące są obecne możliwości synchronizacji także geograficznie rozproszonych systemów. Dokładne sygnały czasu są transmitowane radiowo na kilka sposobów. Komputery można wyposażyć w odbiorniki takich sygnałów uzyskując synchronizację zegarów komputerów w różnych lokalizacjach do tego samego, dokładnego wzorca. Najpowszechniej stosowanym i globalnie dostępnym sygnałem czasu jest sygnał emitowany przez system GPS (Global Positioning System). Przyłączając komputery bezpośrednio do odbiorników GPS można uzyskać synchronizację zegarów o dokładności nawet do 100ns [61], choć jest to kosztowne i mało odporne na awarie rozwiązanie.

Klasa $\mathcal{SRP}$ obejmuje systemy dysponujące standardowymi sieciami komunikacyjnymi, używające tych sieci do wymiany komunikatów służących synchronizacji zegarów. Systemy takie są najbardziej rozpowszechnione. Brak gwarantowanego czasu dostarczenia komunikatu, charakteryzujący stosowane tu sieci komputerowe, nie pozwala na gwarantowaną jakość synchronizacji zegarów. W zależności od protokołu synchronizacji, procesy mogą stracić na pewien czas synchronizację swych zegarów [34], bądź dokładność tej synchronizacji może zmieniać się wraz ze

zmianą aktualnego opóźnienia transmisji. To ostatnie podejście występuje w bardzo popularnym Network Time Protocol (NTP) - protokole synchronizacji zegarów, będącym Internetowym standardem [91]. NTP ustawia lokalny zegar komputera według wskazań dostępnych przez sieć zegarów wzorcowych (wzorcem może być np. maszyna z odbiornikiem GPS, lub zegar atomowy), stosując złożone algorytmy statystyczne i operacje filtrujące. Działając poprzez Internet pozwala uzyskać dokładność rzędu dziesiątek ms, w obrębie LAN można liczyć na dokładność liczoną w setkach μs. Zasadniczą zaletą NTP jest jego dostępność. Oprogramowanie NTP jest standardowo dołączane do wielu systemów operacyjnych, istnieją darmowe implementacje NTP, a podstawowa konfiguracja tej usługi nie nastręcza problemów. W Internecie można znaleźć wiele ogólnie dostępnym zegarów wzorcowych, można też łatwo zbudować własny zegar wzorcowy np. podłączając odbiornik GPS do komputera. Wysoką dokładność synchronizacji wzajemnej zegarów daje zastosowanie Reference Synchronization (RBS) [44]. RBS pozwala utrzymać rozbieżność zegarów w grupie kooperujących maszyn na poziomie zaledwie kilku μs, wymagając od sprzętu jedynie fizycznie realizowanego rozgłaszania. Ten warunek spełniają popularne sieci, jak np. klasyczny Ethernet (bez buforowania ramek w koncentratorach) czy WiFi.

Bieżące trendy w rozwoju metod synchronizacji zegarów w systemach rozproszonych dążą do uzyskania rozbieżności we wskazaniach zegarów wynoszącej parę μs lub mniej, stosując w tym celu połączenie prostych elementów sprzętowych z rozwiązaniami programowymi. Przykładem może być użycie programowalnych kart Myrinet z wbudowanym zegarem, pozwalające ograniczyć różnice we wskazaniach zegarów do 5 μs w ramach klastra [32]. Trudniejszym problemem jest taka synchronizacja zegarów wybranej grupy komputerów, aby nie tylko rozbieżności wewnątrz tej grupy były małe, ale aby jednocześnie wszystkie zegary w grupie były precyzyjnie dostrojone do zewnętrznego wzorca czasu, jakim jest Universal Time Coordinated (UTC). Tego zadania podejmuje się architektura Network Time Interface (NTI), której zadaniem jest dystrybucja sygnału czasu z odbiornika GPS wewnątrz klastra przy użyciu specjalizowanych modułów sprzętowych [103]. NTI pozwala uzyskać zsynchronizowanie z UTC na poziomie 10 μs. Ten wynik bardzo poprawiają prace Horauera, zaprezentowane w najbardziej pełnej postaci w jego pracy doktorskiej [62]. Horauer, pozostawiając Ethernet jako medium przy pomocy którego dystrybuowany jest czas UTC, postuluje m.i. sprzętowe wspomaganie obsługi znaczników czasu wstawianych do komunikatów oraz użycie zmodyfikowanego przełącznika Ethernet i specjalnego zegara lokalnego dostosowanego do współpracy z zewnętrznym wzorcem. Deklarowana dokładność synchronizacji zegarów osiągalna z pomocą opracowanej architektury sięga 100ns, a ponieważ wzorcem czasu może być GPS, tak precyzyjnie zsynchronizowane klastry mogą być dowolnie oddalone od siebie.

2.1.3 Silnie spójne stany globalne

W ogólnej teorii globalnych stanów spójnych można zastosować porządkowanie zdarzeń według wskazań częściowo zsynchronizowanych zegarów lokalnych. Tylko częściowe zsynchronizowanie zegarów uniemożliwia precyzyjne wskazanie momentu zajścia danego zdarzenia w czasie, dlatego pomocne jest tutaj użycie interwałów. Zdarzenie otrzymuje znacznik reprezentujący przedział czasu, w którym nastąpiło. Samo zajście zdarzenia jest natychmiastowe. Wprowadźmy oznaczenia:

$\underline{zr}(e)$ początek przedziału czasu, w którym nastąpiło zdarzenie e,

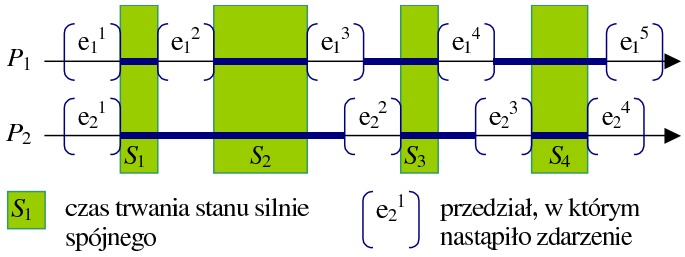
$\overline{zr}(e)$ koniec przedziału czasu, w którym nastąpiło zdarzenie e.

Uporządkowanie według znaczników interwałowych ma być zgodne z rzeczywistą kolejnością występowania zdarzeń. Wymagamy aby:

I1 $\underline{zr}(e) \leq \overline{zr}(e)$ dla każdego zdarzenia e,

I2 $\underline{zr}(e_i^k) \leq \underline{zr}(e_i^{k+1})$ oraz $\overline{zr}(e_i^k) \leq \overline{zr}(e_i^{k+1})$ dla każdego zdarzenia e_i^k i bezpośrednio następującego po nim w tym samym procesie zdarzenie e_i^{k+1},

I3 jeśli e_i nastąpiło w rzeczywistości przed e_j, to $\underline{zr}(e_i) \leq \overline{zr}(e_j)$.



Rysunek 2.1: Stany silnie spójne

Te postulaty można spełnić na przykład przy założeniu, że zegary lokalne ustawiane są według wskazań zegara wzorcowego z dokładnością ε i przyjmując, że $\underline{zr}(e_i) = zr_i(e_i) - \varepsilon$, $\overline{zr}(e_i) = zr_i(e_i) + \varepsilon$. Oczywiście, procedura synchronizacji zegarów nie może powodować cofania zegara, lecz ten warunek jest powszechie przestrzegany przy konstrukcji mechanizmów synchronizacji, np. NTP.

Na podstawie opisanych znaczników interwałowych określimy teraz relację porządkującą zdarzenia:

$$e_i^k \overset{bm}{\rightarrow} e_j^l \equiv \begin{array}{l} i = j \wedge k < l \\ i \neq j \wedge \underline{zr}(e_i^k) \leq \overline{zr}(e_j^l) \end{array}$$

Oznaczenie bm odnosi się do intuicyjnej interpretacji relacji $e_i \overset{bm}{\rightarrow} e_j$: zdarzenie e_i nastąpiło być może przed e_j.

Przypomnijmy, że proces P_i znajduje się w stanie lokalnym s_i^k pomiędzy wystąpieniem zdarzeń e_i^k a e_i^{k+1}. Relację porządkującą stany lokalne definiujemy następująco:

$$s_i^k \overset{bm}{\rightarrow} s_j^l \equiv e_i^{k+1} \overset{bm}{\rightarrow} e_j^l$$

Można to zinterpretować jako „stan s_i^k być może zakończył się zanim s_j^l się rozpoczął“. Stany lokalne są równoległe, jeśli nie są w relacji $\overset{bm}{\rightarrow}$:

$$s_i \parallel^{bm} s_j \Leftrightarrow s_i \overset{bm}{\not\rightarrow} s_j \wedge s_j \overset{bm}{\not\rightarrow} s_i$$

Stosując tę relację równoległości w definicji globalnych stanów spójnych otrzymujemy, że stan globalny $S = \{s_i : i = 1..N\}$ jest spójny, jeśli jego składowe stany lokalne są równoległe w sensie relacji $\parallel^{bm}$. Tak zdefiniowane stany spójne będziemy nazywać *stanami silnie spójnymi*, a zbiór tych stanów oznaczymy przez $SCGS$ (Strongly Consisted Global States). Formalnie:

$$S \in SCGS \Leftrightarrow \forall_{s_i, s_j \in S} s_i \parallel^{bm} s_j \tag{2.1}$$

Zastosowanie relacji $\overset{bm}{\rightarrow}$ w defininicji stanów silnie spójnych powoduje, że mamy pewność, iż stany lokalne wchodzące w skład danego stanu silnie spójnego trwały przez pewien czas jednocześnie. Na rysunku 2.1 widzimy przykład. W każdym okresie trwania globalnego stanu silnie spójnego wiadome jest, w jakim stanie znajdował się każdy z procesów.

Relacja $\preceq$(wprowadzona w punkcie 1.2.1) porządkuje zbiór $SCGS$ liniowo. Dowód tej własności można znaleźć w [112]. Intuicyjnie, uporządkowanie to odwzorowuje kolejność wystąpienia poszczególnych stanów silnie spójnych w czasie rzeczywistym. $SCGS$ uporządkowany relacją $\preceq$ będziemy oznaczać przez $SCGS^{\preceq}$. Rezultatem liniowego uporządkowania jest fakt, że obliczenie oraz jego wszystkie obserwacje są identyczne - obserwowana sekwencja stanów niewątpliwie miała miejsce w rzeczywistości.

Ponieważ relacja $\overset{bm}{\rightarrow}$ nie jest porządkiem częściowym, nie wszystkie własności globalnych stanów spójnych przedstawione w punkcie 1.2.1 są tu obowiązujące. Pełny związek $SCGS$ z ogólną teorią stanów spójnych przedstawił Stoller w [112]. Po pierwsze, wykazał on, że teorię stanów spójnych

opartą na relacji przyczyny i skutku można uogólnić i zastosować w niej częściowy porządek na zdarzeniach określony jako $e_i^k \overset{np}{\rightarrow} e_j^l \equiv \neg e_j^l \overset{bm}{\rightarrow} e_i^k$. Oznaczenie *np* pochodzi od „na pewno": e_i^k na pewno nastąpiło przed e_j^l. Stoller następnie udowodnił, że stany silnie spójne są identyczne ze stanami wspólnymi wprowadzonymi przez Fromentina i Raynala (patrz 1.3), gdy pojęcie stanów wspólnych zastosować w kracie $\overset{np}{CGS}^{\preceq}$. $\overset{np}{CGS}$ oznacza zbiór globalnych stanów spójnych zbudowany w oparciu o relację $\overset{np}{\rightarrow}$. Jedną z konsekwencji tej identyczności jest fakt, że pomiędzy dwoma kolejnymi stanami silnie spójnymi może wystąpić więcej niż jedno zdarzenie. Na przykład na rys. 2.1 stan S_2 od S_3 dzielą dwa zdarzenia. Zatem mogą istnieć stany globalne, który miały miejsce w rzeczywistości, ale które nie będą uznane za stany silnie spójne. Zdolność wykrywania stanów silnie spójnych uzależniona jest od częstotliwości występowania zdarzeń i dokładności synchronizacji zegarów. Dla przykładu, jeśli zegary lokalne procesów są zsynchronizowane względem siebie z dokładnością ε, to stany globalne trwające krócej niż 2ε mogą zostać nierozpoznane jako stany silnie spójne. Mamy tu do czynienia z ograniczeniem znanym, w dużej mierze natury technicznej.

Niezwykle istotna jest niezależność otrzymanego uporządkowanego zbioru $SCGS^{\preceq}$ od komunikacji między procesami. Użyta tu relacja porządkująca zdarzenia nie opiera się na zależności między wysłaniem, a odebraniem komunikatu, a jedynie na czasie wystąpienia zdarzeń. Pozwala to przy użyciu stanów silnie spójnych efektywnie monitorować systemy, w których procesy rzadko lub wcale nie komunikują się (np. czujniki podłączone do centrali).

Liniowy charakter $SCGS^{\preceq}$ oznacza, że moc zbioru $SCGS$ nie przekracza NE. W rezultacie, koszt algorytmów przeglądających stany silnie spójne jest nieporównanie mniejszy niż koszt algorytmów przeglądających $CGS^{\succeq}$. Omówienie algorytmów detekcji stanów silnie spójnych znajduje się w dalszej części tego rozdziału.

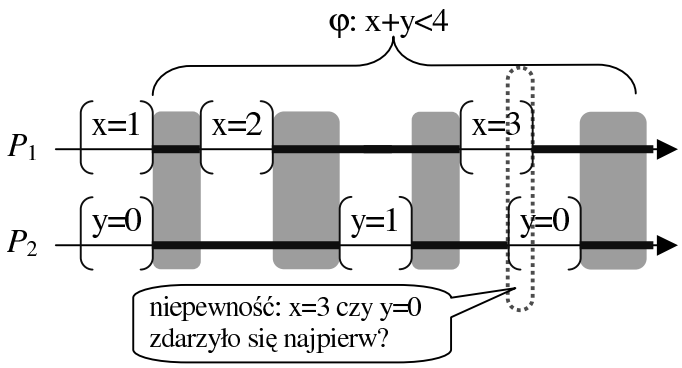
2.1.4 Predykaty określone na silnie spójnych stanach globalnych

Predykaty globalne określone na globalnych stanach silnie spójnych sprawdzane są jako predykaty w modalności *Instantly*. Dla predykatu globalnego φ, Inst(φ) jest spełnione, gdy φ jest spełnione w globalnym stanie silnie spójnym. Wykrycie Inst(φ) gwarantuje, że w rzeczywistym przebiegu obliczeń φ było faktycznie spełnione. Dodatkowo, stan globalny, w którym zaobserwowano Inst(φ) jest rzeczywistym stanem, w którym φ było prawdziwe w obliczeniu. Są to konsekwencje liniowego uporządkowania zbioru $SCGS$ relacją $\preceq$. Natomiast konsekwencją zastosowania znaczników opartych na czasie rzeczywistym jest możliwość określenia kiedy spełnienie Inst(φ) miało miejsce. Stan, w którym φ zostało spełnione opisuje się (oczywiście z pewnym przybliżeniem) wartościami zegara czasu rzeczywistego. Można zweryfikować z pewną dokładnością okres od kiedy do kiedy dany predykat był spełniony. Istnieje zatem możliwość badania predykatów odnoszących się do czasu rzeczywistego, np. czy w procesach P_2 i P_3 jednocześnie wartość zmiennej `temperatura` była większe niż 200 przez ponad minutę? Jednak, gdy okres w którym predykat był spełniony obejmuje kolejne stany silnie spójne, nie zawsze możemy mieć pewność, że spełnienie było nieprzerwane. Rysunek 2.2 ilustruje ten problem. Jeśli w ciągu stanów silnie spójnych pomiędzy dwoma kolejnymi stanami zaszły dwa lub więcej zdarzeń, oznacza to, że pewien stan globalny (lub ich sekwencja) nie został wykryty jako stan silnie spójny i wobec tego nie wiemy, jaka była wartość predykatu w czasie trwania tego stanu. Czas występowania takich okresów niepewności można jednak podać w pewnym przybliżeniu.

Możliwość względnie łatwego przejrzenia całego zbioru $SCGS$ spowodowała, wykrywanie Inst(φ) jest wykonalne on-line dla φ w ogólnej postaci. Nie ma tu powodów, aby rozpatrywać postaci szczególne predykatów.

2.2 Standardowy algorytm detekcji stanów silnie spójnych

Możliwość zastosowania czasu rzeczywistego do wspomagania określania kolejności zdarzeń sugerowali już Marzullo i Neiger [85]. Zdając sobie sprawę z ogromnego kosztu sprawdzania Poss(φ) i Def(φ), zaproponowali wykorzystanie częściowo zsynchronizowanych zegarów w systemach klasy



Rysunek 2.2: Predykat spełniony w ciągu stanów silnie spójnych

$\mathcal{SRO}$ do dodatkowego (uzupełniającego relację przyczyny i skutku) uporządkowania zdarzeń. Jeśli dokładność synchronizacji zegarów wynosi ε, to dla zdarzeń e_i i e_j takich, że $e_i \parallel^{ps} e_j$ oraz $zr(e_i) + \varepsilon < zr(e_j)$ wiemy, że e_i wystąpiło przed e_j. Uzupełniając w ten sposób relację $\stackrel{ps}{\rightarrow}$ redukuje się koszt wyznaczania Poss(φ) i Def(φ) dzięki zmniejszeniu liczby alternatywnych globalnych stanów spójnych.

Predykatami globalnymi w systemach częściowo zsynchronizowanych w sposób systematyczny zajął się Mayo [88]. Zaproponował on dwa sposoby wyznaczania wartości predykatów globalnych w systemach rozproszonych częściowo zsynchronizowanych. Dla predykatu φ_i określonego na stanie lokalnym P_i, $SLP(t, i, \varphi_i)$ jest spełnione, gdy w P_i predykat φ_i był spełniony przez co najmniej ε czasu od chwili t liczonej według zegara P_i (ε to, jak zwykle, dokładność synchronizacji zegarów). Istotna własność SLP polega na tym, że jeśli $SLP(t, i, \varphi_i)$ jest spełnione jednocześnie dla $i = 1..N$ dla pewnej wartości t, to w pewnej chwili w czasie rzeczywistym spełnione jest $\bigwedge_{i=1}^{N} \varphi_i$. Z kolei $GLP(t, \varphi_i)$ jest spełnione, gdy φ_i był spełniony w P_i dla $zr_j = t$, $j = 1..N$, czyli gdy zegary procesów wskazywały tę samą wartość t. Posługując się tak sformułowanymi pojęciami, Mayo wyprowadza rozproszone, symetryczne algorytmy detekcji określonych predykatów, ze szczególnym uwzględnieniem predykatów stabilnych. W końcowej części pracy zajmuje się także scentralizowaną detekcją niestabilnych predykatów koniunkcyjnych. W pracy nie jest używane pojęcie stanu globalnego, autor koncentruje się na wykrywaniu określonych własności osadzonych w czasie dzięki zastosowaniu formuł SLP i GLP, dla każdej własności i formuły stosowany jest osobny algorytm.

Badania spójnych stanów globalnych budowanych w oparciu o wskazania częściowo zsynchronizowanych zegarów przeprowadził Stoller [112]. Zastosował on interwałowe znaczniki czasu dla zdarzeń i podał powiązanie teorii stanów spójnych opartej na relacji przyczyny i skutku z teorią stanów spójnych opartych na wskazaniach częściowo zsynchronizowanych zegarów, patrz 2.1.3. Opracował też algorytm detekcji stanów silnie spójnych, który tu w skrócie opiszemy. Algorytm ten jest adaptacją algorytmu Fromentina i Raynala wykrywania stanów wspólnych. Adaptacja taka była możliwa po wspomnianym uprzednio stwierdzeniu, że stany silnie spójne są identyczne ze stanami wspólnymi w kracie $\stackrel{np}{CGS}^{\preceq}$. Algorytm Fromentina i Raynala działa przy koszcie $O(EN^3)$, lecz uporządkowanie liniowe skalarnych znaczników czasu rzeczywistego (w opozycji do częściowego uporządkowania wektorowych znaczników czasu logicznego) pozwoliło zastosować kolejki priorytetowe i zredukować koszt do $O(EN \log N)$. Algorytm przedstawiony jest w ramce Algorytm 1. Zaznaczamy, że poprawne działanie on-line algorytmów SCGS (wszystkich opisanych w pracy wersji) wymaga dyscypliny FIFO przy dostarczania komunikatów od procesów do monitora, tj. kanały proces-monitor muszą być FIFO.

Przedstawimy teraz pełny i szczegółowy opis algorytmu detekcji globalnych stanów silnie spójnych w nieco zmodyfikowanym wariancie. Przyjmiemy, że każde zdarzenie raportowane do monitora kończy stan lokalny danego procesu i jednocześnie rozpoczyna stan następny. Wystarczy teraz, że komunikaty do monitora zawierać będą informację o pojedynczych zdarzeniach, a nie o parze zdarzeń: rozpoczynającym i kończącym stan. Przyjmiemy także, że dla $i = 1..N$ zegar

Algorithm 1 Standandardowy algorytm SCGS

1. Procesy wysyłają do monitora komunikaty zawierające informacje o swych stanach lokalnych. Każdy komunikat od P_i informuje o jednym stanie lokalnym i zawiera znaczniki interwałowe zdarzeń rozpoczynającego i kończącego dany stan. Przez $\underline{s}$ oznaczymy zdarzenie rozpoczynające, a przez $\overline{s}$ zdarzenie kończące stan s. Nie każdy stan musi być raportowany, czyli zdarzenie kończące dany stan nie musi rozpoczynać stanu kolejnego z punktu widzenia monitora.

2. Monitor umieszcza odebrane komunikaty w kolejkach FIFO, kolejka q_i zawiera komunikaty od P_i.

3. Gdy kolejka q_i jest pusta dla pewnego i, algorytm zawiesza się w oczekiwaniu na jej zapełnienie (na komunikat od P_i).

4. Sprawdzana jest wzajemna współbieżność stanów z czoła kolejek, czyli czy $\forall_{i,j=1..N} \mathrm{head}(q_i) \parallel^{bm} \mathrm{head}(q_j)$. Warunek ten możemy zapisać jako

$$\forall_{i,j=1..N, i\neq j} \overline{zr}(\underline{\mathrm{head}(q_i)}) < \underline{zr}(\overline{\mathrm{head}(q_j)}), \tag{2.2}$$

   a dzięki zastosowaniu operacji min oraz max- w krótszej i łatwiejszej do sprawdzenia postaci:

$$\max_{i=1}^{N} \overline{zr}(\underline{\mathrm{head}(q_i)}) < \min_{i=1}^{N} \underline{zr}(\overline{\mathrm{head}(q_i)}). \tag{2.3}$$

   Wykrycie takiej współbieżności oznacza wykrycie stanu silnie spójnego. Dla wykrytego stanu S jego gwarantowany okres trwania to interwał pomiędzy $zr(\underline{S}) = \max_{i=1}^{N} \overline{zr}(\underline{\mathrm{head}(q_i)})$, a $zr(\overline{S}) = \min_{i=1}^{N} \underline{zr}(\overline{\mathrm{head}(q_i)})$.

5. Gdy stany nie są współbieżne, wybierana jest ta kolejka q_i, dla której $\underline{zr}(\overline{\mathrm{head}(q_i)})$ jest minimalne, czyli ta, która zawiera najwcześniej zakończony stan lokalny spośród stanów z czoła kolejek; z tej kolejki usuwa się czołowy element.

6. Gdy stany są współbieżne (czyli po wykryciu stanu silnie spójnego) poszukiwania są wznawiane przez wybór tej kolejki q_i, dla której $\underline{zr}(\underline{\mathrm{head2}(q_i)})$ jest najmniejsza (funkcja $\mathrm{head2}(q_i)$ zwraca element drugi w kolejności z kolejki q_i). Z tej kolejki usuwany jest czołowy element.

7. Skok do kroku 3.

lokalny procesu P_i jest zsynchronizowany względem zegara wzorcowego zr_W z dokładnością ε_i: $|zr_i - zr_W| < \varepsilon_i$. Wartości ε_i są znane monitorowi, np. procesy dostarczają je przed rozpoczęciem właściwego algorytmu, bądź mogą to być parametry z góry ustalone w danym systemie. Dzięki temu raportowane zdarzenia mogą być oznaczane pojedynczą wartością $zr(e)$ reprezentującą odczyt zegara lokalnego w chwili wystąpienia zdarzenia e. Monitor przekształca takie skalarne znaczniki czasu w znaczniki interwałowe według reguły: $\underline{zr}(e_i) = zr(e_i) - \varepsilon_i$, $\overline{zr}(e_i) = zr(e_i) + \varepsilon_i$. Założenia te pozwalają zredukować koszt komunikacji - dla K kolejnych stanów lokalnych zamiast K komunikatów zawierających po cztery znaczniki czasu wysłanych będzie $K+1$ komunikatów zawierających pojedynczy znacznik czasu. Oryginalny algorytm posługuje się trzema kolejkami priorytetowymi. Nam wystarczą tylko dwie, co upraszcza strukturę programu, zmniejsza zapotrzebowanie na pamięć i eliminuje koszt obsługi trzeciej kolejki.

Algorytm używa następujących struktur danych:

kolejki FIFO Q_i, $i = 1..N$ Q_i zawiera odebrane przez monitor komunikaty o zdarzeniach od procesu P_i. Określone są następujące operacje:

Q_i.first() Zwraca pierwszy element w kolejce. Jeśli kolejka jest pusta, to blokuje proces aż element stanie się dostępny.

Q_i.append(e) Dodaje zdarzenie do końca kolejki.

Q_i.remove() Usuwa pierwszy element.

Q_i.empty() Zwraca TRUE gdy kolejka jest pusta, wpp. zwraca FALSE.

Koszt wymienionych operacji wynosi $O(1)$. Q_i.first() jest zdarzeniem kończącym stan lokalny P_i aktualnie rozważany przez algorytm.

Wektor $R_{1..N}$ zawiera zdarzenia. R_i to zdarzenie rozpoczynające aktualnie rozważany stan procesu P_i. Algorytm sprawdza, czy stany lokalne s_i : $\underline{s_i} = R_i$, $\overline{s_i} = Q_i$.first(), $i = 1..N$ są wzajemnie równoległe (tworzą SCGS).

Kolejka priorytetowa $maxR$ zawiera pary $< zr(R_i) - \varepsilon_i, i >$. Kluczem jest pierwszy składnik pary. Służy do znajdowania zdarzeń w R o maksymalnym znaczniku czasu $\underline{zr}$. Określone są następujące operacje:

insert(t, i) : pointer dodaje rekord w koszcie $O(\log N)$ i zwraca bezpośredni wskaźnik do wstawionego rekordu.

max() zwraca rekord o maksymalnym kluczu w koszcie $O(1)$.

max2() zwraca rekord o kluczu największym po maksymalym w koszcie $O(1)$.

remove(ptr) usuwa rekord wskazywany przez wskaźnik ptr w koszcie $O(\log N)$.

empty() zwraca TRUE jeśli kolejka jest pusta, wpp. zwraca FALSE.

Kolejka priorytetowa $minT$ zawiera trójki $< zr(Q_i.\text{first}()) + \varepsilon_i, i, ptr >$. Kluczem jest pierwszy składnik trójki. ptr to wskaźnik na rekord w $maxR$ z taką samą wartością składnika i, używany do znajdywania w koszcie stałym w $maxR$ zdarzenia rozpoczynającego stan kończony danym zdarzeniem w $minT$. Określone są następujące operacje:

insert(t, i, ptr) dodaje rekord w koszcie $O(\log N)$.

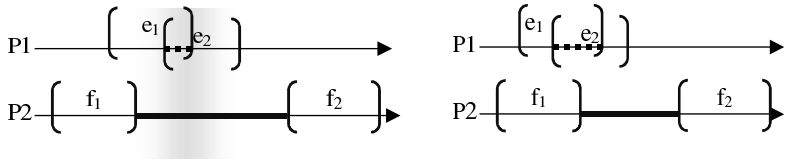
min() zwraca rekord o minimalnym kluczu w koszcie $O(1)$.

min2() zwraca rekord o kluczu najmniejszym po minimalnym w koszcie $O(1)$.

removemin() usuwa rekord o minimalnym kluczu w koszcie $O(\log N)$.

empty() zwraca TRUE jeśli kolejka jest pusta, wpp. zwraca FALSE.

Wektor $maxRptr_{1..N}$ zawiera wskaźniki, $maxRptr_i$ wskazuje na rekord $< t, i, ptr >$w $maxR$. Wektor ten jest używany przez funkcję removemin().



Rysunek 2.3: Krótko trwające stany lokalne ograniczone zdarzeniami o przecinających się interwałach mogą prowadzić do stanów silnie spójnych (lewa część) lub nie (prawa część)

Powyższe struktury są inicjalizowane następująco:

Q_i zawiera dokładnie jeden element i Q_i.first() $= e0$, gdzie $zr(e0) = 0$, $i = 1..N$,

$R_i = e0$, $i = 1..N$,

$maxR$ zawiera rekordy $< 0, i >$, $i = 1..N$,

$maxRptr_i$ wskazuje na rekord $< 0, i >$ w $maxR$, $i = 1..N$,

$minT$ zawiera rekordy $< 0, i, maxRptr_i >$, $i = 1..N$.

Algorytm wykorzystuje dwa wątki, pomocniczy obsługujący przybywające komunikaty, oraz główny działający na zdarzeniach umieszczonych przez wątek pomocniczy w kolejkach Q_i. Ponieważ wątki nie muszą działać jednocześnie, można je traktować jako koprocedury (coroutines), a cały algorytm jako sekwencyjny (możliwy też jest zapis tego algorytmu jako pojedynczej, sekwencyjnej procedury). Pseudokod podany jest w ramce Algorytm 2.

Inicjalnie stanem lokalnym wszystkich procesów jest „sztuczny“ stan o zerowym czasie trwania ograniczony zdarzeniami $e0$. Stan taki nie może być składnikiem stanu globalnego. Drugi stan lokalny każdego procesu, stan pomiędzy $e0$, a pierwszym raportowanym zdarzeniem, oznacza okres przed wystąpieniem pierwszego zdarzenia (i również może mieć zerowy czas trwania, o ile pierwsze zdarzenie otrzymało znacznik czasu równy zero). Początkowo wykrywane stany globalne mogą zawierać tego typu stany lokalne. Dopiero, gdy każdy proces dośle raporty o co najmniej dwóch zdarzeniach, możliwe jest wykrycie normalnych stanów globalnych aplikacji, składających się ze stanów lokalnych ograniczonych dwoma zdarzeniami aplikacyjnymi. Algorytm łatwo zmodyfikować, aby wykrywał dopiero takie stany globalne.

Kod oznaczony komentarzem „obydwa zdarzenia z tego samego procesu“ służy do identyfikacji takich stanów silnie spójnych S, w których dla pewnego i mamy $\underline{zr}(S) = \overline{zr}(\underline{s_i}) > \underline{zr}(\overline{s_i}) = \overline{zr}(S)$. Rysunek 2.3 (lewa część) pokazuje tego rodzaju sytuację. Warunek SCGS w ogólnej postaci (formuła 2.2 na stronie 39) może być spełniony w takim przypadku, podczas gdy warunek zoptymalizowany, którym posługuje się algorytm (formuła 2.3), spełniony nie będzie. Dla zdarzeń pochodzących z tego samego procesu ich nakładające się interwałowe znaczniki czasu nie są przeszkodą w identyfikacji kolejności ich wystąpienia. Trzeba się tylko upewnić, że interwały te nie nakładają się na interwały zdarzeń z innych procesów. Dla takich globalnych stanów silnie spójnych niemożliwe jest ustalenie, według zegara wzorcowego, momentu, w którym stan ten z pewnością trwał. W oryginalnym algorytmie omawiany przypadek jest rozpatrywany w inny sposób, niestety prowadzący do nieprawidłowego wykrywania SCGS w sytuacjach, w których nie ma pewności, czy stan silnie spójny zaistniał. Na przykład na prawej części rysunku 2.3 pokazana jest sytuacja, w której warunek SCGS w postaci ogólnej nie jest spełniony (zatem stan silnie spójny nie zaistniał), lecz algorytm oryginalny zadeklaruje tam znalezienie SCGS.

Sposób znajdywania silnie spójnych stanów globalnych w prezentowanym algorytmie jest identyczny, jak w przypadku algorytmu oryginalnego, zatem pozostaje poprawny. Natomiast komentarza wymaga postępowanie po znalezieniu stanu silnie spójnego. Algorytm powinien szukać kolejnych globalnych stanów silnie spójnych, ponawiając poszukiwania w taki sposób, aby żaden globalny stan silnie spójny nie został pominięty. Choć sposób wznowienia poszukiwań zasadniczo jest taki sam, jak w algorytmie oryginalnym (choć implementacja nieco prostsza, bez trzeciej

Algorithm 2 Zmodyfikowany standardowy algorytm SCGS

Wątek pomocniczy:

```
on „przybył komunikat o zdarzeniu e z procesu P_i“ do
  Q_i.append(e)
  if „monitor był zawieszony w oczekiwaniu na zdarzenie od P_i“ then
    resume monitor
```

Wątek główny:

```
loop
  <t,i,ptr> = minT.min()
  min = t
  <max,j> = maxR.max()
  if min > max then {
    /*SCGS znaleziony, SCGS=<s_1,..,s_N>, gdzie s_ijest stanem lokalnym P_i pomię-
dzy zdarzeniami R_i a Q_i.first(), jego gwarantowany okres trwania to interwał pomię-
dzy max a min*/
    if „to końcowy stan aplikacji“ then
      loopexit
  } else if i==j {
    /*obydwa zdarzenia z tego samego procesu*/
    <min2,,> = minT.min2()
    <max2,> = maxR.max2()
    if (max2<min AND min2>max) then {
      /*SCGS znaleziony */
      if „to końcowy stan aplikacji“ then
        loopexit
    }
  }
  /* szukaj dalej */
  minT.removemin()
  maxR.remove(ptr)
  maxRptr_i= maxR.insert(<t+2ε_i,i>)
  R_i=Q_i.first()
  Q_i.remove()
  t= zr(Q_i.first()) /*może zablokować*/
  minT.insert(t-ε_i, i, maxRptr_i)
endloop
```

kolejki priorytetowej), to oryginał nie uzasadnia należycie poprawności stosowanej procedury. Wypełniamy tę lukę.

Twierdzenie 1. Po znalezieniu globalnego stanu silnie spójnego $S =< s_1, .., s_N >$ nie zostanie pominięty żaden kolejny globalny stan silnie spójny, gdy dalsze poszukiwania rozpoczną się od stanu S' powstałego przez zastąpienie stanu lokalnego s_i w S kolejnym stanem procesu P_i, wybierając i tak, że dla pewnego j

$$\underline{zr}(\overline{s_i}) - \overline{zr}(\underline{s_j}) = \min_{k,l=1}^{N} \underline{zr}(\overline{s_k}) - \overline{zr}(\underline{s_l}).$$

Warunek ten oznacza, że czas zapewnionego współtrwania stanów s_i oraz s_j jest najkrótszy spośród wszystkich par stanów lokalnych składających się na S, przy czym s_i kończy się później niż s_j. Dowód:

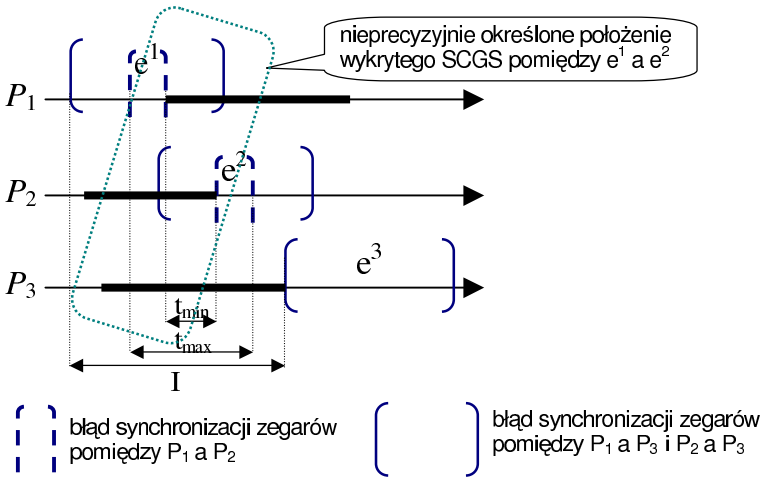
Pokażemy, że nie istnieje takie $m \neq i$, że zastąpienie s_m^k w S przez kolejny stan procesu P_m, czyli przez s_m^{k+1}, może doprowadzić do kolejnego globalnego stanu silnie spójnego nadal zawierającego s_i. Weźmy i oraz j spełniające tezę twierdzenia. Przypuśćmy, że istnieje $m \neq i$ takie, że po zastąpieniu s_m^k w S przez kolejny stan procesu P_m, czyli przez s_m^{k+1}, otrzymamy globalny stan silnie spójny zawierający s_i. Indeksy i oraz j wybraliśmy tak, że $\underline{zr}(\overline{s_i}) - \overline{zr}(\underline{s_j}) \leq \underline{zr}(\overline{s_m^k}) - \overline{zr}(\underline{s_j})$. Ponieważ stan globalny osiągnięty poprzez zastąpienie s_m w S przez kolejny stan procesu P_m ma być stanem silnie spójnym, musi on spełniać warunek 2.2. W szczególności $0 < \underline{zr}(\overline{s_i}) - \overline{zr}(\underline{s_m^{k+1}})$. Zauważmy, że $\overline{s_m^k}$ oraz $\underline{s_m^{k+1}}$ to jedno i to samo zdarzenie. Po podstawieniu oraz dodaniu i uproszczeniu nierówności otrzymujemy $\overline{zr}(\overline{s_m^k}) < \underline{zr}(\overline{s_m^k})$, co jest sprzeczne z założeniem I1 dotyczącym właściwości interwałowych znaczników czasu □

Zmodyfikowany algorytm SCGS oraz powyższy dowód zostały opublikowane w [21].

2.3 Użycie względnej dokładności synchronizacji zegarów lokalnych

Skuteczność działania przedstawionego w poprzednim punkcie standardowego algorytmu wykrywania silnie spójnych stanów globalnych zależy w prostej linii od dokładności synchronizacji zegarów lokalnych. Przypomnijmy - jeśli maksymalna różnica wskazań pomiędzy zegarami lokalnymi w systemie wynosi ε, wówczas stany silnie spójne stany globalne trwające krócej niż 2ε mogą zostać niezauważone. Przykładem jest stan pomiędzy zdarzeniami e_1^3, a e_2^2 na rysunku 2.1. Poprawienie dokładności synchronizacji zegarów w sposób globalny może okazać się kosztowne, trudne w implementacji i skomplikowane organizacyjnie. Dlatego warto zwrócić uwagę na możliwość wykorzystania precyzyjnej synchronizacji zegarów w lokalnym zakresie, w obrębie określonego podzbioru procesów. Zdarza się w praktyce, że w całym systemie możemy zatem wyróżnić podzbiory procesów mające swe zegary zsynchronizowane znacznie lepiej (w ramach danego podzbioru) niż globalna jakość synchronizacji. Znakomitą ilustracją takiej sytuacji jest przykład klastra składającego się z komputerów SMP, czyli mających po kilka procesorów współdzielących pozostałe zasoby komputera. Zegary tych komputerów są zsynchronizowane ze sobą z określoną dokładnością. Zauważmy, że procesy działające na jednej maszynie SMP korzystają z tego samego fizycznego zegara systemowego, zatem można powiedzieć, że ich zegary lokalne są idealnie zsynchronizowane. Wszystkie zdarzenia pochodzące z pojedynczej maszyny można więc uporządkować liniowo (w ramach rozdzielczości zegara). Innym przykładem może być zespół klastrów, każdy dysponujący lokalną szybką siecią i wysokiej jakości lokalnym mechanizmem synchronizacji zegarów, połączone wolniejszą siecią, bez możliwości (z przyczyn technicznych, kosztowych lub administracyjnych) precyzyjnej synchronizacji czasu pomiędzy klastrami.

Nasz pomysł poprawy skuteczności algorytmu wykrywania SCGS zasadza się na wykorzystaniu względnej dokładności synchronizacji zegarów. Przez względną dokładność synchronizacji zegarów rozumiemy znajomość deklarowanej dokładności synchronizacji zegarów danej pary procesów



Rysunek 2.4: Przykład globalnego stanu silnie spójnego wykrywalnego tylko dzięki użyciu względnej dokładności synchronizacji zegarów

względem siebie. Dopuszczalny błąd synchronizacji zegara danego procesu może być różny względem zegarów różnych innych procesów. Dla dwóch procesów P_i i P_j monitor zna wartość $\varepsilon_{i,j}$ oznaczającą maksymalną różnicę wskazań zegarów tych dwóch procesów. Na rysunku 2.4 widzimy przykład globalnego stanu silnie spójnego, który nie byłby wykryty przy zastosowaniu standardowego algorytmu (wykorzystującego globalną dokładność synchronizacji zegarów), lecz który jest wykrywalny przy użyciu względnej dokładności synchronizacji zegarów. Na omawianym rysunku globalna dokładność synchronizacji odpowiada szerszemu z użytych tam interwałów. Interwały te zachodzą na siebie wykluczając wykrycie SCGS. Użycie węższych interwałów odpowiadających dokładniejszej synchronizacji zegara procesu P_1 względem zegara procesu P_2 pozwala na wykrycie dodatkowego SCGS pomiędzy zdarzeniami e^1 a e^2. W przeciwieństwie do stanów silnie spójnych wykrytych zwykłą metodą, czas jego trwania nie jest precyzyjnie oznaczony. Użycie względnej dokładności synchronizacji zegarów nie pozwala na określenie kiedy taki stan na pewno trwał według zegara wzorcowego, gdyż w tej metodzie nie posługujemy się odniesieniem do zegara wzorcowego. Można jedynie ustalić, że musiał on wystąpić w przedziale czasu oznaczonym na omawianym rysunku symbolem I. Można też określić jego maksymalny i minimalny czas trwania - na rysunku oznaczone odpowiednio symbolami t_{max} i t_{min}.

Macierz $\varepsilon_{1..N,1..N}$ określa dokładość względnej synchronizacji zegarów: $|zr_k - zr_l| \leq \varepsilon_{k,l}$. Ma ona następujące własności:

- $\varepsilon_{k,l} \geq 0$ dla $k, l = 1..N$
- $\varepsilon_{k,k} = 0$ dla $k = 1..N$
- $\varepsilon_{k,l} = \varepsilon_{l,k}$ dla $k, l = 1..N$
- $\varepsilon_{k,l} + \varepsilon_{l,m} \geq \varepsilon_{k,m}$ dla $k, l, m = 1..N$

Ostatnią własność łatwo uzasadnić: faktyczna różnica wskazań zegarów P_k i P_m nie może być większa niż $\varepsilon_{k,l} + \varepsilon_{l,m}$. Zatem, jeśli nie mamy dodatkowej informacji o wartości $\varepsilon_{k,m}$, zawsze możemy przyjąć, że $\varepsilon_{k,m} = \varepsilon_{k,l} + \varepsilon_{l,m}$.

Algorytm wykrywania silnie spójnych stanów globalnych z uwzględnieniem względnej dokładności synchronizacji zegarów lokalnych przedstawiony jest w ramce Algorytm 3. Kroki 3-6 tego algorytmu są tożsame z odpowiednimi krokami algorytmu standardowego, co gwarantuje ich poprawność. Niemożliwe tylko okazało się użycie zoptymalizowanej wersji warunku SCGS (warunek 2.3). Zastosowane tam operacje min i max mogą być użyte, gdy istnieje wspólna skala czasu dla wszystkich zegarów lokalnych, czyli gdy wskazania tych zegarów można bezpośrednio porównywać.

Algorithm 3 Wykrywanie SCGS z użyciem względnej dokładności synchronizacji zegarów

1. Procesy wysyłają do monitora komunikaty zawierające informacje o swych stanach lokalnych. Każdy komunikat od P_i informuje o jednym stanie lokalnym i zawiera pojedyczne znaczniki czasu zdarzeń rozpoczynającego i kończącego dany stan. Przez $\underline{s}$ oznaczymy zdarzenie rozpoczynające, a przez $\overline{s}$ zdarzenie kończące stan s, $zr(e)$ oznacza znacznik czasu przypisany zdarzeniu e. Nie każdy stan musi być raportowany, czyli zdarzenie kończące dany stan nie musi rozpoczynać stanu kolejnego z punktu widzenia monitora.

2. Monitor umieszcza odebrane komunikaty w kolejkach FIFO, kolejka q_i zawiera komunikaty od P_i.

3. Gdy kolejka q_i jest pusta dla pewnego i, algorytm zawiesza się w oczekiwaniu na jej zapełnienie (na komunikat od P_i).

4. Znacznik czasu $zr(e)$ każdego zdarzenia e rozpoczynającego lub kończącego stan znajdujący się u czoła kolejki, jest konwertowany na rodzinę znaczników interwałowych $< \underline{zr_{k,l}}(e), \overline{zr_{k,l}}(e) >$, $k,l = 1..N$, według reguły: $\underline{zr_{k,l}}(e) = zr(e) - \frac{\varepsilon_{k,l}}{2}$, $\overline{zr_{k,l}}(e) = zr(e) + \frac{\varepsilon_{k,l}}{2}$.

5. Uwzględniając względną dokładność synchronizacji zegarów sprawdzane jest, czy stany z czoła kolejek tworzą silnie spójny stan globalny, czyli czy

$$\forall_{k,l=1..N,k\neq l} \overline{zr_{k,l}}(\underline{\mathrm{head}(q_k)}) < \underline{zr_{k,l}}(\overline{\mathrm{head}(q_l)})$$

6. Gdy powyższy warunek nie jest spełniony, to $\exists k,l : \overline{zr_{k,l}}(\underline{\mathrm{head}(q_k)}) \geq \underline{zr_{k,l}}(\overline{\mathrm{head}(q_l)})$. Z kolejki q_l usuwany jest czołowy element.

7. Po wykryciu stanu silnie spójnego poszukiwania są wznawiane przez wybór takiego i, że dla pewnego j

$$\underline{zr_{i,j}}(\overline{s_i}) - \overline{zr_{i,j}}(\underline{s_j}) = \min_{k,l=1}^{N} \underline{zr_{k,l}}(\overline{s_k}) - \overline{zr_{k,l}}(\underline{s_l}).$$

   Z kolejki q_i usuwany jest czołowy element.

8. Skok do kroku 3.

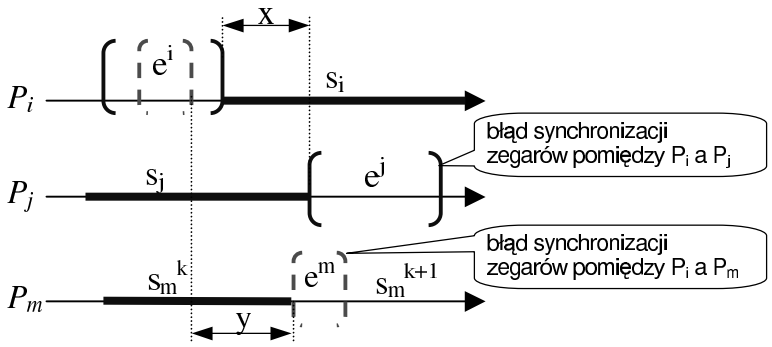
Użycie względnej dokładności synchronizacji zegarów uniemożliwia posługiwanie się wspólną skalą czasu. Podwyższa to koszt algorytmu, czym się zajmiemy dalej. Uzasadnienia wymaga krok 7.

Twierdzenie 2. Po znalezieniu globalnego stanu silnie spójnego $S =< s_1, .., s_N >$ nie zostanie pominięty żaden kolejny globalny stan silnie spójny, gdy dalsze poszukiwania rozpoczną się od stanu S' powstałego przez zastąpienie stanu lokalnego s_i w S kolejnym stanem procesu P_i, wybierając i tak, że dla pewnego j

$$\underline{zr_{i,j}}(\overline{s_i}) - \overline{zr_{i,j}}(\underline{s_j}) = \min_{k,l=1}^{N} \underline{zr_{k,l}}(\overline{s_k}) - \overline{zr_{k,l}}(\underline{s_l}).$$

Warunek ten oznacza, że czas zapewnionego współtrwania stanów s_i oraz s_j jest najkrótszy spośród wszystkich par stanów lokalnych składających się na S, przy czym s_i kończy się później niż s_j. Twierdzenie to ma treść analogiczną do twierdzenia 1. Faktycznie, twierdzenie 1 jest szczególnym przypadkiem niniejszego twierdzenie, przypadkiem w którym wszystkie elementy $\varepsilon_{i,j}$ mają tę samą wartość. Dowód jest nieco inny, gdy musi brać pod uwagę względne dokładności synchronizacji zegarów:

Pokażemy, że nie istnieje takie $m \neq i$, że zastąpienie s_m^k w S przez kolejny stan procesu P_m, czyli przez s_m^{k+1}, może doprowadzić do kolejnego globalnego stanu silnie spójnego nadal zawierającego



Rysunek 2.5: Ilustracja do dowodu twierdzenia 2.

s_i. Weźmy i oraz j spełniające tezę twierdzenia. Przypuśćmy, że istnieje $m \neq i$ takie, że po zastąpieniu s_m^kw S przez kolejny stan procesu P_m, czyli przez s_m^{k+1}, otrzymamy globalny stan silnie spójny zawierający s_i. Indeksy i oraz j wybraliśmy tak, że

$$\underline{zr_{i,j}}(\overline{s_i}) - \overline{zr_{i,j}}(\underline{s_j}) \leq \underline{zr_{m,j}}(\overline{s_m^k}) - \overline{zr_{m,j}}(\underline{s_j}),$$

co na rysunku 2.5 oznacza, że długość odcinka x jest mniejsza lub równa długości odcinka y. Ponieważ stan globalny osiągnięty poprzez zastąpienie s_mw S przez kolejny stan procesu P_m ma być stanem silnie spójnym, musi on spełniać warunek 2.2. W szczególności

$$0 < \underline{zr_{i,m}}(\overline{s_i}) - \overline{zr_{i,m}}(\underline{s_m^{k+1}}),$$

co oznacza, że e^m nastąpiło przed e^j na rysunku 2.5. Napiszemy teraz obie nierówności podstawiając za znaczniki czasu ich definicję z kroku 4 algorytmu:

$$\begin{aligned} zr(\overline{s_i}) - \frac{\varepsilon_{i,j}}{2} - zr(\underline{s_j}) - \frac{\varepsilon_{i,j}}{2} &\leq zr(\overline{s_m^k}) - \frac{\varepsilon_{m,j}}{2} - zr(\underline{s_j}) - \frac{\varepsilon_{m,j}}{2} \\ 0 &< zr(\overline{s_i}) - \frac{\varepsilon_{i,m}}{2} - zr(\underline{s_m^{k+1}}) - \frac{\varepsilon_{i,m}}{2} \end{aligned}$$

Zauważmy, że $\overline{s_m^k}$ oraz $\underline{s_m^{k+1}}$ to jedno i to samo zdarzenie (e^m na rysunku 2.5). Po podstawieniu oraz dodaniu i uproszczeniu nierówności otrzymujemy $\varepsilon_{i,j} > \varepsilon_{i,m} + \varepsilon_{m,j}$, co jest sprzeczne z ostatnim założeniem dotyczącym własności macierzy $\varepsilon_{i,j}$ □

Proponowana implementacja przedstawionego algorytmu wykorzystuje następujące struktury danych:

kolejki FIFO Q_i, $i = 1..N$ Q_i zawiera odebrane przez monitor komunikaty o zdarzeniach od procesu P_i. Określone są następujące operacje:

Q_i.first() Zwraca pierwszy element w kolejce. Jeśli kolejka jest pusta, to blokuje proces aż element stanie się dostępny.

Q_i.append(e) Dodaje zdarzenie do końca kolejki.

Q_i.remove() Usuwa pierwszy element z kolejki.

Q_i.empty() Zwraca TRUE gdy kolejka jest pusta, wpp. zwraca FALSE.

Koszt wymienionych operacji wynosi $O(1)$. Q_i.first() jest zdarzeniem kończącym stan lokalny P_i aktualnie rozważany przez algorytm.

Wektor $R_{1..N}$ zawiera zdarzenia. R_i to zdarzenie rozpoczynające aktualnie rozważany stan procesu P_i. Algorytm sprawdza, czy stany lokalne s_i : $\underline{s_i} = R_i$, $\overline{s_i} = Q_i$.first(), $i = 1..N$ są wzajemnie równoległe (tworzą SCGS)

Macierz $D_{1..N,1..N}$ pomocna przy sprawdzaniu warunku SCGS w kroku 5 algorytmu. Z definicji $D_{i,j} = \underline{zr_{i,j}}(Q_j.\text{first}()) - \overline{zr_{i,j}}(R_i)$. SCGS jest wykryty, gdy $\forall_{i,j=1..N} D_{i,j} > 0$.

Zbiór Z zawierający informacje w postaci par indeksów $< i, j >$ o nie dodatnich elementach macierzy D. Zbiór można zrealizować jako tablicę haszowaną lub podwójnie wiązaną listę. Dla tego drugiego wariantu trzeba, aby każdy element macierzy D zawierał wskaźnik do swego odpowiedniego elementu w Z. Określone są następujące operacje na zbiorze, każda działająca przy koszcie $O(1)$:

$D_{i,j}$.**Zadd**() dodaje parę $< i, j >$do Z,

$D_{i,j}$.**Zremove**() usuwa parę $< i, j >$ z Z,

Z.**empty**() zwraca TRUE gdy zbiór jest pusty, wpp. zwraca FALSE,

Z.**elem**() zwraca dowolną parę zawartą w zbiorze, nie usuwając jej ze zbioru.

Kolejka priorytetowa H zawiera trójkki $< D_{i,j}, i, j >$ odpowiadające wszystkim elementom macierzy D, kluczem jest pierwszy składnik trójki. Każdy element macierzy D zawiera (niejawny) wskaźnik na odpowiadający mu element w H, aby umożliwić dostęp do tego elementu bez wyszukiwania. Kolejka zawiera N^2 elementów, określone są na niej następujące operacje:

$D_{i,j}$.**Hinsert**() wstawia $< D_{i,j}, i, j >$ do kolejki w koszcie $2 \log N$

$D_{i,j}$.**Hupdate**() w trójce $< D_{i,j}, i, j >$ zawartej w H ustawia wartość pierwszego składnika według bieżącej wartości $D_{i,j}$. Pozycja trójki w kolejce zmienia się odpowiednio.

$H.\min()$ zwraca trójkę z najmniejszym kluczem

Powyższe struktury są inicjowane następująco:

Q_i zawiera dokładnie jeden element i $Q_i.\text{first}() = e0$, gdzie $zr(e0) = 0$, $i = 1..N$,

$R_i = e0$, $i = 1..N$,

$D_{i,j} = -\varepsilon_{i,j}$, $i, j = 1..N$.

Z **i** H są wypełnione według definicji zgodnie z zawartością macierzy D.

Algorytm wykorzystuje dwa wątki, pomocniczy obsługujący przybywające komunikaty, oraz główny działający na zdarzeniach umieszczonych przez wątek pomocniczy w kolejkach Q_i. Ponieważ wątki nie muszą działać jednocześnie, można je traktować jako korutyny, a cały algorytm jako sekwencyjny (możliwy też jest zapis tego algorytmu jako pojedynczej, sekwencyjnej procedury). Pseudokod przedstawiony jest w ramce Algorytm 4.

Inicjalnie stanem lokalnym wszystkich procesów jest „sztuczny“ stan o zerowym czasie trwania ograniczony zdarzeniami $e0$. Stan taki nie może być składnikiem stanu globalnego. Drugi stan lokalny każdego procesu, stan pomiędzy $e0$, a pierwszym raportowanym zdarzeniem, oznacza okres przed wystąpieniem pierwszego zdarzenia (i również może mieć zerowy czas trwania, o ile pierwsze zdarzenie otrzymało znacznik czasu równy zero). Początkowo wykrywane stany globalne mogą zawierać tego typu stany lokalne. Dopiero, gdy każdy proces dośle raporty o co najmniej dwóch zdarzeniach, możliwe jest wykrycie normalnych stanów globalnych aplikacji, składających się ze stanów lokalnych ograniczonych dwoma zdarzeniami aplikacyjnymi. Algorytm łatwo zmodyfikować, aby wykrywał dopiero takie stany globalne.

Brak możliwości wykorzystania operacji min i max do wykrywania globalnych stanów silnie spójnych podwyższył koszt algorytmu. Przyjmiemy, jak zwykle, że N to liczba procesów aplikacyjnych, E to maksymalna liczba zdarzeń przypadająca na jeden proces. Inicjalizacja struktur danych wymaga $O(N)$ operacji dla kolejek Q_ii wektora R, $O(N^2)$ operacji dla zbioru Z i macierzy D oraz $O(N^2 \log N)$ operacji dla kolejki H. Pętla zewnętrzna programu powtarzana jest co najwyżej NE razy (każde zdarzenie przetwarzane jest jednokrotnie). W każdej iteracji:

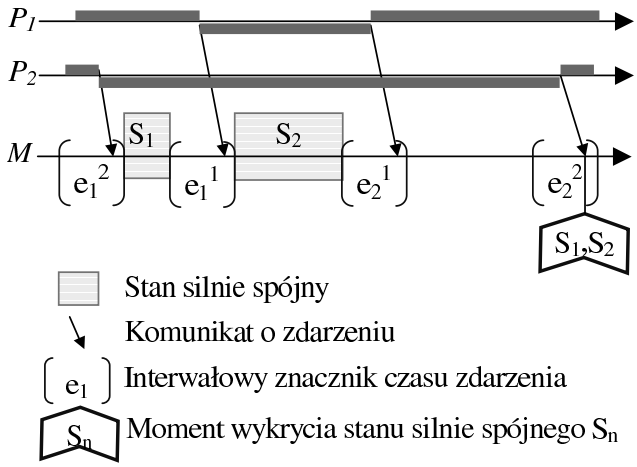
Algorithm 4 Implementacja algorytmu 3

Wątek pomocniczy:

```
on „przybył komunikat o zdarzeniu e z procesu P_i“ do
  Q_i.append(e)
  if „monitor był zawieszony w oczekiwaniu na zdarzenie od P_i“ then
    resume monitor
```

Wątek główny:

```
loop
  if Z.empty() then
    /*SCGS znaleziony, SCGS=<s_1,..,s_N>, gdzie s_ijest stanem lokalnym P_i pomię-
dzy zdarzeniami R_i a Q_i.first() */
    if „to końcowy stan aplikacji“ then loopexit
    <v,i,k> = H.min()
  else
    <i,k> = Z.elem()
  fi
  /*weź kolejny stan P_k*/
  R_k = Q_k.first()
  Q_k.remove()
  diff = zr(Q_k.first()) - zr(R_k) /*może zablokować*/
  /* diff≥0, aktualizuj D, Z, H */
  for i= 1 to N
    if D_{i,k} + diff ≤ 0 and D_{i,k} > 0 then
      D_{i,k}.NPremove()
    fi
    D_{i,k} = D_{i,k} + diff
    D_{i,k}.Hupdate()
  endfor
  for i= 1 to N
    if D_{k,i}- diff > 0 and D_{k,i}≤ 0 then
      D_{k,i}.NPadd()
    fi
    D_{k,i} = D_{k,i} - diff
    D_{k,i}.Hupdate()
  endfor
endloop
```



Rysunek 2.6: Opóźnienie wykrywania silnie spójnych stanów globalnych w standardowym algorytmie SCGS

- Znajdujemy indeks procesu, dla którego bierzemy kolejny stan, w koszcie $O(1)$.
- Pojedynczy wiersz i pojedynczą kolumnę macierzy D oraz zbiór Z aktualizujemy w koszcie $O(N)$ - wykonywane jest co najwyżej N razy Zadd() i co najwyżej N razy Zremove().
- Kolejka H aktualizowana jest w koszcie $O(N \log N)$ - $2N$ aktualizacji, każda kosztuje $O(\log N)$.

Całkowity koszt to $O(EN^2 \log N)$, zatem N razy więcej niż koszt algorytmu standardowego, przy dodatkowym, rzędu $O(N^2)$, zapotrzebowaniu na pamięć.

Opisany powyżej algorytm został opublikowany w pracy [18].

2.4 Użycie niezakończonych stanów lokalnych

Standardowy algorytm wykrywania silnie spójnych stanów globalnych rozpatruje jedynie już zakończone stany lokalne procesów. Wynika to z faktu, że warunek SCGS (warunek 2.2 i 2.3) zawiera odwołania do czasów zakończenia aktualnie rozpatrywanych stanów lokalnych. W praktyce oznacza to, że wykryte stany silnie spójne należą już do przeszłości, nie możliwe jest wykrycie globalnego stanu silnie spójnego w czasie jego trwania, gdy jest on bieżącym stanem systemu. Zatem wykrywanie SCGS następuje zawsze z pewnym opóźnieniem w stosunku do czasu ich rzeczywistego wystąpienia. Opóźnienie to może być dowolnie duże. Rozpatrzmy przypadek, gdy jeden z procesów przez dłuższy czas pozostaje w tym samym stanie lokalnym, podczas gdy inne procesy częściej zmieniają swe stany. Globalne stany silnie spójne powstałe z udziałem takiego długotrwałego stanu lokalnego będą wykryte dopiero, gdy stan ten się zakończy, a zatem początkowe z nich będą długo czekać na wykrycie. Rysunek 2.6 ilustruje ten problem. Stany S_1 i S_2 nie mogą zostać zauważone dopóki monitor nie dowie się o zakończeniu długotrwałego stanu lokalnego procesu P_2.

W wielu sytuacjach takie opóźnienia są nie do przyjęcia. Powodują one pogorszenie własności typu czas reakcji lub zdolność do autokorekty stanu w każdym systemie wykorzystującym monitorowanie on-line stanu globalnego. Szczególnym przypadkiem jest sterowanie on-line wykonaniem aplikacji w oparciu o jej silnie spójne stany globalne. Prawidłowe sterowanie może odbywać się jedynie na podstawie aktualnych danych o stanie programu. Zagadnienie to jest szeroko rozpatrzone w rozdziale 4, gdzie przedstawiony jest ilościowy wpływ opóźnienia w wykrywaniu SCGS na parametry systemu sterowania aplikacją.

Do redukcja omawianego opóźnienia dochodzi w sytuacjach, gdy monitor konstruuje globalne stany silnie spójne używając rozpoczętych, lecz jeszcze nie zakończonych stanów lokalnych. Jedna z metod uzyskania takiego stanu rzeczy polega na wstrzymywaniu procesów przed zmianą ich

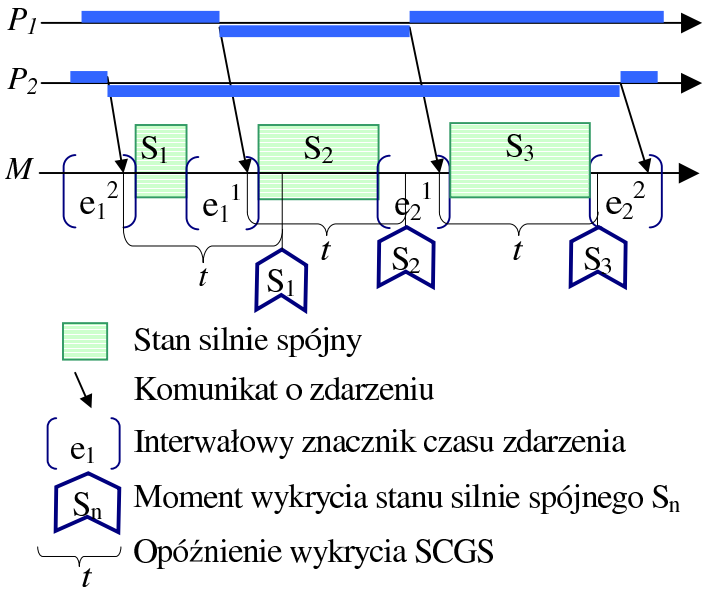
stanu lokalnego. Proces musi otrzymać potwierdzenie monitora, że jego dotychczasowy stan lokalny został uwzględniony, zanim wykona instrukcję programu zmieniającą ten stan. Ewaluacja predykatów globalnych w modalności *Currently* [36] polega właśnie na takim postępowaniu. To podejście daje monitorowi możliwość obserwacji aktualnie trwających stanów globalnych, jednak kosztem specjalnej instrumentacji kodu procesów (konieczne wprowadzenie oczekiwań na zezwolenie na kontynuację od monitora w odpowiednich miejscach kodu) oraz częstego wstrzymywania biegu procesów. Koszty te są dla nas nie do zaakceptowania, gdyż wymagają modyfikacji kodu procesów w zależności od monitorowanych własności, monitorowanie zmienia charakterystykę działania aplikacji przez wprowadzenie blokowania, a samo blokowanie powoduje degradację wydajności aplikacji.

Marzullo i Neiger [85] wskazali (nie prowadząc konkretnych badań w tym kierunku), że uwzględnianie stanów niezakończonych jest możliwe, jeśli procesy wysyłają periodycznie komunikaty potwierdzające trwanie rozpoczętego stanu lokalnego, lub gdy czas transmisji komunikatów do monitora jest ograniczony z góry. Pierwszy pomysł jest łatwy w realizacji, ale posiada istotne wady. Im bardziej aktualną informację ma mieć do dyspozycji monitor, tym częściej trzeba wysyłać potwierdzające komunikaty. Każdy taki komunikat obciąża sieć i musi być przetworzony przez monitor. Musimy wybierać pomiędzy rzadkim udostępnieniem monitorowi danych o trwających stanach lokalnych, a znaczącym (k-krotnym, gdy średni czas trwania stanu lokalnego jest k razy dłuższy niż interwał pomiędzy kolejnymi potwierdzeniami) wzrostem obciążenia sieci i procesora przypisanego monitorowi. Trzeba się liczyć z tym, że duże obciążenie powoduje wydłużenie czasu transmisji oraz obsługi komunikatów, w rezultacie czego informacja przetwarzana przez monitor i tak staje się przestarzała. Wyniki stosownych testów przedstawione są w punkcie 4.2.2.

2.4.1 Użycie ograniczenia na maksymalny czas transmisji komunikatu do monitora

Znajomość maksymalnego czasu transmisji komunikatów prowadzi do bardzo obiecującej metody wczesnego wykrywania silnie spójnych stanów globalnych. Załóżmy, że czas przesłania komunikatu do monitora nie przekracza T, oraz że monitor także dysponuje zegarem lokalnym zsynchronizowanym z zegarami procesów aplikacyjnych z dokładnością ε, czyli że $|zr_i - zr_M| \leq \varepsilon$, $i = 1..N$. Wówczas monitor ma pewność, że niezakończone stany procesów trwały co najmniej do chwili $zr_M - T - \varepsilon$ według wskazań zegarów tychże procesów, gdyż komunikaty o wcześniejszych końcach stanów dotarłyby już wcześniej. Zastowanie tej wiedzy pozwala skrócić czas pomiędzy powstaniem, a wykryciem globalnych stanów silnie spójnych, co ilustruje rysunek 2.7. Zaznaczone tam opóźnienie t musi mieć wartość większą niż $T + \varepsilon$, po jego upływie monitor wie, że dany stan lokalny trwa z pewnością dostatecznie długo, aby go użyć do konstrukcji SCGS.

Zasadniczym problemem przy próbie praktycznego wykorzystania tej metody jest określenie maksymalnego czasu transmisji komunikatów do monitora. Gwarantowany maksymalny czas transmisji to cecha wchodząca w skład pojęcia Quality of Service, o czym pisaliśmy już w punkcie 2.1.2. Wymienione tam rozwiązania gwarantujące QoS są bezpośrednio przydatne też tutaj. Uwagę zwraca wskazana możliwość realizacji QoS nawet w sieciach Ethernet, przy zastosowaniu dodatkowych protokołów sterujących ruchem sieciowym. Ostatnie badania pokazują, że maksymalny czas transmisji w przełączanym Ethernecie można skutecznie przewidzieć i utrzymywać na niskim poziomie stosując zwykłe urządzenia sieciowe i algorytmy kształtowania ruchu sieciowego wbudowane w nowoczesne (np. Linux 2.4, 2.6) systemy operacyjne [82]. Nasze eksperymenty dowiodły, że bez stosowania żadnych dodatkowych technik praktyczne określenie maksymalnego czasu transmisji jest nadal możliwe na podstawie pomiaru obserwowanych opóźnień transmisji. Uzyskane tak ograniczenie jest słabej jakości: 2 do 3 rzędów wielkości większe niż średni czas transmisji i wymaga stałych korekt na podstawie obserwacji stanu sieci - jednak wystarcza do implementacji omawianej metody wczesnego wykrywania silnie spójnych stanów globalnych. Natomiast rozwiązaniem idealnym wydaje się użycie dwóch osobnych sieci w pojedynczym systemie: jednej do wymiany danych - bez QoS, ale z duża przepustowością, drugiej do przesyłania informacji do monitora - gwarantującą QoS i małe opóźnienie.



Rysunek 2.7: Wykrywanie silnie spójnych stanów globalnych przy znanym maksymalnym czasie transmisji komunikatu

Przedstawimy teraz algorytm wykrywania silnie spójnych stanów globalnych pracujący na niezakończonych stanach lokalnych dzięki wykorzystaniu gwarantowanego maksymalnego czasu transmisji komunikatów do monitora. Za punkt wyjścia przyjmiemy algorytm 2. Zasadnicza zasada jego działania pozostanie taka sama, toteż uzasadnienie poprawności będzie tyczyć tylko wprowadzonych zmian. Przypomnijmy, że dla $i = 1..N$ zegar lokalny procesu P_i jest zsynchronizowany względem zegara wzorcowego zr_W z dokładnością ε_i: $|zr_i - zr_W| < \varepsilon_i$. Wartości ε_i są znane monitorowi. Przyjmijmy dla wygody oznaczenie $\varepsilon_{max} = \max_{i=1..N} \varepsilon_i$. Dodatkowo monitor posiada zegar lokalny zr_M i $|zr_M - zr_W| < \varepsilon_M$. W wyjściowym algorytmie modyfikacji wymaga przede wszystkim warunek SCGS (warunek 2.3). Jeśli kolejka q_i jest pusta dla pewnego i, to aktualnie rozpatrywany stan P_i nie zakończył się. W takim przypadku musimy wziąć pod uwagę opóźnienie t i sprawdzić, czy stan(y) niezakończone trwa(ją) dostatecznie długo:

$$\max_{i=1}^{N} \overline{zr}(\underline{\text{head}(q_i)}) < \begin{cases} \min_{i=1}^{N} \underline{zr}(\overline{\text{head}(q_i)}) & \text{gdy } \forall i : \neg\text{empty}(q_i) \\ \min\left(\min_{i=1}^{N}\left(\underline{zr}(\overline{\text{head}(q_i)}) : \neg\text{empty}(q_i)\right), zr_M - T - \varepsilon_M - \varepsilon_{max}\right) & \text{wpp.} \end{cases} \tag{2.4}$$

Przybycie komunikatu rozpoczynającego stan lokalny daje okazję do weryfikacji powyższego warunku. Jednak nie ma sensu przeprowadzać tej weryfikacji zbyt wcześnie. Załóżmy, że proces P_i rozpoczyna nowy stan lokalny s_i zdarzeniem e_i. Gdy monitor odbierze komunikat zawierający informację o e_i wie, że podróżował on nie dłużej niż $zr_M - zr(e) + \varepsilon_i + \varepsilon_M$. Załóżmy, że znany monitorowi bieżący stan s_j procesu P_j, $i \neq j$, jest niezakończony. Wiadomo, że s_j trwał (co najmniej) do $zr_M - T - \varepsilon_j - \varepsilon_M$. Ponieważ $zr_M - T - \varepsilon_j - \varepsilon_M < zr(e_i)$, to na razie nie wiadomo, czy s_j dotrwał do początku s_i. Dopiero po odczekaniu $T - zr_M + zr(e_i) + \varepsilon_i + \varepsilon_M + \gamma$ czasu, gdzie γ jest pewną dodatnią dowolnie bliską zeru wartością, można stwierdzić, że s_i i s_j były przez pewien czas równoległe (o ile nie odebrano w międzyczasie komunikatu o zakończeniu s_j). Odczekiwanie jest zrealizowane w algorytmie przy użyciu procedury `alarmSet(t,i)`. Jej wywołanie powoduje, że po upływie `t` czasu do monitora dostarczany jest komunikat *alarm* powodujący wznowienie działania algorytmu, oraz kasowany jest poprzedni oczekujący (jeśli jest) alarm. Drugi parametr pozwala ustawić osobny alarm dla każdego procesu.

Przyjrzyjmy się teraz procedurze poszukiwania SCGS. Proces, którego kolejny stan ma być wzięty pod uwagę w trakcie poszukiwania SCGS, jest wybierany jako ten, którego obecnie rozpatrywany stan kończy się najwcześniej (informacja o nim jest zawarta w $minT$.first()). Teraz jednak, gdy potencjalnie nie znamy momentów zakończenia wszystkich stanów lokalnych, trzeba

sprawdzać, czy brakujące zakończenia stanów nie mogą okazać się wcześniejsze (gdy komunikaty o nich dotrą do monitora), niż najwcześniejsze spośród znanych zakończeń. Niech t_{min} oznacza czas najwcześniejszego znanego zakończenia stanu spośród aktualnie rozpatrywanej kombinacji stanów lokalnych oraz niech niektóre z tych stanów będą niezakończone. Gdy warunek

$$t_{min} \leq zr_M - T - \varepsilon_M - \varepsilon_{max}, \tag{2.5}$$

nie jest spełniony, wtedy poszukiwania trzeba zawiesić do nadejścia kolejnych komunikatów. Gdy nadejdą, albo będą to informacje o brakujących zakończeniach stanów (i poszukiwania można kontynuować), albo (gdy upłynie stosowny czas) warunek 2.5 będzie już spełniony i dla wybranego procesu można będzie bezpiecznie wziąć jego kolejny stan lokalny kontynuując poszukiwania. Zauważmy, że gdy algorytm zostanie wznowiony przez alarm, to właśnie ta ostatnia sytuacja ma miejsce. Czyli: przybycie komunikatu o zdarzeniu ustawia alarm i może spowodować zawieszenie algorytmu, alarm wznawia algorytm i daje gwarancję, że warunek 2.5 jest spełniony, a zatem że drugiego pod rząd zawieszenia procedury poszukiwania SCGS nie będzie. W rezultacie poszukiwanie SCGS nie będzie nigdy wstrzymywane w nieskończoność.

Pozostało ustalenie, jak poszukiwania są wznawiane po znalezieniu silnie spójnego stanu globalnego. Proces, którego kolejny stan ma być wzięty pod uwagę w celu wznowienia poszukiwania, jest wybierany jako ten, którego obecnie rozpatrywany stan kończy się najwcześniej. Zatem i tu, przy obecności stanów niezakończonych trzeba sprawdzać, czy brakujące zakończenia stanów nie mogą okazać się wcześniejsze (gdy komunikaty o nich dotrą do monitora), niż najwcześniejsze spośród znanych zakończeń, co opisuje warunek 2.5. Jeśli warunek ten nie jest spełniony, to wznowienie poszukiwań się nie udaje i algorytm jest wstrzymywany. O tym, że na pewno zostanie on wznowiony, przekonamy się analizując cztery przypadki wyczerpujące wszystkie możliwości:

1. Koniec wykrytego SCGS stanowi zdarzenie e_i kończące stan s_i. Wznowienie poszukiwań SCGS nastąpiło wskutek alarmu ustawionego przybyciem komunikatu o e_i. Wznowienie się uda, gdyż $t_{min} = zr(e_i)$ (e_i jest najwcześniejszym znanym zakończeniem rozpatrywanych stanów lokalnych) i z definicji alarmu warunek 2.5 jest spełniony.

2. Koniec wykrytego SCGS stanowi zdarzenie e_i kończące stan s_i. Próba wznowienia poszukiwań SCGS nastąpiła na wskutek odebrania komunikatu o zdarzeniu e_j. Próba ta może się nie udać (warunek 2.5 może nie być spełniony), lecz alarm ustawiony przy odbiorze komunikatu o e_j sprowadzi nas do przypadku 1.

3. SCGS został wykryty wskutek alarmu, ustawionego przy odbiorze komunikatu o zdarzeniu e_i. Koniec takiego SCGS nie jest znany, czyli w warunku 2.4 mamy: $\exists i : \neg\text{empty}(q_i)$ i $\min_{i=1..N:\neg\text{empty}(q_i)} \underline{zr}(\overline{\text{head}(q_i)}) > zr_M - T - \varepsilon_M - \varepsilon_{max}$. Wznowienie poszukiwań zostaje zainicjowane kolejnym alarmem. Taka sekwencja alarmów nie jest możliwa. Wykrycie SCGS wskutek alarmu oznacza, że pomiędzy odbiorem komunikatu o e_i, a zadziałaniem wtedy ustawionego alarmu, nie odebrano żadnych komunikatów o zdarzeniach. Zatem nie było okazji, aby ustawić drugi, późniejszy alarm.

4. SCGS został wykryty na wskutek alarmu, jak w przypadku 3. Wznowienie poszukiwań następuje przy najbliższym odbiorze komunikatu o zdarzeniu. Ponieważ teraz to zdarzenie kończy wykryty wcześniej SCGS, dalej postępujemy jak w przypadku 2.

Algorytm używa następujących struktur danych (identycznych, jak algorytm 2):

kolejki FIFO Q_i, $i = 1..N$ Q_i zawiera odebrane przez monitor komunikaty o zdarzeniach od procesu P_i. Określone są następujące operacje:

Q_i.first() Zwraca. pierwszy element w kolejce. Jeśli kolejka jest pusta, to blokuje proces aż element stanie się dostępny,

Q_i.append(e) Dodaje zdarzenie do końca kolejki,

Q_i.remove() Usuwa pierwszy element z kolejki,

Q_i.empty() Zwraca TRUE gdy kolejka jest pusta, wpp. zwraca FALSE.

Koszt wymienionych operacji wynosi $O(1)$. Q_i.first() jest zdarzeniem kończącym stan lokalny P_i aktualnie rozważany przez algorytm.

Wektor $R_{1..N}$ zawiera zdarzenia. R_i to zdarzenie rozpoczynające aktualnie rozważany stan procesu P_i. Algorytm sprawdza, czy stany lokalne s_i : $\underline{s_i} = R_i$, $\overline{s_i} = Q_i$.first(), $i = 1..N$ są wzajemnie równoległe (tworzą SCGS)

Kolejka priorytetowa $maxR$ zawiera pary $< zr(R_i) - \varepsilon_i, i >$. Kluczem jest pierwszy składnik pary. Służy do znajdowania zdarzeń w R o maksymalnym znaczniku czasu $\underline{zr}$. Określone są następujące operacje:

insert(t, i) : pointer dodaje rekord w koszcie $O(\log N)$ i zwraca bezpośredni wskaźnik do wstawionego rekordu.

max() zwraca rekord o maksymalnym kluczu w koszcie $O(1)$.

max2() zwraca rekord o kluczu największym po maksymalym w koszcie $O(1)$.

remove(ptr) usuwa rekord wskazywany przez wskaźnik ptr w koszcie $O(\log N)$.

empty() zwraca TRUE jeśli kolejka jest pusta, wpp. zwraca FALSE.

Kolejka priorytetowa $minT$ zawiera trójki $< zr(Q_i.\text{first}()) + \varepsilon_i, i, ptr >$. Kluczem jest pierwszy skadnik trójki. ptr to wskaźnik na rekord w $maxR$ z taką samą wartością składnika i, używany do znajdywania w koszcie stałym w $maxR$ zdarzenia rozpoczynającego stan kończony danym zdarzeniem w $minT$. Określone są następujące operacje:

insert(t, i, ptr) dodaje rekord w koszcie $O(\log N)$.

min() zwraca rekord o minimalnym kluczu w koszcie $O(1)$.

min2() zwraca rekord o kluczu najmniejszym po minimalnym w koszcie $O(1)$.

removemin() usuwa rekord o minimalnym kluczu w koszcie $O(\log N)$.

empty() zwraca TRUE jeśli kolejka jest pusta, wpp. zwraca FALSE

Wektor $maxRptr_{1..N}$ zawiera wskaźniki, $maxRptr_i$ wskazuje na rekord $< t, i, ptr >$w $maxR$. Wektor ten jest używany przez funkcję

zmienna_logiczna_*wykryty* ma wartość TRUE wtedy i tylko wtedy, gdy wykryto już SCGS będący kombinacją bieżących stanów lokalnych, zapobiega to raportowaniu tego samego SCGS wiele razy.

Powyższe struktury są inicjalizowane następująco:

Q_i zawiera dokładnie jeden element i Q_i.first() $= e0$, gdzie $zr(e0) = 0$, $i = 1..N$,

$R_i = e0$, $i = 1..N$,

$maxR$ zawiera rekordy $< 0, i >$, $i = 1..N$,

$maxRptr_i$ wskazuje na rekord $< 0, i >$ w $maxR$, $i = 1..N$,

$minT$ zawiera rekordy $< 0, i, maxRptr_i >$, $i = 1..N$,

$wykryty = FALSE$,

alarmy nie są ustawione.

Ramka 5 zawiera pseudokod algorytmu. Dodatkowe operacje wprowadzone do algorytmu to: przetwarzanie alarmów oraz kilka więcej porównań podczas przetwarzania każdego zdarzenia i alarmu, wliczając w to test (realizowalny w koszcie stałym) czy $\exists i : Q_i$.empty(). Alarmów jest co najwyżej tyle, ile przetwarzanych zdarzeń. W rezultacie koszt przedstawionego algorytmu jest większy od kosztu algorytmu standardowego o pewien niewielki współczynnik (średnio 2 według testów przedstawionych w punkcie 4.2.2), pozostaje jednak rzędu $O(EN \log N)$. Zamiast N mechanizmów alarmowych, po jednym na proces, można używać pojedynczego mechanizmu, w celu zredukowania liczby generowanych alarmów (zdarzenie e_j może przestawić alarm ustawiony przez e_i zanim alarm zadziała). Jednak w niektórych sytuacjach spowoduje to wykrycie SCGS nieco późniejsze, niż przy użyciu N alarmów.

Opisany w tym punkcie algorytm został przedstawiony w publikacji [21].

Algorithm 5 Wczesne wykrywanie SCGS przy użyciu ograniczonego (<T) czasu transmisji komunikatów do monitora

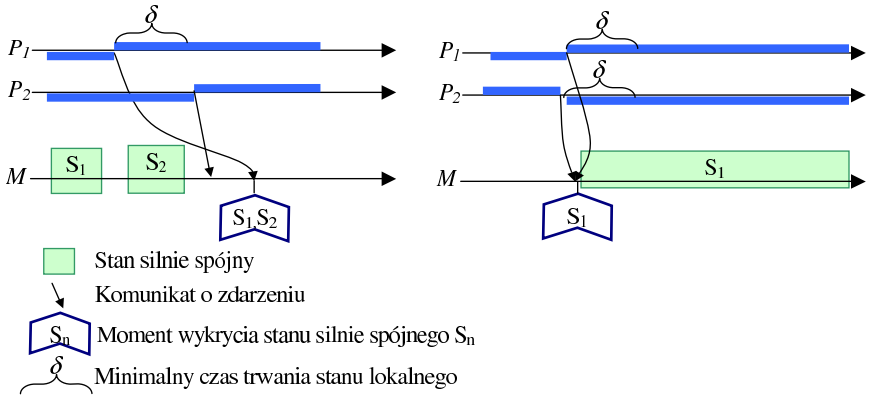
```
loop
  e_i= receive() /*odebrano komunikat o zdarzeniu od P_i*/
  if e_i≠alarm then
    if not Q_i.empty() then
      Q_i.append(e_i)
      alarmSet(T-(zr_M-zr(e_i))+ε_M+ε_max+γ,i)
      checkSCGS()
    else
      Q_i.append(e_i)
      ptr=minT.insert(zr(e_i)-ε_i,i,maxRptr_i)
      alarmSet(T-(zr_M-zr(e_i))+ε_M+ε_max+γ,i)
      checkSCGS()
    fi
  else /*e_i=alarm*/
    checkSCGS()
  fi
endloop

function tryToAdvance() : boolean
  if minT.empty() then return FALSE
  <t_min,i,ptr> = minT.min()
  if ∃i : Q_i.empty() and t_min> zr_M-T-ε_M-ε_max then
    /*ealier termination can appear yet*/
    return FALSE
  fi
  minT.removemin()
  maxR.remove(ptr)
  maxRptr_i= maxR.insert(<t_min+2ε_i,i>)
  R_i= Q_i.first()
  Q_i.remove()
  if not Q_i.empty() then
    t_min= zr(Q_i.first())
    minT.insert(t_min-ε_i,i,maxRptr_i)
  fi
  wykryty = FALSE /*nowa kombinacja stanów lokalnych*/
  return TRUE
endfunc

procedure checkSCGS()
  loop
    if not wykryty then
      if minT.empty() then
        min=zr_M-T-ε_M-ε_max
      else
        <t_min,i,ptr> = minT.min()
        if ∃i : Q_i.empty() and zr_M-T-ε_M-ε_max<t_min then
            min=zr_M-T-ε_M-ε_max
        else
          min=t_min
        fi
      fi
```

```
      <max,i> =maxR.max()
      if min > max then
        /*SCGS znaleziony, SCGS=< s_1,..,s_N >, gdzie s_i to stan lokalny P_i
             pomiędzy zdarzeniami R_i a Q_i.first() */
        if „to końcowy stan aplikacji“ then exit
        wykryty=TRUE
        return
      else if i==j then
        /*zdarzenia ograniczające trwanie SCGS są z tego samego procesu*/
        <min2,i,ptr> = minT.min2()
        <max2,j> = maxR.max2()
        if ∃i : Q_i.empty() and zr_M-T-ε_M-ε_max< min2 then
          /*koniec stanu wcześniejszy niż min2 może jeszcze nadejść*/
          return
        fi
        if (max2<min and min2>max) then
          /*SCGS znaleziony, SCGS=< s_1,..,s_N >, gdzie s_i to stan lokalny P_i
               pomiędzy zdarzeniami R_i a Q_i.first() */
          if „to końcowy stan aplikacji“ then exit
          wykryty=TRUE
          return
        fi
      fi /*i==j*/
    fi /*not wykryty*/
    /* szukaj dalej */
    if not tryToAdvance() then return
  endloop
endproc
```

2.4.2 Ustalenie minimalnego czasu trwania stanów lokalnych

Wczesne wykrywanie silnie spójnych stanów lokalnych możliwe jest także, gdy czas trwania stanów lokalnych jest ograniczony z dołu i ograniczenie to znane jest monitorowi. Przyjmijmy, ze stany lokalne trwają co najmniej czas δ. Przybycie komunikatu informującego o zdarzeniu rozpoczynającym nowy stan lokalny procesu daje wówczas od razu możliwość uwzględnienia tego stanu, jako już trwającgo δ czasu. W pewnych sytuacjach oznacza to natychmiastowe wykrycie kolejnego SCGS. Oczywiście czas trwania (zakończenia) tak rozpoznanego stanu spójnego jest niewiadomy w momencie jego wykrycia. Na rysunku 2.8 po lewej stronie widzimy przykład, w którym przybycie informacji o zdarzeniu od procesu P_1 umożliwiło wykrycie od razu dwóch silnie spójnych stanów globalnych: S_1- gdyż stały się znane zakończenia stanów lokalnych wchodzących w jego skład, oraz S_2- bo gwarantowany czas trwania nowego, bieżącego stanu procesu P_1 pokrywa się ze znanym już okresem trwania stanu procesu P_2. Po prawej stronie rysunku mamy przykład wykrycia SCGS aktualnie jeszcze trwającego, składającego się z niezakończonych stanów lokalnych o pokrywających się gwarantowanych okresach ich trwania.

Określony minimalny czas trwania stanu lokalnego jest spotykany w praktyce. Gdy raporty zawierają rezultaty przeliczenia kolejnych zadań lokalnych, możliwe bywa określenie minimalnej wielkości takiego zadania, a zatem minimalnego czasu pomiędzy przeliczeniem dwóch kolejnych zadań. Innym przykładem są periodyczne raporty o wartości lokalnego obciążenia procesora, lub, powiedzmy, temperatury odczytywanej przez zestaw czujników. Konfiguracja systemu ustala jak często dany czujnik ma sprawdzać temperaturę, zatem dla każdego z nich ustalony jest minimalny czas pomiędzy kolejnymi odczytami, czyli czas trwania stanu lokalnego. Tego rodzaju ograniczenie częstości raportowania zmian można stosować, o ile rzadsze obserwacje nadal umożliwią



Rysunek 2.8: Natychmiastowe wykrywanie SCGS przy znanym minimalnym czasie trwania stanu lokalnego

zauważenie interesujących nas tendencji. Np. bezczynny procesor pozostanie bezczynny, a wzrost monitorowanej temperatury będzie wykryty o ile utrzyma się stosownie długo. Kosztem precyzji monitorowania (zamiast kilku szybko po sobie następujących małych korekt wartości, otrzymamy jedną większa korektę) uzyskuje się ograniczenie liczby komunikatów, co zmniejsza obciążenie monitora i sieci oraz wydłuża czasy trwania stanów lokalnych. Znacząca rola tych parametrów opisana jest w rozdziale 4.

Algorytm wykorzystujący znany minimalny czas trwania stanu lokalnego łatwo otrzymać przez adaptację zmodyfikowanego algorytmu standardowego 2. Struktury danych i ich inicjalizacja są takie same. Dodatkowo przyjmijmy, że dla procesu P_i minimalny czas trwania jego stanu lokalnego wynosi δ_i. Każdy stan lokalny rozpoczęty zdarzeniem e_i o nieznanym jeszcze zakończeniu, otrzymuje tymczasowe zakończenie w postaci zdarzenia $delta$, przy czym $zr(delta) = zr(e_i) + \delta_i$. Zdarzenia faktycznie kończące dany stan zastępują zdarzenia $delta$. Łatwo to wykonać dla kolejek FIFO Q_i (zastąpienie pierwszego elementu), lecz dla kolejki priorytetowej $minT$ wymagane jest użycie dodatkowego wektora (analogicznego do $maxRptr$), przechowywującego dla elementów z czoła kolejek Q_i bezpośrednie wskaźniki do odpowiadających im elementów w $minT$. Ponieważ jest to mechanizm identyczny, jak już stosowny do wymiany elementów w $maxR$, nie będziemy go dokładniej opisywać. Dodamy jeszcze zmienną logiczną $wykryty$. Ma ona wartość TRUE wtw, gdy wykryto już SCGS będący kombinacją bieżących stanów lokalnych, zapobiega to raportowaniu tego samego SCGS wiele razy (inicjalnie $wykryty$=FALSE). Tekst algorytmu podany jest w ramce Algorytm 6. Koszt algorytmu jest zwiększony względem kosztu algorytmu wyjściowego, przez obsługę zdarzeń $delta$, co podwaja sumaryczną liczbę przetwarzanych zdarzeń. Zwykłe zdarzenia mogą być obsługiwane nieco inaczej (zastępują zdarzenia $delta$ w Q_i i $minT$), lecz koszt obsługi pozostaje taki sam. W rezultacie pozostajemy przy koszcie rzędu $EN \log N$.

2.4.2.1 Połączenie z poprzednią metodą

Dwie wyżej przedstawione metody wczesnego wykrywania silnie spójnych stanów globalnych można zastosować jednocześnie, aby połączyć ich możliwości. Na rysunku 2.9 pokazana jest sytuacja, w której uwzględnienie jedynie minimalnego czasu trwania stanów lokalnych nie przynosi pełnego efektu. Stan globalny S_1 daje się szybko wykryć, gdyż okres trwania zakończonego stanu s_2^1 procesu P_2 pokrywa się z gwarantowanym okresem trwania bieżącego stanu procesu P_1. Z tym ostatnim nie pokrywa się jednak gwarantowany okres trwania noworozpoczętego stanu s_2^2. Zatem stan globalny S_2 można wcześnie wykryć jedynie, jeśli czas transmisji komunikatu do monitora jest ograniczony, po upływie opóźnienia t od chwili rozpoczęcia stanu s_2^2. Odpowiednie połączenie algorytmów 5 i 6 wymagało następujących modyfikacji w algorytmie 5:

- W pętli głównej, gdy przybył nowy komunikat od procesu P_i, sprawdzamy, czy kolejka Q_i zawiera zdarzenie *delta*. Jeśli tak (*delta* musi być wtedy jedynym elementem Q_i), to przybyłe zdarzenie zastępuje *deltę* w Q_i oraz w $minT$.

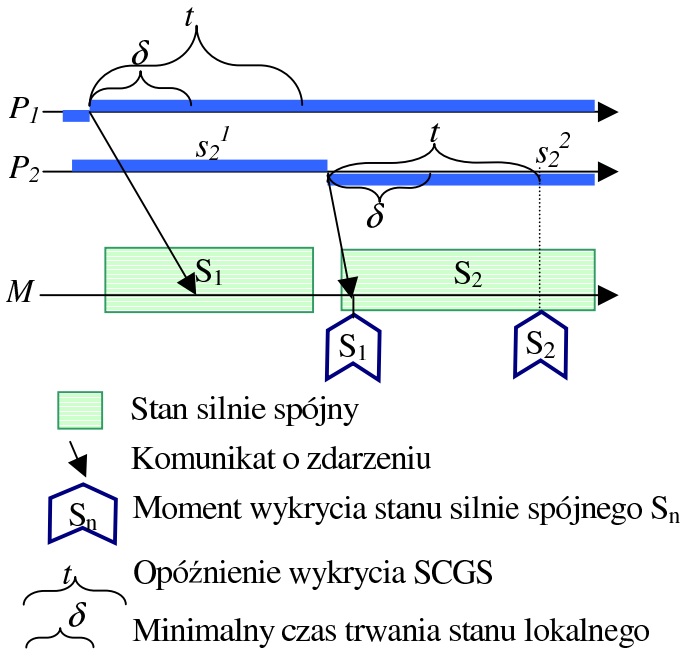
Algorithm 6 Wczesne wykrywanie SCGS przy znanym minimalnym czasie trwania stanów lokalnych

Wątek pomocniczy:

```
on „przybył komunikat o zdarzeniu e z procesu P_i“ do
 if Q_i.first()=delta then
   „zastąp zdarzenie delta zdarzeniem e w Q_i oraz minT“
 else
   Q_i.append(e)
 if „monitor był zawieszony w oczekiwaniu na zdarzenie od P_i“ then
   resume monitor
```

Wątek główny:

```
loop
  <t,i,ptr> = minT.min()
  min = t
  <max,j> = maxR.max()
  if not wykryty then
    if min > max then
      /*SCGS znaleziony, SCGS=<s_1,..,s_N>, gdzie s_ijest stanem lokalnym P_i po-
między zdarzeniami R_i a Q_i.first() */
      wykryty=TRUE
      if „to końcowy stan aplikacji“ then loopexit
    else if i==j
      /*oba zdarzenia z tego samego procesu*/
      <min2,,> = minT.min2()
      <max2,> = maxR.max2()
      if (max2<min AND min2>max) then
        /*SCGS znaleziony */
        wykryty=TRUE
        if „to końcowy stan aplikacji“ then loopexit
      fi
    fi
  fi /*not wykryty*/
  /* szukaj dalej */
  if Q_i.first() == delta then
    /*czekaj na zdarzenie od P_i*/
    suspend
  else
    wykryty=FALSE
    minT.removemin()
    maxR.remove(ptr)
    maxRptr_i= maxR.insert(<t+2ε_i,i>)
    R_i=Q_i.first()
    Q_i.remove()
    if Q_i.empty() then
      Q_i.insert(delta) /*zr(delta) = zr(R_i)+δ_i*/
    fi
    t= zr(Q_i.first())
    minT.insert(t-ε_i, i, maxRptr_i)
  fi
endloop
```



Rysunek 2.9: Łączne użycie dwóch metod wczesnego wykrywania SCGS

- w funkcji tryToAdvance(), gdy chcemy w P_i przejść do jego kolejnego stanu lokalnego, lecz kolejny stan nie jest jeszcze zakończony (Q_i staje się pusta), rozważamy możliwość użycia *delty*. Jeśli czas trwania stanu lokalnego zagwarantowany zdarzeniem *delta* jest dłuższy niż czas trwania wynikły z ograniczenia na czas transmisji komunikatu, czyli jeśli $t_{min} + \delta > zr_M - \mathtt{T} - \varepsilon_M - \varepsilon_{max}$, wówczas wstawiamy *deltę* do Q_i i $minT$.
- w funkcji tryToAdvance(), gdy chcemy w P_i przejść do jego kolejnego stanu lokalnego, a bieżący stan lokalny kończy się zdarzeniem *delta*, wtedy sprawdzamy czy *delta* powinna zostać usunięta. Jeśli czas trwania stanu lokalnego zagwarantowany zdarzeniem *delta* jest krótszy niż czas trwania wynikły z ograniczenia na czas transmisji komunikatu, czyli jeśli $t_{min} > zr_M - \mathtt{T} - \varepsilon_M - \varepsilon_{max}$, wówczas usuwamy deltę z Q_i oraz $minT$. Ta operacja odblokowuje mechanizm alarmów, zablokowany dotąd obecnością *delty*.

2.5 Hierarchiczne wykrywanie stanów silnie spójnych

Przedstawione do tej pory metody detekcji silnie spójnych stanów globalnych są metodami scentralizowanymi. Całe przetwarzanie raportów o zdarzeniach (lub stanach lokalnych procesów) skoncentrowane jest w pojedynczym centralnym monitorze. Rozwiązanie takie jest proste, ale nie pozbawione wad. Wadą najistotniejszą dla nas jest realna możliwość przeciążenia monitora. Duża liczba monitorowanych procesów oraz szybka zmienność stanów lokalnych powodują, że do centralnego monitora kierowany jest obfity strumień komunikatów. Istnieje zagrożenie, że łącza, którymi komunikaty te docierają, zostaną obciążone ponad miarę. Przy tym każdy komunikat o zdarzeniu musi zostać przetworzony przez algorytm wykrywania SCGS, dlatego zagrożeniem jest też potencjalne przeciążenie procesora przypisanego monitorowi. Obserwacja globalnych stanów spójnych jest tylko etapem pośrednim w dążeniu do wyliczania predykatów globalnych określonych na tych stanach. Zbyt duży koszt konstrukcji globalnych stanów spójnych (i ewaluacji predykatów globalnych) zmuszał do pewnych ustępstw. Ograniczano klasy analizowalnych predykatów do takich, dla których istnieją specjalizowane szybkie algorytmy. Szerzej na ten temat pisaliśmy już w punkcie 1.3. Alternatywą było przetwarzanie off-line. Jednak nie ma ono racji bytu, gdy informacja

o stanie aplikacji jest wymagana jeszcze podczas jej działania. Tak właśnie jest w najbardziej interesującym nas przypadku - w przypadku sterowania on-line wykonaniem aplikacji w oparciu o jej stany globalne.

Bardzo istotnym krokiem w kierunku redukcji kosztu wyliczania globalnych stanów spójnych jest już samo zastosowanie silnie spójnych stanów globalnych, których koszt konstrukcji - $O(EN \log N)$ - jest o wiele mniejszy, niż dla stanów spójnych opartych o wektorowe zegary logiczne - $\Omega(E^N)$. Krok ten nie zmienia jednak scentralizowanego charakteru detekcji stanów aplikacji. Nowe podejście, które tu przedstawiamy, polega na podziale i rozproszeniu zadań obliczeniowych i ruchu sieciowego związanych z wykrywaniem globalnych stanów spójnych. Zamiast pojedynczego monitora proponujemy użyć hierarchii monitorów. Pomysł najwygodniej będzie opisać dla hierarchii dwupoziomowej (poziomów może być więcej), zawierającej monitory grupowe (niższego rzędu) i jeden monitor główny (wyższego rzędu). Procesy aplikacyjne dzielimy na grupy $G_i, i = 1..K$, przy czym grupa G_i zawiera N_i procesów, $\sum_{i=1}^{K} N_i = N$. Procesy z grupy G_i współpracują z monitorem grupowym M_i. Monitory grupowe wykrywają grupowe stany silnie spójne i przekazują o nich informację do monitora głównego. Ten konsoliduje otrzymane dane o stanach grupowych i konstruuje globalne stany silnie spójne. Oczekiwane zalety opisanego pomysłu są następujące:

1. Pojedynczy monitor centralny, a dokładniej - procesor mu przypisany, jest potencjalnie wąskim gardłem obliczeniowym z uwagi na konieczność odbierania wszystkich komunikatów o zdarzeniach (stanach lokalnych) i ich przetwarzania przez algorytm konstrukcji globalnych stanów spójnych. Jego zadanie można rozdzielić na większą liczbę procesorów stosując hierarchię monitorów.

2. Pojedynczy monitor centralny odbiera komunikaty od wszystkich procesów aplikacyjnych. Łącze do niego prowadzące może łatwo stać się wąskim gardłem. Stosując hierarchię monitorów chcielibyśmy zorganizować ruch komunikatów o stanach w sposób drzewiasty. Monitory grupowe powinny redukować rozmiar danych otrzymanych od im przypisanych procesów, zanim prześlą je do monitora głównego. Pozwoli to rozłożyć obciążenie sieci wywołanie komunikatami o stanach na większą liczbę łączy.

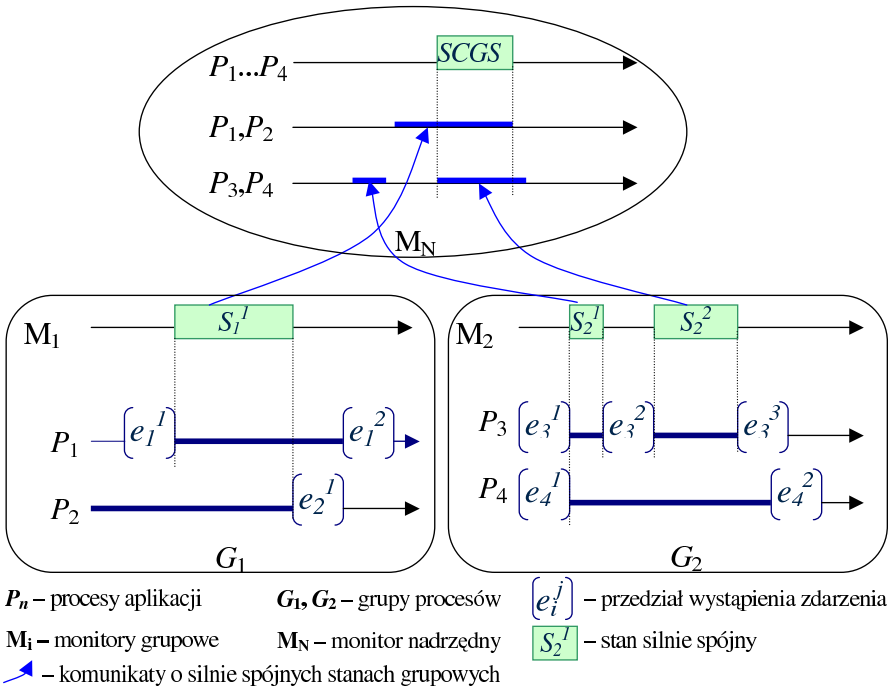
3. Wyliczone Grupowe Stany Spójne mogą być użyteczne już na poziomie danej grupy. Ponieważ ich wyliczanie przez monitor grupowy jest szybsze niż gdyby to miał robić monitor centralny (bo: monitor grupowy jest mniej obciążony i może być umieszczony „bliżej“ procesów swojej grupy niż monitor centralny), można z nich szybciej skorzystać.

4. Monitory grupowe mogą korzystać z lepszej jakości synchronizacji zegarów w obrębie grupy niż globalna jakość tej synchronizacji. Pozwala to, bez zwiększenia kosztu, wykrywać silnie spójne stany grupowe i co za tym idzie silnie spójne stany globalne, niemożliwe do zauważenia z globalnej perspektywy przy uwzględnieniu tylko globalnej jakości synchronizacji zegarów. Ilustracją tej tezy może być rysunek 2.4, gdy uznamy, że P_1 i P_2 są w jednej grupie, P_3 jest w innej, a przedstawioną tam względną dokładność synchronizacji zegarów pomiędzy P_1 a P_3 i P_2 a P_3 potraktujemy jako dokładność globalną.

5. Hierarchicznie wyliczane globalne stany spójne umożliwiają hierarchiczną ewaluację predykatów globalnych. Zalety hierarchicznej ewaluacji predykatów można opisać podając argumenty analogiczne do pierwszych trzech wyżej wymienionych.

6. Podział procesów na grupy i przypisanie im osobnych monitorów daje możliwość modularyzacji aplikacji oraz zdecentralizowania sterowania opartego o predykaty globalne.

7. Większa liczba monitorów pozwala myśleć realnie o wprowadzeniu odporności na awarie monitora. Byłoby to znacznie trudniejsze w przypadku pojedynczego monitora centralnego. W tej pracy jednak nie będziemy się dalej zajmować tym tematem.



Rysunek 2.10: Łączenie grupowych stanów silnie spójnych w stany globalne

Zanim przejdziemy do omówienia hierarchicznych algorytmów wykrywania SCGS, należy zauważyć, że hierachiczne konstruowanie globalnych spójnych stanów opartych o wektorowe zegary logiczne jest całkowicie nieopłacalne. Po pierwsze, przebadanie współbieżności grupowych stanów spójnych przez monitor wyższego poziomu wymagałoby podobnych nakładów obliczeniowych, jak w przypadku pojedynczego monitora centralnego. Po drugie, reprezentacja grupowych stanów spójnych, przekazywana monitorowi wyższego poziomu po każdym zdarzeniu w grupie, musiałaby zawierać wektorowe znaczniki czasu logicznego każdego procesu wchodzącego w skład danej grupy. Spowodowałoby to, że komunikaty zawierające grupowe stany spójne miałyby rozmiar pomiędzy $O(N)$ (dla małych grup) a $O(N^2)$ (dla dużych grup), przy liczbie komunikatów rzędu $O(E^{N_i})$ na grupę (N - liczba procesów w systemie, N_i - liczba procesów w grupie, E - maksymalna liczba zdarzeń w pojedynczym procesie). W rezultacie koszt komunikacji do monitora (monitorów) wyższego poziomu stałby się ogromny. Te obserwacje utwierdzają nas w przekonaniu, że zastosowanie znaczników czasu rzeczywistego i silnie spójnych stanów globalnych było trafną decyzją.

2.5.1 Hierarchiczna wersja standardowego algorytmu wykrywania SCGS

Zasadniczą rolę w standardowym algorytmie SCGS (algorytmy 1 i 2) pełnią operacje min i max. Przy ich pomocy zostają znalezione najwcześniejsze zakończenie i najpóźniejszy początek stanu, spośród aktualnie rozpatrywanej kombinacji stanów lokalnych. Operacje te są łączne i przemienne, toteż w naturalny sposób można je przeprowadzić najpierw na poziomie grup, uwzględniając stany procesów należących do danej grupy, a następnie, działając już na częściowych rezultatach, na poziomie globalnym. Pomysł dla drzewa wysokości 2 ilustruje rysunek 2.10, a opis algorytmu podany jest w ramce Algorytm 7. W kroku numer 6 konieczne jest użycie algorytmu 1 (a nie 2), bo monitor nadrzędny otrzymuje informacje o okresach trwania grupowych stanów silnie spójnych, a nie osobno o ich początkach i końcach. Przy tym koniec grupowego stanu silnie spójnego nie jest nigdy identyczny z początkiem kolejnego grupowego stanu silnie spójnego. Odstęp pomiędzy kolejnymi stanami silnie spójnymi nie może być mniejszy niż $2\varepsilon_i$ dla pewnego i. Pamiętajmy bowiem, że liczy się gwarantowany okres trwania stanu spójnego, trzeba uwzględnienić błędy w synchronizacji zegarów - przypomnijmy sobie rysunek 2.1.

Aby określic koszt hierarchicznej wersji standardowego alorytmu SCGS, dla uproszczenia przyj-

Algorithm 7 Hierarchiczna wersja standardowego algorytmu SCGS

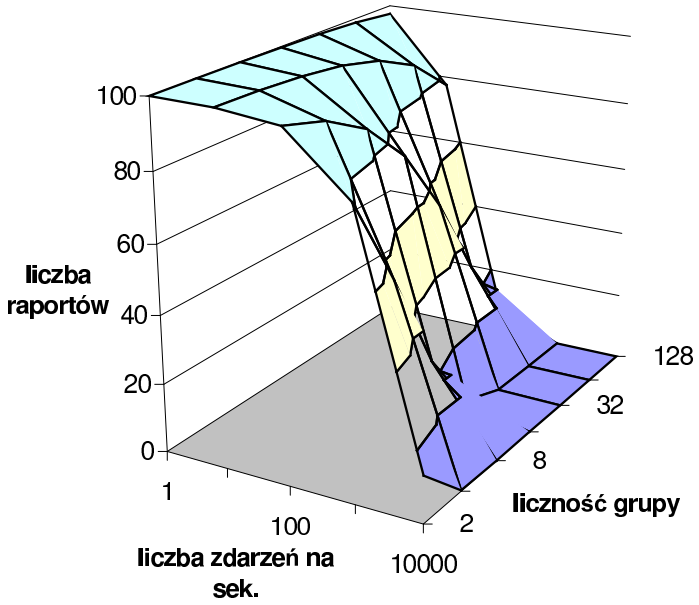
1. N procesów aplikacyjnych podzielonych jest na K grup, grupa G_i zawiera N_i procesów, $\sum_{i-1}^{K} N_i = N$.
2. Każda grupa procesów współpracuje z przydzielonym jej monitorem.
3. Monitory grupowe są liściami w drzewie monitorów. Stopień drzewa i jego wysokość są dowolne. Na wyższych poziomach drzewa znajdują się monitory nadrzędne. W korzeniu znajduje się monitor główny.
4. W obrębie każdej grupy procesów przydzielony jej monitor realizuje standardowy algorytm wykrywania SCGS (algorytm 1 lub 2).
5. Informacja o wykrytych w grupie silnie spójnych stanach grupowych (ich gwarantowane okresy trwania) jest przekazywana do nadrzędnych monitorów.
6. Nadrzędne monitory działają według algorytmu 1, traktując otrzymywane informacje o gwarantowanych okresach trwania stanów grupowych jak komunikaty o okresach trwania stanów lokalnych pojedynczych procesów. Dla stanu grupowego S przyjmuje się, że $\underline{zr}(\underline{S}) = \overline{zr}(\underline{S}) = zr(\underline{S})$ i $\underline{zr}(\overline{S}) = \overline{zr}(\overline{S}) = zr(\overline{S})$ (błędy wskazań zegarów zostały już uwzględnione na pierwszym poziomie monitorów).
7. Jeśli monitor jest korzeniem w drzewie monitorów, znalezione przez niego silnie spójne stany są stanami globalnymi, wpp. informacje o znalezionych stanach spójnych przesyła do nadrzędnego monitora.

mijmy, że grupy procesów są równoliczne, przez $N_g = N/K$ oznaczmy liczbę procesów w grupie. Drzewo hierarchii monitorów jest pełne, ma stopień N_g i wysokość $L = \log_{N_g} N$. E to maksymalna liczba zdarzeń w procesie. Koszt obliczeniowy w obrębie jednej grupy jest rzędu $O(EN_g \log N_g)$. Liczba silnie spójnych stanów w grupie nie może być większa niż EN_g. Przenosząc te zależności na wyższe poziomy hierarchii otrzymujemy wzór

$$koszt = \sum_{q=1}^{L} \frac{N}{q} E^q \log N_g = ENL \log N_g = EN \log {}^{L} = EN \log N.$$

Czyli sumaryczny koszt obliczeniowy (uwzględniający wszystkie monitory) algorytmu hierarchicznego jest identyczny, jak dla algorytmu scentralizowanego. Udało się uzyskać rozproszenie obliczeń bez zwiększania ich kosztu.

Zajmiemy się teraz kosztem komunikacji. Każdy monitor na poziomie q może otrzymać do E^q komunikatów o stanach, przy czym na poziomie q jest $N/^q$ monitorów. Zatem sumaryczna liczba komunikatów wynosi nie więcej niż LEN. Ta liczba jest L razy większa niż w algorytmie scentralizowanym. Przy tym, niestety, liczba komunikatów otrzymywanych przez monitor główny ma takie samo ograniczenie górne, jak liczba komunikatów otrzymywanych przez pojedynczy centralny monitor. Jednak w praktyce nie każde zdarzenie lokalne rozpoczyna nowy silnie spójny stan grupowy (np. zdarzenie e_2^2 na rysunku 2.1) i nie każdy silnie spójny stan grupowy przyczynia się do powstania silnie spójnego stanu grupowego wyższego poziomu (np. stan S_2^1 na rysunku 2.10). Dzięki temu zjawisku monitory otrzymują więcej raportów niż wysyłają, odciążając monitory wyższego poziomu. Przyjrzyjmy się bliżej temu zjawisku na poziomie pierwszym, gdzie monitory odbierają od procesów informacje o zdarzeniach. Z uwagi na nakładanie się przedziałów wystąpienia zdarzeń, stan spójny nie jest wykrywany pomiędzy zdarzeniami z różnych procesów odległymi w czasie o mniej niż 2ε (przyjmujemy jednorodną dokładność synchronizacji zegarów ε względem zegara wzorcowego). Przyjmijmy, że zdarzenia zachodzą niezależnie od siebie i mają jednostajny rozkład w czasie, czas działania aplikacji wynosi T i $\varepsilon \ll T$. Prawdopodobieństwo, że w czasie $< zr(e), zr(e) + 2\varepsilon >$ w wybranym innym procesie nie nastąpiło zdarzenie, wynosi



Rysunek 2.11: Oczekiwana liczba raportów o stanach

$(1 - \frac{2\varepsilon}{T})^E$. Prawdopodobieństwo, że że w czasie $< zr(e), zr(e) + 2\varepsilon >$ nie nastąpiło zdarzenie w żadnym innym procesie, czyli, że e rozpocznie nowy stan silnie spójny, wynosi $(1 - \frac{2\varepsilon}{T})^{E(N_g-1)}$. Przy podanych założeniach, średnio o taki współczynnik liczba raportów wysyłanych do monitora wyższego poziomu będzie mniejsza od liczby otrzymanych raportów. Dla ilustracji znaczenia otrzymanej formuły rozpatrzmy przykład. Mamy dwa poziomy monitorów: grupowe i główny. Przyjmijmy $\varepsilon = 100\mu s$, zdarzenia w procesie zachodzą średnio w tempie 100 na sekundę, grupa liczy 32 procesy. Wówczas oczekujemy, że monitory grupowe spowodują dwukrotną redukcją liczby komunikatów, czyli że monitor główny dostanie 2 razy mniej raportów, niż otrzymałby pojedynczy centalny monitor. Na rysunku 2.11 prezentujemy zależność oczekiwanej liczby komunikatów w monitorach grupowych pierwszego poziomu od częstotliwości występowania zdarzeń i liczności grupy. Za 100% przyjęliśmy liczbę komunikatów odbieranych przez monitor centralny w algorytmie standardowym. Widoczna jest znaczna redukcja liczby komunikatów nadchodzących do monitora głównego. Zauważmy, że mniejsza liczba komunikatów przekazywanych do monitorów wyższego poziomu oznacza zmniejszenie ich obciążenia obliczeniowego. Ponieważ opisywana redukcja zachodzi tym efektywniej, im częściej występują zdarzenia, bardzo dobrze łagodzi ona chwilowe szczyty aktywności procesów, mogące w innym razie tymczasowo przeciążyć monitor główny.

2.5.2 Synchronizacja zegarów w grupach procesów

Standardowy algorytm SCGS zakłada, że zegary procesów są zsynchronizowane globalnie, np. są synchronizowane względem jednego globalnego zegara wzorcowego. Można oczekiwać, że synchronizacja zegarów osobno w ramach wydzielonej grupy procesów może osiągnąć większą dokładność, niż synchronizacja globalna. Przykłady środowisk, w których taka sytuacja ma miejsce podaliśmy w punkcie 2.3. Wykorzystanie wewnątrz grupowej dokładności synchronizacji zegarów (lepszej niż dokładność synchronizacji globalnej) pozwala wykrywać grupowe stany silnie spójne niemożliwe do zauważenia przy użyciu globalnej dokładności synchronizacji zegarów. Stany takie mogą w dalszej kolejności prowadzić do wykrywania takich globalnych stanów silnie spójnych, które inaczej nie byłyby zauważone. Pisaliśmy już o tym na początku punktu 2.5 i podaliśmy tam stosowny przykład w tezach (teza 4) opisujących oczekiwane zalety hierarchicznego wykrywania SCGS.

Przyjmijmy, że w grupie G_i zegary procesów są zsynchronizowane względem siebie z dokładnością $2\varepsilon_i$, np. poprzez ich synchronizację z lokalnym dla grupy zegarem wzorcowym z dokładnością ε_i. Globalna dokładność synchronizacji zegarów to $2\varepsilon_G$. Zmodyfikujemy algorytm 7 w następujący sposób: wysokość drzewa monitorów ograniczymy do dwóch (jedynie dla prostoty opisu), a krok 6 algorytmu zmienimy na:

6. Monitor główny działa według algorytmu 1, traktując otrzymywane informacje o gwarantowanych okresach trwania stanów grupowych jak komunikaty o okresach trwania stanów lokalnych pojedynczych procesów. Dla stanu grupowego S_i z grupy G_i przyjmuje się, że $\underline{zr}(\underline{S_i}) = \overline{zr}(\underline{S_i}) = zr(\underline{S_i}) - \varepsilon_i + \varepsilon_G$ i $\underline{zr}(\overline{S_i}) = \overline{zr}(\overline{S_i}) = zr(\overline{S_i}) + \varepsilon_i - \varepsilon_G$ (przechodzimy od błędów wskazań zegarów na poziomie grupy do błędów na poziomie globalnym).

Zastosowane tu przeliczenie okresów gwarantowanego trwania stanów grupowych może spowodować, że po przeliczeniu otrzymamy $zr(\underline{S_i}) > zr(\overline{S_i})$. Nie jest to problemem. Taki przypadek będzie potraktowany identycznie, jak przypadek dwóch, następujących szybko jeden po drugim, zdarzeń pochodzących z jednego procesu. Dla stanu lokalnego s ograniczonego takimi zdarzeniami mamy $\overline{zr}(\underline{s}) > \underline{zr}(\overline{s})$ i jest to poprawnie obsługiwane przez algorytm, czyli taki stan może wejść w skład globalnego stanu silnie spójnego.

2.5.3 Względna dokładność synchronizacji zegarów między grupami

Algorytm detekcji SCGS biorący pod uwagę względną dokładność synchronizacji zegarów, opisany w punkcie 2.3, można zastosować także przy hierarchicznej konstrukcji silnie spójnych stanów globalnych. Wyobraźmy sobie dla przykładu system komputerowy, składający się z pewnej liczby klastrów połączonych sieci rozległą. Synchronizacja zegarów wewnątrz każdego klastra jest realizowana z użyciem lokalnie ustalonego wzorca czasu. Dla grup posługujących się tym samym wzorcem (np. ten sam radiowy sygnał czasu) wiadomo, że różnica wskazań zegarów dla procesów należących do tych grup będzie mała. Jednocześnie dla grup regulujących swe zegary w różny sposób (np. jedna według lokalnego odbiornika GPS, inna według odległego serwera czasu NTP) możliwe będą większe rozbieżności we wskazaniach zegarów. W przedstawionej sytuacji warto zastosować wariant hierarchicznego dwupoziomowego algorytmu SCGS, potrafiący wykorzystać informacje o międzygrupowej dokładności synchronizacji zegarów, podany w ramce Algorytm 8. W algorytmie tym koszt obliczeniowy dla monitora głównego wynosi $O(ENK^2 \log K)$, zatem koszt sumaryczny jest nieco wiekszy niż dla podstawowego algorytmu hierarchicznego. Z drugiej strony koszt ten jest znacznie mniejszy niż dla scentralizowanej wersji, także wykorzystującej względną dokładność synchronizacji.

2.5.4 Uwzględnianie niezakończonych stanów lokalnych

Pomysł przyśpieszenia momentu wykrycia silnie spójnych stanów globalnych, poprzez uwzględnianie niezakończonych stanów lokalnych, daje się zastosować też przy hierarchicznej metodzie konstrukcji SCGS. Gdy czas transmisji komunikatów do monitorów jest ograniczony, proponowany wariant algorytmu przedstawia się następująco:

- Monitory grupowe posługują się nieco zmodyfikowanym algorytmem 5. Algorytm ten wykrywa silnie spójne stany grupowe, dla których czas ich zakończenia nie zawsze jest znany już w momencie wykrycia. Modyfikacja polega na tym, że do monitorów wyższego poziomu raportowane są a) informacje o gwarantowanych okresach trwania zakończonych spójnych stanów grupowych, b) informacje o czasie rozpoczęcia spójnych stanów grupowych, dla których ich koniec nie jest jeszcze znany, c) informacje o czasie zakończenia poprzednio raportowanych niezakończonych spójnych stanów grupowych.
- Monitory wyższego poziomu posługują się inną modyfikacją algorytmu 5, taką, że algorytm przyjmuje trzy wyżej wymienione rodzaje komunikatów a), b), c), a nie tylko komunikaty o zdarzeniach kończących dany stan lokalny i jednocześnie rozpoczynających następny.

Algorithm 8 Hierarchiczne wykrywanie SCGS z użyciem względnej dokładności synchronizacji zegarów między grupami

1. N procesów aplikacyjnych podzielonych jest na K grup, grupa G_i zawiera N_i procesów, $\sum_{i-1}^{K} N_i = N$.
2. Każda grupa procesów współpracuje z przydzielonym jej monitorem, monitor nadrzędny współpracuje z monitorami gruppowymi.
3. Monitory grupowe posługują się zmodyfikowanym standardowym algorytmem SCGS (algorytm 2).
4. Dla silnie spójnego stanu grupowego rozpoczynającego się zdarzeniem e_i i kończącego się zdarzeniem e_j, zamiast jego gwarantowanego okresu trwania $< \overline{zr}(e_i), \underline{zr}(e_j) >$, raportowany do monitora głównego jest interwał $< zr(e_i), zr(e_j) >$.
5. Monitor nadrzędny dysponuje informacją o względnej dokładności synchronizacji zegarów pomiędzy grupami. $\varepsilon_{i,j} = \max\{|zr_k - zr_l| \text{ dla } k, l \text{ takich, że: } P_k \in G_i, P_l \in G_j\}$.
6. Monitor główny posługuje się algorytmem 3, traktując otrzymywane informacje o okresach trwania silnie spójnych stanów grupowych jak informacje o czasie trwania stanów lokalnych.

Możliwe opóźnienie, z jakim monitor główny dowiaduje się o zdarzeniach, zależy tu już nie tylko od charakterystyki i obciążenia sieci, ale też od czasu przetwarzania komunikatów o zdarzeniach przez monitory pośrednie. Opóźnienie to powinno być stale monitorowane, aby na podstawie bieżących obserwacji ustalać odpowiednio odległy czas alarmu.

2.5.5 Podsumowanie przedstawionych hierarchicznych algorytmów wykrywania SCGS

Oczekiwania wiązane z hierarchicznym wykrywanien SCGS przedstawiliśmy na wstępie punktu 2.5. Możemy teraz powiedzieć, że opracowane i przedstawione powyżej algorytmy hierarchicznego wykrywania SCGS spełniają wymienione postulaty (poza odpornością na awarię - tym się nie zajmowaliśmy). Teoretyczna analiza tych algorytmów wykazała ich skuteczność w rozpraszaniu obliczeń związanych z wykrywaniem SCGS, oraz dowiodła, że monitory grupowe mogą znacznie redukować liczbę komunikatów przekazywanych do monitora wyższego poziomu, dzięki temu rozdzielając równomiernie ruch sieciowy i redukując niebezpieczeństwo przeładowania łączy. Odpowiednie wersje hierarchicznego algorytmu wykrywania SCGS wykorzystują niejednakową jakość synchronizacji zegarów w poszczególnych grupach, umożliwiając wykrycie większej liczby stanów spójnych niż przy użyciu pojedynczej globalnej dokładności synchronizacji. Uzyskiwana dzięki przedstawionym algorytmom hierarchia stanów spójnych stanowi świetną podstawę do budowy hierarchicznych technik sterowania opartych na stanach spójnych.

Hierarchiczne algorytmy wykrywania SCGS zostały przedstawione w publikacjach [20, 19].

Uzupełnieniem niniejszego punktu, opisującego hierarchiczne algorytmy wykrywania SCGS, jest następny punkt (2.6) traktujący o hierarchicznym wyliczaniu predykatów globalnych oraz praktyczna ocena efektywności sterowania opartego o hierarchicznie wykrywane stany spójne przedstawiona w punkcie 4.2.5.

2.6 Hierarchiczne wartościowanie predykatów na stanach globalnych

Predykaty globalne to funkcje logiczne określone na stanach globalnych. Konstrukcja spójnych stanów globalnych jest pierwszym krokiem do użycia predykatów globalnych. Mając dany glo-

balny stan spójny, można wartościować na nim określone warunki logiczne. Dopiero spełnienie (lub nie) tych warunków niesie ze sobą istotną informację dla obserwatora. Rozłożenie obciążenia związanego z wykrywaniem silnie spójnych stanów globalnych, dzięki stosowaniu hierarchicznych algorytmów wykrywania SCGS, może okazać się mało istotne, jeśli ewaluacja predykatów globalnych odbywa się centralnie w monitorze głównym. Predykaty mogą mieć złożone formy, może być ich wiele, skutkiem czego moniotr główny łatwo może stać się przeciążony obliczeniami związanymi z wartościowaniem predykatów. Prostym rozwiązaniem tego problemu jest przydzielenie oddzielnych monitorów do ewaluacji predykatów związanych z różnymi monitorowanymi własnościami. Każdy z takich monitorów może otrzymywać informacje jedynie o zdarzeniach interesującego go rodzaju, np. jeden monitor o liczbie zadań, które dany proces ma jeszcze do rozwiązania, drugi o wynikach obliczeń. Nadal jednak, w ramach pojedynczego predykatu, mamy do czynienia z jego scentralizowaną ewaluacją. Konstruowanie globalnych stanów spójnych w sposób rozproszony, najpierw częściowo, na poziomie grup, pozwala, w niektórych przypadkach, na równie rozproszone i stopniowe wyliczanie predykatów globalnych. Dla predykatu φ postaci $\varphi = \varphi_1 \diamond ... \diamond \varphi_N$, gdzie $\diamond$ jest operacją łączną i przemienną, naturalne staje się wartościowanie φ w sposób hierarchiczny. W grupie G_i wyliczane jest $\diamond_{k:P_k \in G_i} \varphi_k$, rezultaty częściowe wykorzystywane są w monitorach wyższych rzędów. Jeśli przydział procesów do grup jest sekwencyjny, tzn. zbiór $\{k : P_k \in G_i\}$ jest zbiorem kolejnych liczb naturalnych dla $i = 1..K$, to wymóg przemienności staje się zbędny.

Jeśli predykat nie ma opisanej wyżej wygodnej postaci, można spróbować użyć innych technik. Załóżmy, że chcemy wykryć czy i kiedy predykat globalny φ został spełniony. Hierarchicznie konstruowane globalne stany spójne mają postać $S =< S^1, .., S^K >$, gdzie S^i to stan spójny grupy G_i. Załóżmy, że znamy takie predykaty φ_i, że $\varphi_i(S^i) \Rightarrow \neg\varphi(S)$. Gdy monitor grupowy stwierdza, że φ_i jest spełnione, wówczas nie musi on informować monitora głównego o zaistnieniu stanu S^i. Zyskujemy dzięki temu redukcję liczby komunikatów do monitora głównego i, co za tym idzie, redukcję obciążenia związanego z konstrukcją SCGS i wyliczaniem predykatów globalnych.

Predykaty globalne dające się zapisać w formie $\varphi = f(f_1(S^1), .., f_K(S^K))$ także dają możliwość usprawnienia ich ewaluacji przy zastosowaniu hierarchicznej metody konstrukcji globalnych stanów spójnych. Wartości $f_i(S^i)$ można wyliczać na poziomie grupy, oszczędzając pracy monitorowi głównemu. Jeśli rozmiar danych potrzebnych do opisania stanu S^i jest większy, niż rozmiar wartości (w sensie liczby bajtów) zwracanych przez funkcję f_i, to wysyłajac do monitora głównego wartości $f_i(S^i)$, zamiast danych opisujących stan S^i, dodatkowo redukujemy ruch sieciowy generowany przez algorytm.

2.7 Podsumowanie

Przedstawione podstawy teoretyczne dotyczące silnie spójnych stanów globalnych (SCGS), oraz algorytmy ich wykrywania, stanowią solidny fundament do dalszych prac nad wykorzystaniem SCGS do sterowania wykonaniem aplikacji równoległych. Zaprezentowane warianty algorytmów wykrywania SCGS, z uwagi na ich niewielką złożoność obliczeniową, powinny pracować wystarczająco efektywnie, aby użyć ich do monitorowania SCGS na bieżąco. Poszczególne warianty pozwalają wykorzystać specyficzne cechy środowiska obliczeniowego lub sterowanej aplikacji, i w ten sposób poprawić jakość pracy danego algorytmu. Opracowanie omawianych wariantów stanowi znaczący wkład własny w całość pracy. Wspomniana przed chwilą jakość pracy algorytmów wykrywania SCGS została przebadana i opisana w rozdziale 4.

Wysoką wydajność obliczeń można uzyskać stosując dużą liczbę procesorów, na których uruchamia się równoległy program. Jednak warunkiem osiągnięcia dobrej wydajności w ten sposób jest zapewnienie dobrej skalowalności, czyli podwyższania wydajności pracy przy zwiększaniu liczby użytych procesorów, zarówno środowiska sprzętowo-systemowego, jak również zastosowanego algorytmu równoległego. Proponowana w tej pracy metoda sterowania, wykorzystująca monitorowanie stanów globalnych aplikacji, jest fragmentem środowiska sprzętowo-systemowego. Dzięki opracowaniu hierarchicznych algorytmów wykrywania SCGS oraz zasad hierarchicznego wartościowania predykatów globalnych używanych do sterowania, umożliwiliśmy uzyskanie dobrej skalowalności proponowanej metody sterowania. Metoda ta może być zatem użyta w wysoko wydajnych obli-

czeniach równoległych realizowanych za pomocą dużej liczby procesorów.

Rozdział 3

Sterowanie w programach równoległych za pomocą predykatów na stanach globalnych

Na początku niniejszej pracy przedstawiliśmy szereg znanych metod sterowania w programach równoległych. Ich analiza doprowadziła do wniosków sformułowanych w punkcie 1.1.3: potrzebna jest nowa metoda sterowania, która działa efektywnie w środowiskach rozproszonych, daje programiście wygodę charakterystyczną dla metod bazujących na wspólnych zasobach, pozwala na posługiwanie się pojęciami wysokiego poziomu przy definiowaniu sterowania oraz jest niezależna od mechanizmów przekazywania danych. Wymagania te może spełnić sterowanie oparte o predykaty globalne.

Przez *sterowanie oparte o predykaty globalne* rozumiemy wpływanie na przebieg wykonania programu na podstawie bieżąco wartościowanych, odpowiednio dobranych do konkretnej aplikacji, predykatów globalnych. Zauważmy, że sterowanie oparte o predykaty globalne spełnia wszystkie przedstawione postulaty. Predykaty globalne określone są na stanach globalnych, stany te są wspólne dla wszystkich procesów i mogą być wyznaczane w systemach rozproszonych. Predykaty można definiować w oparciu o własności konkretnej aplikacji i wyrażać nimi złożone zależności. Spełnienie takich zależności może wyzwalać pewne akcje sterujące. W oczywisty sposób wyliczanie predykatów globalnych oraz uruchamianie na tej podstawie akcji sterujących nie stanowi mechanizmu przekazywania danych. Rozdział ten omawia szczegółowo pomysł systemu sterowania programami równoległymi/rozproszonymi w oparciu o predykaty globalne, analizuje związane z tym problemy proponuje konkretne rozwiązania.

3.1 Wybór i dostosowanie form ewaluacji predykatów globalnych dla potrzeb mechanizmu sterującego

3.1.1 Modalności

Dotychczasowe badania dotyczące predykatów globalnych (patrz 1.3.1) odbywały się w większości w kontekście ich zastosowania do budowy rozproszonych monitorów i programów uruchomieniowych. Tymczasem sterowanie na bieżąco wykonaniem aplikacji równoległej (rozproszonej) stawia specyficzne wymogi dla metody sprawdzającej spełnienie predykatów globalnych, na której to sterowanie chcielibyśmy oprzeć. W naszym rozumieniu, aby skutecznie sterować bieżącym wykonaniem aplikacji, trzeba:

- znać rzeczywisty przebieg wykonania aplikacji i móc go analizować,
- uzyskiwać informacje o przebiegu wykonania na bieżąco, z możliwie małym opóźnieniem,

- zbierając informacje o przebiegu wykonania nie wprowadzać dużego dodatkowego obciążenia systemu

Przyjrzymy się teraz, jak opisane w punkcie 1.3 oraz w rozdziale 2 metody wyznaczania stanów spójnych i ewaluacji predykatów globalnych (modalności), pasują do wymienionych wymagań. Zamieścimy tutaj tylko krótką ich charakterystykę, zawierającą jedynie najbardziej istotne, pod względem ich przydatności do sterowania aplikacją, elementy.

- Modalność *Possibly* - spełnienie Poss(φ) nie oznacza, że φ zostało spełnione w rzeczywistości, a jedynie, że było spełnione w pewnym obliczeniu równoważnym obliczeniu rzeczywistemu w sensie relacji przyczyny i skutku. Reakcja na spełnienie Poss(φ) może więc okazać się reakcją nieodpowiedną do faktycznie zaistniałej sytuacji. Ponieważ Poss(φ) wykrywane jest w momencie zauważenia pierwszego takiego spójnego stanu globalnego, w którym φ jest spełnione, moment wykrycia Poss(φ) nie musi mieć nic wspólnego z momentem rzeczywistego spełnienia φ (o ile taki moement w rzeczywistości w ogóle ma miejsce). Koszt wykrywania spełnienia Poss(φ) bez ograniczania postaci predykatu φ jest bardzo duży.

- Modalność *Definitely* - spełnienie Def(φ) daje gwarancję, że φ spełnione było faktycznie, gdyż oznacza, że w każdym obliczeniu zgodnym w sensie relacji przyczyny i skutku z uzyskaną obserwacją istnieje stan spełniający φ. Nie wiemy jednak, który z tych stanów miał miejsce w rzeczywistości (zatem nie znamy szczegółów zaistniałej sytuacji), ani kiedy (zapewne decyzja sterująca powinna być inna, gdy φ było spełnione przed pięcioma minutami, inna, gdy spełnione jest właśnie teraz). W rzeczywistym wykonaniu aplikacji φ mogło być spełnione, podczas gdy Def(φ) nie jest wykrywane. W rezultacie nie można zagwarantować, że spełnienie φ wywoła określoną reakcję. Koszt wykrywania spełnienia Def(φ) bez ograniczania postaci predykatu φ jest bardzo duży.

- Modalność *Currently* - wykrywanie Curr(φ) wymaga kooperacji procesów aplikacyjnych z monitorem. Procesy muszą wiedzieć, jakie predykaty są monitorowane, aby przed każdą akcją mogąca potencjalnie zanegować wartość predykatu prosić monitor o pozwolenie na kontynuację działania. Taka kooperacja powoduje duże koszty komunikacji i jest niewygodna z punktu widzenia programisty, bowiem wymaga dostosowania kodu procesów przy każdorazowej modyfikacji obserwowanych predykatów. Samo zawieszanie działania procesów w oczekiwaniu na pozwolenie kontynuacji od monitora powoduje niedopuszczalne dla nas straty w efektywności obliczeń wykonywanych przez procesy.

- Modalność *Properly* (z użyciem zegarów logicznych) - Prop(φ) ewaluuje predykat φ jedynie na wspólnych globalnych stanach spójnych. Pozwala to na skutecznie wykrywanie spełnienia predykatów o postaci będącej koniunkcją predykatów lokalnych - przy odpowiednim potraktowaniu stanów lokalnych procesów, predykaty takie są spełniane wyłącznie w stanach wspólnych. Niestety, dla innych form predykatów użycie tej modalności traci sens - predykaty o innych postaciach mogą być spełniane poza stanami wspólnymi i przy użyciu modalności *Properly* ich spełnienie nie będzie zauważane.

- Modalność *Instantly* - Inst(φ) wartościuje φ na silnie spójnych stanach globalnych, a zatem na stanach rzeczywiście zaistniałych. Stany te można znajdować stosunkowo niewielkim kosztem. Dla każdego znalezionego stanu znany jest (co najmniej przybliżony) czas jego wystąpienia i trwania. Spełnienie φ w stanach trwających mniej niż ε, gdzie ε to dokładność synchronizacji zegarów lokalnych procesów, może być niezauważone. Ta zależność wyznacza maksymalne tempo zmian stanów, dla którego możliwe jest skuteczne monitorowanie. Dla ustalonego tempa skuteczność monitoringu można ulepszyć poprawiając jakość synchronizacji zegarów.

Rezultatem niniejszego zestawienia jest wniosek, że modalność *Instantly* najlepiej nadaje się do wykorzystania w mechanizmie sterującym wykonaniem aplikacji, podejmującym decyzje sterujące na podstawie wartościowania predykatów globalnych.

3.1.1.1 Stany obserwowane

Do tej pory przyjmowaliśmy założenie, że predykaty globalne są wartościowane na spójnych stanach globalnych. W pewnych sytuacjach spójność stanu globalnego jest wymogiem nadmiarowym, niepotrzebnie obciążającym system sterujący i opóźniającym podjęcie decyzji sterującej. Proponowaną prostszą alternatywą jest ewaluowanie predykatów na stanach globalnych bezpośrednio dostępnych obserwatorowi - czyli na stanach obserwowanych. W tym modelu każda zmiana stanu lokalnego procesu, o której dowiaduje się monitor, powoduje ewaluacje predykatów, bez sprawdzenia, czy otrzymany nowy stan globalny jest spójny. W rezultacie decyzje sterujące mogą być podejmowane natychmiast po odebraniu każdego pojedynczego raportu o stanie procesu. Raporty są przetwarzane w kolejności FCFS, przy czym ta kolejność nie musi odpowiadać rzeczywistej kolejności zmian stanów procesów. Opisany sposób użycia predykatów globalnych można stosować tylko w przypadkach szczególnych:

1. Gdy predykat globalny, na podstawie którego zbudowana jest funkcja sterująca, ma postać alternatywy predykatów lokalnych (czyli predykatów zależnych od stanu pojedynczego procesu). Każde przybycie raportu zmienia stan pojedynczego procesu i może zmienić wartość predykatu lokalnego z tym procesem związanego. Ze względu na strukturę predykatu globalnego jego wartość może być rozpatrzona niezależnie od stanów pozostałych procesów.

2. Gdy predykat globalny, na podstawie którego zbudowana jest funkcja sterująca, ma postać $F(x_1, .., x_n) > K$, gdzie x_i jest zmienną procesu P_i i funkcja F jest monotonicznie rosnąca, tj $F(x_1, .., x_i^j, .., x_n) \leq F(x_1, .., x_i^{j+1}, .., x_n)$. Dla tak określonego predykatu spełniona jest zasada żywotności (zauważone w końcu zostanie jego spełnienie) i zapewniania (jeśli wykryto spełnienie predykatu, to spełniony jest on w rzeczywistości).

Gdy powyższe warunki są spełnione, zalecane jest stosowanie stanów obserwowanych w celu przyśpieszenia i uproszczenia procesu podejmowania decyzji sterujących na podstawie predykatów globalnych.

3.1.2 Spełnienie predykatu a reakcje procesów

W proponowanym mechaniźmie sterowania aplikacją równoległą, spełnienie predykatu globalnego ma powodować pewne reakcje procesów, wpływając na przebieg ich wykonania. Związek pomiędzy spełnieniem predykatu a wywoływaną reakcją może przybierać różne formy i teraz zastanowimy się nad możliwościami tu istniejącymi. Celem tych rozważań jest znalezienie takiego sposobu powiązania spełnienia predykatu z reakcjami procesów, który umożliwi wygodną i wszechstronną kontrolę nad aplikacją.

Najprostszą formę związku pomiędzy spełnieniem predykatu a wywoływaną reakcją określamy jako *jednorodne reakcje globalne*, czyli że spełnienie globalnego predykatu powoduje ustaloną globalną reakcję. Oznacza to, że wszystkie procesy dowiadują się o spełnieniu predykatu i wszystkie reagują w ten sam sposób. Operacja synchronizująca bariery jest tu dobrym przykładem: predykat φ= „czy wszystkie procesy osiągnęły barierę?“; *reakcja* = „przekrocz barierę“. Innym przykład może stanowić wykrywanie zakończenia działania aplikacji rozproszonej [42]. Zakładamy, że procesy komunikują się przez komunikaty i że znajdują się w stanie aktywnym (wtedy mogą wysyłać komunikaty) lub pasywnym (wtedy nie wysyłają komunikatów). Proces może samoistnie przejść ze stanu aktywnego w pasywny, natomiast z pasywnego przechodzi w aktywny gdy odbiera komunikat. Aplikacja jest zakończona, gdy wszystkie procesy są pasywne i wszystkie komunikaty zostały odebrane. Predykat φ= „czy wszystkie procesy są pasywne i wszystkie komunikaty odebrane?“; *reakcja* = „zakończ proces“. Ten prosty schemat powiązania spełnienia predykatu z reakcją procesu jest wystarczający tylko w takich prostych przypadkach. W innych, reakcje procesów powinny być zróżnicowane, np. po wykryciu zakleszczenia wystarczy reakcja jednego procesu, aby przerwać cykl zależności; po wykryciu niezrównoważenia obciążenia w systemie tylko nadmiernie/za słabo obciążone procesy powinny realizować procedurę wyrównania obciążenia. To rozumowanie prowadzi nas do pomysłu predykatów globalnych wartościowanych względem procesu.

Predykat globalny φ wartościowany względem procesu oznaczamy jako $\varphi(i)$, gdzie $i = 1..N$ jest numerem procesu. Predykat taki jest wartościowany osobno dla każdego procesu. Gdy $\varphi(i)$ jest spełnione, proces P_i powinien zareagować. Oto przykład: $\varphi(i)$ = „czy P_i uczestniczy w zakleszczeniu i czy powinien w związku z tym zwolnić zarezerwowane zasoby?“; *reakcja* = „zwolnij zarezerwowane zasoby“. Drugi przykład dotyczy równoległych rozproszonych algorytmów branch-and-bound [63, 6]. Poszukiwane jest najlepsze rozwiązanie. Przestrzeń rozwiązań dzieli się na podobszary, przy czym dla każdego podobszaru znane jest górne oszacowanie znajdujących się tam rozwiązań. Procesy przeszukują podobszary, pomijając te, dla których oszacowanie jest gorsze niż znane już rozwiązanie. Nowo znalezione rozwiązanie może okazać się lepsze niż oszacowania dla aktualnie przeszukiwanych przez inne procesy podobszarów. Dlatego warto wprowadzić następującą zasadę wyrażoną predykatem wartościowanym względem procesu: $\varphi(i)$ = „czy P_i przeszukuje podobszar, dla którego oszacowanie jest gorsze niż aktualne najlepsze rozwiązanie?“; *reakcja* = „porzuć przeszukiwanie bieżącego podobszaru i zajmij się następnym“.

W bardziej złożonych sytuacjach powyższa propozycja jest nadal niewystarczająca. Oto przykład: dla strategii równoważenia obciążenia, nakazującej przeciążonym procesom oddać część swych zadań procesom najmniej obciążonym, spełnienie predykatu postaci $\varphi(i)$ = „czy P_i jest przeciążony?“ nie sugeruje, komu i ile zadań P_i ma oddać. Można próbować wartościować predykat względem pary procesów: $\varphi(i, j)$ = „czy P_i jest przeciążony, a P_j niedociążony?“. Powoduje to jednak zwielokrotnienie liczby wartościowań predykatu, i sprawia, że reakcje są podejmowane na podstawie relacji pomiędzy parami procesów, zamiast na podstawie stanu globalnego. Jeśli *reakcja* = „oddaj połowę zadań dla P_j“, to jej zastosowanie będzie nieprawidłowe, gdy mamy cztery przeciążone procesy i wszystkie one będą dociążać ten sam bezczynny proces P_j. Natomiast w opisanym wyżej przykładzie dotyczącym algorytmu branch-and-bound, możnaby uniknąć części wystąpień reakcji porzucenia (zapewne dość kosztownych), gdyby każda taka reakcja niosła procesom informację o nowym globalnym najlepszym rozwiązaniu, pozwalając procesom w przyszłości lokalnie pominąć nieperspektywiczne podobszary.

Zasygnalizowane problemy mają wspólne źródło: procesy nie znają globalnego stanu, w którym dany predykat globalny został spełniony powodując reakcję, a do wyboru prawidłowej reakcji taka wiedza jest potrzebna. Dlatego proponujemy, aby wywołując reakcję w procesie, dostarczać mu dodatkowo informację o stanie globalnym tę reakcję powodującym. Nie musi to być informacja w pełni opisująca stan globalny, potrzebne są tylko dane niezbędne do prawidłowego wyboru reakcji. Przykład dotyczący równoważenia obciążenia przybierze teraz postać następującą: $\varphi_1(i)$ = „czy P_i jest przeciążony?“; *info* = „liczba przeciążonych i lista niedociążonych procesów“; *reakcja* = „wybierz odbiorców z otrzymanej listy, wylicz liczbę zadań do przekazania i przekaż je“. $\varphi_2(i)$ = „czy P_i jest niedociążony?“; *info* = „lista przeciążonych procesów“; *reakcja* = „odbierz zadania od procesów z otrzymanej listy“. W przykładzie dotyczącym algorytmu branch-and-bound wystarczy przekazywać *info* = „nowe najlepsze globalne rozwiązanie“.

Informowanie procesu o stanie, który wywołał reakcję, ma jeszcze inną istotną funkcję. W rozważanych systemach przesłanie komunikatu do monitora stanów globalnych (i spowrotem) nie jest natychmiastowe, zatem rozpoznanie stanu globalnego, ewaluacja predykatu oraz wywołanie reakcji w procesie, wymagają pewnego czasu. W tym czasie stan globalny aplikacji może zmienić się znacząco. W stanie bieżącym reakcja wywołana przez stan historyczny może okazać się błędna. Dzięki załączonej informacji, proces może w pewnym zakresie ocenić, czy wywołanie reakcji nie następuje zbyt późno i czy nie powinno być zaniechane. Wrócmy jeszcze raz do przykładu opartego na algorytmie branch-and-bound. Proces P_i przeszukuje podobszar o_k, dla którego oszacowanie znajdujących się tam rezultatów wynosi δ_k. W tym czasie P_j znajduje nowe rozwiązanie r lepsze od δ_k. Predykat $\varphi(i)$ = „czy P_i przeszukuje podobszar, dla którego oszacowanie jest gorsze niż aktualne najlepsze rozwiązanie?“ jest spełniony, zatem P_i powinien porzucić właśnie przeszukiwany podobszar. Jednak, P_i dowiaduje się o tym, gdy zakończył już przeszukiwanie o_k i teraz przeszukuje o_{k+1}, przy czym δ_{k+1} jest lepsza niż r. Przerwanie przeszukiwania o_{k+1} byłoby błędem. Jeśli proces wie, że reakcja tyczy się stanu, w którym zajmował się on jeszcze podobszarem o_k, ma możliwość zablokowania reakcji. W przypadku silnie spójnych stanów globalnych, ich identyfikację łatwo zrealizować znakując je czasem ich wystąpienia. Każdy proces może pamiętać, od kiedy zajmuje się bieżącym zadaniem i może reagować, o ile stan powodujący reakcję pochodzi z okresu

pracy nad bieżącym, a nie wcześniejszym, zadaniem.

W punkcie tym ustaliliśmy, że predykaty globalne chcemy wartościować względem każdego procesu osobno, aby uzyskać zróżnicowanie reakcji procesów powodowanych spełnieniem predykatu globalnego. Ponieważ do podjęcia właściwych reakcji często niezbędne są dodatkowe informacje o okolicznościach, w jakich reakcje te wywołano (czyli o stanie globalnym), przy wywoływaniu reakcji procesom dostarcza się potrzebne dane. Dane te powinny zawierać m.i. identyfikator stanu globalnego wywołującego reakcję.

Przedstawione powyżej badania zostały opublikowane w pracy [17].

3.2 Synchronizatory jako strukturalne elementy sterowania w programach

Realizacja sterowania w programach równoległych w oparciu o predykaty globalne, wymaga obserwacji stanów globalnych aplikacji, ewaluacji zdefiniowanych predykatów globalnych na tych stanach, oraz wywoływania reakcji procesów na spełnienie predykatu. W systemie musi istnieć proces (procesy) odpowiedzialny za te zadania. Samo monitorowanie wykonania aplikacji, czyli konstrukcja spójnych stanów globalnych oraz ewaluacja predykatów jest realizowana przez proces zwany zwykle monitorem. Jednak sterowanie wiąże się z aktywnym działaniem, którego nie wykazują monitory. Dlatego zamiast terminu monitor, będziemy używać nazwy *synchronizator*, określając w ten sposób proces obserwujący stany globalne aplikacji, podejmujący decyzje sterujące na podstawie predykatów globalnych ewaluowanych na tych stanach i wysyłający sygnały sterujące do procesów.

Synchronizatory można łączyć z procesami aplikacyjnymi, dodając do każdego procesu specjalizowany moduł. Taka metoda opisana jest w [88], lecz opiera się ona na użyciu dedykowanych synchronizatorów i algorytmów, potrafiących wykrywać tylko jeden z góry określony predykat globalny. Stwarza to duże utrudnienia i ogranicza zastosowania. Efektywna realizacja uniwersalnych synchronizatorów zintegrowanych z procesami nie jest w tym rozwiązaniu możliwa, głównie z uwagi na koszt propagacji informacji o stanach procesów. Dla przykładu: w systemie Meta [86] stosować można dowolny predykat, ale dobrą efektywność uzyskuje się, gdy do wartościowania predykatu jest użyta lokalnie dostępna informacja i gdy powodowane spełnieniem predykatu reakcje są lokalne. Zastosowanie prawdziwie globalnych predykatów, powodujących reakcje wszystkich (lub wielu) procesów, powoduje tam duże obciążenie, głównie ze względu na koszty komunikacji.

W rozwiązaniu proponowanym w rozprawie synchronizator jest procesem logicznie odrębnym od procesów aplikacyjnych. Obserwuje on spójne stany aplikacji i może na nich wartościować dowolne zaprogramowane predykaty. Dla synchronizatora Z, wśród wszystkich procesów aplikacyjnych $P_1, \ldots, P_N$, wyróżniamy procesy, których stan Z bierze pod uwagę, oraz procesy, na które Z może wpływać. Pierwszy zbiór oznaczamy przez Z_{in}, drugi przez Z_{out}. W najprostszym przypadku $Z_{in} = Z_{out} = \{P_1, \ldots, P_N\}$. Z każdym monitorowanym aspektem działania aplikacji i związanym z nim sterowaniem możemy związać osobny synchronizator Z i odpowiednio dobrać zbiory Z_{in} i Z_{out}. Dążymy do tego, aby całość sterowania odpowiedzialna za współpracę procesów była realizowana jako reakcje na spełnienie odpowiednich predykatów globalnych, wartościowanych przez synchronizatory. Rozdzielamy w ten sposób sekwencyjne sterowanie w ramach pojedynczego procesu od sterowania na poziomie równoległej aplikacji. Celem jest umożliwienie zmiany każdego z obydwu rodzajów (poziomów) sterowania niezależnie - sterowanie lokalne zmieniamy przez modyfikację kodu procesu, sterowanie na poziomie aplikacji przez modyfikację predykatów globalnych w synchronizatorach i reakcji wywoływanych spełnieniem predykatu.

Synchronizatory można łączyć ze sobą w hierarchię drzewiastą. Wówczas, synchronizatory na niższym poziomie powiadamiają synchronizatory nadrzędne o rezultatach wartościowania swych predykatów. Na podstawie tych informacji synchronizatory nadrzędne mogą wartościować swoje predykaty, czyniąc je w ten sposób w pewnym sensie predykatami wyższego rzędu. Decyzje sterujące mogą być wypracowywane na dowolnym poziomie hierarchii i są przekazywane na niższe poziomy hierarchii. Synchronizator powiadomiony o decyzji sterującej synchronizatora nadrzęd-

nego obsługuje tę decyzję, w rezultacie czego powiadamia wybrane synchronizatory niższego poziomu (lub już procesy). Taki schemat umożliwia strukturalizację i skalowalność mechanizmu monitorowania i sterowania aplikacją. Szczególnie dobre efekty stosowania hierarchii synchronizatorów otrzymujemy przy zastosowaniu modalności *Instantly* do ewaluacji predykatów globalnych. Hierarchiczny sposób konstrukcji globalnych stanów silnie spójnych oraz hierarchiczne metody wykrywania spełnienia predykatów globalnych opisaliśmy w punktach odpowiednio 2.5 i 2.6.

3.3 Sposoby reakcji procesów aplikacyjnych na spełnienie predykatu sterującego synchronizatora

Nasze dotychczasowe rozważania pomijały kwestię sposobu reakcji procesów na spełnienie predykatu. Poprzestawaliśmy na ogólnym stwierdzeniu, że synchronizator komunikuje procesom fakt spełnienia określonego predykatu, a procesy reagują na to. Teraz opiszemy mechanizmy realizujące te reakcje. Na poziomie procesu trzeba wyspecyfikować jak proces zareaguje, gdy otrzyma informację o spełnieniu predykatu globalnego, oraz jak i kiedy informację taką może odebrać. Przedstawimy szereg propozycji rozwiązań, dzieląc je na dwie zasadnicze grupy:

Synchroniczne - reakcja na spełnienie predykatu jest możliwa dopiero, gdy sterowanie w procesie dotrze do odpowiedniej instrukcji testującej predykat.

Asynchroniczne - reakcja na spełnienie predykatu jest uruchamiana od razu po otrzymaniu informacji o spełnieniu predykatu, niezależnie od aktualnie wykonywanych obliczeń.

3.3.1 Synchroniczne globalne instrukcje sterujące

Synchroniczne globalne instrukcje sterujące są częścią sekwencyjnego kodu procesu, są wykonywane gdy sterowanie w procesie do nich dotrze. W tym sensie instrukcje te nie różnią się od zwykłych instrukcji sterujących, takich jak *if* czy *while*. Dobrą ilustracją jest tu pomysł omówiony w [116], a wspomniany już w punkcie 1.3.1. Globalne instrukcje sterujące, np. *if* wykonywany jednocześnie w wielu procesach, badają warunki oparte o globalnie replikowane zmienne. Instrukcja taka musi poczekać, aż dostępne będą aktualne kopie wszystkich używanych zmiennych replikowanych. To postępowanie dzieli program na synchronicznie wykonywane fazy, odpowiadające: wymianie danych zawierających wartości replikowanych zmiennych, sprawdzeniu warunku i wykonaniu warunkowego kodu. Możliwa jest modyfikacja przedstawionego pomysłu, w której globalne instrukcje sterujące testowałyby prawdziwość jawnie określonych predykatów globalnych. Przez synchroniczne globalne instrukcje sterujące rozumiemy właśnie instrukcje typu $if(\varphi)$ czy $while(\varphi)$, stosowane w zwykły sposób w kodzie procesów, gdzie φ jest predykatem globalnym. Tak określone sterowanie składa się z czterech warstw, są to:

1. sposób wyrażania stanów procesów,
2. definicje użytych predykatów globalnych,
3. modalność predykatów określająca, dla których stanów globalnych są wyliczane i brane pod uwagę wartości predykatów,
4. kod procesów, w którym predykatów używają globalne instrukcje sterujące.

Stan procesów proponujemy utożsamiać z wartościami wytypowanych zmiennych. Na tych zmiennych programista definiuje predykaty globalne, o formie właściwej dla konkretnej aplikacji, a następnie pisze kod poszczególnych procesów. Istotny jest wybór stanu globalnego, na którym należy wartościować użyty predykat globalny φ. Przedyskutujemy kilka takich możliwości.

Pełna synchronizacja. W najprostszym rozwiązaniu, globalną instrukcję sterującą wykonać muszą wszystkie procesy i instrukcja ta działa jako bariera. Dopiero, gdy wszystkie procesy

ją osiągną, w sposób deterministyczny określa się stan globalny aplikacji i na jego podstawie ewaluuje predykat użyty w instrukcji sterującej. Takie postępowanie przypomina sposób wartościowania predykatów globalnych w modalności *Currently*, patrz punkt 1.3.

Podział na fazy. Aby procesy wcześniej dochodzące do globalnej instrukcji sterującej nie musiały zawsze czekać bezczynnie, można ustalić, że predykaty użyte w globalnych instrukcjach sterujących będą wartościowane na określonych historycznych stanach, znanych odpowiednio wcześnie. Możnaby tu adaptować model BSP (patrz punkt 1.1) i w danej fazie obliczeń brać pod uwagę stan globalny osiągnięty przez aplikację na koniec fazy poprzedniej. Blisko tego rozwiązania mieści się propozycja z pracy [116], gdzie replikowana (globalna) zmienna może być czytana dopiero po tym, gdy zakończy się proces, który nadał jej nową wartość.

Migawka. Proces wykonujący globalną instrukcję sterującą może próbować ustalić stan globalny aplikacji stosując algorytm migawki (ang. snapshot) [71]. Musiałby wtedy zaczekać, aż stan zostanie określony, ponadto niezależne określanie stanu globalnego przez wiele procesów powodowałoby spory narzut. Zauważmy, że stan globalny otrzymany przy użyciu algorytmu migawki nie jest deterministycznie określony, co znacznie utrudnia jego interpretację i wykorzystanie.

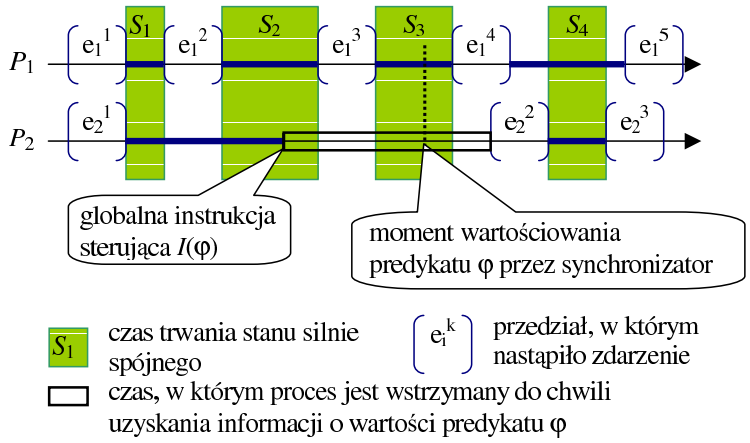
First, Recent, Time. Dany proces wykonujący globalną instrukcję sterującą ma dodatkowe możliwości wyboru stanu globalnego, na którym wartościouje predykat. Proces taki interesuje wartość predykatu w stanie globalnym zawierającym jego bieżący stan lokalny. Jednak takich stanów globalnych może być wiele. Proponujemy wyróżnić tu trzy możliwości:

First(φ, i, s_i) Wartościowanie predykatu φ przy użyciu najwcześniejszego globalnego stanu spójnego, zawierającego bieżący stan lokalny s_i procesu P_i, wykonującego globalną instrukcję sterującą $I(\varphi)$. Stan taki jest jednoznacznie określony, gdy aplikacja jest monitorowana przy użyciu globalnych stanów silnie spójnych. Na rysunku 3.1 widzimy przykład posługujący się właśnie silnie spójnymi stanami globalnymi. W przykładzie tym wartościowanie predykatu φ przy użyciu modalności First$(\varphi, 2, s_2)$ nastąpiłoby na stanie S_1.

Recent(φ, i, s_i) Wartościowanie predykatu φ przy użyciu najpóźniejszego znanego globalnego stanu spójnego, zawierającego bieżący stan lokalny s_i procesu P_i, wykonującego globalną instrukcję sterującą $I(\varphi)$. Stan taki jest jednoznacznie określony, gdy aplikacja jest monitorowana przy użyciu globalnych stanów silnie spójnych. Modalność ta oznacza użycie możliwie najbardziej „bieżącego" stanu globalnego, a dokładniej - ostatniego znanego stanu globalnego. Może to być stan globalny, który w rzeczywistości rozpoczął się już po dojściu P_i do instrukcji $I(\varphi)$. Jednak stan taki zawiera najbardziej aktualne informacje o działaniu aplikacji. W przykładzie z rysunku 3.1 wartościowanie predykatu φ przy użyciu modalności Recent$(\varphi, 2, s_2)$ nastąpiłoby przy użyciu stanu S_2 (gdy wartościujący proces jeszcze nic nie wie o zdarzeniu e_1^3), lub S_3 (gdy już wie).

Time$(\varphi, zr(I))$ Wartościowanie predykatu φ przy użyciu globalnego stanu spójnego zawierającego moment dotarcia danego procesu do globalnej instrukcji sterującej $I(\varphi)$, $zr(I)$ oznacza wartość znacznika czasu przypisanego do momentu dotarcia procesu do instrukcji I. Dla tej modalności wygodne jest użycie znaczników czasu rzeczywistego, gdyż pozwalają one osobno oznaczyć określony moment w czasie trwania jednego stanu lokalnego. W przykładzie z rysunku 3.1 wartościowanie predykatu φ przy użyciu modalności Time$(\varphi, zr(I))$ nastąpiłoby na stanie S_2. Gdy spójny stan globalny odpowiadający podanej definicji nie daje się określić, używamy pierwszego globalnego stanu spójnego następującego po dotarciu do $I(\varphi)$.

Predykaty globalne mogą być wartościowane przez zainteresowane procesy. Ma to dwie wady: a) procesy muszą poznać stan globalny, co znacznie podwyższa koszt komunikacji, oraz b) wartościowanie predykatu może być niepotrzebnie wykonywane wielokrotnie, osobno w każdym procesie. Alternatywnie, miejscem wartościowania predykatów może być synchronizator. Wówczas, proces



Rysunek 3.1: Wartościowanie predykatu globalnego na potrzeby globalnej instrukcji sterującej

wykonujący globalną instrukcję sterującą jedynie pyta synchronizator o wartość predykatu w stanie globalnym odpowiadającym bieżącemu stanowi lokalnemu tego procesu. Oczywiście, i tu trzeba ustalić modalność stosowaną przez synchronizator do wartościowania predykatów.

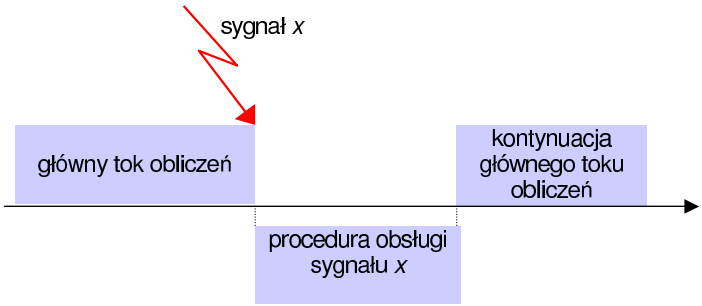
Dodatkowej uwagi wymaga kwestia realizacji oczekiwania przez procesy na spełnienie danego predykatu. Klasyczne instrukcje sterujące (if, while, ...) nie dają możliwości wstrzymania procesu, przy ich pomocy można zaprogramować tylko czekanie aktywne. Proponujemy zatem wprowadzenie specjalnej instrukcji sterującej $wait(\varphi)$, powodującej zawieszenie wykonywania procesu do czasu spełnienia predykatu globalnego φ.

3.3.2 Asynchroniczne powiadamianie i reakcje

Wyodrębnienie synchronizatora jako osobnego procesu stale nadzorującego działanie aplikacji równoległej, umożliwia kontrolę nad aplikacją w sposób zbliżony do sposobu stosowanego w systemach reaktywnych (patrz punkt **??**). Synchronizator, na podstawie obserwowanych wartości predykatów globalnych, może generować sygnały sterujące dla procesów, a procesy mogą reagować na otrzymane sygnały. W przeciwieństwie do omówionych wyżej synchronicznych instrukcji sterujących, zakładamy teraz, że reakcje na spełnienie predykatu są wyzwalane w sposób asynchroniczny względem głównych obliczeń realizowanych przez procesy. Nie wymagamy, aby w kodzie procesu trzeba było umieszczać instrukcje testowania wartości predykatu globalnego lub odbioru sygnału sterującego powiadamiającego o spełnieniu predykatu od synchronizatora. Zakładamy, że przybycie sygnału sterującego automatycznie uruchomi wskazaną procedurę, być może powodując chwilowe wstrzymanie głównego biegu wykonywanych obliczeń. Rysunek 3.2 ilustruje przedstawioną idee. Brak jawnych żądań ewaluacji predykatu globalnego pozwala uniknąć, związanego z takimi żądaniami, oczekiwania na wynik. Gdy konieczne jest stałe (częste) sprawdzanie pewnego warunku globalnego (np. w pętli), każdorazowe oczekiwanie na wynik powodowałoby znaczną redukcję wydajności realizowanych obliczeń. Jedną z głównych zalet stosowania asynchronicznych reakcji jest pozbycie się tego problemu.

Sterowanie z użyciem asynchronicznych reakcji składa się z sześciu elementów:

1. sposób wyrażania stanów procesów,
2. definicje użytych predykatów globalnych,
3. modalność predykatów określająca, dla których stanów globalnych są wyliczane i brane pod uwagę wartości predykatów,
4. sygnały sterujące niosące informacje o spełnieniu predykatu,



Rysunek 3.2: Asynchroniczna reakcja na sygnał wygenerowany na podstawie obserwowanych wartości predykatu globalnego

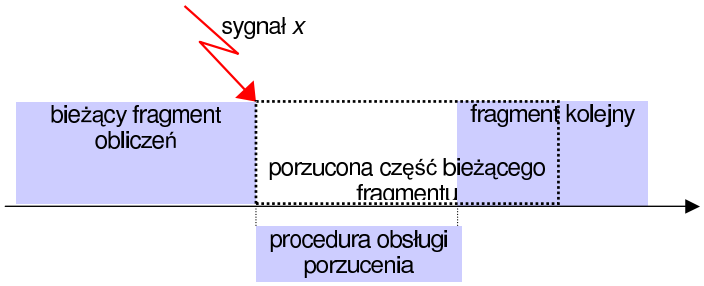
5. działania aktywowane przez sygnały,

6. reguły przyjmowania sygnałów przez procesy.

Stan procesów można utożsamiać z wartościami wytypowanych zmiennych. Na tych zmiennych programista definiuje predykaty globalne, o formie właściwej dla konkretnej aplikacji. Zgodnie z wnioskami z punktu 3.1.1 planujemy stosować modalność *Instantly*. Procesy mają dowiadywać się o spełnieniu predykatu za pośrednictwem sygnałów sterujących, trzeba zatem ustalić listę rodzajów takich sygnałów, oraz ich związek ze spełnieniem poszczególnych predykatów (związek 1-1 jest najprostszy). Sygnały muszą zostać jawnie wysłane do procesów, aby wywołać reakcje. Reakcje te mają mieć postać asynchronicznego uruchomienia powiązanych z danym sygnałem procedur. Programista obowiązany jest do utworzenia tych procedur. Zauważmy, że procedury te, dedykowane do realizacji sterowania i synchronizacji, są oddzielone od głównego kodu procesów, co jest korzystne z punktu widzenia strukturalizacji i łatwości modyfikacji aplikacji. Ostatni wymieniony element, czyli reguły przyjmowania sygnałów przez procesy, jest przydatna do ograniczenia pełnej asynchroniczności omawianych reakcji, integruje ona mechanizm asynchronicznego uruchamiania procedur z przebiegiem głównych obliczeń w procesie. Jej zadaniem jest umożliwienie zignorowania lub czasowego wstrzymania przyjmowania wskazanych sygnałów. Przewidujemy bowiem, że w pewnych sytuacjach sygnały sterujące nie powinny przerywać bieżących obliczeń, czy to ze względów technicznych, czy też dlatego, że tak może uznać proces otrzymujący sygnał, dysponujący nowszymi informacjami o swym stanie, niż synchronizator w momencie, gdy dany sygnał generował. Element ten może mieć postać specjalnych instrukcji, umieszczanych przez programistę w głównym kodzie procesów.

Jako uzupełnienie opisanego modelu proponujemy wprowadzić przeciwieństwo pojęcia aktywacji przez sygnał. Jest to pojęcie porzucenia bieżących obliczeń. Przybycie sygnału ma powodować przerwanie wykonywania bieżącego fragmentu kodu i skok do innego fragmentu. Wyniki „niedokończonych“ obliczeń są tracone. Procedury aktywowane przez sygnały uzyskują tu postać procedur obsługi porzucenia fragmentu obliczeń, a warstwa reguł aktywacji procedur odpowiada za definicję tychże fragmentów. Sens takiej operacji widać na przykładzie aplikacji zawierającej równoległe przeszukiwanie przestrzeni rozwiązań. Gdy jeden proces znajdzie dobre rozwiązanie, pozostałe mogą natychmiast przerwać swoje poszukiwania i przejść do kolejnego etapu aplikacji. Zasoby sytemowe używane przez procedurę przeszukiwania są niezwłocznie zwalniane i mogą zostać od razu wykorzystane do realizacji dalszych zadań. Ilustrację porzucenia obliczeń widzimy na rysunku 3.3.

W konkluzji punktu 3.1.2 ustaliliśmy, że powinna istnieć możliwość sygnalizowania spełnienia predykatu globalnego osobno każdemu procesowi oraz, że informacja o spełnieniu predykatu powinna być wzbogacona informacją o stanie globalnym, w których predykat został spełniony. Proponowana struktura sterowania z użyciem asynchronicznych reakcji może doskonale spełnić wymienione postulaty. Procedura wyliczania wartości predykatu może zawierać kod selektywnie wysyłający sygnały do wybranych procesów. Sygnały mogą przenosić dane (podobnie jak zwykłe komunikaty) i dane te mogą opisywać stan globalny w którym spełnieniony został dany predykat.



Rysunek 3.3: Porzucenie obliczeń na skutek odebrania sygnału sterującego

Proponujemy, aby dane przekazywane wraz z sygnałami były udostępniane procedurom aktywowanym przez sygnały jako parametry wywołania.

3.3.2.1 Reguły przyjmowania sygnałów przez procesy

Zdecydowaliśmy się dokładnie przebadać oraz zaimplementować drugi z przedstawionych powyżej sposobów reakcji procesów na spełnienie predykatu, czyli asynchroniczne powiadamiania i reakcje. Za tą decyzją przemawiały przede wszystkim trzy następujące motywy.

- Eliminacja oczekiwania na decyzję sterującą potencjalnie umożliwia polepszenie efektywności działania aplikacji.
- Oddzielna specyfikacja głównego kodu obliczeniowego programu od kodu odpowiedzialnego za sterowanie (za reakcje na spełnienie predykatów sterujących) pozwala na daleko idącą strukturalizację programu równoległego, a co za tym idzie ułatwia jego tworzenie i modyfikacje.
- Wbudowanie proponowanego mechanizmu sterowania w system P-GRADE (patrz rozdział 5), dzięki asynchronicznym reakcjom, znakomicie uzupełni dostępny w P-GRADE model komunikacji. Model ten nie zawiera instrukcji nieblokującego (warunkowego) odbioru komunikatu ani instrukcji testu (czy dany komunikat już przybył), przez co implementacja nieregularnych aplikacji w P-GRADE jest trudna i mało efektywna.

Eksperymenty praktycznego zastosowania sterowania opartego o predykaty globalne z wykorzystaniem asynchronicznego powiadamiania pokazały, że warstwa reguł przyjmowania sygnałów odgrywa dużą rolę. Istnieje wiele możliwości co do sposobu działania tej warstwy. Oto kilka z pytań, na które musieliśmy sobie odpowiedzieć.

- Czy aktywowane obliczenia powinny działać współbieżnie z głównym wątkiem obliczeń, czy (jak na rysunku 3.2) jako korutyny?
- Co robić, gdy w trakcie działania aktywowanego kodu przychodzi nowy sygnał sterujący?
- Co robić, gdy sygnał sterujący przybywa w momencie, w którym aktywacja procedury jego obsługi jest technicznie niemożliwa?
- Sygnał może przybyć z dużym opóźnieniem w stosunku do wystąpienia stanu globalnego powodującego wygenerowania tego sygnału. Co robić, gdy wielkość tego opóźnienia ma znaczenie dla poprawności sterowania?

Postanowiliśmy ustalić możliwie proste reguły jak najmniej komplikujące sposób działania procesu przyjmującego sygnały.

- Aktywowanie procedury zawiesza bieg głównych obliczeń, aby nie wprowadzać współbieżności wewnątrz pojedynczego procesu.

- Na raz może działać tylko jedna aktywowana sygnałem procedura obsługi sygnału. Sygnały przychodzące przez zakończeniem poprzednio aktywowanej procedury są kolejkowane. Unikamy w ten sposób przerywania i porzucania procedur aktywowanych sygnałami i związanych z tym problemów semantycznych oraz kwestii technicznych.

- Wprowadzamy możliwość określenia fragmentu kodu jako nieprzerywalnego przez sygnały, gdy z powodów technicznych nie powinien on być przerwany lub porzucony.

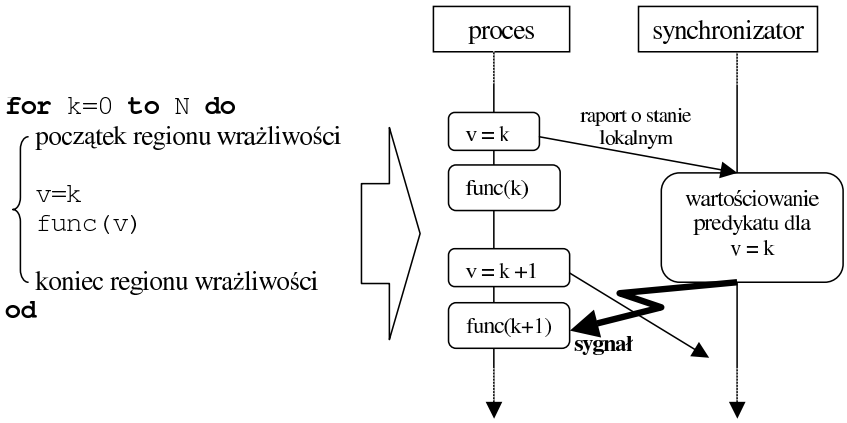
- Po zakończeniu wykonania aktywowanej procedury zawieszone obliczenia mogą zostać wznowione lub porzucone. W przypadku porzucenia program jest wznawiany za końcem bieżącego regionu wrażliwości na sygnał, zdefiniowanego poniżej.

- Proces określa sam, na jak bardzo spóźniony sygnał ma reagować. Określenia tego dokonuje się oznaczając fragment kodu jako *region wrażliwości na sygnał*. Proces reaguje na sygnały jedynie, gdy sterowanie procesu znajduje się w takim regionie, po opuszczeniu regionu przychodzące sygnały są ignorowane. Rysunek 3.4 ilustruje dodatkową trudność pojawiającą się, gdy regiony wrażliwości umieszczone są w pętli. Synchronizator dokonał wartościowania predykatu globalnego w stanie globalnym zawierającego stan lokalny procesu, odpowiadający wykonywaniu iteracji k-tej, i na tej podstawie wysłał sygnał sterujący. Sygnał przybył, gdy proces znajdował się już w kolejnym stanie, wykonując k+1-wszą iterację pętli. Łatwo wyobrazić sobie, że reakcja na sygnał dotyczący poprzedniej iteracji mogłaby być niewskazana. Np. sygnał mógł oznaczać „porzuć obliczanie func(k) (ale nie func(k+1))". Podobne trudności w systemie Dagger [57] zostały pokonane przez dodanie do każdego wysłanego komunikatu zarządzanego jawnie przez programistę licznika i sprawdzanie przy odbiorze komunikatu wartości dołączonego licznika. Nasze rozwiązanie tego problemu będzie nie będzie wymagało wkładu pracy programisty, oprzemy je na pojęciu *wykonania regionu wrażliwości*. Region, podobnie jak dowolna instrukcja, jest wykonywany, gdy sterowanie procesu dojdzie do niego. Jeśli region wrażliwości umieszczony jest w pętli, to jest wykonywany wielokrotnie. Proponujemy następujące uściślenie podanego wyżej warunku uruchamiania reakcji na sygnał:

  Proces reaguje na dany sygnał jedynie wtedy, gdy sterowanie procesu znajduje się w regionie wrażliwości na sygnał oraz przybyły sygnał został wygenerowany na podstawie stanu globalnego następującego już po rozpoczęciu bieżącego wykonania regionu wrażliwości.

Bardziej formalnie: oznaczmy przez e_k rozpoczęcie bieżącego wykonania regionu wrażliwości przez proces P_k, zdarzenie to nie wpływa na brany pod uwagę stan lokalny procesu. Proces ten otrzymuje sygnał wygenerowany na podstawie stanu globalnego $S =< s_1, .., s_k, .., s_N >$. Reakcja nastąpi jedynie, gdy sterowanie procesu nadal znajduje się wewnątrz regionu wrażliwości i $e_k \rightarrow \underline{s_k}$ (e_k poprzedza $\underline{s_k}$) lub $e_k = \underline{s_k}$. Przy postulowanym zastosowaniu silnie spójnych stanów globalnych i relacji następstwa zdarzeń opartej o znaczniki czasu rzeczywistego, warunek ten przybiera łatwą do sprawdzenia uproszczoną postać: $zr(e_k) < zr(\underline{S})$. Jeśli uruchomiona procedura reakcji na sygnał zostanie zakończona poleceniem porzucenia obliczeń, wówczas program będzie kontynuowany od punktu kończącego bieżący region wrażliwości. Obszerniejszy opis semantyki reakcji procesu na sygnały przedstawiony z zastosowaniem Kolorowanych Sieci Petriego (CPN) można znaleźć w [16].

Należy zauważyć, że tak określony mechanizm sterowania jest niedeterministyczny. Czas jaki upływa od wystąpienia stanu globalnego do próby realizacji decyzji nim spowodowanej zależy od czasu przesłania informacji o stanach lokalnych, czasu detekcji stanu globalnego i czasu propagacji decyzji. Choć możliwe jest otrzymanie górnego ograniczenia tych czasów (patrz punkty 2.1.2 i 2.4.1), to ich faktyczne wartości mogą być inne przy każdym wykonaniu programu. Niedeterminizm nie jest nowym zjawiskiem w programach rozproszonych, gdyż niedeterministyczne opóźnienia w komunikacji są obecne też w klasycznych modelach sterowania. W naszej propozycji stwierdzamy jawnie, już na poziomie modelu, że pewne decyzje nie będą realizowane, o ile przesłanki do ich podjęcia będą znane zbyt późno. W naszej koncepcji do określenia czasu trwania regionu wrażliwości nie używamy zegara podając warunki typu „czekaj na potencjalną decyzję przez 200 ms".



Rysunek 3.4: Region wrażliwości na sygnał umieszczony w pętli

Czas trwania regionu wrażliwości ustalany jest syntaktycznie na poziomie kodu procesu, gdzie oznacza się jego początek i koniec. Wiąże to omawiane regiony ze strukturą programu i pozwala na konstrukcje typu „czekaj na potencjalną decyzję do końca bieżącej iteracji pętli". Dyskutowany niedeterminizm postrzegamy nie jako wadę, lecz jako odzwierciedlenie rzeczywistych cech rozproszonego systemu komputerowego w modelu sterowania programów działających na takim sprzęcie. Odzwierciedlenia tego należy używać, jeśli jest ono przydatne w sformułowaniu rozwiązania danego problemu. Jeśli nie jest przydatne, wówczas wystarczy zawrzeć cały program w regionie wrażliwości, aby uzyskać gwarancję, że żadne reakcje nie zostaną pominięte.

Opisane w tym punkcie zagadnienia są tematem publikacji [14, 15].

3.4 Modelowanie sterowania opartego na predykatach globalnych

W niniejszej pracy rozważamy sterowanie działaniem aplikacji w oparciu o predykaty globalne. Przez sterowanie aplikacją rozumiemy sterowanie poszczególnymi procesami, nie stosujemy ani nie będziemy modelować sterowania programem równoległym jako nierozłączną całością. Sterowanie w każdym procesie z osobna, realizowane w oparciu o współdzieloną informację, jaką są stany globalne i predykaty na nich określone, tworzy sterowanie całością aplikacji równoległej. Najprostszy model to przedstawiający wygląda następująco:

Proces P_k znajdujący się w stanie lokalnym s_k^i, dla którego $\underline{s_k^i} = I(\varphi)$ (stan kończy się globalną instrukcją sterującą), przechodzi do kolejnego stanu, wybierając ten kolejny stan z dwóch możliwych na podstawie wartości predykatu globalnego φ określonego na stanie globalnym $S =< s_1, .., s_k^i, .., s_N >$:

$$next(s_k^i, S) = \begin{cases} s_k^{i+1} & \text{jeśli } \varphi(S), \\ , & \text{jeśli } \neg\varphi(S). \end{cases}$$

Model ten odzwierciedla reguły sterowania przedstawione w punkcie 3.3.1 (za wyjątkiem reguły „podział na fazy", patrz 3.3.1). Wybrana jedna z opisanych tam modalności określa, na którym z potencjalnie alternatywnych stanów globalnych S będzie wartościowany predykat.

Sterowanie według reguły „podział na fazy" (patrz 3.3.1) można przedstawić przy pomocy modelu jak wyżej, lecz wartościując predykat na stanie $S =< s_1, .., s_k^{i-1}, .., s_N >$, czyli na stanie zawierającym stan P_k z poprzedniej fazy obliczeń. Reguły przejścia aplikacji między fazami muszą zagwarantować, że w S stany lokalne innych procesów również pochodzą z poprzedniej fazy, czyli że cały stan globalny S pochodzi z tejże poprzedniej fazy.

Sterowanie z zastosowaniem asynchronicznych reakcji jest bardziej skomplikowane. Wystąpienie stanu globalnego powodującego spełnienie predykatu sterującego oraz reakcja procesu na

to spełnienie są oddalone w czasie. Ponadto chcielibyśmy uwzględnić mechanizm ograniczający opóźnienie, z jakim proces może zareagować na spełnienie predykatu globalnego (czemu służą zaproponowane w punkcie 3.3.2.1 regiony wrażliwości). Opis stanu lokalnego procesu będzie teraz miał następującą postać: $< s, r, p >$, gdzie $r = 1$ gdy proces jest gotowy do reakcji na sygnał (sterowanie procesu znajduje się w regionie wrażliwości), 0 wpp., $p > 0$ gdy spełniony został predykat globalny, lecz proces jeszcze nie zareagował na to, 0 wpp., s reprezentuje stan lokalny procesu zawierający pozostałe istotne aspekty. Stan procesu zmienia się według reguł:

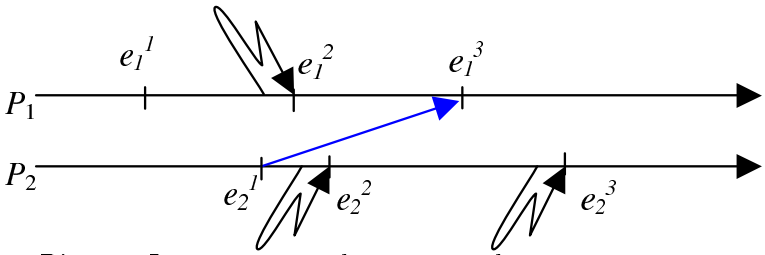
1. Przy przekroczeniu limitu oczekiwania na reakcję na spełnienie predykatu
   - $next(< s^i, 1, * >, S) =< s^{i+1}, 0, 0 >$ gdy $\overline{s^i}$ jest zdarzeniem zakończenia oczekiwania na reakcję (jest wyjściem z regionu wrażliwości)
2. Spełnieniu predykatu globalnego jest zapamiętywane, o ile nastąpiło w odpowiednim momencie
   - $next(< s^i, 0, x >, S) =< s^{i+1}, 0, x >$
   - $next(< s^i, 1, x >, S) =< s^{i+1}, 1, x + 1 >$ gdy $\varphi(S)$
   - $next(< s^i, 1, x >, S) =< s^{i+1}, 1, x >$ gdy $\neg\varphi(S)$
3. Reakcja na spełnienie predykatu może nastąpić w każdym momencie, dopóki proces jeszcze na nią oczekuje:
   - $next(< s^i, 1, 0 >, S) =< s^{i+1}, 1, 0 >$
   - dla $x > 0$: $next(< s^i, 1, x >, S) = \begin{cases} < s^{i+1}, 1, x > & \text{gdy reakcja nie nastąpiła} \\ < ', 0, x - 1 > & \text{gdy reakcja nastąpiła} \end{cases}$
4. Po wykonaniu danej reakcji możliwa jest realizacja dalszych oczekujących reakcji
   - $next(< s^i, 0, x >, S) =< s^{i+1}, 1, x >$ gdy $\overline{s^i}$ jest zdarzeniem kończącym daną reakcję (jest ostatnim zdarzeniem w procedurze reagowania na sygnał)

3.4.1 Reprezentacja graficzna

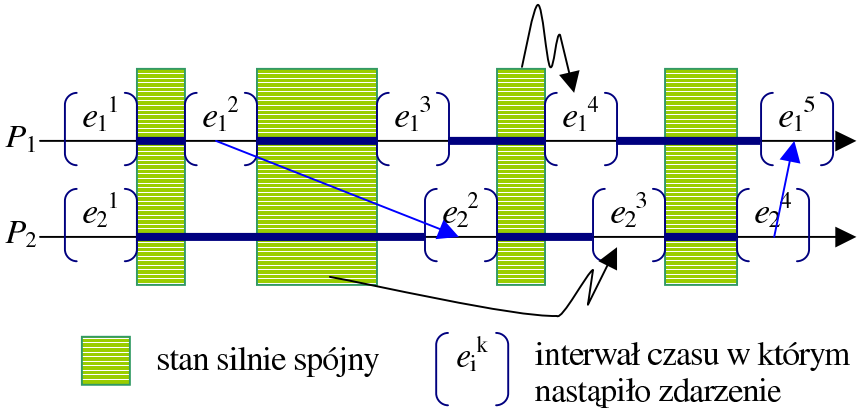
Nasze doświadczenie wskazuje, że niezwykle pomocne w zrozumieniu i interpretacji działania mechanizmu sterowania w programach równoległych jest posłużenie się schematami rysunkowymi. Standardowe diagramy Lamporta (np. rysunek 1.1) operują trzema rodzajami zdarzeń: odebranie i wysłanie komunikatu oraz zdarzenie wewnętrzne procesu. Diagramy te przedstawiają przebieg wykonania programu równoległego, lecz nie pokazują decyzji sterujących działaniem programu prowadzących do takiego właśnie przebiegu, ani czynników wpływających na te decyzje. Opracowaliśmy rozszerzenie diagramów Lamporta pozwalające na ilustrację działania mechanizmu sterującego wykonaniem programu równoległego opartego o predykaty określone na silnie spójnych stanach globalnych.

Proponujemy wprowadzić nowe zdarzenie na poziomie procesu - *decyzja sterująca*. Zdarzenie to wyznacza element w działaniu procesu, w którym wybrano jedną z kilku możliwych ścieżek wykonania, np. wykonano instrukcję `if` lub aktywowano procedurę obsługi. Uwidacznianie są tylko te decyzje, które mają istotne znaczenie z punktu widzenia całości programu równoległego. Do zdarzenia prowadzi strzałka typu błyskawica ukazująca na podstawie czego podjęto daną decyzję. W klasycznym modelu sterowania decyzje są podejmowane na podstawie wartości zmiennych lokalnych procesu, toteż decyzje sterujące stanowią tam jedynie pewien podzbiór zdarzeń wewnętrznych, co widzimy to na rysunku 3.5.

Nasz model sterowania zakłada, że podstawą do decyzji sterującej jest stan spójny aplikacji, a dokładniej spełnienie predykatu określonego na takim stanie. Diagram powinien pokazywać, który stan spójny spowodował którą decyzję. W idealnym przypadku stan spójny prowadzący do danej



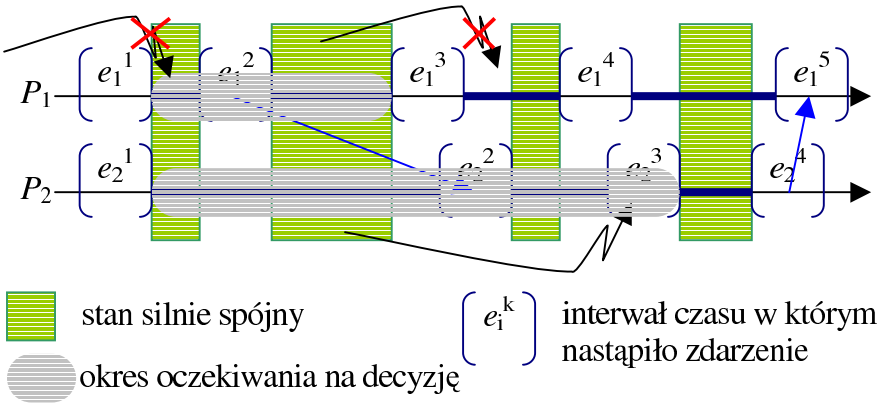
Rysunek 3.5: Diagram Lamporta z uwidocznionymi decyzjami sterującymi



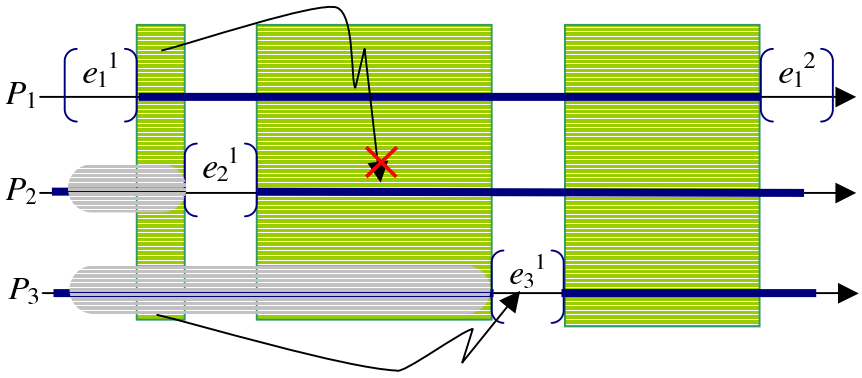
Rysunek 3.6: Decyzje sterujące podejmowane na podstawie stanów silnie spójnych

decyzji sterującej bezpośrednio ją poprzedza, tak jak dla decyzji e_1^4 na rysunku 3.6. Jednak w praktyce trzeba się liczyć z opóźnieniami w przesyłaniu informacji. Opóźnienia te spowalniają detekcję stanu spójnego i procesy mają możliwość poznania stanów aplikacji jedynie sprzed pewnego czasu. Zatem stosowne decyzje sterujące powodowane tymi stanami będą opóźnione, np. decyzja e_2^3 na rysunku 3.6. Dopuszczalne opóźnienie decyzji kontrolowne jest na poziomie procesu przez okresy oczekiwania, odpowiadające okresom, w których sterowanie procesu przebywa w regionie wrażliwości na sygnał sterujący. Pokazane jest to na rysunku 3.7. Decyzja e_2^3 znajduje się w tym samym okresie oczekiwania, co stan, na podstawie którego została podjęta, zatem jest realizowana. Decyzje w procesie P_1 nie są realizowane, bo pierwsza z nich została podjęta na podstawie stanu sprzed bieżącego okresu oczekiwania, a druga jest gotowa do realizacji gdy proces już na nią nie czeka.

Za pomocą wprowadzonej reprezentacji graficznej zilustrujemy teraz przebieg fragmentu obliczeń typu dziel i ograniczaj (and. branch and bound) przeprowadzonych przez trzy współbieżne



Rysunek 3.7: Okresy oczekiwania na decyzje sterujące



Rysunek 3.8: Reprezentacja fragmentu obliczeń dziel i ograniczaj

procesy. Procesy przeszukują przestrzeń potencjalnych rozwiązań, odrzucając te częściowe rozwiązania (podproblemy), które prowadzą do wyników gorszych niż znane już pełne rozwiązanie. Na rysunku 3.8 zdarzenie e_1^1 oznacza znalezienie nowego najlepszego globalnie rozwiązania przez proces P_1. W momencie wystąpienia tego zdarzenia procesy P_2 i P_3 pracowały nad rozwiązaniami częściowymi, prowadzącymi do wyników gorszych niż nowo znalezione rozwiązanie. Z tego powodu powzięte zostały decyzje sterujące nakazujące procesom P_2 i P_3 zaniechanie dalszego rozpatrywania bieżących podproblemów. Proces P_3 zrealizował decyzję sterującą, co reprezentuje zdarzenie e_3^1. Zdarzenie to oznacza zaprzestanie prac nad bieżącym rozwiązaniem częściowym i rozpoczęcie prac nad kolejnym podproblemem. Inaczej przedstawia się sytuacja w przypadku procesu P_2. Proces ten szybko zakończył rozpatrywać podproblem, którego dotyczyła powzięta decyzja sterująca, co reprezentuje zdarzenie e_2^1. Decyzja była gotowa do realizacji, gdy P_2 pracował już nad kolejnym podproblemem, dlatego została zignorowana.

Rozdział 4

Badania wydajności sterowania w programach wykorzystującego predykaty globalne

W poprzednich rozdziałach przedstawiliśmy podstawy teoretyczne stanów spójnych oraz mechanizm sterowania wykonaniem programów równoległych z niej korzystający. Efektywność tego mechanizmu wymaga weryfikacji. Ponieważ jest to nowa idea, nie sprawdzona w praktyce, nie wiemy, czy teoretycznie określone zasady funkcjonowania tego mechanizmu będą działać wydajnie w określonych środowiskach sprzętowo-systemowych. W tym rozdziale zajmujemy się tym problemem. Badania wydajnościowe przeprowadziliśmy w oparciu o symulację środowiska komputerowego. Takie podejście uniezależniło nas od dostępności sprzętu, pozwaliło modelowo sprawdzić rozmaite, istotnie różniące się konfiguracje oraz było wykonalne szybciej i znacznie niższych kosztem niż testy w rzeczywistych środowiskach. Opracowując symulator, przyjęliśmy założenia zgodne z wnioskami zaprezentowanymi w punktach 3.1.1, 3.1.2, 3.3.2.1:

- stany spójne aplikacji będą wyznaczane przez specjalnie do tego przeznaczone procesy, tzw. synchronizatory,
- wartościowanie predykatów globalnych odbywać się będzie w synchronizatorach,
- zastosujemy modalność *Instantly*, a zatem do wyznaczania stanów spójnych użyjemy częściowo zsynchronizowanych zegarów lokalnych,
- spełnienie predykatu spowoduje przesłanie sygnałów sterujących do wskazanych procesów, sygnały te powodują uruchomienie skojarzonych z nimi reakcji natychmiast po swym przybyciu, asynchronicznie względem głównego toku obliczeń.

4.1 Symulator wykonania programów

Do budowy symulatora wykorzystaliśmy specjalizowany pakiet oprogramowania do tworzenia symulacji typu Discreet Event Simulation o nazwie OMNeT++ [3]. Pakiet ten składa się z kilku zasadniczych elementów:

- biblioteka klas C++ tworzących infrastrukturę symulacji - sieć modułów wymieniających komunikaty,
- język opisu konfiguracji symulacji (NED), określa które moduły i jak połączone ze sobą mają tworzyć daną symulację, oraz definiuje parametry używane przez moduły,

- biblioteka klas pomocniczych (kolejki, tablice, generatory liczb pseudolosowych, statystyki, ..),
- graficzny interfejs użytkownika, umożliwia podgląd działania symulacji (istniejące komunikaty, wartości zmiennych, zawartości struktur danych itp), pełni rolę programu uruchomieniowego oraz pozwala na demonstracje,
- język opisu parametrów wykonania symulacji, pozwala określić wartości parametrów startowych symulacji oraz opisać szereg symulacji do wykonania jako jedno zadanie.

Zadaniem programisty tworzącego symulację jest implementacja modułów symulacji jako klas w C++, przy użyciu klas bibliotecznych. Moduły muszą implementować reakcje na nadejście komunikatów i same mogą wysyłać komunikaty. Komunikat jest obiektem klasy pochodnej od bibliotecznej klasy cMessage. Kod klas oraz stosowny opis połączeń modułów w języku NED są kompilowane i łączone z wybranym interfejsem użytkownika - graficznym (do demonstracji i uruchamiania) lub tekstowym.

Symulator zawiera trzy klasy modułów: proces aplikacyjny, synchronizator i sieć, które są opisane poniżej.

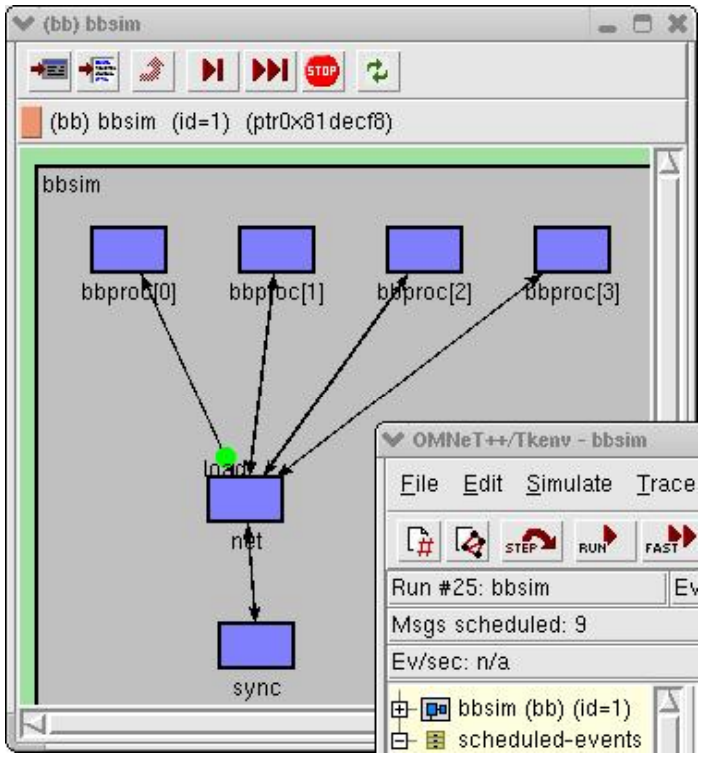
Proces aplikacyjny Procesy aplikacyjne, tworzone w liczbie zadanej parametrem, są implementowane osobna dla każdej symulowanej aplikacji. Zawierają kod odpowiedzialny za symulację stosowanych obliczeń, raportują swoje stany lokalne, odbierają sygnały sterujące od synchronizatorów, mogą wymieniać miedzy sobą komunikaty z danymi.

Synchronizator Odbiera raporty o stanach lokalnych od procesów aplikacyjnych, na ich podstawie buduje stany silnie spójne oraz wylicza predykaty sterujące na zaobserwowanych stanach spójnych. W wyniku tych wyliczeń synchronizator może wysłac procesom sygnały sterujące. Synchronizatory można łączyć w hierarchie według reguł opisanych w punkcie 2.5 oraz 4.2.5.

Sieć Procesy aplikacyjne oraz synchronizator(y) połączone są (w sensie opisu w języku NED) z modułem sieci tworząc topologie gwiazdy, której centrum stanowi sieć (patrz rys. 4.1). W ten sposób moduł sieci symuluje rzeczywistą sieć komputerową o zadanych cechach. Moduł ten określa takie elementy jak: protokół dostępu do łącza, topologia połączeń, przepustowość, i inne.

Przeprowadzenie symulacji miało posłużyć zbadaniu wydajności proponowanego mechanizmu sterowania. W związku z tym, model systemu równoległego zaimplementowany w symulatorze musiał obejmować aspekty wydajności obliczeń równoległych. Postanowiliśmy oprzeć się na znanym i sprawdzonym modelu LogP [38]. Model ten wykorzystuje cztery parametry opisujące system równoległy: L - maksymalny (przy pewnych założeniach) czas transmisji komunikatu, o - czas procesora niezbędny przy wysłaniu i odebraniu komunikatu, g - minimalny odstęp pomiędzy kolejnymi komunikatami nadawanymi/odbieranymi przez dany proces, P - liczba procesów w systemie. Wartości *Log* są zwykle określane względem czasu pojedynczego cyklu procesora. Model LogP sprawdza się, gdy modelowana aplikacja równoległa przesyła małe komunikaty podobnej wielkości - a właśnie tym charakteryzuje się nasz system sterowania. Wariant tego modelu - LogGP - uwzględnia dodatkowo przepustwość (G) i pozwala modelować dłuższe komunikaty [5]. Zaletą modeli rodziny LogP jest ich prostota, pozwalająca na skuteczną analizę efektywności projektowanych algorytmów równoległych. Jednak w naszym przypadku musieliśmy poświęcić część tej prostoty na rzecz dokładności modelu.

Parametr L normalnie określany jest dla ustalonej (zwykle niewielkiej) intensywności ruchu w sieci, po czym jest traktowany jako faktyczny czas transmisji każdego komunikatu. Tymczasem w praktyce średni czas przesłania komunikatu zwykle okazuje się znacznie krótszy niż czas maksymalny. Wydajność proponowanej metody sterowania zależy przede wszystkim od czasu średniego, dlatego nie mogliśmy przyjąć takiego uproszczenia. W symulatorze czas transmisji komunikatu T_t wynika z parametrów o i G, rozmiaru komunikatu R oraz opóźnienia wprowadzanego przez sieć T_o. Wyraża się on wzorem $T_t = 2o + R/G + T_o$. Ponieważ chcieliśmy móc badać rozmaite rodzaje sieci



Rysunek 4.1: Schemat połączeń modułów w symulacji

przy szerokim i zmiennym zakresie intesywności ruchu, postanowiliśmy wyznaczać L dynamicznie w trakcie symulacji i bez ograniczania liczby transmitowanych jednocześnie komunikatów, a nie w specjalnych wstępnych eksperymentach dla każdego zestawu parametrów.

Po wprowadzeniu parametru T_o w symulatorze musieliśmy określić jego wartość. Prostym i najdokładniejszym rozwiązaniem okazało się symulowanie działania sieci, z uwzględnieniem topologii połączeń i protokołu dostępu do łącza. Tym samym zrezygnowaliśmy z założenia modelu LogP mówiącego, że faktyczna struktura sieci jest nieistotna, gdyż charakterystykę działania sieci wystarczająco określają parametry modelu. Jednocześnie w ten sposób zapewniliśmy sobie możliwość przebadania wpływu architektury sieci na wydajność mechanizmu sterowania.

Uruchomienie symulacji wymaga podania (zwykle w pliku) szeregu parametrów, których część omówiliśmy powyżej. Najważniejsze z nich to:

1. parametry aplikacji
   - liczba procesów aplikacyjnych (P w modelu LogP),
   - czas trwania stanów lokalnych procesów (okres raportowania stanu lokalnego),
   - liczebność grupy procesów (dla rozwiązań hierarchicznych),
   - specyficzne dla symulowanej aplikacji, np. czas obliczenia pojedynczego zadania, okres wysyłania i rozmiar komunikatów do innych procesów, itp.
2. parametry sieci komputerowej
   - topologia i protokół dostępu (nazwa modułu implementującego dany typ sieci),
   - przepustowość (G w modelu LogGP),
   - dodatkowe opóźnienie,

- Maximal Transfer Unit (MTU, największy rozmiar danych możliwy do przesłania w pojedynczej transmisji),

3. parametry systemowe

   - dokładność synchronizacji zegarów lokalnych względem zegara wzorcowego ε,
   - gwarantowany czas trwania stanów lokalnych δ,
   - czas pojedynczego wykonania algorytmu wykrywania SCGS,
   - czas pojedynczej ewaluacji predykatu,
   - czas CPU wykorzystywany do nadania/odbioru komunikatu (o w modelu LogP),
   - minimalny czas pomiędzy kolejnymi operacjami nadania/odbioru komunikatu przez dany proces (g w modelu LogP).

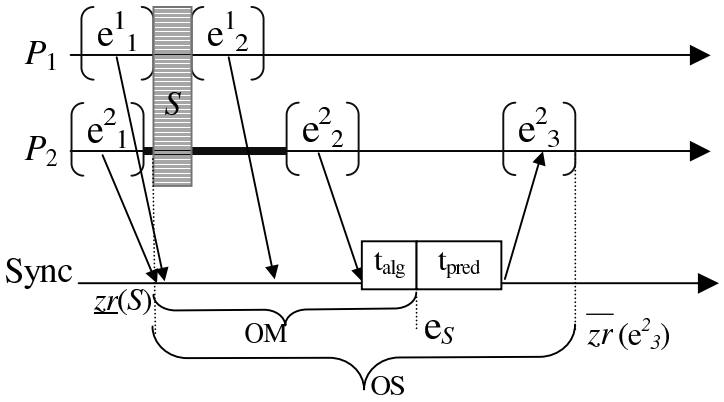
4.2 Eksperymenty symulacyjne dla wybranych zastosowań

Eksperymenty symulacyjne miały dwa cele: porównanie alternatywnych rozwiązań (np. różnych wersji algorytmów wykrywania SCGS), oraz szacowanie rzeczywistej wydajności badanego mechanizmu sterowania. Uzyskanie wiarygodnych wyników wymagało wstępnego skalibrowania symulacji i wyznaczenia właściwych wartości parametrów środowiska systemowo-sprzętowego. W tym celu przeprowadziliśmy szereg eksperymentów pomiarowych oraz dokonaliśmy przeglądu literatury. Symulator został skalibrowany tak, aby odpowiadał sieci komputerów PC z procesorem 2GHz, połączonych siecią Ethernet 100BaseT i stosujących bibliotekę komunikacyjną MPICH [54], Ethernet 1000BaseT z MPICH lub Myrinet 2000 z biblioteką MPICH-GM [10]. Ponieważ asynchroniczna reakcja na sygnały sterujące nie jest standardowym mechanizmem w istniejących systemach komputerowych ogólnego przeznaczenia, przyjęliśmy, że zostanie ona zaimplementowana tak, jak w systemie PS-Grade, opisanym w rozdziale 5 (komunikat od synchronizatora do docelowego węzła i dalej sygnał czasu rzeczywistego systemu UNIX wewnątrz węzła do docelowego procesu).

4.2.1 Miary jakości sterowania

Przeprowadzenie systematycznych badań ilościowych proponowanego mechanizmu sterowania wymaga określenia miar, przy użyciu których będziemy klasyfikować wyniki. Istnieją tu dwa podejścia. W pierwszym, dla danej aplikacji można zdefiniować specyficzną dla niej miarę w kategoriach pojęć z dziedziny tej aplikacji. Miara taka zwykle oddaje uzyskaną jakość sterowania z punktu widzenia potrzeb konkretnej aplikacji. Mierzony jest końcowy efekt działania sterowania, czyli skuteczność mechanizmu sterowania do realizacji wybranego zagadnienia. Użyte przez nas miary specyficzne dla aplikacji omówimy dalej, przy okazji opisywania konkretnych eksperymentów. Niestety, tego rodzaju miary muszą być osobno definiowane dla każdej aplikacji. Powoduje to, oprócz każdorazowego wysiłku z tym związanego, że miary takie są nieporównywalne ze sobą i niewiele mówią o samym systemie sterowania. Drugie podejście oznacza stosowanie miar niezależnych od aplikacji, czyli miar ogólnych. Miary te biorą pod uwagę tylko cechy wyizolowanego systemu sterowania, bez zwracania uwagi na kwestię czym system ten steruje. Miary ogólne powinny być dobrane tak, aby ulepszenie mechanizmu sterującego poprawiające wartości miar ogólnych, nie pogarszało jakości tego mechanizmu według miar specyficznych dla aplikacji. Proponujemy dwie ogólne miary jakości mechanizmu sterowania opartego o predykaty: czas reakcji systemu sterowania oraz częstość reakcji systemu sterowania.

Na rysunku 4.2 widzimy schemat ciągu zdarzeń prowadzący od zaistnienia silnie spójnego stanu globalnego S, przez jego wykrycie (standardowym algorytmem SCGS, 1), do wywołania reakcji spowodowanej spełnieniem pewnego predykatu globalnego w S. Interesująca nas wielkość *opóźnienie sterowania* OS, to czas, jaki upływa pomiędzy zaistnieniem S, a momentem wywołania reakcji w procesie, czyli według oznaczeń z rysunku $\overline{zr}(e_3^2) - \underline{zr}(S)$. Ponieważ wartość $\underline{zr}(S)$ jest odczytywana z zegara synchronizatora, a wartość $\overline{zr}(e_3^2)$ z zegara danego procesu, OS wyliczane



Rysunek 4.2: Opóźnienie sterowania OS i opóźnienie monitorowania OM

jest z dokładnością 2ε. W symulacji, w pełni kontrolując parametry środowiska, mogliśmy łatwo wyeliminować tą niedokładność.

Opóźnienie sterowania OS jest to czas pomiędzy początkiem gwarantowanego okresu istnienia silnie spójnego stanu S, a górnym ograniczeniem czasu zajścia zdarzenia e_i, gdzie e_i jest wywołaniem reakcji w procesie P_i, spowodowanej spełnieniem pewnego predykatu globalnego w stanie S. Czyli $OS = \overline{zr}(e_i) - \underline{zr}(S)$.

Powyższa definicja określa bezwzględną wartość opóźnienia. Wartość ta zależy od wielu czynników, jej interpretacja może nastręczać trudności. Prezentując wyniki eksperymentów będziemy stosować miarę OS unormowaną względem wybranych parametrów symulacji, tak aby podkreślić istotne zależności między tymi parametrami a stosowaną miarą.

Decyzje sterujące synchronizatora są wynikiem ewaluacji predykatów na stanach spójnych aplikacji. Predykaty są wartościowane jednokrotnie na każdym stanie, zaraz po zaobserwowaniu danego stanu, toteż synchronizator ma okazje podejmować decyzje sterujące tak często, jak często obserwowane są stany spójne aplikacji. Częstotliwość pojawiania się silnie spójnych stanów globalnych z punktu widzenia synchronizatora jest zatem istotna, bowiem determinuje ona stopień ciągłości sterowania sprawowanej przez synchronizator.

Częstotliwość sterowania CzS to średnia liczba stanów spójnych aplikacji obserwowanych przez synchronizator w ciągu sekundy, będąca jednocześnie liczbą potencjalnie podejmowanych decyzji sterujących w ciągu sekundy. CzS zależy od szeregu parametrów systemu i nadzorowanej aplikacji. Dlatego miarę CzS będziemy normować względem wybranych parametrów symulacji, tak aby podkreślić istotne zależności między tymi parametrami a stosowaną miarą.

Proponowany mechanizm sterowania bazuje na mechaniźmie monitorowania stanów spójnych aplikacji. Właściwości mechanizmu monitorującego wpływają silnie na skuteczność pracy systemu sterowania. Dlatego proponujemy dodatkowo miary jakości samego mechanizmu monitorującego.

Opóźnienie monitorowania OM jest to czas pomiędzy początkiem gwarantowanego okresu istnienia silnie spójnego stanu S, a górnym ograniczeniem czasu zajścia zdarzenia e_S, gdzie e_S oznacza wykrycie tego stanu przez synchronizator. Czyli $OM = \overline{zr}(e_S) - \underline{zr}(S)$. Podobnie jak OS, OM wyliczane jest z dokładnością 2ε. W symulacji, w pełni kontrolując parametry środowiska, mogliśmy łatwo wyeliminować tą niedokładność.

Przyjrzymy się teraz sytuacji, w której procesy nie informują synchronizatora o wszystkich zmianach swych stanów lokalnych. Dopuszczamy tę możliwość, gdy zmiany stanów lokalnych mają charakter stopniowy. Przykładem może być monitorowana temperatura pewnego elementu konstrukcyjnego lub średnie obciążenia procesora. Rzadsze raportowanie bieżącego stanu może być

wskazane, aby zmniejszyć obciążenie sieci przesyłaniem oraz synchronizatora przetwarzaniem komunikatów zawierających informacje o stanie procesów. Synchronizator, zamiast wielu drobnych korekt monitorowanej wielkości, otrzyma większe korekty, lecz rzadziej. W rezultacie obserwowane stany aplikacji będą zawierały informacje częściowo nieaktualne już w czasie istnienia danego stanu. Do pomiaru aktualności informacji skojarzonej z danym stanem spójnym aplikacji użyjemy miary aktualności stanów AS.

Aktualność stanu SCGS $S =< s_1, .., s_N >$ jest to liczba $AS = \sum_{i=1}^{N} \overline{zr}(e_S) - \overline{zr}(\underline{s_i})$, gdzie e_S oznacza zaobserwowanie stanu S przez synchronizator.

AS można wyznaczyć z dokładnością $2N\varepsilon$. W symulacji, w pełni kontrolując parametry środowiska, mogliśmy wyeliminować tą niedokładność. Podobnie jak OS, tak i AS można normalizować względem wybranych parametrów symulacji, np. średniej długości czasu trwania stanów lokalnych. Miara ta pozwala określić ilościowo na ile aktualne (albo przestarzałe) są informacje o stanie systemu używane do podejmowania decyzji sterujących.

4.2.2 Testy wybranych algorytmów wykrywania SCGS

W rozdziale 2 zaproponowaliśmy kilka wariantów algorytmów wykrywania SCGS. Wybraliśmy do porównań cztery z nich, wszystkie stosujące globalną dokładność synchronizacji zegarów procesów, a zatem nadające się do zastosowania w najczęściej spotykanych środowiskach. Testy rozwiązań hierarchicznych opisane są dalej w punkcie 4.2.5. Porównywane algorytmy to:

T standardowy algorytm pracujący na zakończonych (*Terminated*) stanach lokalnych (algorytm 2),

DT algorytm wykorzystujący minimalny czas trwania stanów lokalnych (*Delta,* algorytm 6)

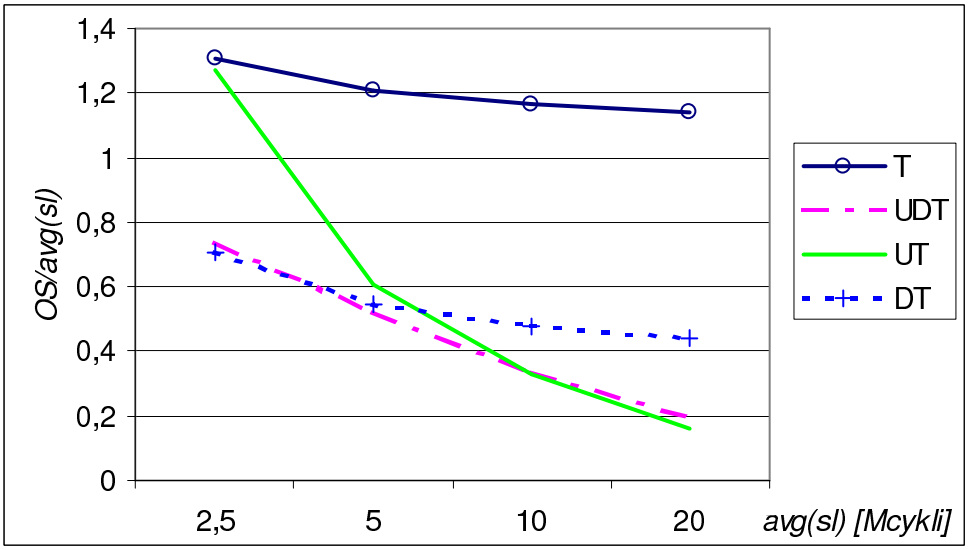
UT algorytm wykorzystujący ograniczenie na maksymalny czas przesłania komunikatu i pracujący na niezakończonych stanach lokalnych (*Unterminated+Timeout*, algorytm 5)

UDT algorytm UT dodatkowo wykorzystujący minimalny czas trwania stanów lokalnych (*Unterminated+Delta+Timeout*, punkt 2.4.2.1)

Aplikacja wykorzystywana w testach miała następującą konstrukcję: stan procesu aplikacyjnego P_i charakteryzowany był przez liczbę rzeczywistą zwaną wartością kontrolną WK_i. Procesy wykonywały ciąg zadań obliczeniowych, po zakończeniu każdego zadania przyrostowo zmieniały swe WK zgodnie z określonym wzorem i z udziałem czynnika losowego. Raporty o aktualnej wartości kontrolnej wysyłane były do synchronizatora periodycznie. Synchronizator sprawdzał, czy wartości te mieszczą się w dopuszczalnym przedziale wokół obserwowanej przez niego średniej arytmetycznej $\overline{WK} = \sum_{i=1}^{N} WK_i/N$. Do procesów charakteryzujących się zbyt dużą lub zbyt małą WK wysyłano sygnał sterujący nakazujący korektę wartości. Opisana aplikacja pozwala łatwo zaobserwować różnice w jakości sterowania, oraz, co opisano dalej, umożliwia wprowadzanie wariantów odzwierciedlających pewne generyczne zachowania procesów aplikacyjnych. Pozwala też łatwo zdefiniować hierarchiczne schematy sterowania, co wykorzystanow w punkcie 4.2.5.

Dla tej aplikacji zdefiniowaliśmy następującą miarę jakości sterowania specyficzną dla tej aplikacji Q: sumowaliśmy różnice $|\overline{WK}_R - WK_i|$ mierzone po każdym końcu zadania w każdym procesie, gdzie $\overline{WK}_R$ oznacza rzeczywistą i aktualną (a nie obserwowaną przez synchronizator) średnią arytmetyczną wartości kontrolnych. Otrzymaną sumę dzieliliśmy przez globalną liczbę wykonanych zadań obliczeniowych. Uzyskana wartość, oznaczana symbolem Q, reprezentuje poziom odchylenia wartości kontrolnych w procesach od ich globalnej średniej w czasie działania aplikacji.

Symulowaliśmy 32 procesy aplikacyjne, dokładność synchronizacji zegarów ustaliliśmy na 120000 cykli CPU, co jest wartością możliwą do osiągnięcia programowo, przy użyciu NTP, o ile zegar wzorcowy dostępny jest w sieci lokalnej.



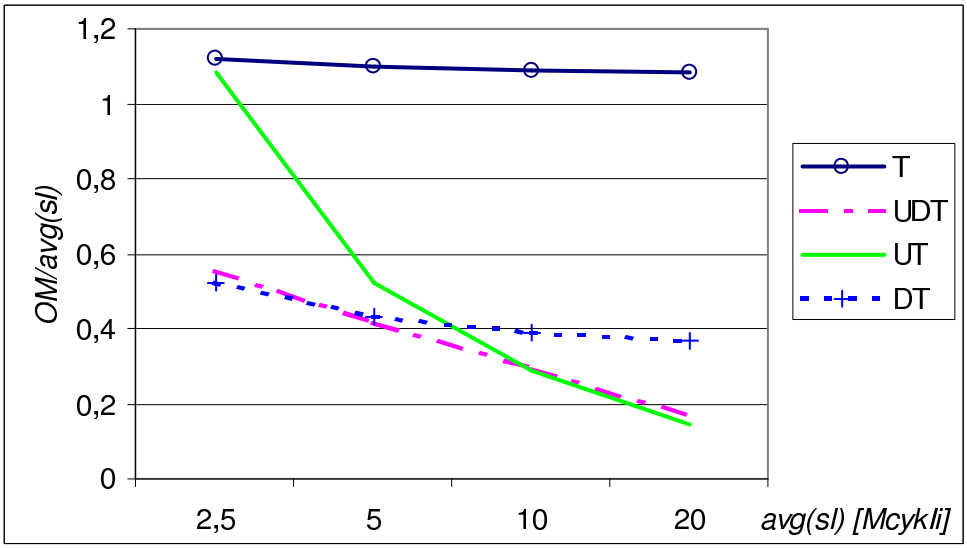
Rysunek 4.3: Opóźnienie sterowania OS normalizowane średnim czasem trwania stanów lokalnych avg(sl)

4.2.2.1 Charakterystyka Opóźnienia Sterowania OS dla czterech algorytmów wykrywania SCGS

Zbadaliśmy uzyskiwane wartości OS w opisanej aplikacji przy zastosowaniu czterech wymienionych algorytmów SCGS, zmieniając długości trwania stanów lokalnych procesów sl. Na rysunku 4.3 wartości na osi poziomej oznaczają średni czas trwania stanów lokalnych avg(sl) (czyli średni okres pomiędzy kolejnymi raportami o wartościach kontrolnych wysyłanymi przez dany proces). Na osi pionowej mamy opóźnienie sterowania OS znormalizowane względem avg(sl). Ponieważ w eksperymentach dalsze działanie systemu po zaobserwowaniu SCGS było identyczne w każdym przypadku (wartościowane były te same predykatów, wysyłane takie same sygnały sterujące poprzez taką samą sieć), to wartość $OS-OM$ była stała. Zatem ukazane wyniki powinny być uwarunkowane efektywnością samego monitorowania. Rzeczywiście, potwierdzenie tej tezy znajdziemy na rysunku 4.4. Widzimy tam zmiany opóźnienia monitorowania OM, także znormalizowanego względem średniego czasu trwania stanów lokalnych avg(sl), które odzwierciedlają wiernie zmiany OS z rysunku 4.3. Widoczny na rysunku 4.3 silniejszy wzrost znormalizowanej wartości OS dla krótkich stanów lokalnych wynika z proporcjonalnie większej roli stałej wartości $OS - OM$. Po ustaleniu, że nasze zainteresowanie w tym przypadku możemy ograniczyć do miary OM, zajmiemy się dalej analizą wyników z rysunku 4.4.

Czas potrzebny do zaobserwowania SCGS przez algorytm T jest ściśle skorelowany z długością stanów lokalnych procesów, używanych do budowy SCGS. Średnie czasy trwania stanów lokalnych avg(sl) zmienialiśmy od 20 do 2,5 miliona cykli CPU, otrzymując za każdym razem OM znormalizowane względem avg(sl) równe ok. 1,1. Jest to naturalna konsekwencja sposobu działania tego algorytmu - musi on poczekać na zakończenie stanu lokalnego, zanim weźmie taki stan pod uwagę.

Algorytm UT sprawuje się tym lepiej, im dłuższe są stany lokalne, na których pracuje. Wynika to z faktu, że kluczową rolę odgrywa tu opóźnienie alarmu t, patrz rysunek 2.7. Opóźnienie to było stałe w całym eksperymencie z uwagi na niezmienne środowisko systemowo-sprzętowe. W przypadku długich stanów lokalnych, t jest wielokrotnie mniejsze od sl, w rezultacie czego otrzymujemy OM wielokrotnie mniejsze, niż w przypadku algorytmu T. Gdy czasy trwania stanów lokalnych maleją zbliżając się do t, wtedy algorytmy T i UT dają jednakowe opóźnienie monitorowania OM. W dodatkowej serii eksperymentów z algorytmem UT przeanalizowaliśmy wartość OM znormalizowaną względem opóźnienia alarmu t. Dla ustalonego czasu trwania stanów lokalnych zmienialiśmy wartość t. Uzyskane wyniki mieściły się w przedziale pomiędzy wartościami 1,05 a 1,2. Dowodzi to, że czas potrzebny do zaobserwowania SCGS przez algorytm UT jest ściśle



Rysunek 4.4: Opóźnienie monitorowania OM normalizowane średnim czasem trwania stanów lokalnych avg(sl)

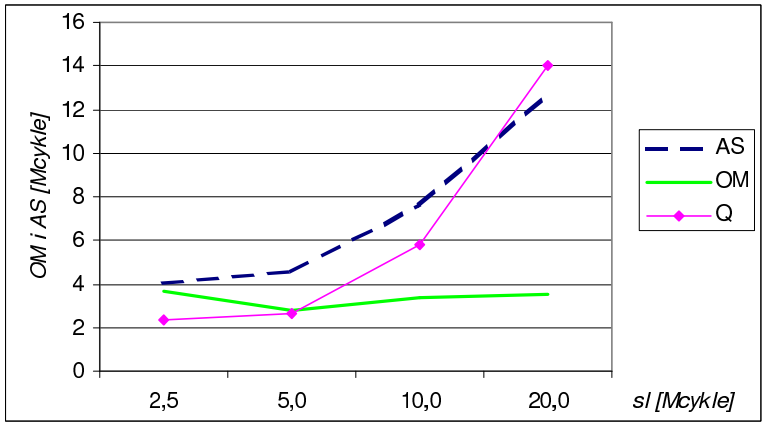
skorelowany ze stosowanym opóźnieniem alarmu t i wynosi między $1,05t$ a $1,2t$.

Bardzo interesująco zaprezentował się algorytm DT. Daje on wyniki lepsze 2-3 krotnie niż T w całym zbadanym przedziale długości stanów lokalnych. O ile przegrywa z UT w zakresie długotrwałych stanów lokalnych, to jest od niego znacznie lepszy w przypadku krótkich stanów lokalnych. To zachowanie stanie się zrozumiałe, gdy przeanalizujemy rolę gwarantowanego minimalnego czasu trwania stanów lokalnych δ. W eksperymentach przyjęliśmy, że $\delta = 0,75\text{avg}(sl)$. Zatem okres, w którym użycie delty nie pomaga i synchronizator musi poczekać na zakończenie danego stanu lokalnego, trwa najwyżej $0,25\text{avg}(sl)$. Gdy doliczymy do tego pewien stały narzut (czas transmisji komunikatu o zakończeniu stanu plus czas działania algorytmu wykrywania SCGS), mający większe względne znaczenie dla krótkich stanów lokalnych, uzyskamy pełne uzasadnienie wyników widocznych na rysunku 4.4.

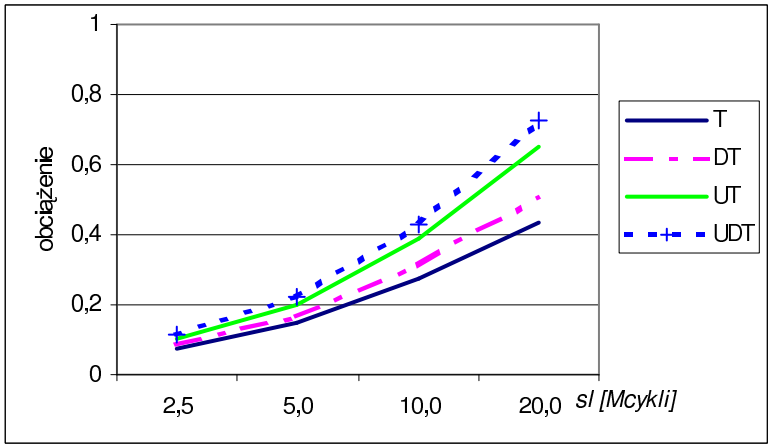
W dodatkowej serii eksperymentów przebadaliśmy zachowanie algorytmu DT dla pełnego przedziału $0 < \delta < sl$. Wyniki potwierdziły nasze rozumowanie. Nienormalizowane OM dla algorytmu DT wyraża się wzorem $OM = sl - \delta + t_{kom} + t_{alg}$, gdzie t_{kom} to czas komunikacji, czas w którym informacja o stanie lokalnym procesu dociera do synchronizatora, t_{alg} to czas działania algorytmu wykrywania SCGS. W rezultacie wartość nienormalizowanego OM początkowo maleje odwrotnie proporcjonalnie do wzrostu wartości δ. Gdy δ zbliża się do sl, wartość OM zbliża się do wartości $t_{kom} + t_{alg}$, lecz nigdy nie spada poniżej.

Algorytm UDT przejmuje najlepsze cechy algorytmów UT i DT: dla krótszych stanów lokalnych zachowuje się jak DT, dla dłuższych jak UT. Połączenie mechanizmu alarmu i gwarantowanego minimalnego czasu trwania stanów lokalnych zaowocowało elastycznym zachowaniem algorytmu.

W przeprowadzonym eksperymencie procesy przesyłały do synchronizatora raporty o swej wartości kontrolnej co k-tą jej zmianę (a nie przy każdej zmianie). Im k było bliższe jedności, tym krócej trwały z punktu widzenia synchronizatora stany lokalne. Zgodnie z opisaną charakterystyką działania algorytmu T, powodowało to zmniejszenie opóźnienia monitorowania OM dla tego algorytmu. Jednocześnie, zgodnie z intuicją, zmniejszała się wartość miary Q (patrz 4.2.2). Natomiast w przypadku algorytmu UT, mimo zwiększania częstotliwości raportowania wartości kontrolnych, OM pozostawało na stałym poziomie, a jedynie Q malało. Tutaj miara OM okazała się niewystarczająca do uchwycenia zmian charakterystyki pracy systemu sterowania. Użyliśmy zatem innej miary ogólnej - aktualności stanu AS. Na rysunku 4.5 pokazane są nienormowane miary OM i AS oraz miara Q dla algorytmu UT. Jak widać, w tym przypadku jakość uzyskiwanej kontroli według miary specyficznej dla aplikacji Q jest znacznie lepiej odwzorowywana przez miarę ogólną



Rysunek 4.5: Miary OM, AS i Q dla algorytmu UT



Rysunek 4.6: Obciążenie CPU synchronizatora dla czterech algorytmów SCGS

AS, niż przez OM. Duża wartość OM implikuje dużą wartość AS (patrz definicje), natomiast przy małej wartości OM AS może być duże. W rozpatrywanej sytuacji uzyskanie małej wartości miary Q jest możliwe tylko wtedy, gdy zarówno OM, jaki i AS mają odpowiednio małe wartości.

Osobnej uwagi wymaga obciążenie synchronizatora (tj. procesora, na którym on pracuje). Obciążenie to jest inne dla poszczególnych algorytmów SCGS. Cztery testowane algorytmy mają co prawda taką samą złożoność, ale widoczną rolę odgrywają tu stałe współczynniki inne dla każdego wariantu algorytmu. Rzeczywiste wartości tych współczynników zależą od konkretnej implementacji algorytmów SCGS, toteż przedstawione tu wyniki należy traktować orientacyjnie. Względny koszt poszczególnych wersji algorytmów ustaliliśmy metodą porównania wykonywanej liczby operacji na zdarzenie, z uwzględnieniem zdarzeń dodatkowych (delta, alarm). Rysunek 4.6 ukazuje otrzymane wyniki. Najbardziej złożony algorytm (UDT) powoduje około dwukrotnie większe obciążenie procesora, niż algorytm najprostszy (T). Pamiętajmy jednak, że w wypadku użycia skomplikowanych predykatów, a także większej liczby procesów, czas działania algorytmu SCGS (logarytmiczny względem liczby procesów) będzie miał niewielkie znaczenie w porównaniu z czasem wartościowania predykatów (zwykle liniowym względem liczby procesów).

4.2.2.2 Wpływ charakterystyki sterowanej aplikacji na uzyskiwaną jakość sterowania

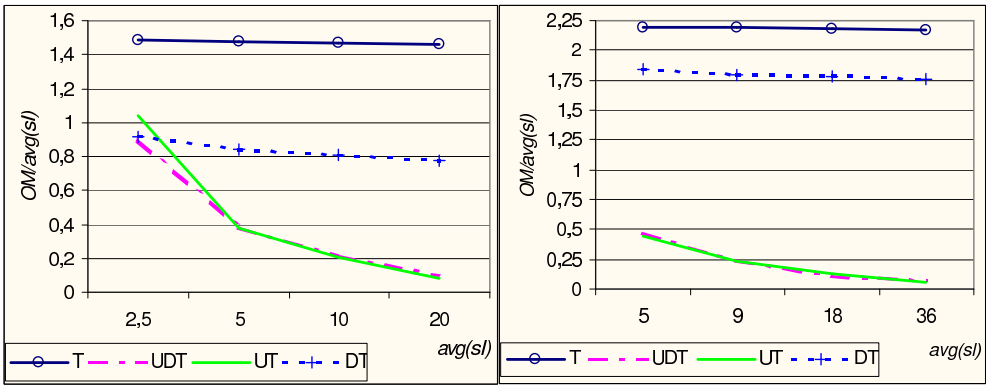
Rezultaty przedstawione w poprzednim punkcie wyraźnie wskazują, że na mierzone parametry sterowania wpływają trzy zasadnicze czynniki:

- czas trwania stanów lokalnych w procesach aplikacyjnych sl,
- minimalny czas trwania stanu lokalnego δ w stosunku do faktycznego czasu trwania stanu lokalnego sl,
- czas opóźnienia alarmu t

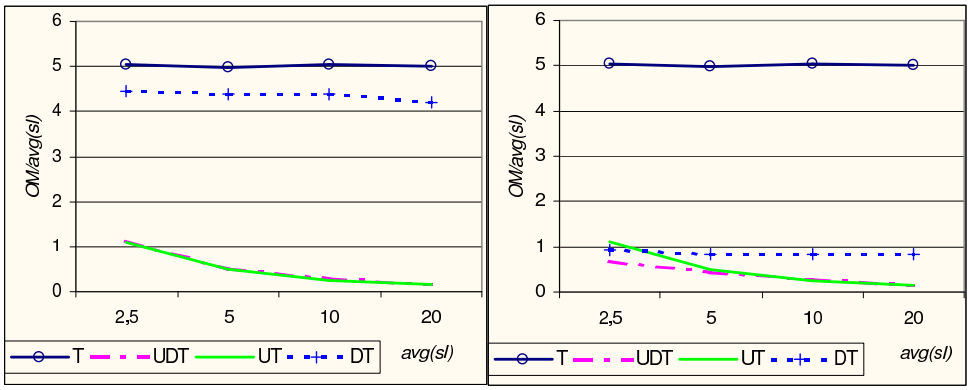
Parametr t zależy od środowiska obliczeniowego i na razie nie będziemy się nim zajmować. Parametry sl oraz δ stanowią pewne charakterystyki samej aplikacji równoległej i przyjrzymy się im teraz dokładniej. W opisanych powyżej eksperymentach badaliśmy OM dla zakresu wartości $\text{avg}(sl)$. Czas teraz doprecyzować, że przyjęty wtedy rozkład losowy czasów trwania stanów lokalnych sl był identyczny we wszystkich procesach aplikacyjnych. Zastosowaliśmy rozkład normalny o średniej $\text{avg}(sl)$ i odchyleniu standardowym $\text{stddev}(sl) = \frac{1}{10}\text{avg}(sl)$. Teraz zbadamy wpływ wartości $\text{stddev}(sl)$ na otrzymywane wyniki. Na rysunku 4.7 widzimy wartości OM uzyskane dla czterech badanych algorytmów wykrywania SCGS przy odchyleniu standardowym czasu trwania stanów lokalnych $\text{stddev}(sl) = \frac{3}{10}\text{avg}(sl)$ i $\frac{6}{10}\text{avg}(sl)$. Porównując je dodatkowo z rysunkiem 4.4 odpowiadającym $\text{stddev}(sl) = \frac{1}{10}\text{avg}(sl)$ dochodzimy do następujących wniosków:

- Wartość $\text{stddev}(sl)$ nie ma wpływu na wartości OM uzyskiwane dla algorytmu UT. Niższa wartość normalizowanego OM w przypadku $\text{stddev}(sl) = \frac{6}{10}\text{avg}(sl)$ wynika z faktu, że t pozostało niezmienne, podczas gdy $\text{avg}(sl)$ zwiększyło się (zwiększenie średniej było wymuszone zwiększeniem odchylenia standardowego - czasy nie mogły przybierać ujemnych wartości).
- Wartości OM uzyskiwane dla algorytmu T wyraźnie pogarszają się, gdy $\text{stddev}(sl)$ rośnie. W teście zwiększenie $\text{stddev}(sl)$ n-krotnie powodowało $\frac{n}{3}$-krotne zwiększenie OM. Powodem jest oczekiwanie na zakończenia stanów lokalnych - niektóre z nich trwają szczególnie długo dla dużych wartości odchylenia standardowego.
- Wartości OM uzyskiwane dla algorytmu DT są gorsze dla większych wartości $\text{stddev}(sl)$. Wyjaśniamy to następująco: wartości δ były ustawiane według reguły $\delta \approx \frac{3}{4}\text{avg}(sl)$. Dla dużych odchyleń standardowych $\text{stddev}(sl)$ otrzymywane różnice $ls - \delta$ często osiągały duże wartości. Algorytm może czekać $ls_i - \delta$ na zakończenie każdego z rozpatrywanych stanów lokalnych s_i (patrz 2.4.2). Oczekiwanie jest zdominowane przez $\max_{i=1..N} ls_i - \delta$, zatem wyraźnie się wydłuża gdy $\text{stddev}(sl)$ rośnie.

Obowiązującym to tej pory założeniem w naszych eksperymentach była homogeniczność procesów. Postanowiliśmy to zmienić. W kolejnym teście wróciliśmy do ustawienia $\text{stddev}(sl) = \frac{1}{10}\text{avg}(sl)$ i zmodyfikowaliśmy proces P_1 tak, aby jego stany lokalne były średnio 10 razy dłuższe niż stany lokalne pozostałych procesów. Spowodowało to zwiększenie $\text{stddev}(sl)$ do $\frac{1}{2}\text{avg}(sl)$, tym razem jednak tylko pojedynczy proces był odpowiedzialny za ten wzrost odchylenia standardowego. Zaobserwowane wartości znormalizowanego OM dla algorytmów T i DT okazały się kilkukrotnie większe niż w przypadku użycia podobnej wartości $\text{stddev}(sl)$ dla homogenicznych procesów, co widać na rysunku 4.8 po lewej stronie. Znając odmienną charakterystykę P_1 zrezygnowaliśmy z globalnego charakteru parametru δ i wprowadziliśmy δ_i właściwe dla procesu P_i. δ_1 otrzymało wartość 10 razy większą niż $\delta_i, 1 < i \leq N$. Uzyskana ogromna poprawa charakterystyki algorytmu DT widoczna jest na rysunku 4.8. Można powiedzieć zatem, że DT działa efektywnie, jeśli znamy minimalne czasy trwania stanów lokalnych poszczególnych procesów i jeśli faktyczne czasy trwania stanów lokalnych nie są wiele dłuższe od czasów minimalnych. Innymi słowy zarówno $\sum_{i=1}^{N}(\text{avg}(sl_i) - \delta_i)$ jak i $\sum_{i=1}^{N}\text{stddev}(sl_i)$ powinny by bliskie zeru. W skrajnie korzystnym przypadku, gdy stany lokalne trwają dokładnie δ_i, OS jest równe $2t_{msg} + D_{alg} + D_{pred} + t_{react}$, gdzie



Rysunek 4.7: Normalizowane OM dla zakresu avg(sl) przy stddev(sl) = $\frac{3}{10}$avg(sl) (z lewej) i $\frac{6}{10}$avg(sl) (z prawej)



Rysunek 4.8: Znormalizowane OM dla heterogenicznych procesów z użyciem globalnego parametru δ (z lewej) i specyficznego dla procesu δ_i (z prawej)

t_{msg} to czas przesłania komunikatu, D_{alg} to czas działania algorytmu SCGS, D_{pred} czas ewaluacji predykatu sterującego, t_{react} opóźnienie, z jakim realizowana jest reakcja na sygnał sterujący po jego dostarczeniu do docelowego węzła.

4.2.2.3 Podsumowanie charakterystyk wydajnościowych wybranych algorytmów wykrywania SCGS

Przeanalizowaliśmy efektywność wg miary OM czterech wybranych algorytmów wykrywania SCGS: T, UT, DT i UDT. Najkrócej ich charakterystyki można opisać następująco:

- Standardowy algorytm T niemal zawsze potrzebuje więcej czasu na zaobserwowanie SCGS, niż pozostałe algorytmy. Jednak nie wymaga żadnych dodatkowych założeń względem charakterystyki aplikacji i systemu.
- Stosowanie algorytmu UT opłaca się, gdy maksymalny czas transferu komunikatu do monitora jest istotnie (wielokrotnie) krótszy niż czasy trwania stanów lokalnych procesów.
- Algorytm DT zdecydowanie warto stosować, gdy czasy trwania stanów lokalnych są ograniczone z dołu. Jeśli czas trwania stanów lokalnych możemy przewidzieć dokładnie, wtedy DT okazuje się najszybszym algorytmem z czwórki prezentowanych.
- Opóźnienie monitorowania dla algorytm UDT odpowiada wartości mniejszej z opóźnień algorytmów DT i UT przy takich samych parametrach systemu i aplikacji. Jednak jest to

Tablica 4.1: Podsumowanie uzyskiwanych wartości OM dla testowanych algorytmów

algorytm	OM	uwagi
T	sl	OM istotnie wzrasta gdy stddev(sl) rośnie - dominujące znaczenie mają długotrwałe stany lokalne
UT	t	t jest wprost proporcjonalne do maksymalnego czasu transferu komunikatu do monitora
DT	$sl - \delta$	OM istotnie wzrasta gdy stddev($sl-\delta$) rośnie - dominujące znaczenie mają największe rozbieżności pomiędzy minimalnym a rzeczywistym czasem trwania stanów lokalnych
UDT	$\min(t, sl - \delta)$	Generuje parokrotnie większe obciążenie CPU niż T i DT

najbardziej złożony algorytm.

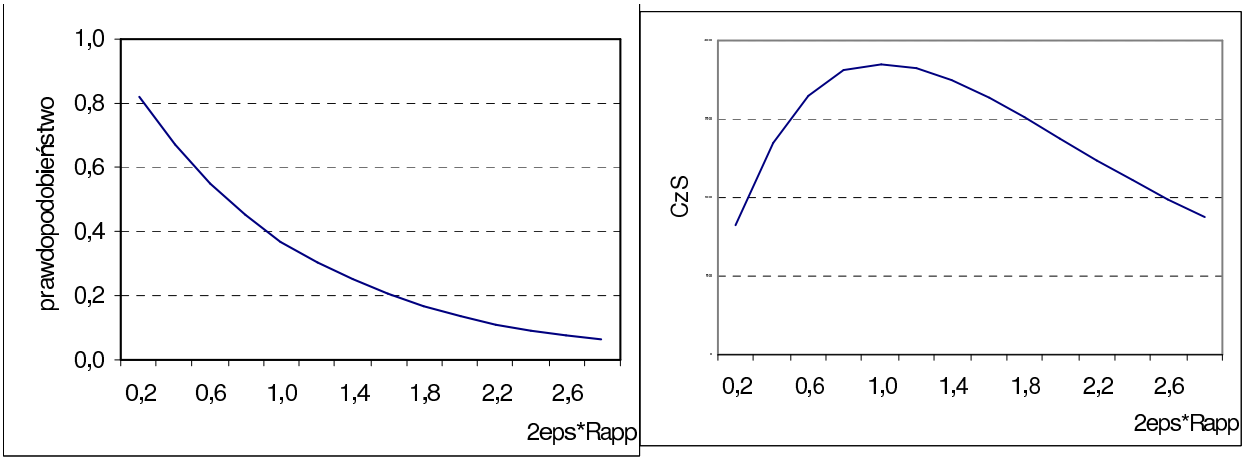
4.2.3 Badanie częstotliwości sterowania

Niedokładności w synchronizacji zegarów procesów, wymuszające użycie interwałowych znaczników czasu (patrz 2.1.3), uniemożliwiają analizę stanów globalnych następujących dowolnie szybko jeden po drugim. Przy dokładności synchronizacji zegarów względem wzorca równej ε, rzeczywista kolejność zdarzeń następujących po sobie w odstępie do 2ε nie może być jednoznacznie określona przez monitor (wyjątek stanowią zdarzenia z tego samego procesu). Na przykład na rysunku 2.1 interwały zdarzeń e_1^3 oraz e_2^2 pokrywają się, monitor nie wie które z tych zdarzeń było pierwsze i nie deklaruje SCGS pomiędzy nimi. To zjawisko powoduje, że część zmian stanów lokalnych, powodująca oczywiście zmiany stanu globalnego, jest pomijana przez monitor. Im więcej mamy takich pominięć, tym mniej silnie spójnych stanów globalnych zauważa monitor. Ponieważ decyzje sterujące w naszym modelu są podejmowane na podstawie wartości predykatów globalnych, a predykaty wartościuje się na zauważonych SCGS, to opisywane zjawisko prowadzi do redukcji liczby przypadków, w których można podjąć decyzje sterujące.

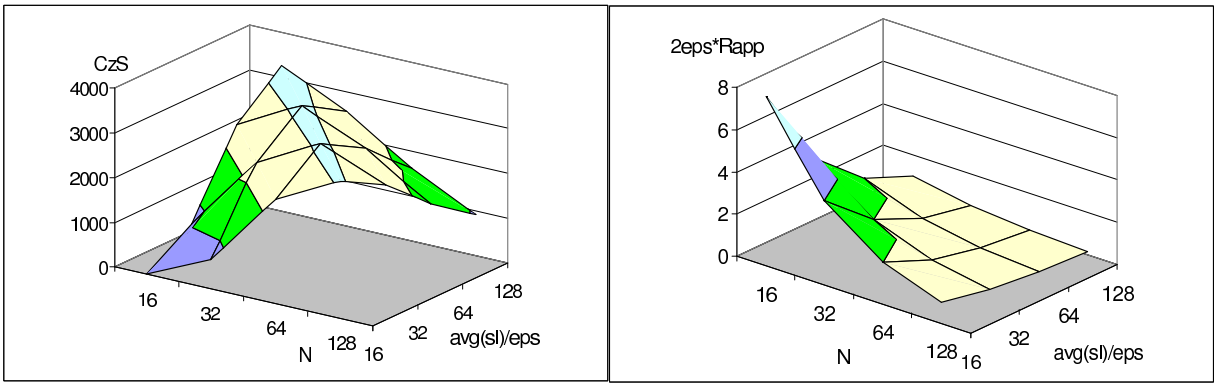
Problem ten analizowaliśmy już z innych powodów w punkcie 2.5.1. Przedstawione tam rozumowanie doprowadziło do wniosku, że oczekiwana liczba zdarzeń, które według monitora rozpoczynają SCGS wynosi $S_{exp} = (1 - \frac{2\varepsilon}{T})^{E(N-1)}$, gdzie T to czas działania aplikacji, $\varepsilon \ll T$, N to liczba procesów, E to liczba zdarzeń w jednym procesie. Przekształcając tę formułę otrzymaliśmy wykres obrazujący prawdopodobieństwo, że zmiana stanu lokalnego (zdarzenie) będzie dostrzeżona jako początek nowego SCGS przy różnych częstościach zmian stanów lokalnych $R_{app} = \frac{NE}{T}$. Aby uwzględnić rolę dokładności synchronizacji zegarów, wartości R_{app} na osi X przemnożyliśmy przez 2ε, patrz rysunek 4.9, część lewa. Dla przykładu, gdy $2\varepsilon R_{app} = 1$, czyli gdy zsumowane długości interwałów zdarzeń są równe T, widzimy, że 4 na 10 zdarzeń rozpoczyna SCGS.

Przemnożenie S_{exp} przez R_{app} pozwala określić bezwzględną liczbę SCGS obserwowaną przez monitor w ciągu sekundy, czyli miarę CzS. Prawa część rysunku 4.9 pokazuje rezultaty takiego mnożenia. Faktyczne wartości na osi Y nie grają tu roli (zależą od ustawień parametrów), interesujący natomiast jest kształt otrzymanej krzywej, czyli przebieg zmienności miary CzS, niezależny od zastosowanych wartości parametrów ε i R_{app}. Przy małej częstości zmian stanów lokalnych R_{app} w naturalny sposób SCGS powstają rzadko. Zwiększanie R_{app} zwiększa częstotliwość powstawania SCGS, ale tylko do pewnego momentu. Dalej liczba pomijanych zdarzeń rośnie tak bardzo, że częstość obserwowanych SCGS już maleje. Maksimum osiągane jest przy $2\varepsilon R_{app} = 1$.

Przeprowadziliśmy szereg eksperymentów symulacyjnych w celu weryfikacji powyższych analiz. Używając 8 do 64 procesów zmienialiśmy avg(sl) w zakresie 16ε do 128ε, przy $\varepsilon = 120000$ cykli CPU. Wyniki w pełni potwierdziły doychczasowe ustalenia. Na lewej części rysunku 4.10 grzbiet wykresu oznaczający najwyższe uzyskane wartości CzS powstał przy takich kombinacjach wartości na osi X (liczba procesów) i Y (avg(sl)/ε), dla których na prawej części rysunku $2\varepsilon R_{app}$ ma wartość zbliżoną do 1.



Rysunek 4.9: Prawdopodobieństwo, że zdarzenie rozpocznie SCGS (z lewej) i oczekiwana wartość CzS (z prawej) (eps=ε)



Rysunek 4.10: Eksperymentalnie wyznaczona CzS (z lewej) i odpowiadająca jej zeskalowana częstotliwość występowania zdarzeń $2\varepsilon R_{app}$ (z prawej), eps=ε

4.2.4 Analiza wpływu parametrów systemowych na uzyskiwaną jakość sterowania

Efektywność oraz skalowalność proponowanego mechanizmu sterowania zależy także od parametrów sprzętu oraz środowiska. Na początek przeanalizujmy Opóźnienie Sterowania OS pod tym kątem. Na OS składają się:

1. czas wysyłania komunikatu do sieci (tym oczekiwanie na dostęp do łącza, formowanie pakietu, jego transmisja),
2. czas transmisji komunikatu w sieci,
3. czas operacji odebrania komunikatu przez monitor,
4. czas, po którym informacja zawarta w komunikacie zostanie wzięta pod uwage, analizowany w poprzednim punkcie 4.2.2,
5. czas wykonania algorytmu wykrywania SCGS t_{alg},
6. czas ewaluacji predykatu t_{pred},
7. czas wysyłania sygnału sterującego,
8. transfer komunikatu z sygnałem,
9. czas operacji odebrania komunikatu z sygnałem,
10. uruchomienie właściwej reakcji procesu związanej z sygnałem.

4.2.4.1 Sieć

Sieć obsługująca nasz system sterowania odpowiada za:

- przesyłanie komunikatów o stanach lokalnych procesów do monitora,
- przesyłanie sygnałów sterujących z monitora do procesów.

Obydwa te zadania powinny być wykonywane szybko, tj. z małym opóźnieniem, gdyż opóźnienie to stanowi składowe 1-3 i 7-9 całego OS. Rozpatrzymy teraz wpływ kilku konkretnych parametrów sieci na sprawność działania mechanizmu sterującego.

Opóźnienie minimalne transmisji komunikatu T_{min}. Opóźnienie działania mechanizmu sterowania zawsze musi być większe od $2T_{min}$, gdyż komunikaty o stanach muszą najpierw dotrzeć do monitora, a następnie odwrotną drogę musi przebyć sygnał sterujący.

Przepustowość sieci. Sieć musi być w stanie przesłać sumaryczny ruch komunikatów o stanach generowany przez wszystkie monitorowane/sterowane procesy (ruch powodowany przez sygnały sterujące jest zwykle nieporównanie mniejszy). Od przepustowości (m.in.) zależy zatem jak wiele procesów może liczyć sterowany system i jak częste zmiany stanów lokalnych można będzie obsłużyć.

Częstość odbioru/wysłania komunikatów. Znana jako parametr g w modelu LogP, określa minimalny czas pomiędzy kolejnymi operacjami wysłania/odbioru pakietu. W praktyce determinuje ona ile najwięcej zdarzeń na sekundę potrafi na bieżąco przetworzyć monitor, czyli jak wiele procesów i jak częste zmiany stanów lokalnych można skutecznie monitorować. Parametr ten zależy od sprzętu, np. dla kart Ethernet wynosi ok $20\mu s$, dla kart Myrinet ok. $10\mu s$.

Opóźnienie średnie transmisji komunikatu T_{avg}**.** Opóźnienie działania mechanizmu sterowania w średnim przypadku będzie zawierać składnik $2T_{avg}$. Opóźnienie to może zależeć od wielu czynników, z reguły jednak rośnie gdy intensywność ruchu w sieci zbliża się do przepustowości tej sieci.

Opóźnienie maksymalne transmisji komunikatu T_{max}**.** Rola tego parametru jest dwojaka. Po pierwsze określa on jak duży wpływ na opóźnienie sterowania w pesymistycznym przypadku może mieć sieć. Po drugie, gdy wykorzystywany jest algorytm UT, T_{max} stanowi o opóźnieniu t, z jakim rozpoczęty, a jeszcze nie zakończony stan lokalny może być wzięty pod uwagę. Jest to stały składnik OS dla algorytmu UT. T_{max} zależy od protokołu dostępu do łącza, reguł przesyłania pakietów w sieci i obciążenia sieci.

Opóźnienie minimalne zależy od prędkości przesyłu danych, jest odpowiednio niższe w szybszych sieciach. Zależy też od złożoności protokołu transmisji i stopnia zaangażowania systemu operacyjnego. W sieciach korzystających ze stosu TCP/IP i usług systemu operacyjnego (obsługa protokołu, kopiowanie danych), np. Ethernet, T_{min} osiąga duże wartości, rzędu dziesiątek czy nawet setek mikrosekund. Sieci posługujące się uproszczonym protokołem realizowanym przez interfejsy sieciowe z pominięciem systemu operacyjnego (np. Myrinet, RDMA) umożliwiają redukcję T_{min} do wartości kilku mikrosekund, a nawet mniejszych. Na przykład, w sieci DIMMnet opóźnienie minimalne odpowiada długości jednego cyklu zapisu do pamięci, zatem jest na poziomie nanosekund [46].

Dwa ostatnie parametry: T_{avg} i T_{max} wymagają nieco szerszego omówienia. Opóźnienie maksymalne oraz średnie może być bardzo duże, gdy sposób działania sieci zakłada powstawanie nieudanych transmisji. Takie transmisje trzeba powtórzyć (a powtórki mogą znowu się nie udać), co oznacza, że efektywny czas przesłania komunikatu może być bardzo duży. Przykładami takich sieci są klasyczny Ethernet stosujący CSMA/CD oraz sieci radiowe oparte na CSMA/CA. Takie sieci nie nadają się do realizacji proponowanego mechanizmu sterującego.

Współcześnie stosowana odmiana Ethernetu, polegająca na użyciu przełączników pracujących w trybie store-and-forward wraz z łączami full-duplex, zapewnia niemal zawsze skuteczne przesłanie komunikatu (o ile tylko pojemność bufora w przełączniku nie zostaje przekroczona). Przy tym komunikaty oczekujące w buforze ustawiane są w kolejkę FIFO, dzięki czemu nie występuje zagłodzenie (czyli nieproporcjonalnie długi czas transmisji dla jednego komunikatu, przy małym czasie dla pozostałych). Taka sieć, oraz inne oparte na przełącznikach (np. Myrinet) są odpowiednie dla naszych potrzeb.

Pewne rodzaje sieci, np. Token Ring lub isochroniczny tryb IP over IEEE1394 pozwalają zagwarantować maksymalny czas oczekiwania na wysłanie pakietu, co przy pewnych założeniach można przełożyć na gwarantowany maksymalny czas transmisji. Cecha ta pozwala wyliczyć precyzyjnie T_{max} przy określonych limitach przepustowości przyznanych poszczególnym procesom. Jest to korzystne, choć cytowane wcześniej prace na temat przełączanego Ethernetu wskazują, że i w przełączanym Ethernecie określenie T_{max} jest możliwe przy podobnych założeniach.

Przeprowadzone badania symulacyjne potwierdziły nasze przypuszczenia, że dodatkowy ruch między procesami aplikacyjnymi w sieci opartej na przełącznikach (przełączany Ethernet) wpływa tylko w niewielkim stopniu na Opóźnienie Sterowania - przy założeniu, że przełącznik jest w stanie obsłużyć występujące jednocześnie transmisje. Komunikacja do/z monitora odbywa się wtedy bez większych zakłóceń. Natomiast każde dodatkowe obciążenie w sieci typu Token Ring/Bus powoduje zauważalne zwiększenie OS, gdyż dodatkowe komunikaty zajmują wspólne pasmo, z którego korzysta też mechanizm sterowania. W rezultacie, dobrym uzupełnieniem algorytmu UT może być stosowanie sieci pozwalającej precyzyjnie przewidzieć T_{max} (np. Token Ring) dedykowanej mechanizmowi sterowania, oraz innej sieci dla pozostałych transmisji.

4.2.4.2 Uruchomienie reakcji procesu na sygnał sterujący

Składowa numer 10 Opóźnienia Sterowania obejmuje rozpoznanie treści sygnału sterującego, sprawdzenie czy sygnał powinien spowodować reakcję w bieżącej sytuacji, uruchomienie samej reakcji. Ten ostatni element jest najważniejszy. Przyjęliśmy asynchroniczny sposób reakcji na sygnał

Tablica 4.2: Czasy reakcji na sygnał w systemach Unix-owych

System	czas	uwagi
Alpha / TruUnix64 v5.1	0,7s	
Linux / Intel Pentium	0,1-0,01s	czas zależny od liczby współbieżnych procesów
Hitachi SR2201	0,004s	komputer wieloprocesorowy z pamięcią rozproszoną
HP class D 2CPU SMP	140μs	procesy wysyłający i odbierający sygnał umieszczone na różnych procesorach
Linux / Intel Pentium 2CPU SMP	8μs	procesy wysyłający i odbierający sygnał umieszczone na różnych procesorach
zmodyfikowany Linux / Intel Pentium	5-10μs	niezależnie od liczby współbieżnych procesów

(punkt 3.3.2.1). Oznacza to, że komunikat niosący sygnał musi być odebrany i obsłużony niezależnie od głównego toku obliczeń, oraz że główny tok obliczeń może zostać przerwany w celu wykonania procedury reakcji na sygnał. Współcześnie dominujące systemy operacyjne (Unix/Linux, Windows) oraz biblioteki komunikacyjne (PVM, MPI) nie pozwalają łatwo i swobodnie stosować proponowanych asynchronicznych rozwiązań. Mechanizm sygnałów w Unix pozwala zrealizować wymagane zadania, lecz w zwykłych implementacjach bywa on powolny. Uzyskany w testach czas pomiędzy wysłaniem sygnału, a uruchomieniem procedury jego obsługi jest pokazany w tabeli 4.2. Widzimy, że pewne rozwiązania sprzętowe, np. komputery SMP na bazie procesorów Intela, oraz modyfikacje systemu operacyjnego, mogą znacznie zmniejszyć wartośc rozważanej składowej OS. Zastosowana przez nas modyfikacja jądra Linuxa polega na:

1. Zwiększeniu dynamicznego priorytetu procesu odbierającego sygnał, tak aby najbliższy przyznany kwant czasu nie był bardzo krótki.

2. Odebraniu procesora nadawcy sygnału i natychmiastowym przyznaniu procesora odbiorcy.

Powszechnie stosowane biblioteki, np. standardowa biblioteka C, PVM, MPI, zawierają funkcje niereentrantne, czyli takie, których wykonanie musi się zakończyć przed kolejnym ich wywołaniem. W rezultacie funkcje te działają niepoprawnie w środowisku współbieżnym - wykonanie takich funkcji nie może być bezpiecznie wstrzymane w celu wykonania innego fragmentu programu, po czym wznowione. Stwarza to poważny problem. Praktycznie oznacza, że pewne rejony kodu nie mogą asynchronicznie reagować na sygnały sterujące, reakcja musi poczekać aż sterowanie wyjdzie z takich rejonów (np. zakończy się wywołanie określonej funkcji). To zjawisko może znacznie zwiększyć obserwowaną wartość OS.

4.2.4.3 Wydajność obliczeniowa procesora monitora

Składowe 5 i 6 Opóźnienia Sterowania zależą od czasu wykonania stosownych obliczeń przez monitor. O ile czas działania algorytmu SCGS nie jest duży, z uwagi na prostotę algorytmu (koszt logarytmiczny względem liczby procesów, dla obsługi pojedynczego zdarzenia), to duża wydajność obliczeniowa procesora używanego przez monitor jest kluczowa do sprawnej ewaluacji złożonych predykatów.

Prędkość procesora monitora wpływa też w pewnym stopniu na inne składowe OS, np. na prędkość przełączania kontekstu przy uruchamianiu procedury obsługi sygnału oraz na czas procesora wykorzystywany do nadania/odbioru komunikatu - parametr o w modelu LogP.

4.2.4.4 Możliwości skalowania

Skalowalność jest własnością bardzo istotną w systemach równoległych. Określa ona jak bardzo można przyśpieszyć obliczenia przeznaczając do ich wykonania większą liczbę procesorów. Sprawdziliśmy zachowanie omawianego systemu sterowania w środowiskach z różną liczbą procesów przy

założeniu, że każdy proces działa na osobnym procesorze. Celem eksperymentów było ustalenie wąskiego gardła, ograniczającego możliwość skalowania systemu. Stwierdziliśmy, że dwa elementy mają krytyczne znaczenie: moc obliczeniowa procesora monitora oraz częstotliwość, z jaką monitor może przyjmować komunikaty. Przy dużej liczbie procesów zwykle wąskim gardłem staje się moc obliczeniowa procesora monitora. Koszt ewaluacji predykatów i działania algorytmu SCGS jest wtedy tak duży, że może spowodować przeciążenie procesora. Przy mniejszej liczbie procesów koszt tych obliczeń (zależny przecież od liczby procesorów) jest mniejszy. Przy zwiększaniu częstości zmian stanów lokalnych, a więc i raportów wysyłanych do monitora, procesor jeszcze będzie miał rezerwę mocy, podczas gdy interfejs sieciowy może już nie nadążać z odbiorem nawały komunikatów. Parametr g modelu LogP kontroluje ten aspekt, patrz 4.1. W sieci Ethernet g wynosi ok. 20 μs, w sieci Myrinet ok. 10μs. Poprawę skalowalności można zatem osiągnąć zwiększając moc obliczeniową procesora monitora i stosując sieci o małej wartości parametru g. Alternatywą są rozwiązania hierarchiczne, przedstawione w punkcie 2.5 i przetestowane w punkcie 4.2.5. Tam też można się zapoznać z ilościowymi aspektami skalowalności.

4.2.5 Hierarchiczne rozwiązania strukturalne sterowania

W punkcie 2.5 przedstawiliśmy szereg oczekiwanych korzyści, które miałoby przynieść zastosowanie hierarchicznej konstrukcji SCGS. W celu weryfikacji tych przypuszczeń opisany w punkcie 2.5.1 algorytm hierarchiczny został zaimplementowany w środowisku symulacyjnym i poddany badaniom. Aplikację użytą w poprzednich testach i opisaną w punkcie 4.2.2 zmodyfikowaliśmy w następujący sposób:

1. Podzieliliśmy procesy aplikacyjne na G równolicznych grup.
2. Każda grupa otrzymała swój monitor grupowy, monitory grupowe połączone zostały z monitorem głównym.
3. Monitor grupowy M_k śledził wartości $\overline{WK}_k$, $1 \leq k \leq G$, czyli średnie wartości kontrolne w swojej grupie i raportował je do monitora głównego.
4. Monitor główny wyliczał globalną średnią $\overline{WK}$ i rozsyłał ją do monitorów grupowych.
5. Każdy monitor grupowy sprawdzał, czy otrzymywane od procesów wartości WK_i mieszczą się w zadanym przedziale. Jeśli nie - do danego procesu P_i wysyłany był sygnał powodujący korektę WK_i.

Dla grupy 256 procesów przeprowadziliśmy porównanie jakości sterowania dla czterech przypadków:

Wariant C Zastosowano standardowy scentralizowany algorytm SCGS. Stany globalne są wyliczane oraz decyzje sterujące są podejmowane centralnie.

Wariant CH Stany globalne aplikacji konstruowano za pomocą opisanego w punkcie 2.5.1 algorytmu hierarchicznego SCGS. Wyznaczona przez monitor główny globalna średnia $\overline{WK}$ była dystrybuowana do monitorów grupowych. Każdy monitor grupowy sprawdzał, czy otrzymywane od procesów wartości WK_i mieszczą się w przedziale $< \overline{WK} - T, \overline{WK} + T >$. Jeśli nie - do danego procesu P_i wysyłany był sygnał powodujący korektę WK_i.

Wariant HH Sterowanie uzyskało bardziej hierarchiczny charakter. Monitor główny utrzymywał średnie grupowe $\overline{WK}_k$ w przedziale $< \overline{WK} - \frac{T}{2}, \overline{WK} + \frac{T}{2} >$, a monitory grupowe utrzymywały wartości WK_i swych procesów w przedziale $< \overline{WK}_k - \frac{T}{2}, \overline{WK}_k + \frac{T}{2} >$. Dopóki średnia grupowa $\overline{WK}_k$ monitora M_k utrzymywała się w przedziale $< \overline{WK} - \frac{T}{2}, \overline{WK} + \frac{T}{2} >$, M_k nie raportował jej nowych wartości do monitora głównego.

Tablica 4.3: Wybrane miary dla wariantu C sterowania

zdarzeń/sek	Q	CzS	OS	omp
1000	50,63	708	1,2E+09	3
2000	23,98	1413	6,3E+08	5
4000	10,35	2318	3,0E+08	10
8000	4,21	3008	1,5E+08	19
16000	1,71	2464	7,7E+07	33
32000	0,73	891	4,1E+07	54
64000	6,45E+13	80	3,5E+08	89

Wariant HHUT Tak jak dla HH, dodatkowo zastosowano algorytm SCGS UT (patrz 2.4.1, algorytm 5) zarówno w monitorach grupowych, jak i w monitorze głównym. Jednak do tej pory algorytm UT zakładał, że każde zdarzenie kończy i zarazem rozpoczyna kolejny stan lokalny danego procesu. Uwzględniono, że na poziomie monitora głównego rolę stanów lokalnych odgrywają stany silnie spójne grupy procesów, a sekwencja stanów silnie spójnych może zawierać przerwy, tj. koniec stanu silnie spójnego nie musi być początkiem następnego. Stosowne korekty zostały wprowadzone.

W rozwiązaniach hierarchicznych zastosowaliśmy grupy 32 procesowe. Podobnie jak do tej pory, symulowaliśmy komputery o mocy odpowiadającej Pentium 2GHz połączone siecią GigaEthernet i komunikujące się poprzez MPICH. Dla każdego opisanego wyżej wariantu sterowania przeprowadziliśmy serię eksperymentów zmieniając liczbę zdarzeń na sekundę zachodzących w systemie. Przy ustalonej liczbie procesów przekładało się to na zmianę długości czasu trwania stanów lokalnych procesów (podobny efekt uzyskalibyśmy ustalając czas trwania stanów lokalnych procesów, a zwiększając liczbę procesów). Z uwagi na mnogość badanych parametrów wyniki prezentujemy w postaci tabel. Oznaczenia tam występujące to:

zdarzeń/sek -łączna liczba raportowanych zdarzeń w systemie zachodzących w ciągu sekundy,

Q -miara określona w punkcie 4.2.2,

CzS -Częstotliwość Sterowania na poziomie globalnym, miara określona w punkcie 4.2.1,

$CzS\,gr$ -Częstotliwość Sterowania na poziomie grupy,

OS -Opóźnienie Sterowania na poziomie globalnym, miara określona w punkcie 4.2.1, gdzie stan spójny S jest stanem globalnym,

$OS\,gr$ -Opóźnienie Sterowania na poziomie grupy, miara określona w punkcie 4.2.1, gdzie stan spójny S jest stanem grupy procesów,

opm -obciążenie procesora monitora głównego w procentach,

$opm\,gr$ -obciążenie procesora monitorów grupowych.

Wariant C wykazał się najgorszą skalowalnością. Pojedynczy monitor centralny radził sobie z obsługą maksymalnie ok. 50 tys. zdarzeń na sekundę. Wąskim gardłem okazała się przepustowość karty sieciowej w sensie liczby obsługiwanych komunikatów na sekundę. Parametr g modelu LogP kontroluje ten aspekt, w sieci Ethernet g wynosi ok. 20 μs i stąd otrzymany wynik. Zauważmy przy tym, że obciążenie CPU monitora (kolumna opm) osiągnęło znaczącą wartość, ponad 50%. Gdyby zastosowano do sterowania bardziej złożone obliczeniowo predykaty, lub bardziej wydajną sieć, wówczas wąskim gardłem stałaby się moc procesora obsługującego monitor.

Hierarchiczna konstrukcja SCGS użyta w wariancie CH umożliwiła bezproblemową obsługę nawet 128 tys. zdarzeń na sekundę. Wartości OS uzyskane dla wariantów C i CH są niemal identyczne, co świadczy, że dodatkowe opóźnienia wywołane obecnością monitorów grupowych okazały

Tablica 4.4: Wybrane miary dla wariantu CH sterowania

zdarzeń/sek	Q	CzS	$CzS\,gr$	OS	$OS\,gr$	opm	$opm\,gr$
1000	54,36	689	99	1,3E+09	1,2E+09	6	1
2000	24,08	1365	214	7,2E+08	6,0E+08	13	3
4000	9,64	2306	439	3,4E+08	2,9E+08	23	5
8000	3,64	2992	837	1,7E+08	1,4E+08	32	7
16000	1,37	2460	1495	8,2E+07	7,1E+07	34	9
32000	0,57	863	2361	4,5E+07	3,7E+07	30	10
64000	0,38	71	3035	5,4E+07	2,1E+07	30	14
128000	0,27	70	3052	5,5E+07	1,7E+07	30	14

Tablica 4.5: Wybrane miary dla wariantu HH sterowania

zdarzeń/sek	Q	CzS	OS	$OS\,gr$	om	$om\,gr$
1000	54,44	698	1,3E+09	1,1E+09	7	1
2000	24,50	1368	7,3E+08	6,0E+08	13	3
4000	10,04	2247	3,4E+08	3,0E+08	22	4
8000	4,66	2918	2,0E+08	1,5E+08	30	7
16000	2,10	2955	1,2E+08	7,9E+07	34	9
32000	2,82	2189	9,0E+07	4,8E+07	32	12
64000	5,56	925	7,6E+07	3,0E+07	27	16
128000	5,57	907	6,5E+07	2,5E+07	26	16

się mało istotne. Uzyskaliśmy lepszą wartość miary Q (0,27), dzięki dobrej jakości sterowania na poziomie lokalnym ($CzS\,gr = 3052$), niestety przy globalnej Częstotliwości Sterowania wynoszącej zaledwie $CzS = 70$. Tak niska globalna Częstotliwość Sterowania spowodowana została tym, że silnie spójne stany grup procesów trwały zwykle bardzo krótko i pomiędzy nimi występowały dłuższe przerwy. Rzadko więc zdarzało się, aby okresy trwania tych stanów ze wszystkich grup pokryły się, tworząc tym samym globalny stan silnie spójny. Innymi słowy liczba zdarzeń na sekundę w stosunku do dokładności synchronizacji zegarów stała się tak duża, że wartość CzS dramatycznie zmalała (patrz 4.2.3). Sytuację mogły poprawić dwie rzeczy: ulepszenie dokładności synchronizacji zegarów, oraz zmniejszenie liczby zdarzeń na sekundę. Postanowiliśmy zająć się drugą możliwością, jako nie ingerującą w środowisko obliczeniowe, i nie tak oczywistą, jak pierwsza.

Wariant HH zakładał, że monitory grupowe nie raportowały swych średnich grupowych przy każdym ich wyliczeniu (przy każdym grupowym stanie silnie spójnym), lecz jedynie, gdy średnie te odchylały się poza przyjęty przedział tolerancji. Czas, w którym średnia $\overline{WK}_k$ utrzymywała się w przedziale tolerancji mógł być stosunkowo długi. Z punktu widzenia monitora głównego był to pojedynczy, długotrwały stan grupy - monitor M_k raportował tylko jego początek i koniec (wartość średniej po wyjściu poza dozwolony przedział). W efekcie liczba zdarzeń zmniejszyła się i - zgodnie z oczekiwaniem - poprawiła się wartość CzS (do 907). Niestety wartość miary Q nie osiągnęła tak niskich wartości, jak w poprzednim przypadku. Założyliśmy, że problem stanowi zbyt duże Opoźnienie Sterowania OS. Stosowany tu algorytm T (w wersji hierarchicznej), jak wykazaliśmy punkcie 4.2.2.2, jest mocno wrażliwy w sensie miary OM i co za tym idzie OS, na zwiększenie wartości stddev(sl), a taki efekt powodują sporadycznie występujące długotrwałe stany lokalne. Podejrzewaliśmy, że w rezultacie decyzje sterujące następowały zbyt późno i nie nadążały za zmieniającą się sytuacją.

Testy z wariantem HHUT potwierdziły powyższą hipotezę. Zastosowanie algorytmu UT umożliwiło uwzględnianie długotrwałych stanów lokalnych (grupowych) już w trakcie ich trwania. Co prawda maksymalne opóźnienie przesłania komunikatu z procesu przez monitor grupowy do monitora głównego okazało się być bardzo znaczące, ponad 10 razy większe niż w relacji proces -

Tablica 4.6: Wybrane miary dla wariantu HHUT sterowania

zdarzeń/sek	Q	CzS	OS	$OS\,gr$	om	$om\,gr$
1000	55,27	821	5,9E+07	1,6E+07	10	3
2000	27,15	1 473	3,6E+07	1,5E+07	18	5
4000	11,93	2 284	3,3E+07	1,5E+07	29	8
8000	4,39	2 981	3,4E+07	1,6E+07	41	12
16000	1,70	2 985	3,5E+07	1,8E+07	49	15
32000	0,44	2 188	3,0E+07	1,4E+07	47	19
64000	0,08	1 740	3,0E+07	1,4E+07	40	25
128000	0,04	1 685	3,0E+07	1,4E+07	40	26

monitor grupowy, lecz wyniki i tak okazały się znakomite. Opóźnienie Sterowania na poziomie globalnym zmalało dwukrotnie do poziomu $3 * 10^7$ cykli, podobnie niemal dwukrotnie na poziomie grupy do $1,4 * 10^7$. W końcowym efekcie osiągnęliśmy $Q = 0,04$ przy $CzS = 1685$.

W punkcie 2.5 przedstawiliśmy szereg oczekiwań odnośnie hierarchicznego wykrywania stanów spójnych aplikacji oraz powiązanego z tym sterowania. W zakresie objętym opisanymi powyżej testami oczekiwania te zostały potwierdzone.

- Obciążenie CPU pojedynczego monitora zostało rozłożone pomiędzy monitory grupowe i monitor główny.
- Wąskie gardło, jakie stanowiła ograniczona przepustowość interfejsu sieciowego monitora, zostało usunięte dzięki rozdzieleniu strumienia komunikatów pomiędzy monitory grupowe.
- Szybkość reakcji systemu sterowania na poziomie grupy (przedstawiona w tabelach w kolumnie $OS\,gr$) okazała się nawet dwukrotnie lepsza, niż szybkość sterowania centralnego w wariancie C (przedstawiona w kolumnie OS).

Osiągnięcie dobrych rezultatów sterowania wymagało nie tylko użycia hierarchicznej konstrukcji SCGS, ale też zastosowania hierarchicznego sterowania. Monitory grupowe podejmowały decyzje sterujące na podstawie stanów spójnych grupy procesów, dzięki czemu na poziomie lokalnym decyzje były podejmowane szybciej i częściej. Monitor główny przekazywał monitorom grupowym informacje globalne, pozwalając im dostosować lokalne decyzje sterujące do stanu całości aplikacji.

Dodatkowo rozwiązania hierarchiczne pozwoliły na częściowe uniknięcie problemu redukcji Częstotliwości Sterowania przy rosnącej liczbie zdarzeń na sekundę, opisanego w punkcie 4.2.3. Efekt ten widoczny jest najlepiej dla wariantu CH w tabeli 4.4 w kolumnach CzS i $CzS\,gr$. Na poziomie globalnym wartości CzS stawały się niskie, podczas gdy na poziomie grupy utrzymywały wartości wysokie. Dla wariantów HH i HHUT kolumna $CzS\,gr$ prezentowała się identycznie jak w przypadku CH, toteż została już pominięta.

4.2.6 Podsumowanie

Dzięki przeprowadzonym badaniom udało nam się określić ilościowo efektywność sterowania w programach równoległych przy zastosowaniu systemu sterowania wykorzystującego wybrane algorytmy wykrywania SCGS. Uzyskane rezultaty umożliwiają dobór odpowiednich algorytmów wykrywania SCGS do konkretnego zadania. Możliwe jest też oszacowanie oczekiwanej efektywności sterowania, wyrażonej w terminach zaproponowanych miar efektywności, dla określonego środowiska i aplikacji.

Eksperymenty pozwoliły wyodrębnić najistotniejsze aspekty efektywności sterowania przez predykaty globalne. Zauważone potencjalne problemy, takie jak spadek częstotliwości sterowania, udało się skorygować wprowadzając odpowiednie rozwiązania, np. sterowanie hierarchiczne.

Większość zaprezentowanych w tym rozdziale rezultatów została opublikowana w pracach [22, 23]. Badania efektywności hierarchicznego wykrywania SCGS oraz hierarchicznego sterowania zostały opublikowane w pracach [20, 19].

Rozdział 5

Realizacja sterowania opartego na predykatach stanów spójnych aplikacji w systemie graficznego projektowania PS-GRADE

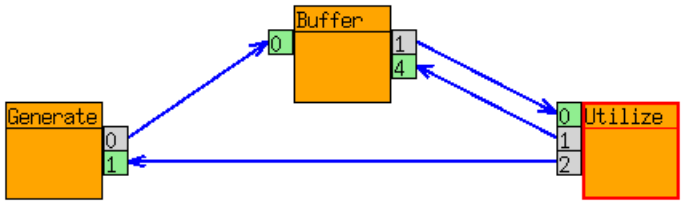
P-GRADE jest kompletnym graficznym systemem projektowania aplikacji równoległych opartych o przesyłanie komunikatów, stworzonym w Laboratorium Systemów Równoległych i Rozproszonych Instytutu SZTAKI Węgierskiej Akademii Nauk [69, 70]. P-GRADE umożliwia graficzne zaprojektowanie programu, automatyczną translację powstałej graficznej specyfikacji na język C, kompilację do postaci programu wykonywalnego, dystrybucję kodu w obrębie klastra oraz uruchomienie. Projektowanie programu rozłożone jest na trzy etapy, odzwierciedlające rozpatrywany poziom szczegółowości.

W etapie pierwszym definiuje się procesy oraz połączenia pomiędzy nimi, patrz rysunek 5.1. Prostokaty reprezentują procesy, linie je łączące to kanały przesyłania komunikatów, kanały rozpoczynają się i kończą portami komunikacyjnymi przypisanymi poszczególnym procesom.

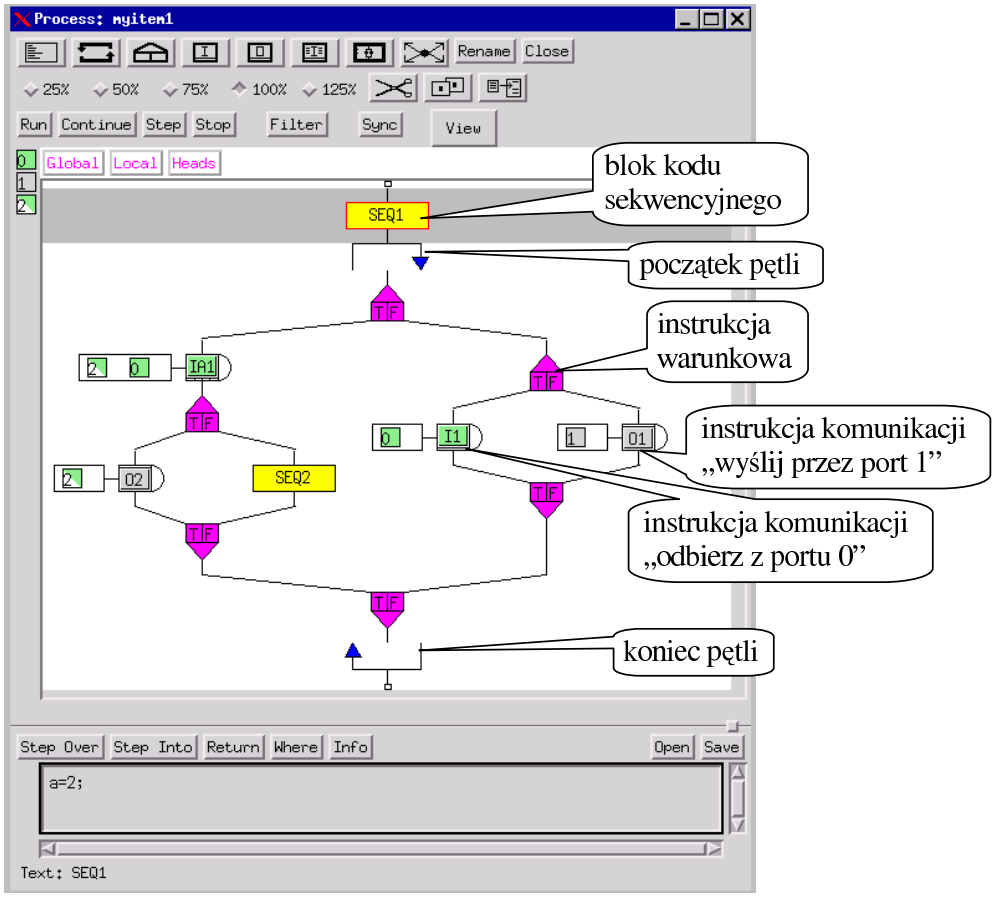
W drugim etapie dla każdego procesu określa się sposób jego działania poprzez narysowanie grafu przepływu sterowania. Przykład widzimy na rysunku 5.2. Instrukcje sterujące takie jak pętle czy instrukcje warunkowe, otrzymują reprezentację graficzną. Dodatkowo specjalne węzły grafu oznaczają instrukcje komunikacji. Instrukcje takie skojarzone są z określonym portem komunikacyjnym procesu, a poprzez ten port i przyporządkowany kanał z innym procesem, oraz ze zmienną, której wartość mają przesłać (odebrać). Przedstawione graficznie instrukcje komunikacji są zamieniane w procesie kompilacji na wywołania funkcji biblioteki przekazywania komunikatów (MPI lub PVM).

Etap trzeci to wypełnienie grafów przepływu sterowania kodem sekwencyjnym. Do węzłów grafu widocznych na rysunku5.2 jako prostokąty nazwane „SEQ“ wstawia się zwykły kod sekwencyjny w języku C. Fragmenty kodu procesu nie zawierające instrukcji komunikacji można zdefiniować tekstowo, bez rysowania odpowiadającego im grafu. Można też w ten sposób użyć gotowego istniejącego sekwencyjnego kodu, wstawiając go w odpowiednie miejsce definiowanych procesów, przyśpieszając tworzenie aplikacji równoległej.

System P-GRADE stanowił dla nas materiał wyjściowy. Pierwsze prace nad jego rozszerzeniem w kierunku zwiększenia możliwości synchronizacji procesów podjęte zostały w [119]. Zaproponowano tam dodanie rodzaju grafu przepływu sterowania na poziomie aplikacji, ustalającego podział obliczeń na fazy i warunki przejścia pomiędzy fazami. Nasze wysiłki poszły w innym kierunku, koncentrując się nad implementacją proponowanych w tej pracy mechanizmów sterowania.



Rysunek 5.1: Przykład schematu procesów i ich połączeń w P-GRADE



Rysunek 5.2: Okno edytora grafu przepływu sterowania pojedynczego procesu w P-GRADE z przykładowym grafem

5.1 PS-GRADE - system P-GRADE z synchronizatorami

Naszym celem było stworzenie w pełni funkcjonalnego systemu programowania równoległego wykorzystującego sterowanie wykonaniem programów opartego na predykatach stanów spójnych aplikacji. Aby to osiągnąć, P-GRADE musiał zostać rozszerzony w dwóch obszarach:

- Trzeba było dodać infrastrukturę monitorowania stanów aplikacji, wraz z monitorem wartościującym zadane predykaty globalne.
- Graf przepływu sterowania opisujący działanie procesu musiał zostać rozszerzony o definicję sposobu reakcji na sygnały sterujące, informujące o spełnieniu danego predykatu.

Wyczerpujący opis systemu PS-GRADE można znaleźć w pracach [74, 73]. Poniżej podajemy opis skrótowy.

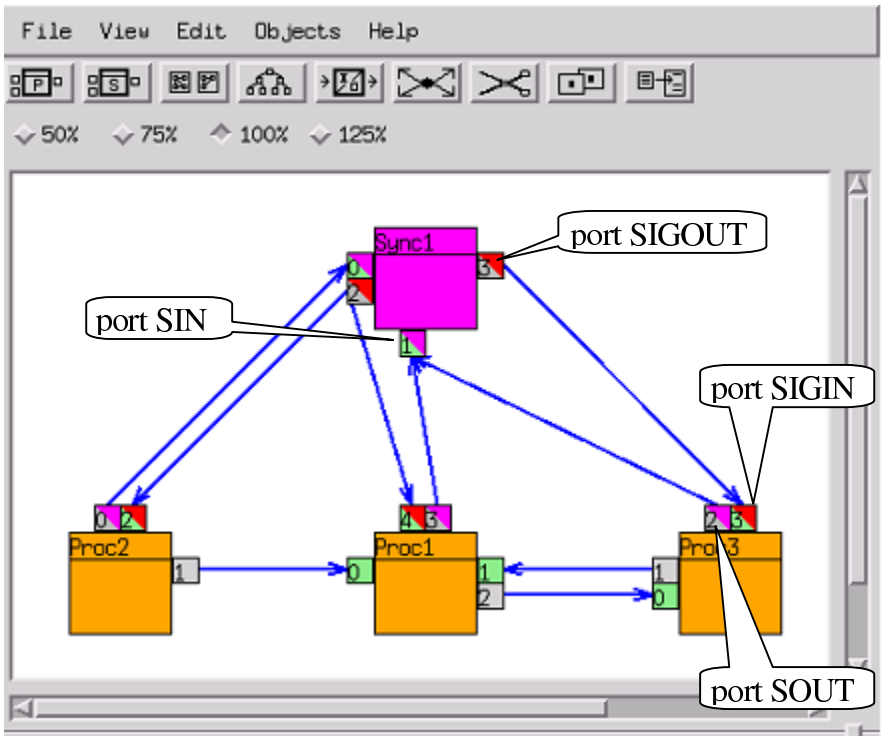
5.1.1 Monitorowanie stanów aplikacji

Na rysunku 5.3 widzimy przykładową aplikację zaprojektowaną w PS-GRADE. Składa się ona z trzech procesów oraz dodatkowego, specjalnego procesu monitorującego stany tej aplikacji i podejmującego na ich podstawie decyzje sterujące. W PS-GRADE proces taki nazywany jest synchronizatorem. Specjalnie nowe rodzaje portów komunikacyjnych służą do łączenia procesów aplikacyjnych z synchronizatorem. Porty te są zakończeniem kanałów komunikacyjnych. Kanały te, tak jak i kanały standardowe, rysuje programista. Procesy wysyłają raporty o swym stanie poprzez porty typu SOUT, a synchronizator odbiera je przez porty SIN. Raporty są automatycznie znakowane czasem, aby umożliwić konstrukcję silnie spójnych stanów aplikacji. Synchronizacja zegarów komputerów w klastrze obliczeniowym nie wchodzi obecnie w zakres funkcjonalności PS-GRADE i jest realizowana niezależnie przez protokół NTP [91], zobacz też punkt 2.1.2.

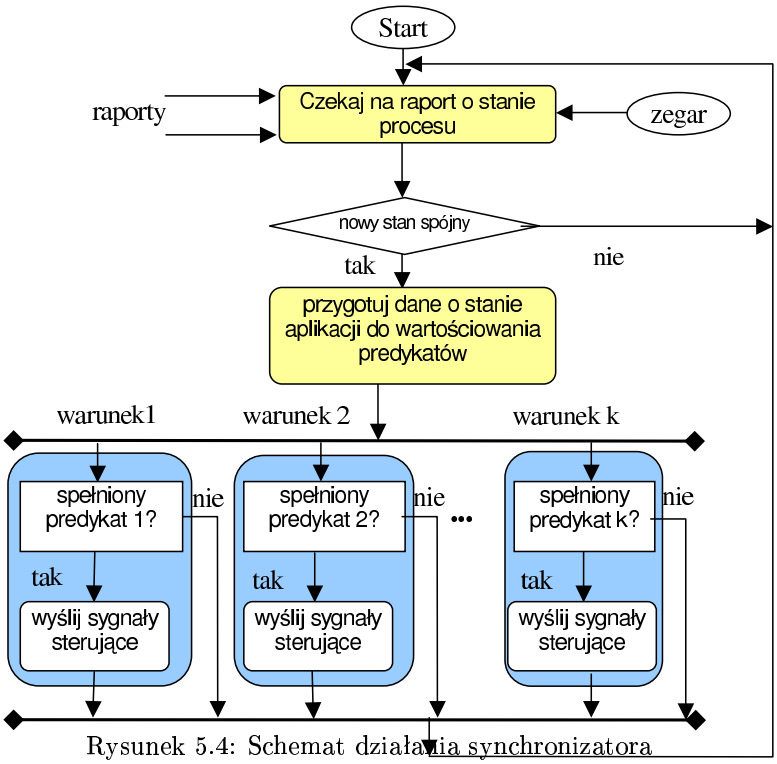
Sygnały sterujące synchronizator przesyła poprzez porty typy SIGOUT, a ich odbiór w procesach następuje przez porty SIGIN. Zwykle raporty dotyczące pojedynczego aspektu stanu procesów, np. długość kolejki zadań, są przesyłane do osobnego portu SIN synchronizatora, natomiast raporty dotyczące innego aspektu, np. znalezione w procesie najlepsze rozwiązanie, odbierane są przez inny port SIN. To podejście, jak opisano dokładniej poniżej, pozwala na łatwą selekcję istotnych elementów stanu procesu przy podejmowaniu określonych decyzji sterujących.

Programista może utworzyć wiele synchronizatorów i poprzydzielać im osobne zadania, przy czym dany synchronizator może obsługiwać tylko pewien podzbiór procesów. Dodatkowo programista definiuje które raporty (z których portów SIN oraz od których procesów dołączonych do tych portów) mają być użyte przy tworzeniu stanów spójnych aplikacji. Takich definicji może być wiele, każda określa tzw. *region*. Najprostszym regionem jest region globalny, dla którego brane są wszystkie raporty od wszystkich procesów. Gdy zachodzi taka potrzeba, przez utworzenie odpowiedniego regionu możemy posługiwać się *regionalnymi* stanami spójnymi budowanymi na podstawie wybranej części raportów np. przesyłanych tylko od procesów P_1 i P_2 na port SIN o numerze 1. W ten sposób spośród wszystkich procesów oraz wszystkich monitorowanych aspektów ich pracy możliwe jest wybranie potrzebnego podzbioru, co upraszcza projektowanie sterowania i ogranicza koszt realizacji takiego sterowania.

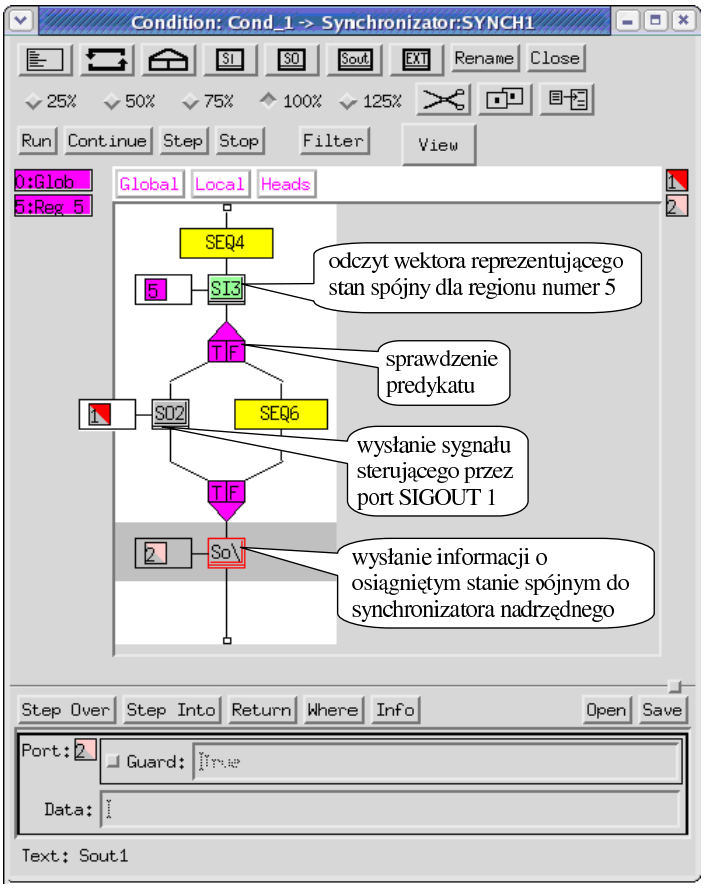
Schemat działania synchronizatora ilustruje rysunek 5.4. Każdorazowo po odebraniu raportu o zdarzeniu w procesie (każde zdarzenie kończy poprzedni i zaczyna następny stan lokalny procesu), sprawdane jest, w których zdefiniowanych regionach osiągnięto stan spójny. Zegar umożliwia zastosowanie algorytmu SCGS opisanego w punkcie 5. Jeśli osiągnięto stan spójny danych regionie, to wartościowane są wybrane predykaty określone na tym regionie. Mechanizm warunków wstępnych ewaluacji predykatów pozwala związać ewaluację danego predykatu z wystąpieniem określonego regionalnego stanu spójnego. W ten sposób inne predykaty mogą być sprawdzane gdy np. wykryliśmy spójny stan procesów $P_1..P_4$ ze względu na raporty odbierane przez port SIN 1, a inne gdy pojawił się stan spójny np. procesów $P_1..P_3$ ze względu na raporty odbierane przez port SIN 2. Możliwe jest także wartościowanie predykatów na stanach obserwowanych zamiast na stanach spójnych, zgodnie z opisem z punktu 3.1.1.1.



Rysunek 5.3: Synchronizator monitorujący aplikację w PS-GRADE



Rysunek 5.4: Schemat działania synchronizatora



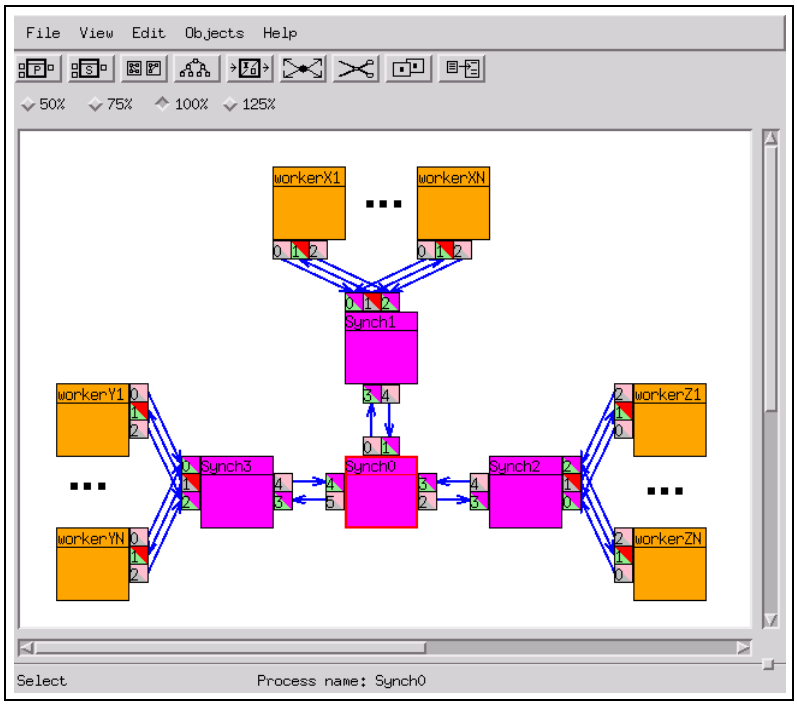
Rysunek 5.5: Przykład definicji predykatu w PS-GRADE

Programista definiuje predykaty posługując się edytorem grafu przepływu sterowania, podobnie jak przy definiowaniu zwykłego procesu. O ile jednak procesowi odpowiada pojedynczy graf, tak w synchronizatorze dla każdego predykatu mamy osobny graf przepływu sterowania. Przykład widzimy na rysunku 5.5. Stan poszczególnych procesów składający się na wybrany regionalny stan spójny odczytywany jest jako wektor z portu reprezentującego odpowiedni region. Treść predykatu sterującego zaimplementowana jest jako graf przepływu sterowania, wykonujący pewne obliczenia na odczytanym wektorze. W grafie tym można umieścić instrukcje wysłania sygnałów sterujących do procesów. W systemie nie ma ograniczeń na rodzaj czy treść definiowanych predykatów. Kod wyliczający określa programista w sposób dowolny. Możliwe jest m.in. zapamiętywanie poprzednich stanów globalnych i uwzględnianie ich historii. W ten sposób można np. zapewnić, że dany sygnał sterujący będzie wysłany do określonego procesu tylko raz w całej sekwencji stanów globalnych zawierających niezmieniony stan lokalny tego procesu.

Synchronizatory można łączyć w hierarchie, zgodnie z ideą opisaną w punkcie 2.5, co widać na rysunku 5.6. Wówczas w synchronizatorach podrzędnych w grafy definiujące sposób ewaluacji predykatów można wstawić instrukcję wysłania określonej danej, reprezentującej stan grupy procesów, do synchronizatora nadrzędnego. Przykład widzimy na rysunku 5.5.

5.1.2 Określenie sposobu reakcji procesu na sygnały sterujące

Graf przepływu sterowania definiujący sposób działania procesu został rozszerzony o nowe elementy pozwalające określić reakcję procesu na sygnały sterujące. Zaimplementowane reguły przyjmowania sygnałów odpowiadają propozycji opisanej w punkcie 3.3.2.1. Rzecz ilustruje przykład



Rysunek 5.6: Hierarchia synchronizatorów w PS-GRADE

na rysunku 5.7. Poniżej wyjaśniamy widoczne tam oznaczenia numeryczne:

1. Węzeł oznaczony numerem 1 oznacza początek regionu kodu procesu, w którym mogą być przyjmowane sygnały sterujące przychodzące na port SIGIN 1, jednocześnie w węźle tym określa się, którym zmiennym system ma przypisać dane przesłane razem z sygnałem.

2. Gałąź grafu oznaczona numerem 2 definiuje normalny tok obliczeń.

3. Gałąź oznaczona numerem 3 jest aktywowana asynchronicznie przybyciem sygnału sterującego. W czasie jej wykonania obliczenia realizowane w gałęzi 2 są zawieszane. Przesłane razem z sygnałem dane są dostępne jako wartości zmiennych wyspecyfikowanych w węźle 1.

4. Węzeł 4 oznacza polecenie zakończenia obsługi sygnału i wznowienie zwieszonych obliczeń.

5. Węzeł 5 to polecenie zakończenia obsługi sygnału i porzucenia zawieszonych obliczeń, sterowanie programu zostanie przeniesione do węzła numer 8.

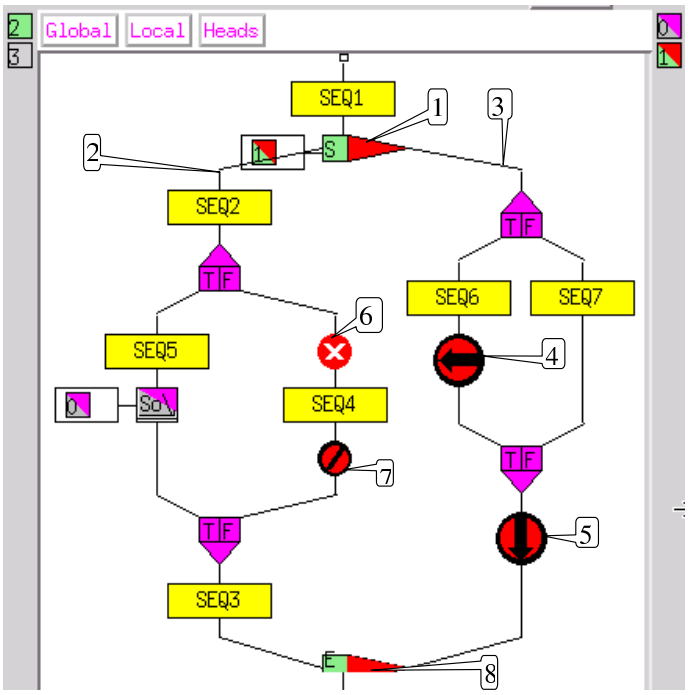
6. Węzeł 6 to czasowe zablokowanie możliwości reagowania na sygnały, sygnały zamiast być obsługiwane od razu, będą kolejkowane.

7. Węzeł 7 to zniesienie tej blokady, zakolejkowane sygnały zostaną tu obsłużone.

8. Węzeł 8 oznacza koniec regionu kodu procesu, w którym mogą być przyjmowane sygnały sterujące przychodzące na port SIGIN 1.

5.1.3 Techniczne aspekty realizacji systemu PS-GRADE

Rozszerzenie systemu P-GRADE do PS-GRADE wymagało znacznej pracy programistycznej. Jej szczegóły są opisane w pracach [68, 74, 95]. Tutaj przedstawimy tylko fragmentaryczny zarys, ze szczególnym uwzględnieniem zagadnień opracowanych przez autora tej pracy.



Rysunek 5.7: Przykład definicji sposobu reakcji na sygnały sterujące

System został zaimplementowany w środowisku Linux. Synchronizację zegarów poszczególnych komputerów zapewnia usługa NTP [91]. Odpowiednie rozszerzenie funkcji wysłania komunikatu istniejącej w P-GRADE automatycznie znakuje wysyłane do synchronizatora raporty znacznikiem czasu.

Synchronizator jest osobnym procesem oczekującym na raporty od procesów. Po każdorazowym odbiorze raportu uruchamiany jest moduł wykrywania silnie spójnych stanów dla dostarczonych definicji regionów. Moduł ten zwraca informację czy i które stany spójne zostały osiągnięte. Moduł implementuje algorytm SCGS wykorzystujący górne ograniczenie na czas przesłania raportu (algorytm 5).

Sygnały przekazywane są do docelowych komputerów jako komunikaty zarządzane przez bibliotekę PVM. Na miejscu są one odbierane przez dedykowany proces systemowy. Proces ten wstawia definicję (numer portu SIGIN) odebranego sygnału sterującego oraz skojarzone z nim (i przysłane w komunikacie dane) do obszaru pamięci dzielonej. Następnie sygnał czasu rzeczywistego POSIX 1b (SIGRT) wysyłany jest do docelowego procesu obliczeniowego, w celu powiadomienia go o przybyłym sygnale sterującym. Wartość skojarzona z sygnałem SIGRT określa obszar pamięci dzielonej, do którego wpisano stosowne informacje. Sygnał SIGRT, w przciwieństwie do innych sygnałów systemów UNIX/Linux mają gwarancję doręczenia i pozwalają na przesłanie wraz z nimi danej. Są to cechy bardzo ważne w naszym systemie.

Wysłanie sygnału SIGRT powoduje, że wykonanie procesu obliczeniowego jest przerywane przez system operacyjny i uruchamiana jest zarejestrowna w systemie operacyjnym procedura obsługi sygnału SIGRT. Procedura ta sprawdza, czy proces wykonuje aktualnie region kodu wrażliwy na sygnały sterujące, jeśli tak to dane z pamięci dzielonej są odczytywane i uruchamiona zostaje procedura obsługi sygnału sterującego (zdefiniowaną jako gałąź grafu oznaczona numerem 3 na rysunku 5.7).

Wejście sterowania programu w region wrażliwy na sygnały sterujące odnotowywane jest poprzez ustawienie odpowiedniej flagi oraz wykonanie funkcji `sigsetjmp()` ze standardowej biblioteki C. Funkcja ta, wraz z funkcją `siglongjmp()` implementuje mechanizm porzucenia obliczeń. Węzeł porzucenia obliczeń (numer 5 na rysunku 5.7) przekształcany jest na wywołanie

funkcji `siglongjmp()`, która to przenosi sterowanie do miejsca zapamiętanego uprzednio przez `sigsetjmp()`, czyli do początku regionu wrażliwego na sygnał. Cały region zamknięty jest w instrukcji warunkowej `if`. Instrukcja ta sprawdza, czy osiągnięto ją za pomocą skoku przez `siglongjmp()`. Jeśli tak, to fragment kodu zawierający region wrażliwy na sygnał jest pomijany.

Czasowe blokowanie przyjmowania sygnałów realizowane jest przez ustawianie odpowiedniej maski sygnałów na poziomie systemu operacyjnego. W ten sposób blokuje się obsługę sygnału SIGRT, sygnały te są kolejkowane przez system operacyjny do momentu przywrócenia maski pozwalającej na obsługę tego sygnału.

5.1.4 Podsumowanie realizacji systemu PS-GRADE

Dodanie sterowanie opartego o predykaty określone na globalnych stanach aplikacji w istotny sposób rozszerzyło możliwości systemu P-GRADE. Graficzny charakter systemu został w pełni zachowany. Wprowadzone synchronizatory i dodatkowe elementy służące sterowaniu udało się bardzo dobrze zintegrować z wyjściowym systemem. Stosunkowo niewielkie modyfikacje w interfejsie użytkownika okazały się wystarczające, przez co sposób pracy z systemem niewiele się zmienił - nowe możliwości sterowania udało się wyrazić przy tylko niewielkim poszerzeniu środków dostępnych programiście. To potwierdza zasadność wyboru systemu P-GRADE jako bazy do implementacji systemu sterowania opartego o predykaty globalne.

Wprowadzone asynchroniczne reakcje na sygnały sterujące bardzo znacznie powiększyły możliwości systemu P-GRADE, szczególnie w zakresie efektywności realizacji nieregularnych aplikacji równoległych, charakteryzujących się nieprzewidywalnymi wzorcami komunikacji. Asynchroniczne reakcje zastępują skutecznie nieistniejące w P-GRADE instrukcje nieblokującego odbioru (lub testu przybycia) komunikatu.

W trakcie realizacji napotkaliśmy na szereg problemów technicznych, głównie związanych z asynchroniczną reakcją na sygnały. Taka funkcjonalność, dostępna swobodnie programiście jako jeden z podstawowych mechanizmów sterowania, nie jest standardem w powszechnie stosowanych językach programowania i co za tym idzie, nie jest bezpośrednio obsługiwana przez system operacyjny. Udowodnilismy jednak, że tę funkcjonalność można zrealizować w standardowym środowisku systemowo sprzętowym.

Skromne środki, przy użyciu których zrealizowano system PS-GRADE, oraz rezygnacja z jakichkolwiek ponadstandardowych elementów w wykorzystywanym środowisku systemowo-sprzętowym pozwalają sądzić, że istnieje jeszcze szerokie pole do poprawy wydajności zastosowanych rozwiązań. Efektywność opisanej w niniejszej pracy implementacji sterowania opartego o predykaty globalne w systemie PS-GRADE, na podstawie wybranych aplikacji, została przedstawiona w rozdziale 6.

System PS-GRADE został opisany w wielu publikacjach, m.in. w [24, 27, 28, 117].

Rozdział 6

Badania efektywności proponowanej metody sterowania w zastosowaniach

6.1 Przegląd wybranych zastosowań proponowanej metody sterowania

Na wstępie tego rozdziału dokonamy krótkiego przeglądu możliwych zastosowań proponowanej metody sterowania. Krótko zatrzymamy się na poszczególnych przykładach wskazując jak moga być one zaimplementowane przy zastosowaniu sterowania przez predykaty globalne. Następnie, dla wybranych zastosowań, przedstawimy szczegółowe badania efektywności zastosowanego sterowania, korzystając z wyników rzeczywistych obliczeń oraz symulacji.

Sterowanie w obliczeniach nieregularnych W obliczeniach równoległych, w których zależności pomiędzy składowymi procesami są nieprzewidywalne, konieczne jest podejmowanie decyzji sterujących na bieżąco. Gdy decyzje te zależą od wszystkich lub od określonego podzbioru procesów aplikacyjnych, wtedy warto wykorzystać proponowany w rozprawie mechanizm sterowania. Przeszukiwanie przestrzeni rozwiązań z eliminacją obszarów nieperspektywicznych jest przykładem tego rodzaju obliczeń. Decyzje o eliminacji takich obszarów zależą od wyników przeszukiwania we wszystkich procesach. Infrastruktura proponowanego mechanizmu sterowania zapewnia sprawną propagację stosownych informacji i umożliwia szybką reakcję na nie. Przykłady obliczeń nieregularnych w postaci implementacji algorytmu dziel i ograniczaj (*branch and bound*) oraz całkowania adaptacyjnego opisaliśmy szczegółowo w punktach odpowiednio 6.2 i 6.3. Dodatkowo, w obliczeniach nieregularnych konieczne jest równoważenie obciążenia w celu zapewnienia dobrej wydajności obliczeń - patrz punkt 6.4.

Koordynacja współpracujących aplikacji W systemach rozproszonych często zdarza się konieczność zapewnienia współpracy osobnych aplikacji. Gdy współpraca ta ma mieć charakter bardziej skomplikowany niż proste następstwo, jej organizacja staje się problematyczna. Po części tego typu problematyką zajmowano się w pracy [86], modyfikując istniejące aplikacje tak, aby mogły one współpracować z nadzorującym je systemem sterującym. W naszym podejściu zakładamy, że aplikacje nadzorowane to programy równoległe już wyposażone w lokalne mechanizmy sterowania przez predykaty wyznaczane na stanach globalnych, a hierarchicznie nadrzędny mechanizm zarządza całym zbiorem takich aplikacji. O hierarchicznych metodach sterowania pisaliśmy w punktach 2.5, 2.6 i 4.2.5. Tu metody te są wykorzystywane.

Koordynacja działania równoległych aplikacji ma szczególne znaczenie w środowisku Grid, z uwagi na dynamiczny rozwój technologii Grid i jej zastosowań. Zwrócono na to uwagę w [83] rozszerzając system GRADE o mechanizm przepływu pracy (workflow) pozwalający projektować ciągi aplikacji, uruchamianych sukcesywanie w Grid. Zastosowane rozwiązanie ma jednak ograni-

czone możliwości wyrazu. W pracach [76, 75] zajęliśmy się problematyką koordynacji aplikacji w Grid w zakresie znacznie szerszym niż badania dotychczasowe. Odpowiednio rozbudowując system PS-GRADE umożliwiliśmy realizację następujących wzorców sterowania w Grid:

1. Uruchomienie danej aplikacji zależnie od łącznego stanu innych, już działających aplikacji. Możliwa jest reakcja oparta na dowolnym warunku, przedstawionym jako predykat. Programista decyduje, jakie aspekty stanów aplikacji mają znaczenia dla sterowania i w związku z tym jakie informacje powinny być przesyłane z aplikacji do synchronizatora gridowego.

2. Zakończenie danej aplikacji zależnie od łącznego stanu innych aplikacji. Aplikacja składowa powinna zostać natychmiast poprawnie zakończona, gdy wymaga tego sytuacja. Np. przy wyszukiwaniu z udziałem kilku wyszukujących aplikacji, gdy jedna z nich znalazła satysfakcjonujące rozwiązanie, wówczas pozostałe powinny zostać zatrzymane. Podobnie w obliczeniach spekulatywnych, gdy jednocześnie stosuje się wiele algorytmów rozwiązujących to samo zagadnienie, gdy pierwszy się zakończy, pozostałe należy zatrzymać.

3. Wpływ na bieżące działanie aplikacji, zależnie od łącznego stanu wszystkich (lub grupy) aplikacji. Na podstawie informacji otrzymywanych od poszczególnych aplikacji synchronizator gridowy wysyła sygnały sterujące do synchronizatorów aplikacyjnych. W każdej aplikacji synchronizatory reagują na otrzymane sygnały wykonując zdefiniowaną procedurę, która (z reguły) powoduje wysłanie sygnałów sterujących do procesów w danej aplikacji.

Wymienione wzorce można stosować w wielu obszarach, gdzie standardowy mechanizm przepływu danych (makro dataflow), odzwierciedlający tylko zależności zadań od dostępności danych, nie wystarcza. Dla każdego wybranego zastosowania można opracować osobne, dedykowane rozwiązanie sterowania. Natomiast nasza propozycja ma charakter jednego uniwersalnego modelu, który można użyć w następujących różnorodnych, przykładowych sytuacjach:

1. Długotrwałe obliczenia szukające optymalnego rozwiązania są wspomagane przez szybciej wyliczone heurystyki. Obydwa typy obliczeń są realizowane jednocześnie. Dobre rozwiązania heurystyczne są przekazywane do głównych obliczeń, powodując ich przyspieszenie.

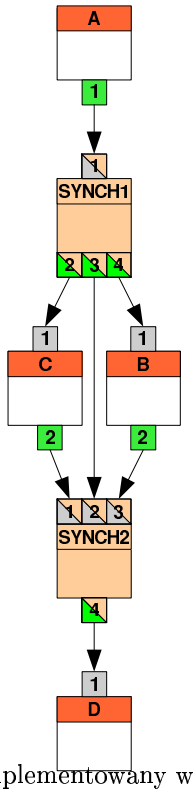
2. W zależności od stanu końcowego danej aplikacji składowej, jedna z kilku alternatywnych aplikacji jest uruchamiana aby kontunuować wieloetapowe przetwarzanie.

3. Synchronizator gridowy może zarządzać równoważeniem obciążenia na poziomie użytkownika. Może on monitorować wydajność aplikacji składowych i decydować o transferze zadań pomiędzy nimi.

4. Równoległe i rozproszone przeszukiwanie z eliminacją podprzestrzeni rozwiązań wymaga dystrybucji najlepszych rozwiązań i globalnej informacji o wyeliminowanych podprzestrzeniach. Synchronizator gridowy może wykonywać te zadania.

5. Synchronizator gridowy może zbierać raporty od aplikacji, ale także od komponentów systemowych. Otrzymując informacje o stanie środowiska może reagować na jego zmiany, np. może uruchomić wcześniej składową aplikację, gdy zwiększyły się dostępne zasoby.

Implementacja dynamicznych schematów przepływu pracy (workflow) Omówiona powyżej koordynacja aplikacji w środowisku Grid może być potraktowana jako przypadek szczególny realizacji schematów przepływu pracy (workflow), w którym poszczególne czynności są równoległymi aplikacjami. Klasyfikacja schematów przepływu pracy w oderwaniu od konketnych implementacji/zastosowań została przedstawiona w pracy [40]. Schematy te zawierały sytuacje proste, w rodzaju alternatywy następstw, bardziej złożone jak np. pętle czy aktywacja wielu kopii danej czynności, oraz najbardziej zaawansowane, wymagające wiedzy o stanie fragmentów systemu w celu prawidłowej dynamicznej interpretacji dalszej części danego schematu. Wyodrębniona została ponadto kategoria schematów zawierających zaprzestanie - dezaktywację pojedynczej lub wskazanej grupy czynności.



Rysunek 6.1: Schemat przepływu implementowany w rozszerzonym systemie PS-GRADE

Dla wszystkich tych schematów zaproponowaliśmy ich implementacje w rozszerzonym środowisku PS-GRADE, wykorzystując synchronizatory jako elementy decyzyjne w schematach przepływu [25]. Synchronizatory zbierają informacje o stanie tylko tych czynności, od których zależy wykonanie schematu, zbieranie informacji o stanie globalnym systemu nie jest tu celowe. Dla przykładu, na rys. 6.1 widzimy schemat realizujący dwa wzorce: synchronizujące złączenie (ang. synchronizing merge) i wielo-złączenie (ang. multi merge) bez synchronizacji. Realizowany wariant określony jest predykatami zdefiniowanymi w synchronizatorach. Synchronizujące złączenie polega na oczekiwaniu na zakończenie wszystkich rozpoczętych ścieżek (na rysunku mamy dwie ścieżki z czynnosciami odpowiednio C i B) przed uruchomieniem czynności dalszych (D na rysunku). Synchronizator SYNCH2 musi wiedzieć, czy w danym wykonaniu wzorca ma oczekiwać na zakończenie ścieżki lewej, prawej, czy obu. Informacja ta jest mu przekazywana bezpośrednim kanałem od synchronizatora SYNCH1, odpowiedzialnego za aktywację ścieżek.

Klasyczne schematy przepływu pracy traktują czynność w schemacie przepływu w sposób atomowy - jest to czarna skrzynka, którą można tylko uruchomić i oczekiwać na zakończenie jej działania. Przez dynamiczne schematy przepływów rozumiemy takie schematy, w których decyzje aktywowania czynności są podejmowane przed zakończeniem czynności poprzedzających, na podstawie informacji o wewnętrznym stanie aktualnie aktywnych czynności. Umożliwia to nowe podejście do zadań, w których czynności trwają długo, są złożone, i które trudno traktować jako niepodzielne całości. Zaproponowaliśmy dwa rodzaje dynamicznych schematów przepływu:

Wspomagający schemat przepływu: zadanie realizowane przez zbiór aktywnych czynności wymaga uruchomienia dodatkowych, wspomagających czynności. Rezultat działania czynności wspomagających przekazywany jest czynnościom zasadniczym. Np. w zgrubnej symulacji poruszających się obiektów nastapiła kolizja. Precyzyjna symulacja tej kolizji wymaga uruchomienia stosowanych dodatkowych czynności. Wyniki precyzyjnej symulacji kolizji udostępniane są symulacji głównej, dzięki czemu może ona kontynuować swoje działanie.

Sprzężony schemat przepływu: globalny stan zbioru aktywnych czynności jest monitorowany.

Na podstawie tego stanu modyfikowane są parametry działania wybranych czynności.

Sterowanie procesami przemysłowymi Działanie wielu systemów przemysłowych oparte jest o ciągłe monitorowanie parametrów pracy urządzeń, oraz reagowanie na obserwowane zmiany wartości tych parametrów. Proponowana metoda sterowania pozwala użyć gotowej infrastruktury monitorującej system, badającej wybrane aspekty jego stanu oraz przekazującej polecenia sterujące do wybranych elementów systemu. Przykładem może być system wentylacji kopalni. Zespół pomieszczeń i tuneli wymaga nieustannej wentylacji. Stan bieżący atmosfery w wybranych punktach raportowany jest do centrum sterującego. Stan ten obejmuje skład powietrza (np. zawartość CO_2), ciśnienie i temperaturę. Na podstawie tego stanu można określić oczekiwany stan atmosfery w kopalni w najbliższej przyszłości, uwzględniając ruchy powietrza wywołane cisnieniem i temperaturą. Zadaniem systemu kontrolującego wentylację jest odpowiednie modyfikowanie tego oczekiwanego stanu poprzez regulację pracy systemu wentylacyjnego - tłoczenie powietrze, zamykanie/otwieranie śluz.

6.2 Problem komiwojażera

Problem komiwojażera, znany jako TSP (Traveling Salesman Problem), oznacza zadanie znalezienia jak najkrótszej (najtańszej) drogi przechodzącej przez wszystkie miasta z określonego zbioru, gdy dane są odległości (koszty) podróży pomiędzy każdą parą miast. Jest to problem NP-zupełny. Z uwagi na jego złożoność obliczeniową oraz na wielość zastosowań do optymalizacji rozmaitych zagadnień, równoległe implementacje TSP mają bardzo istotne znaczenie. Powstało wiele prac na ten temat, np. [6, 35, 63, 64].

Powszechnie wykorzystywaną metodą rozwiązania problemu TSP jest metoda dziel i ograniczaj (branch&bound, B&B) [122]. Polega ona na rozpatrywaniu drzewa potencjalnych rozwiązań (liście reprezentują pełne rozwiązanie, węzły wewnętrzne reprezentują rozwiązania częściowe), gdzie dla każdego podrzewa mamy określone dolne ograniczenie (dla problemów minimalizacji) rozwiązań mogących znajdować się w tym podrzewie. Porównując to ograniczenie ze znanym do tej pory najlepszym rozwiązaniem wiemy, czy lepsze rozwiązania mogą znajdować się w danym podrzewie. Jeśli tak, to je przeszukujemy rekurencyjnie, jeśli nie, to je porzucamy.

Efektywna równoległa implementacja B&B napotyka na kilka istotnych problemów.

- Cała przestrzeń rozwiązań musi zostać rozdzielona pomiędzy procesy równoległe.
- Dynamiczny charakter B&B powoduje, że podział ten trzeba stale uaktualniać w trakcie obliczeń, aby zapewnić zrównoważenie obciążenia, czyli aby każdy proces aplikacji miał co liczyć - niezbędny jest dynamiczne równoważenie obciążenia.
- B&B działa efektywnie, jeśli nowo znalezione najlepsze rozwiązanie jest natychmiast użyte do eliminacji nieperspektywicznych poddrzew. W przeciwnym wypadku proces może próbować znaleźć rozwiązanie w podrzewie, o którym wiadomo z globalnej perspektywy, że nie zawiera ono rozwiązania lepszego niż już znane. W implementacji równoległej bez pamięci dzielonej oznacza to konieczność bezzwłocznego rozgłaszania znalezionego rozwiązania pozostałym procesom oraz stałą gotowość do odebrania informacji o nowym rozwiązaniu.
- Pełną efektywność równoległy B&B może osiągnąć wtedy, gdy bieżąca analiza podrzewa jest przerywana i porzucana, po tym jak inny proces znalazł rozwiązanie lepsze niż dolne ograniczenie właśnie analizowanego poddrzewa. Chodzi tu o uwzględnianie nowych najlepszych rozwiązań nie tylko przy podejmowaniu decyzji czy przeszukiwać dane poddrzewo, ale też już w trakcie przeszukiwania.

Dynamiczny charakter obliczeń B&B oraz konieczność sterowania na poziomie globalnym powodują, że stanowią one dobry przykład do zilustrowania przydatności proponowanej w tej pracy metody sterowania.

6.2.1 Testy w rzeczywistych systemach

Nasze pierwsze równoległe implementacje problemu TSP w systemie P-GRADE (bez synchronizatorów) okazywały się być bardzo nieefektywne, już gdy liczba procesów przekraczała 4. W implementacjach tych procesy otrzymywały od procesu sterującego paczki zadań do rozwiązania (poddrzewa), po rozwiązaniu otrzymanego zbioru zadań przesyłały wynik i dostawały kolejną paczkę zadań wraz z globalnie najlepszym rozwiązaniem. Informacja o nowych najlepszych globalnie rozwiązaniach dosyłana była razem z kolejną paczką zadań. Im mniejsze były paczki, tym większy stawał się narzut związany z komunikacją i tym więcej czasu proces spędzał oczekując na nowe zadania. Im paczki były większe, tym rzadziej procesy były informowane o nowych najlepszych globalnie rozwiązaniach. Przypominamy, że w P-GRADE nie istnieje komunikacja nieblokująca, przy pomocy której proces mógłby sprawdzać bez zawieszania obliczeń, czy otrzymał informację o nowym najlepszym rozwiązaniu. W pierwszej testowanej wersji dla 6 procesów zdarzało się, że nawet 30% czasu obliczeń marnowane było na rozpatrywanie zadań już nieperspektywicznych, o czym dany proces mógł się dowiedzieć dopiero podczas kolejnej komunikacji.

W kolejnej implementacji B&B w P-GRADE zastosowana została bardziej efektywna dystrybucja zadań, dzięki odpowiedniemu dostosowaniu wielkości przesyłanych paczek zadań. Implementacja ta została porównana z implementacją zrealizowaną z pomocą mechnizmów globalnego sterowania w PS-GRADE.

6.2.1.1 Opis implementacji algorytmu TSP w systemie PS-GRADE

W PS-GRADE zastosowaliśmy pojedynczy synchronizator monitorujący dwa aspekty stanu aplikacji:

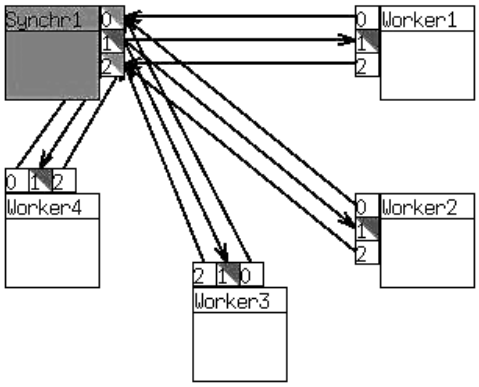
- ilość danych wejściowych każdego procesu,
- najlepsze rozwiązanie znalezione do tej pory.

Schemat połączeń pomiędzy procesami pokazany jest na rysunku 6.2. Od każdego procesu do synchronizatora wiodą dwa kanały: jednym przysyłana jest informacja o bezczynności procesu (*DataReq*) powodowanej brakiem danych wejściowych, drugim znalezione lokalnie najlepsze rozwiązania (*LocMinDist*). Kanał od synchronizatora do procesu służy do dostarczania sygnałów sterujących. Użyte są dwa typy sygnałów: *newData* informujący o dostarczeniu nowej paczki danych wejściowych, oraz *newMinDist* informujący o nowym globalnie najlepszych wyniku.

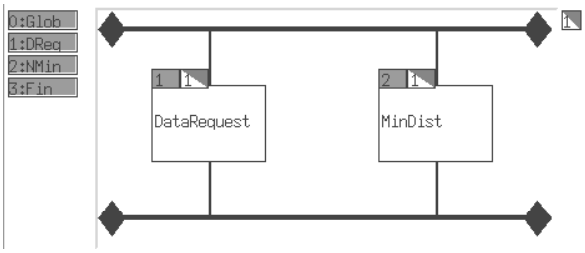
W synchronizatorze zdefiniowano dwa regiony: *Dreg* oparty na raportach o obciążeniu od wszystkich procesów, oraz *NMin* oparty na raportach o lokalnie najlepszych rozwiązaniach od wszystkich procesów. Na tych regionach określono dwa predykaty, *DataRequest* na *Dreg* i *MinDist* na *NMin*. Rysunek 6.3 pokazuje okno definicji predykatów w systemie PS-GRADE. W tym oknie tworzy się predykaty i określa kiedy (dla stanów spójnych określonych na których regionach) mają być one wartościowane. Treść predykatów precyzują diagramy przepływu sterowania. Rysunek 6.4 pokazuje te diagramy dla obydwu omawianych predykatów. Pierwsza instrukcja w nich to wczytanie wektora wartości reprezentujących stan spójny aplikacji, w którym mają być one wartościowane. Rola predykatów w sterowaniu jest następująca:

DataRequest Czy jest proces bezczynny? Jeśli tak to wyślij mu paczkę danych wejściowych. Dane są przesyłane wraz z sygnałem. Synchronizator w przedstawionym rozwiązaniu pełni rolę centralnego dystrybutora zadań. Zostało to tak zrealizowane, gdyż procesy nie mogą przesyłać zadań bezpośrednio między sobą, z uwagi na ograniczenia systemu P-GRADE - każdy proces musiałby być połączony kanałem z każdym innym.

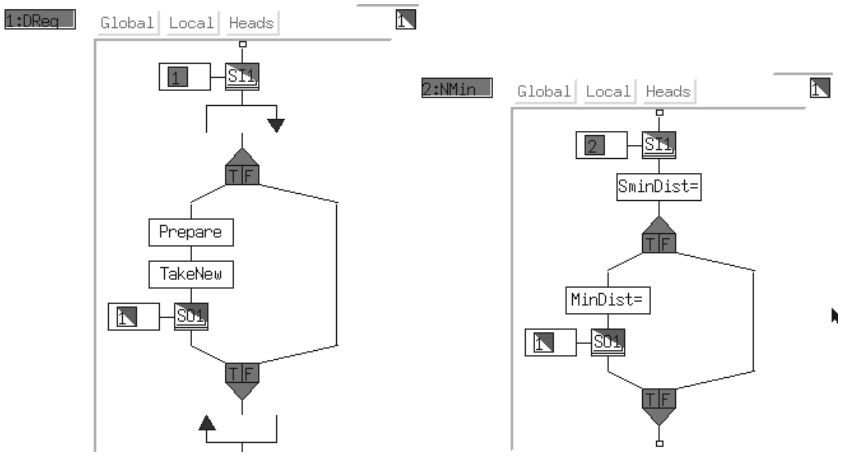
MinDist Czy rozwiązanie znalezione przez proces jest globalnie najlepsze? Jeśli tak, to roześlij je wszystkim procesom. Predykat ten wyliczany jest na stanach obserwowanych (patrz punkt 3.1.1.1), w celu przyśpieszenia dystrybucji nowych rozwiązań.



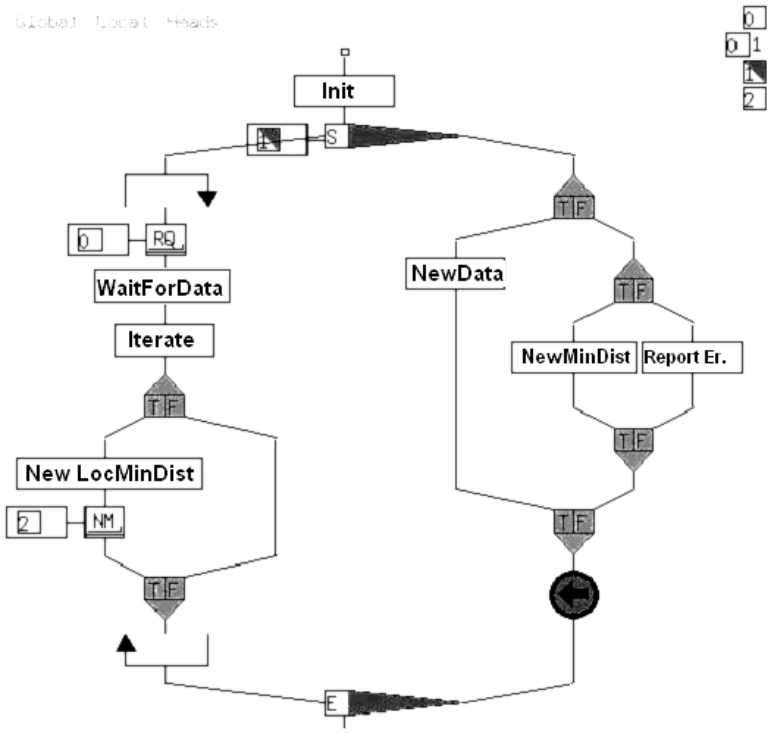
Rysunek 6.2: Schemat połączeń procesów i synchronizatora w aplikacji TSP w systemie PS-GRADE



Rysunek 6.3: Okno definicji predykatów w aplikacji TSP w systemie PS-GRADE



Rysunek 6.4: Diagramy przepływu sterowania predykatów *Dreq* (po lewej) i *NMin* (po prawej)



Rysunek 6.5: Diagram przepływu sterowania procesu w aplikacji TSP

Tablica 6.1: Przyspieszenie obliczeń TSP zrealizowanych w P-GRADE i PS-GRADE

	min	max	średnio
P-GRADE	4,67	9,04	7,15
PS-GRADE	4,95	14,06	9,92
$\frac{\text{PS-GRADE}}{\text{P-GRADE}}$	1,06	1,79	1,37

Treść procesów obliczeniowych pokazana jest na rysunku 6.5. Lewa gałąź diagramu reprezentuje normalnie przebiegające obliczenia TSP. Prawa gałąź określa sposób reakcji na przybycie sygnału sterującego. Najpierw sprawdza się typ sygnału (pierwsze rozgałęzienie warunkowe TF). Dla sygnału *newMinDist* proces akceptuje nową wartość najlepszego znanego rozwiania przysłaną wraz z sygnałem (blok NewMinDist). W przypadku sygnału *newData* otrzymane dane są dołączane do danych lokalnych (blok NewData).

6.2.1.2 Rezultaty testów w systemie PS-GRADE

Użycie synchronizatora uprościło tworzenie struktury programu i umożliwiło łatwe definiowanie warunków sterujących. Dzięki dystrybucji nowych rozwiązań przez asynchronicznie odbierane sygnały spodziewaliśmy się uzyskać lepsze wyniki, niż dla wersji zrealizowanej w P-GRADE.

Zrealizowano serię eksperymentów dla 15 miast, liczonych w klastrze zawierającym 21 heterogenicznych komputerów, klasy od Pentium II 400MHz do Athlon 2 GHz połączonych siecią Ethernet 100BaseT. Wyniki przedstawia Tabela 6.1. Z uwagi na heterogeniczność środowiska, wartości przyśpieszeń nie mają tu istotnego znaczenia. Znaczenie posiada natomiast stosunek uzyskanych przyśpieszeń widoczny w ostatnim wierszu tabeli. Zostosowanie proponowanych mechanizmów sterowania pozwoliło uzyskać średnio 1,37-krotnie wyższe przyśpieszenie obliczeń. Wyniki te należy uznać za bardzo dobre wobec znanych niedoskonałości bieżącej implementacji systemu PS-GRADE:

- Do przesyłania raportów o stanach procesów i sygnałów sterujących do komputerów docelowych wykorzystuje się zwykłą sieć Ethernet 100BaseT i bibliotekę PVM opartą o TCP/IP, taka wolna sieć daje względnie duże opóźnienie transmisji i obniża wydajność obliczeń równoległych.

- W sieci tej opóźnienie t używane przez algorytm wykrywania SCGS wykorzystujący górne ograniczenie na czas przesłania raportu (algorytm 5, punkt 2.4.1) musi być duże, przez co reakcje sterujące mogą być znacznie opóźnione.

- Synchronizacja zegarów protokołem NTP w sieci Ethernet może być poprawiona o co najmniej rząd wielkości gdyby zastosować sieć szybszą (np. Myrinet), lub jedno z rozwiązań opisanych w punkcie 2.1.2. Pozwoliłoby to uzyskać wyższą częstotliwość sterowania.

- Implementacja synchronizatora pozostawia nadal wiele miejsca dla różnego rodzaju optymalizacji zwiększających prędkość przetwarzania raportów, np. zamiana czysto obiektowej implementacji, obficie posługującej się alokacją i dealokacją pamięci, na implementację wykorzystującą statyczne struktury danych.

- Sygnały sterujące dostarczane są procesom poprzez dodatkowy pośredni proces systemowy PS-GRADE oraz przez sygnały systemu operacyjnego. Eliminacja pośredniego procesu oraz przyśpieszenie obsługi sygnałów systemu operacyjnego (patrz wyniki przedstawione w punkcie 4.2.4.2) pozwoliłoby na znaczną redukcję opóźnienia sterowania.

Próby z użyciem hierarchicznych synchronizatorów nie dały lepszych wyników. Nie stanowi to niespodzianki, gdyż już w badaniach symulacyjnych stwierdziliśmy zasadność użycia hierarchii synchronizatorów dopiero dla znacznie większej niż 20 liczby procesów.

6.2.2 Badania symulacyjne

Będąc świadomym wymienionych niedoskonałości rzeczywistego systemu PS-GRADE, zdecydowaliśmy się przebadać algorytm TSP w środowisku symulacyjnym opisanym w punkcie 4.1. W tym celu przyjęliśmy udoskonaloną wersję samej aplikacji, jak i lepsze środowisko obliczeniowe (względem implementacji w PS-GRADE). Oto lista udoskonaleń:

- Procedura równoważenia obciążenia reaguje zanim proces staje się bezczynny i pozwala na transfer zadań bezpośrednio pomiędzy procesami. Procesy przesyłają monitorowi periodycznie liczbę posiadanych zadań do rozpatrzenia. Monitor w każdym globalnym stanie silnie spójnym wybiera parę procesów: najmniej i najbardziej obciążony, i wysyła im sygnał polecający pierwszemu procesowi oczekiwać na zadania i drugiemu przesłać określoną liczbę zadań. Para ta jest zapamiętywana, aby w najbliższych kolejnych stanach spójnych nie powtarzać takich samych poleceń, zanim obciążenie nie zostanie zrównoważone i informacja o tym dotrze do monitora. Tak prosta procedura równoważenia obciążenia została wybrana celowo, aby jej koszt obliczeniowy był niski, ponieważ ma ona być wykonywana dla każdego globalnego stanu silnie spójnego, zatem często. Z drugiej strony jej częste wykonanie zapewni, że mimo swej prostoty spełni ona powierzone zadanie.

- Komputery połaczone są siecią Myrinet, w porównaniu z siecią FastEthernet posiadającą ponad 10-krotnie lepszą przepustowość, mniejsze opóźnienie i mniej obciążającą procesory przetwarzaniem komunikatów. Związana z tym jest uzyskana ok. 10-krotnie lepsza dokładność synchronizacji zegarów lokalnych.

- Modyfikacje jądra systemu operacyjnego umożliwiają 10-100-krotnie szybszą reakcję procesów na sygnały systemu operacyjnego (patrz 4.2.4.2).

- Lepsza implementacja monitora (pominięte koszty związane z tworzeniem i usuwaniem obiektów) pozwala szybciej przetwarzać raporty.

- Dostępna jest większa liczba jednorodnych procesorów.

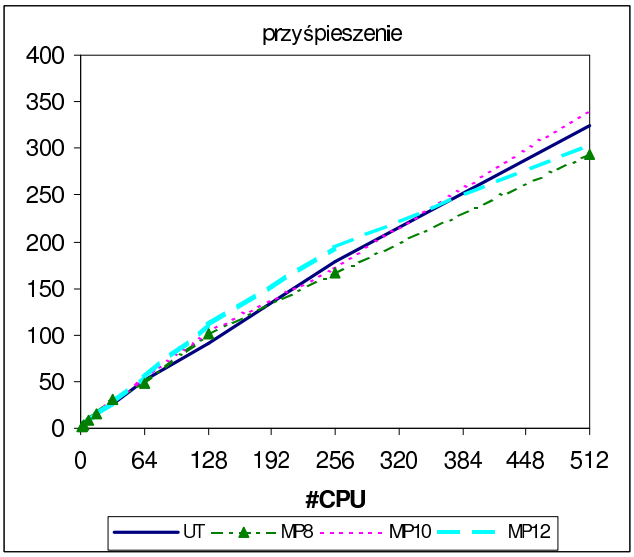
Odpowiednio przygotowaliśmy też wersję TSP zrealizowaną z użyciem klasycznych metod sterowania (czyste przekazywanie komunikatów bez monitorowania stanów globalnych aplikacji). Staraliśmy się uzyskać przy tym jak najlepszą wydajność, korzystając z pełnego wachlarza metod przekazywania komunikatów. Zachowaliśmy opisaną wyżej strategię równoważenia obciążenia, przy czym proces realizujący tę strategię przetwarzał raporty o obciążeniu w kolejności ich przybywania. Informację o nowych najlepszych wynikach procesy rozsyłały bezpośrednio do innych procesów, co stanowiło przewagę nad wersją sterowaną predykatami, korzystającą z pośrednictwa synchronizatora (monitora). Użyliśmy nieblokujących procedur odbioru komunikatu, aby proces liczący mógł co jakiś czas sprawdzić, czy nie otrzymał komunikatu związanego z równoważeniem obciążenia lub dostarczającego nowe najlepsze rozwiązanie. Wywołanie takiej procedury kosztuje niewiele w razie braku komunikatu (wzorowaliśmy się na czasie działania funkcji pvm_nrecv()), jednak aby opóźnienia w obsłudze oczekującego komunikatu były jak najmniejsze, procedura ta musi być uruchamiana periodycznie i odpowiednio często. W strukturze algorytmu TSP jest wiele zagnieżdżonych pętli. Umieszczając nieblokujący odbiór komunikatu wewnątrz bardziej lub mniej wewnętrznej pętli mogliśmy kontrolować, jak często następuje sprawdzenie, czy komunikat nadszedł. Przeprowadziliśmy testy dla kilku takich wariantów. W rezultacie powstał program wykorzystujący klasyczną metodę sterowania, przy tym działający w sposób zbliżony do wersji ze sterowaniem przez predykaty. Przygotowanie takiego programu kosztowało sporo pracy, gdyż całe sterowanie trzeba było zaprogramować od początku z użyciem tylko prostych poleceń wysłania i odebrania komunikatu, ale bez użycia infrastruktury istniejącej w proponowanej przez nas metodzie sterowania (kolejki komunikatów, algorytm SCGS, pośrednictwo monitora) dawało nadzieję uzyskania dobrej wydajności.

Porównanie przyśpieszenia obliczeń uzyskane dla implementacji TSP opartej na proponowanej metodzie sterowania przez predykaty określone na stanach globalnych (z algorytmem UT, patrz punkt 4.2.2) i dla implementacji opartej na zwykłym przekazywaniu komunikatów, widoczne jest na rysunku 6.6. Symbol UT oznacza wersję stosującą sterowanie przez predykaty. Symbole MPxx oznaczają wersje programu oparte na przekazywaniu komunikatów, gdzie xx oznacza poziom zagnieżdżenia pętli, w której sprawdzano czy nadszedł nowy komunikat. Im większy poziom, tym częstsze sprawdzanie. Na rysunku widać, że wariant MP8 sprawuje się gorzej od pozostałych. Z kolei wariant MP12, początkowo najlepszy, uzyskuje nieco słabsze przyśpieszenie dla dużej liczby procesów. Widzimy, że różnice nie przekraczają kilkunastu procent. Różnice pomiędzy wersjami są lepiej widoczne na rysunku 6.7, gdzie pokazany jest procent czasu spędzonego bezczynnie przez procesy, w oczekiwaniu na reakcję procedury równoważenia obciążenia. Systematycznie gorszy wynik wariantu MP8 (o kilka, kilkanaście procent) znajduje swe odzwierciedlenie w mniejszym przyśpieszeniu, i jest wynikiem zbyt długiego oczekiwania na obsługę przybyłych komunikatów. Nieprzedstawione tu warianty MP6 i MP4 dawały znacznie gorsze rezultaty. Najlepszy wynik implementacji opartej na predykatach globalnych można przypisać zarówno szybkiej (asynchronicznej) reakcji na sygnały sterujące, jak też realizacji równoważenia obciążenia w oparciu o stany silnie spójne, dające nieprzekłamany obraz stanu systemu. Obraz widziany przez wersje MPxx mógł momentami być niespójny i zatem znacznie odbiegać od faktycznej sytuacji. Nasuwają się tu następujące wnioski:

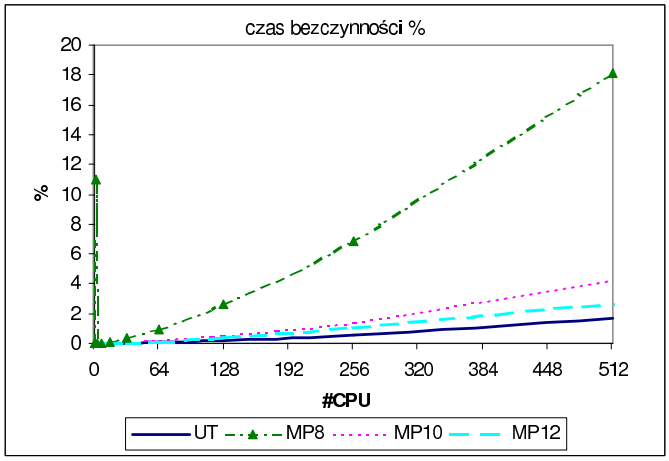
W1: W implementacjach posługujących się klasycznym przekazywaniem komunikatów optymalny dobór częstości sprawdzania przybycia komunikatów wymaga prób doświadczalnych w docelowym środowisku. Nieoptymalny dobór powoduje zauważalną degradację wydajności takich implementacji.

W2: Oparcie procedury równoważenia obciążenia na obserwacji globalnych stanów silnie spójnych poprawia jakość równoważenia obciążenia względem stosowania zwykłej obserwacji opartej o analizę raportów w kolejności ich przybywania (stany obserwowane, patrz 3.1.1.1).

W3: Sterowanie przez predykaty przy użyciu proponowanej gotowej infrastruktury sterującej daje podobną wydajność obliczeń, jak ręcznie eksperymentalnie dopraco-



Rysunek 6.6: Przyśpieszenia dla różnych implementacji TSP w symulacji

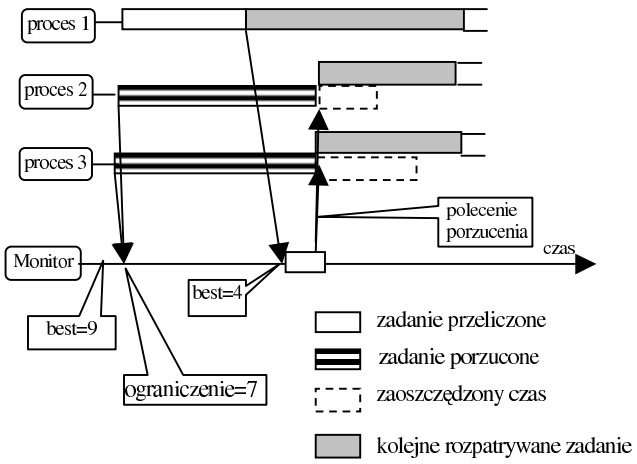


Rysunek 6.7: Czas bezczynności procesów dla różnych implementacji TSP w symulacji

wana implementacja przy użyciu pełnego wachlarza metod przekazywania komunikatów (gdy obie implementacje działają na identycznym sprzęcie).

W4: Zastosowanie asynchronicznej reakcji na sygnały sterujące, będące częścią proponowanej metody sterowania przez predykaty, uwalnia programistę od konieczności eksperymentalnego dopracowywania częstości sprawdzania nadejścia komunikatów bez groźby znacznego pogorszenia wydajności obliczeń równoległych.

W bardziej zaawansowanej implementacji TSP opartej o predykaty globalne chcieliśmy wykorzystać możliwość porzucania obliczeń (patrz punkty 3.3.2 i 3.3.2.1), aby zaprzestać przeliczania zadań, które stały się nieperspektywiczne dopiero w trakcie ich rozpatrywania. Rozważane do tej pory implementacje pozwalały na odrzucenie zadania tylko przed rozpoczęciem jego rozpatrywania. Porzucanie obliczeń ilustruje rysunek 6.8. W przedstawionym tam scenariuszu zdarzeń początkowo najlepsze znane rozwiązanie ma koszt 9. Procesy numer 2 i 3 rozpoczynają rozpatrywanie zadań, dla których wiadomo, że nie mogą dać wyniku lepszego niż 7. To ograniczenie jest raportowane do monitora. W trakcie gdy procesy 2 i 3 przeliczają swe zadania, proces numer 1 znajduje nowe rozwiązanie o koszcie 4 i informuje o tym monitor. Monitor nakazuje procesom 2



Rysunek 6.8: Porzucanie nieperspektywicznych zadań w algorytmie TSP

i 3 porzucić aktualnie przeliczane zadania, gdyż i tak nie mogą one dać wyniku lepszego niż już znany. Procesy 2 i 3 natychmiast rozpoczynają rozpatrywanie kolejnych zadań.

Przeprowadzone badania symulacyjne wskazały, że powyższa idea funkcjonuje zgodnie z oczekiwaniem, lecz faktycznie uzyskiwana poprawa wydajności obliczeń jest niestety znikoma w problemie TSP. Powody tego są dwa:

Problem 1 W rozpatrywanych problemach TSP (B&B, dziel i ograniczaj) na każdy milion rozpatrywanych zadań tylko kilkadziesiąt z nich generuje nowy najlepszy wynik. Zatem tylko tyle razy może zaistnieć sytuacja przedstawiona na rysunku 6.8. W rezultacie bardzo niewielki procent zadań może zostać porzuconych, przez co wpływ porzucania jest niski.

Problem 2 Przedstawiona idea wymaga, aby proces wysyłał raport o dolnym ograniczeniu wyniku dla każdego aktualnie rozpatrywanego zadania, aby synchronizator mógł dla każdego z nich podjąć ewentualną decyzję o przerwaniu. Częstotliwość wysyłania tych raportów powoduje, że interfejs sieciowy monitora nie nadąża przetwarzać nadchodzących komunikatów - nie chodzi tu o nominalną przepustowość w bajtach na sekundę, lecz praktycznie osiągalną przepustowość w komunikatach (pakietach) na sekundę.

Porzucanie obliczeń może dawać istotne zyski, gdy liczba porzucanych zadań w stosunku do liczby zadań rozpatrywanych jest znacząca. W omawianym przypadku, porzucanie może dać znaczące efekty, gdy:

- struktura rozwiązywanego problemu będzie taka, że znaczący procent zadań generuje nowe lepsze rozwiązania,
- wykorzystamy system masywnie równoległy - zawierający wiele tysięcy procesorów - z siecią rozgłaszającą sygnały sterujące, dzięki czemu pojedyncze nowe lepsze rozwiązanie będzie mogło spowodować porzucenie wielu tysięcy zadań, a nowe najlepsze rozwiązania będą szybko przekazywane wszystkim procesom,
- rozwiążemy problem wydajności komunikacji odpowiedzialnej za sterowanie opisany powyżej jako problem 2.

Problem 2 udało się w tym przypadku rozwiązać przez całkowitą eliminację wysyłania raportów o dolnym ograniczeniu wyniku dla rozpatrywanych zadań. Dokonaliśmy tego stosując następujący schemat: każdy nowy znaleziony najlepszy globalnie rezultat był rozsyłany przez synchronizator

jako sygnał sterujący do wszystkich procesów. Reakcja procesu na ten sygnał polegała na zapamiętaniu załączonej w sygnale wartości nowego rozwiązania oraz na autonomicznej decyzji, czy porzucić aktualnie liczone zadanie.

W dalszym ciągu prac zbadaliśmy wpływ parametrów sieci na uzyskiwaną wydajność. Po dokonaniu opisanej wyżej korekty w symulacji obliczeń TSP przeprowadziliśmy dodatkowe symulacje z zastosowaniem podwójnej sieci komunikacyjnej, stosując:

sieć przesyłania danych o charakterystyce odpowiadającej sieci Myrinet, wykorzystywaną do transmisji danych przez procedury równoważenia obciążenia,

sieć sterującą używaną do przesyłania raportów o stanie procesów i sygnałów sterujących z możliwością rozgłaszania. Sieć ta miała swoje charakterystyki zmienione względem stosowanego modelu sieci Myrinet w następujący sposób:

- 40-krotnie mniejszy czas przestoju pomiędzy przesłaniem kolejnych komunikatów (parametr g w modelu LogP),
- 10-krotnie mniejsze minimalne opóźnienie transmisji,
- 10-krotnie lepsza dokładność synchronizacji zegarów,
- możliwość rozgłaszania sygnałów od synchronizatora do procesów.

Postulowane parametry nie wybiegają poza możliwości współczesnych technologii. Rozgłaszanie jest dostepne w istniejących sieciach (jak Myrinet, Ethernet), toteż w tym względzie jedynie dostowaliśmy nasz model do rzeczywistości. Lepszą dokładność zegarów (względem osiąganej przy użyciu standardowego Myrinetu + NTP) można uzyskać na wiele sposobów, patrz punkt 2.1.2. Mniejsze opóźnienie (niż dla sieci Myrinet) spotykane jest w przypadku tylko niektórych rozwiązań, takich jak RDMA w komputerze SR2201 [84], czy DIMMnet [46], jest jednak osiągalne. Najtrudniej jest uzyskać postulowaną małą wartość parametru g, co przekłada się na dużą przepustowość przy wykorzystaniu komunikatów o małych rozmiarach. W częściej spotykanych sieciach wartość g nie spada poniżej 2μs, podczas gdy nas interesują wartości rzędu 0.1μs. Do tego wyniku zbliżają się sieci bazujące na bezpośrednich operacjach na pamięci, takie jak RDMA [111] oraz wspominany DIMMnet.

Dla 512 liczących procesów (na 512 procesorach) porównaliśmy czas obliczeń problemu TSP dla środowiska z pojedynczą i opisaną powyżej podwójną siecią komunikacyjną z wykorzystaniem porzucania obliczeń. Ponieważ struktura problemu TSP pozostała zasadniczo nie zmieniona, nadal mieliśmy do czynienia z wymienionym powyżej problemem numer 1. W rezultacie samo porzucanie zadań powodowało redukcję czasu wykonania programu o 0.1% do 0.2% - wystarczająco, aby potwierdzić poprawne działanie zaprzestania, za mało jednak, aby stanowiło to znaczący ilościowo rezultat w analizowanej aplikacji. Szybsza sieć sterująca pokazała mimo to swoje zalety. Umożliwiła ona szybsze informowanie synchronizatora o obciążeniu i szybsze wykonywanie decyzji synchronizatora dotyczących równoważenia obciążenia (mniejsze Opóźnienie Sterowania, patrz punkt 4.2.1). Nowe najlepsze wyniki algorytmu TSP były szybciej rozpowszechniane wśród liczących równolegle procesów, dzięki czemu procesy szybciej zaczynały eliminować zadania nieperspektywiczne. Wykonane eksperymenty wykazały, że

> **zastosowanie specjalizowanej sieci komunikacyjnej do sterowania (bez zmiany konfiguracji parametrów aplikacji) skracało czas wykonania testowej aplikacji TSP średnio 1.4 raza.**

Specjalizowana sieć komunikacyjna pozwolała na szybsze przesyłanie i przetwarzanie komunikatów o stanach. W badanej konfiguracji 512 procesów nie mogliśmy jednak zwiększyć częstości raportowania stanu przez procesy, gdyż powodowałoby to przeciążenie synchronizatora. Zmieniliśmy zatem konfigurację obliczeń TSP. Zastosowaliśmy 128 procesów liczących (na 128 procesorach) i powiększyliśmy częstości raportowania stanu przez procesy pięciokrotnie. Dodatkowo zmniejszyliśmy czas obliczenia pojedynczej iteracji w algorytmie TSP o dwa rzędy wielkości. Przez to

uzyskaliśmy symulację odpowiadającą łatwiejszemu obliczeniowo problemowi TSP, obliczanemu w systemie z szybszymi procesorami. Ta wersja symulacji charakteryzowała się bardzo dynamicznymi zmianami obciążenia procesorów. Paczka zadań stanowiąca początkowo duże obciążenie mogła zostać wykonana bardzo szybko. W rezultacie szybkość działania sterowania realizującego równoważenie obciążenia okazywała się kluczowa dla efektywności obliczeń równoległych. Po uśrednieniu wyników stwierdziliśmy, że

> **zastosowanie specjalizowanej sieci komunikacyjnej do sterowania i konfiguracja parametrów aplikacji w sposób odpowiednio wykorzystujący możliwości tej sieci przyspieszało wykonanie testowej aplikacji TSP średnio 3.4 raza.**

Realizacja proponowanej metody sterowania z użyciem odpowiednio dobranych struktur sprzętowych, czyli jako zintegrowanego rozwiązania sprzętowo-programowego, jest wysoce efektywna i pozwala na nawet kilkukrotne zwiększenie wydajności realizowanych obliczeń równoległych.

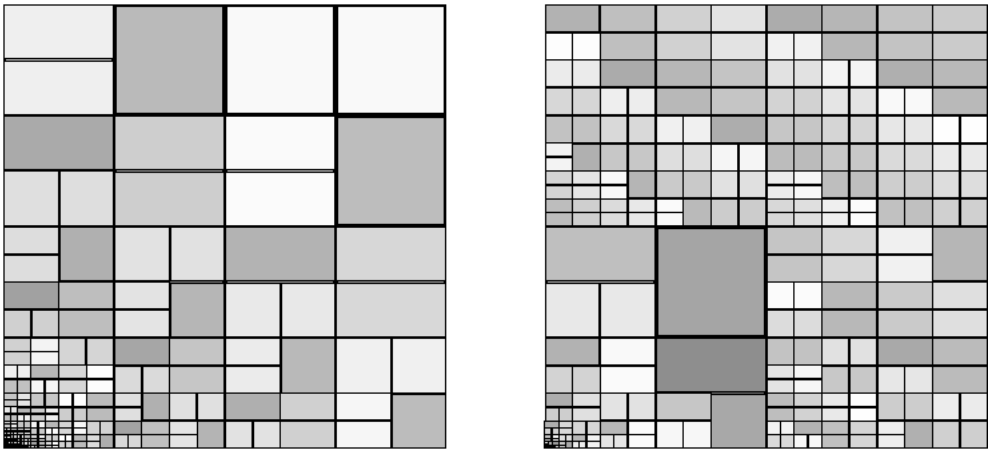
6.3 Całkowanie adaptacyjne

Standardowy algorytm całkowania numerycznego dzieli obszar całkowania na jednakowe fragmenty (podobszary całkowania) i dla każdego wylicza przybliżenie wartości całki. Suma tych przybliżeń stanowi obliczoną wartość całki w zadanym obszarze. Dokładność obliczeń jest tym lepsza, im mniejsze są rozmiary podobszarów. Jednak drobniejsze podobszary oznaczają ich większą liczbę. Ponieważ dla każdego podobszaru trzeba wyliczyć wartość funkcji podcałkowej (n-krotnie, w zależności od metody całkowania numerycznego), to czas obliczeń rośnie liniowo wraz z liczbą podobszarów. Dla funkcji nieregularnych, posiadających ostre grzbiety czy piki, błąd całkowania może być bardzo znaczący dla podobszarów obejmujących tego typu nieregularności. Chcąc uzyskać odpowiednią dokładność obliczeń trzeba by zmiejszyć rozmiar podobszarów tak znacznie, że czas obliczeń wydłużyłby się niedopuszczalnie. Przy tym dla regularnych (gładkich) fragmentów całkowanej funkcji podobszary o większym rozmiarze byłyby wystarczające. Ta obserwacja doprowadziła do powstania całkowania adaptacyjnego [102].

Całkowanie adaptacyjne zaczyna obliczenia od pojedynczego fragmentu obejmującego cały obszar całkowania. Wyliczany i zapamiętywany jest szacowany błąd całkowania dla każdego podobszaru. Podobszary składowane są w kolejce priorytetowej, w której priorytetem jest szacowany błąd całkowania. Zawsze brany jest podobszar z czoła kolejki z największym błędem - jest on dzielony, a powstałe mniejsze podobszary są przeliczane i ponownie wstawiane do kolejki. Dzięki takiej procedurze zawsze zajmujemy się podobszarami z największym błędem całkowania (tymi, w których funkcja podcałkowa jest najbardziej nieregularna) i w ten sposób najefektywniej dążymy do uzyskania dokładnego wyniku. Innymi słowy uzyskamy określoną dokładność przy minimalnej liczbie podobszarów całkowania, czyli przy minimalnym nakładzie obliczeniowym.

Efektywna równoległa implemetacja całkowania adaptacyjnego jest zadaniem trudnym [124, 104]. Gdy każdy proces utrzymuje własną kolejkę priorytetową, wówczas optymalizacja wyboru podobszaru do dalszego podziału jest lokalna dla procesu. Wyniki takiej optymalizacji mogą być bardzo odległe od wyników optymalizacji globalnej. Przykład widoczny jest na rysunku 6.9, gdzie dla funkcji posiadającej pik w lewym dolnym rogu mamy przedstawiony podział na podobszary optymalny globalnie (po lewej) i robiony niezależnie przez 4 procesory, każdy w swojej ćwiartce. W pierwszym przypadku prawa górna ćwiartka obszaru całkowania została podzielona na 5 podobszarów, aby uzyskać błąd całkowania na tym samym poziomie co w lewej dolnej ćwiartce. W przypadku 4 procesów każda ćwiartka została podzielona na 60 regionów. Uzyskana w ten sposób bardzo wysoka dokładność obliczeń w prawej górnej ćwiartce jest niestety przesłaniana niską dokładnością uzyskaną w lewej dolnej ćwiartce.

Z drugiej strony utrzymywanie centralnej puli regionów, gwarantujące optymalną pracę algorytmu, jest nieefektywne - dostęp do tej centralnej puli stanowi wąskie gardło, ponadto sam koszt dostępu do zdalnych danych może być nieopłacalnie wysoki. Praktycznie stosowanym rozwiązaniem jest utrzymywanie lokalnych kolejek priorytetowych przez procesy oraz wymiana danych



Rysunek 6.9: Podział na regiony całkowania dla jednego (z lewej) i 4 procesów bez komunikacji (z prawej)

pomiędzy nimi. Proces posiadający wiele podobszarów o dużym błędzie całkowania powinien przekazać część z nich innym procesom, tak aby wszystkie pracowały nad podobszarami o największych globalnie (w danej chwili) błędach całkowania. W oparciu o te założenia zrealizowaliśmy równoległe całkowanie adaptacyjne w systemach P-GRADE i PS-GRADE. Wykorzystaliśmy pomocnicze, a przede wszystkim numeryczne procedury szeregowego całkowania adaptacyjnego z pakietu ParInt [1], tworząc w oparciu o nie oryginalne wersje równoległe.

6.3.1 Testy w rzeczywistych systemach

Wersja P-GRADE całkowania adaptacyjnego to klasyczna aplikacja oparta na przekazywaniu komunikatów z procesem koordynujacym. Ma ona charakter fazowy, każdy proces wykonuje następujący algorytm:

1. Całkuj adaptacyjnie lokalnie przez K iteracji.
2. Raportuj koordynatorowi bieżący lokalny błąd całkowania (suma błędów z posiadanych podobszarów).
3. Czekaj na odpowiedź od koordynatora zawierającą wskazówki komu wysłać/od kogo odebrać wskazaną część lokalnego błędu całkowania poprzez transfer informacji o podobszarach.
4. Wyślij/odbierz parametry podobszarów.
5. Idź do 1.

Koordynator zatrzymuje procesy po osiągnięciu wymaganej dokładności globalnej.

Fazowy charakter algorytmu jest wymuszony ograniczeniami systemu P-GRADE - brakiem instrukcji odbioru nieblokującego. System PS-GRADE nie ma tego ograniczenia, gdyż w dowolnym momencie obliczenia mogą zostać przerwane sygnałem sterującym, sam sygnał może zawierać parametry, a kod obsługi sygnału może obejmować komunikację. Implementacja całkowania adaptacyjnego w PS-GRADE korzysta z tych cech. Procesy periodycznie wysyłają raporty o lokalnym błędzie całkowania. Synchronizator analizuje stan globalny aplikacji i dla każdego stanu spójnego wysyła sygnał do pary procesów o największym i najmniejszym błędzie, inicjując w ten sposób wymianę informacji o podobszarach pomiędzy nimi. Po wymianie procesy od razu raportują swój nowy stan.

Ogólny schemat sterowania jest tu podobny do zastosowanego w implementacji problemu komiwojażera, jednak napotkaliśmy nowe trudności:

- Błąd raportowany widoczny dla synchronizatora może znacznie się różnić (na plus lub na minus) od błędu faktycznego istniejącego w momencie otrzymania sygnału sterującego inicjującego wymianę informacji o podobszarach. Opisywane w punkcie 4.2.2.1 opóźnienie sterowania odgrywa tu istotną rolę. W rezultacie para procesów mających wymienić podobszary musi się najpierw bezpośrednio porozumieć, aby ustalić czy i jaka wymiana podobszarów jest aktualnie potrzebna. Problem ten nie występuje w wersji P-GRADE, gdyż fazowa natura tamtej implementacji wymusza czekanie na decyzję koordynatora - koordynator ma do dyspozycji w pełni aktualne dane.

- Dane związane z właśnie przesyłanymi podobszarami stanowią część rozwiązania - wartość całki z nimi związana nie jest wliczona w lokalną wartość całki w żadnym procesie. To samo tyczy błędu całkowania. Dlatego synchronizator musi brać pod uwagę także stan komunikacji, dodatkowo raportowany przez procesy. Program można zakończyć i podać poprawny wynik tylko, gdy nie ma trwających transmisji. Alternatywnie, w rozwiązaniu globalnym należy uwzględniać dane związane z podobszarami będącymi w trakcie przesyłania.

- Błąd lokalny szybko rozkłada się na dużą liczbę drobnych podobszarów w każdym procesie, każdy podobszar niesie tylko drobną część błędu całkowania. Z tego powodu wyrównywanie wartości błędów pomiędzy procesami wymaga przesyłania informacji o dużej liczbie podobszarów. To zjawisko nasila się przy zwiększaniu liczby procesów liczących. W rezultacie skalowalność całkowania adaptacyjnego jest ograniczana narastającą intensywną wymianą informacji o podobszarach.

W implementacji całkowania adaptacyjnego w systemie PS-GRADE zastosowaliśmy jeden synchronizator, co dało układ procesów i ich połączeń taki, jak na rysunku 6.10. Co ustaloną parametrem liczbę lokalnych iteracji każdy proces wysyła do synchronizatora swój stan lokalny obejmujący:

- lokalny szacowany błąd całkowania,

- lokalny rezultat całkowania,

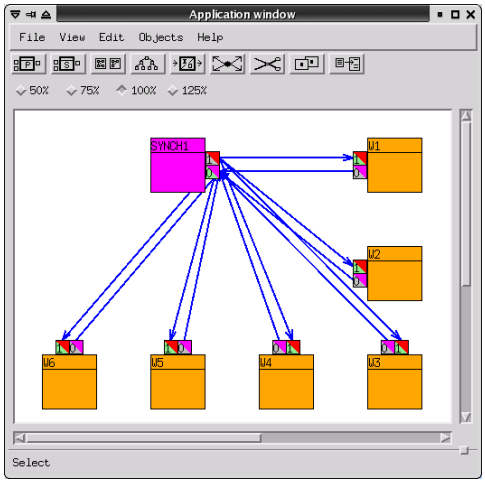
- liczbę podobszarów wysłanych w procedurze równoważenia obciążenia,

- liczbę podobszarów odebranych w procedurze równoważenia obciążenia.

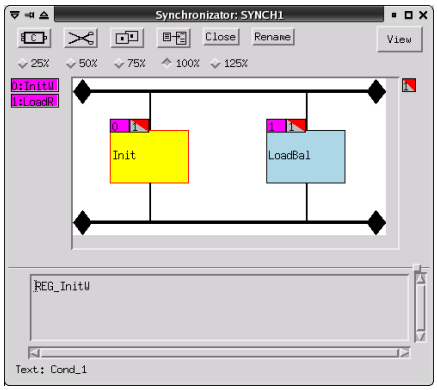
Dwa predykaty są zdefiniowane w synchronizatorze, patrz rysunki 6.11 i 6.12. Predykat *Init* ma znaczenie jedynie przy starcie aplikacji i odpowiada za równy inicjalny podział obszaru całkowania pomiędzy procesory. Predykat *LoadBal* wartościowany jest na każdym stanie spójnym i służy do wyznaczenia pary procesów o największym i najmniejszym lokalnym szacowanym błędzie całkowania (blok CompLoad na rysunku 6.12). Wybrana para procesów otrzymuje sygnał sterujący (widoczne na rysunku dwie instrukcje wysłania sygnału, umieszczone w osobnych gałęziach instrukcji warunkowej), który pobudza je do bezpośredniej komunikacji. Procesy te ustalają bieżącą różnicę ich lokalnego szacowanego błędu całkowania, po czym proces z większym błędem przesyła temu drugiemu stosowną liczbę regionów ze swojej puli. Ta wymiana komunikatów zrealizowana została za pomocą funkcji komunikacyjnych na poziomie języka C, aby zapewnić bezpośrednią komunikację procesów bez rysowania kanałów pomiędzy każdą parą procesów.

Diagram przepływu sterowania procesu liczącego pokazuje rysunek 6.13. Pierwsza część diagramu (blok *Init* i pierwszy region kodu przyjmujący sygnały) odpowiedzialny jest za początkową dystrybucję podobszarów całkowania. Właściwa pętla obliczeniowa, zawierająca blok *Iterate* oraz instrukcję wysłania stanu lokalnego procesu do synchronizatora, zamknięta jest wewnątrz dwóch regionów kodu przyjmujących sygnały. Regiony te umożliwiają procesowi reakcję na sygnał nakazujący przekazanie informacji o części podobszarów (obsługa sygnału w bloku *SendData*) oraz na sygnał nakazujący odebranie informacji o podobszarach (obsługa sygnału w bloku *RecData*).

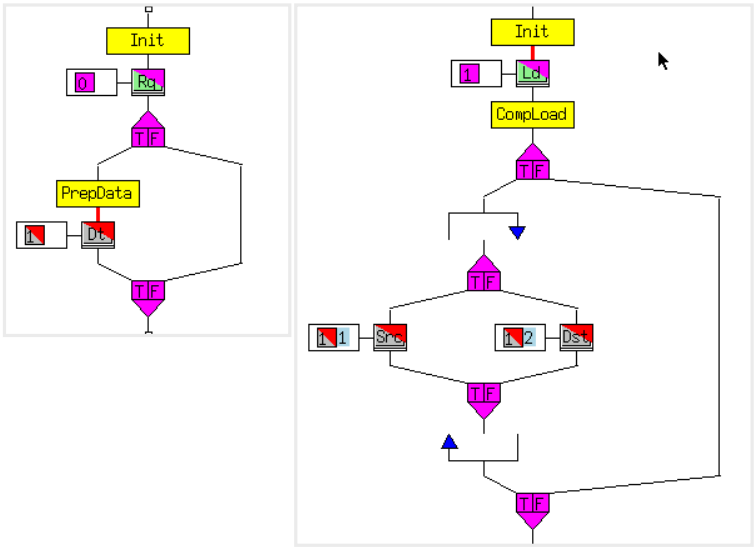
Gdy szacowany błąd całkowania jest odpowiednio mały, synchronizator zaprzestaje realizować równoważenie obciążenia i zaczyna zwracać uwagę na globalną sumę wysłanych i odebranych



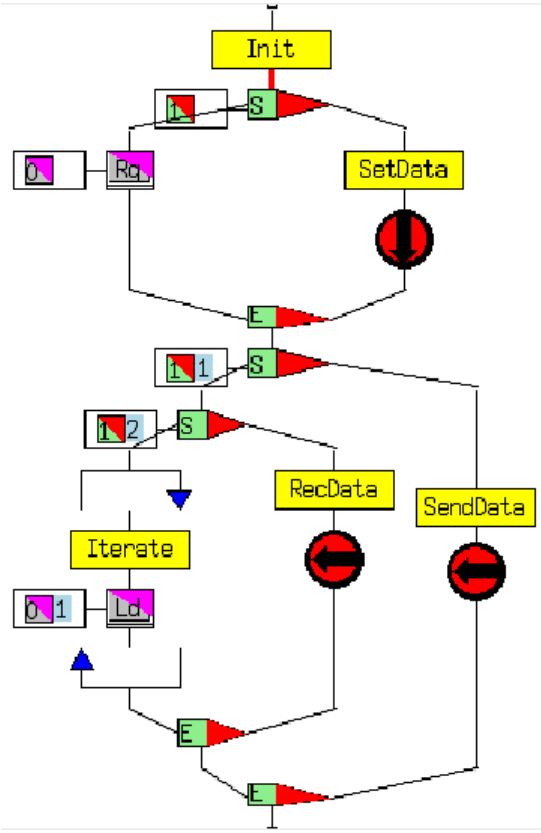
Rysunek 6.10: Schemat połączeń procesów i synchronizatora w programie całkowania adaptacyjnego w systemie PS-GRADE



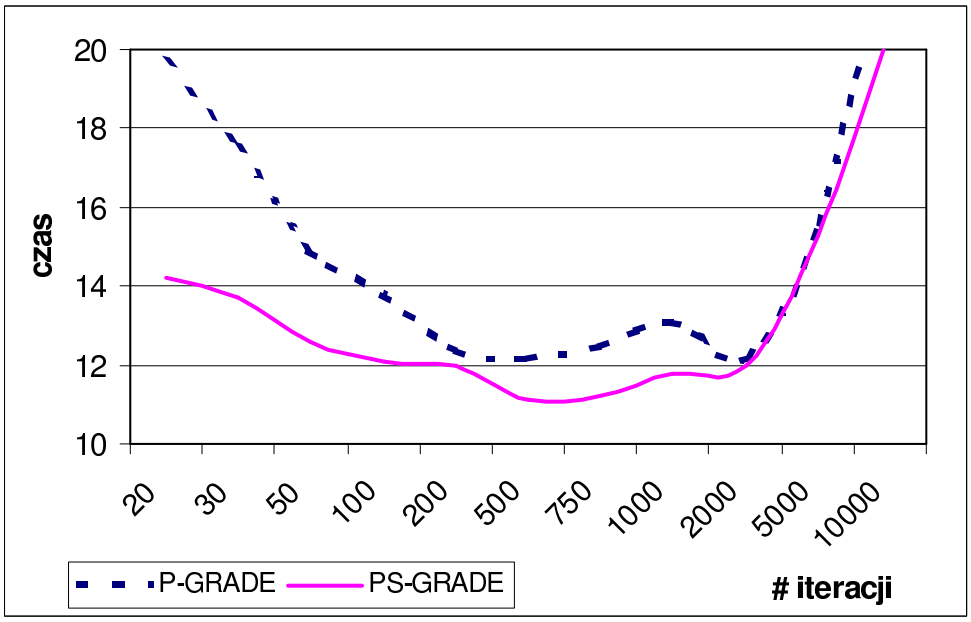
Rysunek 6.11: Okno definicji predykatów dla całkowania adaptacyjnego



Rysunek 6.12: Diagramy przepływu sterowania predykatów *Init* (z lewej) i *LoadBal* (z prawej)



Rysunek 6.13: Diagram przepływu sterowania procesu liczącego w programie całkowania adaptacyjnego



Rysunek 6.14: Czas równoległego całkowania adaptacyjnego

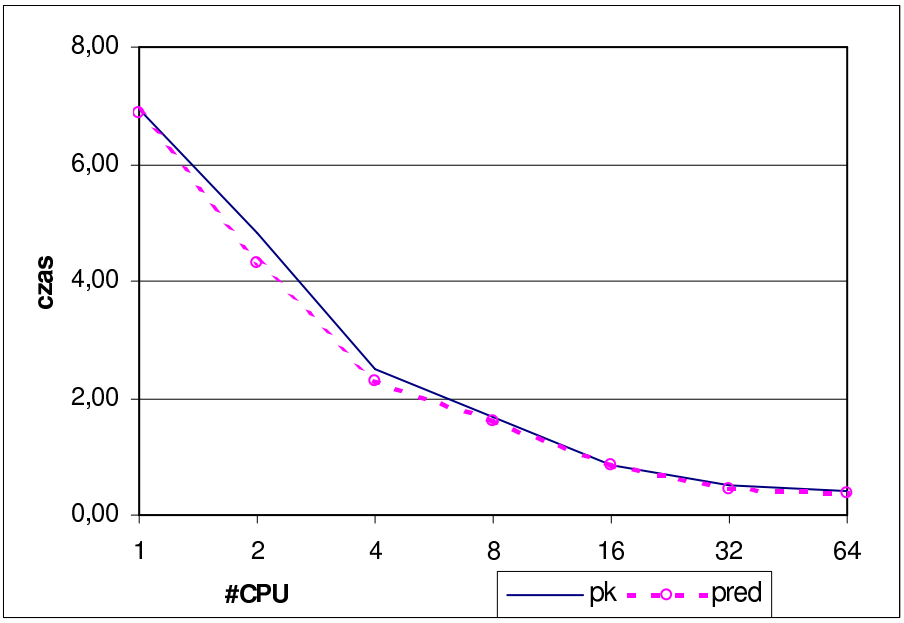
podobszarów. Gdy suma ta równa jest zero, wówczas żadne dane o podobszarach nie są przysyłane i suma lokalnych wyników całkowania jest prawidłowym wynikiem globalnym. Ta suma jest przekazywana jako wynik działania programu, po czym aplikacja jest zatrzymywana.

Przeprowadziliśmy badania obu wersji na klastrze 4 procesorowym, plus piąty procesor dla synchronizatora/koordynatora. Rezultaty, dla szerokiego przedziału częstości raportowania błędu (czyli długości fazy w wersji P-GRADE) liczonego jako liczba iteracji całkowania pomiędzy raportami, są przedstawione na rysunku 6.14. Proponowana metoda sterowania daje wyraźnie lepszą efektywność obliczeń (krótszy czas wykonania) dla krótkich faz (20-100 iteracji) i o kilka-kilkanaście procent lepsze w pozostałych przypadkach.

6.3.2 Badania symulacyjne

Podobnie jak w przypadku problemu komiwojażera, także tu zdecydowaliśmy się na badania w środowisku symulacyjnym opisanym w punkcie 4.1. W tym celu przyjęliśmy udoskonaloną wersję samej aplikacji, jak i lepsze środowisko obliczeniowe. Oto lista udoskonaleń:

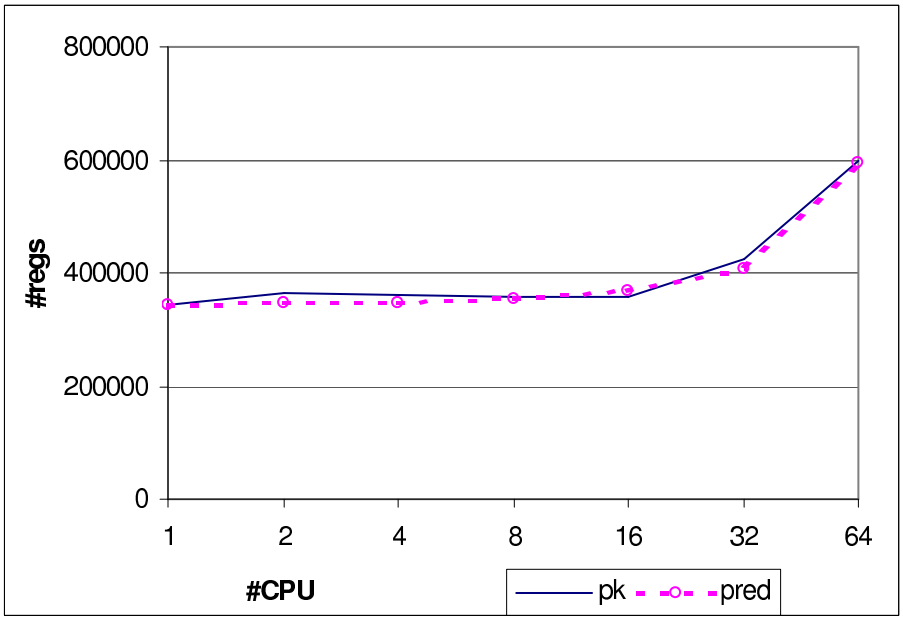
- Komputery połaczone są siecią Myrinet, w porównaniu z siecią FastEthernet posiadającą ponad 10-krotnie lepszą przepustowość, mniejsze opóźnienie i mniej obciążającą procesory przetwarzaniem komunikatów. Związana z tym jest uzyskana ok. 10-krotnie lepsza dokładność synchronizacji zegarów lokalnych.
- Modyfikacje jądra systemu operacyjnego umożliwiają 10-100-krotnie szybszą reakcję procesów na sygnały systemu operacyjnego (patrz 4.2.4.2).
- Lepsza implementacja monitora (pominięte koszty związane z tworzeniem i usuwaniem obiektów) pozwala szybciej przetwarzać raporty.
- Dostępna jest większa liczba jednorodnych procesorów, na każdym alokowaliśmy pojedynczy proces.
- Dla implemetacji opartej na przekazywaniu komunikatów dostępne są instrukcje odbioru nieblokującego. Sprawdzenie, czy przybył komunikat kosztuje bardzo niewiele w porównaniu z pojedynczą iteracją całkowania.



Rysunek 6.15: Czas równoległego całkowania adaptacyjnego w symulacji

- Implementacja oparta na przekazywaniu komunikatów nie ma charakteru fazowego. Procesy raportują periodycznie swój lokalny błąd całkowania i po każdej iteracji obliczeń obsługują przybyłe komunikaty. Ponieważ pojedyncza iteracja jest intensywna obliczeniowo, dodanie do każdej z nich sprawdzenia, czy są komunikaty gotowe do odebrania nie stanowi zauważalnego narzutu.

Częsta wymiana informacji o podobszarach dla większej liczby procesorów spowodowała, że kończenie programu w sytuacji, gdy żadne informacji o podobszarach nie są przesyłane, okazało się nieefektywne. Zbyt rzadko następowały takie okazje. Dlatego sposób raportowania stanów przez procesy zmieniliśmy tak, aby monitor wiedział jaki sumaryczny błąd całkowania i wartość całki są w trakcie przesyłania. Dzięki temu aplikacja mogła zostać zakończona niezależnie od stanu komunikacji, natychmiast, gdy osiągnięto zadaną dokładność wyniku. Rezultaty zaprezentowane na rysunku 6.15 świadczą o tym, że przy całkowaniu adaptacyjnym implementacja z zastosowaniem proponowanej przez nas gotowej infrastruktury sterowania (pred) pozwala na uzyskanie takiej samej, a nawet nieco lepszej (dla 32-64 procesorów) wydajności obliczeń, niż eksperymentalnie dopracowana implementacja oparta o przekazywanie komunikatów (pk), z wykorzystaniem pełnej gamy instrukcji komunikacyjnych (w szczególności nieblokujacego odbioru komunikatu). Rysunek 6.16 pokazuje natomiast, jak rośnie liczba podobszarów, na które podzielony został obszar całkowania w celu uzyskania żądanej dokładności obliczeń, w miarę zwiększania liczby liczących procesów. Powyżej 16 procesów wymiana informacji o podobszarach staje się zbyt wolna i dla części procesów algorytm liczący zaczyna całkować podobszary nieoptymalne z globalnego punktu widzenia. Implementacje z użyciem przekazywania komunikatów i z użyciem sterowania przez predykaty globalne wykazują bardzo podobną efektywność w tym względzie, jednak z paruprocentową przewagą proponowanej przez nas gotowej infrastruktury sterowania. Jak wykazały dalsze badania, zaobserwowane ograniczenie skalowalności całkowania adaptacyjnego, powodowane jest przez częste przesyłanie dużej liczby danych o podobszarach w celu optymalnej dystrybucji błędu całkowania. Zastosowanie dodatkowej sieci komunikacyjnej dla potrzeb systemu sterowania nie daje tu znaczącej poprawy, gdyż wąskim gardłem jest przepustowość zwykłej sieci transmisji danych, realizującej przesyłanie informacji o podobszarach, oraz obsługa tych transmisji.



Rysunek 6.16: Liczba regionów całkowania względem liczby procesów całkujących

6.4 Równoważenie obciążenia procesorów

W tym bardzo szerokim problemie można wydzielić dwa przeciwstawne nurty, których skrajne warianty to: równoważenie obciążenia procesorów na poziomie systemu, niewidoczne dla programisty, oraz równoważenie obciążenia na poziomie użytkownika, realizowane przez aplikację na podstawie pomiarów jej parametrów realizowanych przez tę aplikację. Proponowana metoda sterowania znakomicie ułatwia implementację drugiego z podejść, oferując także elementy tego pierwszego. Infrastruktura zbierania informacji o stanach procesów może bezpośrednio służyć do gromadzenia aktualnej informacji o obciążeniach procesorów. Sposób określania tego obciążenia może definiować programista - odpowiedni kod w procesach aplikacyjnych powinien raportować wartości obciążenia do monitora. Predykaty sterujące w monitorze definiują strategię podejmowania decyzji równoważących obciążenie. Poprzez te predykaty można określić kiedy, komu i jaką część swych zadań ma oddać dany proces. To wyodrębnienie części decyzyjnej pozwala łatwo modyfikować stosowaną strategię równoważenia obciążenia. Przy przypisaniu po jednym procesie liczącym na procesor, obciążenie procesora można określać z poziomu aplikacji, badając stan samego procesu liczącego, np. długość jego kolejki zadań.

Asynchroniczna reakcja na sygnały sterujące umożliwia naturalną implementację procedur równoważenia obciążenia na poziomie procesu. Podjęcie decyzji komunikowane jest procesowi jako sygnał sterujący, kod aktywowany tym sygnałem zajmuje się wysłaniem lub odebraniem wskazanej części pracy, jaką ma do wykonania proces. I tu wyodrębnienie kodu związanego z równoważeniem obciążenia owocuje łatwością wprowadzania zmian.

Główną wadą równoważenia obciążenia na poziomie użytkownika jest konieczność jej implementacji osobno w każdej aplikacji. Dostępność odpowiedniej infrastruktury w systemie sterowania pronoowanym w niniejszej pracy znakomicie ułatwia to zadanie. W pracy [118] zajmowaliśmy się różnorakimi sposobami realizacji równoważenia obciążenia w oparciu o opisany w rozdziale 5 system PS-GRADE. Przedstawione tam propozycje obejmują następujące rozwiązania:

- Redystrybucja zadań według wskazówek monitora. Monitor otrzymuje raporty na temat obciążenia poszczególnych procesów i wysyła sygnały sterujące przekazaniem zadań z/do wskazanego procesu.
- Modularna konstrukcja aplikacji, która poprzez mechanizm zaprzestania i aktywacji umożli-

wia przeniesienie (migrację) wykonania określonego fragmentu kodu na wskazany procesor. Proponowany mechanizm sterowania odpowiedzialny jest za całość zadań związanych z równoważeniem obciążenia.

- Wykorzystanie biblioteki realizującej tworzenie obrazu procesu (ang. checkpointing) i odtworzenie stanu procesu. Na podstawie danych o obciążeniu monitor podejmuje decyzje o przeniesieniu wykonania procesu do określonego węzła i realizuje to dzięki zastosowaniu zewnętrznej biblioteki.

- Dodatkowe procesy monitorujące stan węzłów obliczenionych umożliwiają wykorzystanie systemowej (dostarczanej przez system operacyjny) miary obciążenia węzłów, także dla węzłów na których nie ma w danym momencie procesów aplikacyjnych

W implementacji problemu komiwojażera zastosowaliśmy równoważenie obciążenia za pomocą pierwszej z wymienionych metod (patrz opis 6.2). Jak już zauważyliśmy, czas bezczynności procesorów okazał się być najmniejszy, gdy równoważenie obciążenia było realizowane przy użyciu proponowanej metody sterowania. Teraz przyjrzymy się tej kwestii dokładniej.

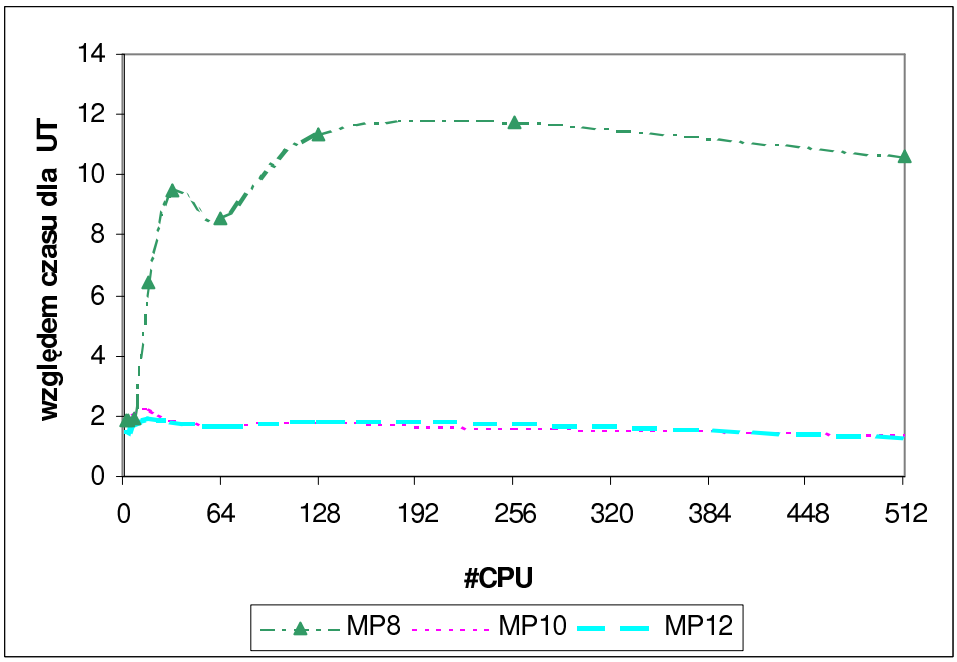
Na rysunku 6.17 widać stosunek czasu poświęconego na równoważenie obciążenia przy zastosowaniu zwykłego przekazywania komunikatów (MP8, MP10, MP12), do poświęconego czasu gdy użyte jest sterowanie przez predykaty (UT). Użycie przekazywania komunikatów daje wyraźnie gorsze efekty. Jeśli przybycie komunikatu weryfikowane jest zbyt rzadko (MP8), to różnica jest ponad jedenastokrotna. Mimo tak znaczącego czasu przeznaczonego na równoważenie obciążenia, efekty tego równoważenia dla wariantu MP8 pozostają gorsze niż dla innych wariantów (patrz rys. 6.7). Gdy często sprawdzamy, czy przybył komunikat (MP10, MP12) równoważenie obciążenia potrzebuje 1,3-2 razy więcej czasu niż w przypadku sterowania przez predykaty. Widać przy tym, że pomiędzy wariantami MP10 i MP12 praktycznie brak różnicy. Dalsze zwiększanie częstotliwości sprawdzenia przybycia komunikatów nie rokuje zatem nadziei na poprawę. Odwrotnie, bardzo częste wywoływanie procedury sprawdzającej obecność komunikatu będzie powodować istotny narzut.

Liczba przesłań zadań pomiędzy procesami zrealizowana w ramach równoważenia obciążenia pokazana jest na rysunku 6.18. Wariant MP8 realizuje najmniej przesłań - zbyt rzadkie sprawdzanie przybycia komunikatu ogranicza liczbę komunikatów, która może być obsłużona w danych czasie. Zestawiając wykresy dla wariantu MP8 na rysunkach 6.17 i 6.18 widzimy, że wariant ten potrzebuje kilkakrotnie więcej czasu niż warianty M10 i MP12 na wykonanie znacząco mniejszej liczby transmisji. Dobór właściwej częstotliwości sprawdzania przybycia komunikatu jest więc niezwykle istotny. Z drugiej strony, sterowanie przez predykaty (UT) jest w stanie dokonać największej liczby przesłań, zużywając na to mniej czasu i uzyskując najmniejszy czas bezczynności procesów (rys. 6.7).

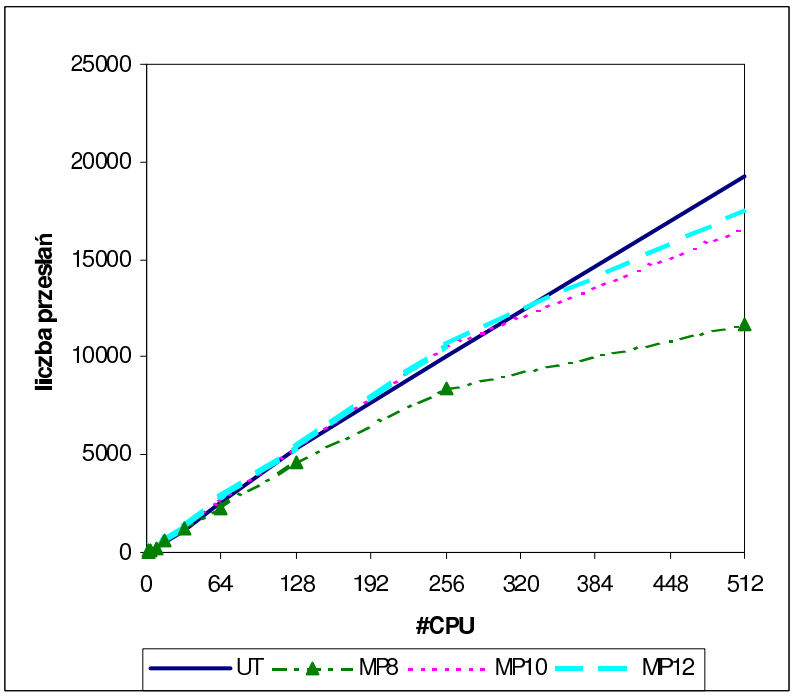
W dalszych symulacjach zweryfikowaliśmy, że opisana metoda równoważenia obciążenia w klastrach wykorzystujących sieć Myrinet pozwala skutecznie rozdzielić obliczenia zainicjowane na pojedynczym procesorze. Początkowa sytuacja: jeden proces w pełni obciążony, a pozostałe bezczynne, pozwaliła dobrze ocenić skuteczność działania wybranej metody równoważenia obciążenia. W ciągu sekundy w obliczenia zostawało wciągnięte średnio 635 początkowo biernych procesorów. Zwróćmy przy tym uwagę, że w problemie komiwojażera nie jest możliwe rozłożenie obciążenia na kilkadziesiąt-kilkaset procesorów w jednym ruchu już na początku obliczeń. Pula podzadań musi najpierw rozrosnąć się na tyle, aby można ją było rozdzielić pomiędzy bezczynne procesory. Opisana powyżej metoda równoważenia obciążenia dobrze tu pasuje. Przydziela ona stopniowo pracę kolejnym procesorom, każdy z nich powiększa swą pulę podzadań umożliwiając dokonanie przydziału procesorom następnym.

6.5 Podsumowanie badań

Przedstawiliśmy szereg wyników eksperymentów w systemach rzeczywistych oraz symulacyjnych, porównujących efektywność sterowania poprzez predykaty globalne z klasycznym sterowaniem z



Rysunek 6.17: Czas na równoważenie obciążenia przy przekazywaniu komunikatów (czas przy sterowaniu predykatami = 1)



Rysunek 6.18: Równoważenie obciążenia - liczba przesłań zadań pomiędzy procesami

użyciem przesyłania komunikatów. Stosując standardowe środowisko sprzętowe uzyskiwaliśmy zauważalnie lepszą wydajność dla programów posługujących się proponowaną metodą sterowania z użyciem predykatów globalnych. Możemy stwierdzić, że dodanie do systemu P-GRADE możliwości sterowania predykatami znacznie poprawiło wydajność tego systemu, pokrywając istniejące w nim oryginalnie braki, prowadzace do nieefektywności w obliczeniach równoległych. Badania symulacyjne natomiast pokazały, że proponowana metoda sterowania wykazuje niewielką przewagę efektywności także względem systemów korzystających z zaawansowanych procedur przekazywania komunikatów, w szerokim zakresie rozmiarów systemu mierzonego liczbą procesorów.

Praktyczne i symulacyjne eksperymenty potwierdziły, że proponowana metoda sterowania przez predykaty wartościowane na stanach globalnych pozwala łatwo i modularnie wyspecyfikować sterowanie aplikacją. Wyodrębnienie predykatów sterujących ułatwia sprecyzowanie warunków sterujących oraz ich modyfikację. Asynchroniczne reakcje na spełnienie predykatów pozwalają budować efektywnie działające aplikacje bez konieczności eksperymentalnego dostosowywania częstości prób odebrania komunikatów.

Wyodrębnienie podsystemu sterowania jako synchronizatora i używanych przez niego kanałów komunikacyjnych pozwoliło w naturalny sposób na wprowadzenie dodatkowego sprzętu, w postaci dedykowanej sieci komunikacyjnej, obsługującego ten podsystem. Symulacje pokazały, że połączenie koncepcji sterowania przez predykaty globalne z rozszerzeniem sprzętowym wspomagającym to rozwiązanie umożliwia uzyskanie znaczącej poprawy wydajności (średnio 3.4 w obliczeniach TSP) względem standardowego systemu wykorzystującego przesyłanie komunikatów.

Część opisanych w tym rozdziale wyników została opublikowana w pracy [26].

Rozdział 7

Podsumowanie wyników rozprawy

W pracy tej zidentyfikowano wyraźną lukę w spektrum istniejących metod sterowania w programach równoległych. Luka ta pozostawiała miejsce dla globalnej metody sterowania, niezależnej od przekazywania danych między procesami i pozwalającej na definiowanie złożonych akcji sterujących wykonaniem procesów. Wypełniając tę lukę zaproponowano globalną metodę sterowania spełniająca powyższe kryteria, dodatkowo dającą programiście gotowy syntetyczny obraz całości stanu aplikacji i pozwalającą na łatwe wyrażanie warunków sterujących wykonaniem programu. Czynności sterujące w procesach pobudzane spełnieniem wspomnianych warunków są ustalane przez programistę zgodnie z wymaganiami danej aplikacji, przez co programista uzyskuje wygodny zestaw wysokopoziomowych prymitywów dostosowanych do określonej aplikacji.

Spośród przeanalizowanych możliwych metod powiązania przebiegu wykonania procesu z decyzjami podejmowanymi przez system sterujący, wybrano asynchroniczną aktywację kodu i porzucanie obliczeń. Metoda ta umożliwia eliminację oczekiwania na podjęcie decyzji oraz wyraźną separację kodu związanego ze sterowaniem na poziomie aplikacji od zasadniczego kodu obliczeniowego procesu.

Dzięki wykorzystaniu silnie spójnych stanów globalnych programów (SCGS) w znacznym stopniu opanowano problem skalowania (tj. efektywności przy zastosowaniu większej liczby procesorów) dla proponowanej globalnej metody sterowania. SCGS uzyskuje się stosunkowo niskim kosztem, przez co możliwe jest uzyskanie sterowania na nich opartego na bieżąco. Hierarchiczne metody sterowania, oparte o hierarchiczne konstruowane SCGS, pozwalają stosować proponowaną metodę sterowania także w większych systemach komputerowych i uzyskiwać lepszą jakość sterowania.

W pracy przedstawiono nowe warianty algorytmów SCGS, posiadające lepsze charakterystyki działania niż algorytm standardowy znany w literaturze. Charakterystyki te zostały przebadane symulacyjnie od strony ich wpływu na jakość uzyskiwanego sterowania. Wykazano, które warianty algorytmu SCGS pracują najefektywniej w jakich warunkach i jakie wartości parametrów opisujących jakość sterowania można otrzymać z ich pomocą.

Zwrócono też uwagę na część systemowo-sprzętową proponowanej metody sterowania. Podano ograniczenia efektywności metody sterowania, na jakie napotykamy stosując współczesne systemy operacyjne i sprzęt komputerowy. Wskazano, które ich cechy są odpowiedzialne za powstanie tych ograniczeń i w niektórych przypadkach zaproponowano pewne udoskonalenia. Zaproponowano m.in. wprowadzenie dodatkowej szybkiej sieci przeznaczonej do przesyłania informacji związanych ze sterowaniem. Symulacje tego rozwiązania potwierdziły jego efektywność.

Proponowana metoda sterowania została zaimplementowana jako rozszerzenie graficznego systemu programowania równoległego P-GRADE. Powstały system PS-GRADE udowodnił realność realizacji sterowania przez predykaty globalne. Graficzna filozofia tworzenia aplikacji P-GRADE dobrze współgrała z postulowanym i uzyskanym w proponowanej metodzie sterowania oddzieleniem elementów odpowiedzialnych za sterowanie od zasadniczej (obliczeniowej) części programu. W PS-GRADE to oddzielenie przybrało postać osobnych okien dialogowych bądź oddzielnych gałęzi w grafie przepływu sterowania, dających programiście jasny przegląd struktury programu równoległego i pozwalający łatwo modyfikować przyjęte na poziomie aplikacji reguły sterowa-

nia. Przeprowadzone porównania efektywności obliczeń realizowanych w klasycznym systemie P-GRADE oraz zmodyfikowanym PS-GRADE wykazały, że sterowanie przez predykaty skojarzone z asynchroniczną reakcją na decyzje sterujące może poprawić efektywność obliczeń. Dodatkowe symulacje wskazały, że ustrukturalnienie sterowania w programie i wysoki poziom abstrakcji oferowany programiście w przypadku sterowania przez predykaty, nie powodują narzutów pogarszających istotnie wydajność, w porównaniu z eksperymentalnie dopracowaną implementacją stosującą klasyczne sterowanie. Co więcej, wykorzystanie spójnego obrazu (stanów) aplikacji może dawać pewną przewagę sterowaniu przez predykaty w niektórych sytuacjach, czego przykładem jest równoważenie obciążenia. Realizacja systemu PS-GRADE niewielkimi środkami przy użyciu standardowego sprzętu w standardowym środowisku systemowym pozwala sądzić, że istnieje tu jeszcze szerokie pole do ulepszeń.

Wyżej przedstawione fakty pozwalają powiedzieć, że w niniejszej rozprawie zrealizowano wszystkie postawione w punkcie 1.4 cele. Sumując całość przedstawionych dokonań możemy stwierdzić, że dowiedziona została prawdziwość postawionej na wstępie niniejszej pracy tezy, a mianowicie, że sterowanie oparte o analizę stanów globalnych aplikacji dla wybranych klas problemów umożliwia poprawną i wygodną strukturalizację sterowania w programach, przy uzyskaniu nie gorszej, a w wielu wypadkach lepszej, wydajności obliczeniowej, niż przy użyciu klasycznych metod sterowania, polegających na specyfikacji sterowania i obliczeń przemieszanych z komunikacją poprzez przesyłanie komunikatów. Zgodnie z planem, obrona tezy wymagała realizacji wytyczonych w punkcie 1.4 zadań pośrednich, które zostały wykonane z powodzeniem.

Uzyskane w niniejszej rozprawie wyniki były szeroko publikowane, m.in. w [14, 15, 24, 16, 29, 27, 17, 18, 28, 20, 19, 21, 22, 26].

Rozdział 8

Dalsze badania

W dotychczasowych pracach dotyczących sterowania poprzez predykaty globalne stosowaliśmy Silnie Spójne Stany Globalne (SCGS). Miało to uzasadnienie semantyczne (stan SCGS zaistniał faktycznie, toteż na jego podstawie można podjąć wiążącą i sensowną decyzję sterującą) i wydajnościowe (koszt obserwacji stanów SCGS jest stosunkowo niewielki). Jednak w pewnych sytuacjach stosowanie SCGS może być niewłaściwe ze względu na to, że monitorowanie silnie spójnych stanów globalnych aplikacji nie jest monitorowaniem ciągłym (patrz 4.2.3). Pomiędzy kolejnymi stanami SCGS może wystąpić interwał nie objęty żadnym stanem dostępnym dla monitora. W przedstawionych w pracy przykładach takie sytuacje nie powodowały błędów, co najwyżej wpływały na uzyskaną jakość sterowania. W przyszłości jednak warto zająć się tym problemem. W szczególności warto rozpatrzyć użycie stanów globalnych budowanych w oparciu o opisaną w punkcie 2.1.3 relację $\xrightarrow{np}$porządkującą częściowo zdarzenia. Spójne stany globalne tworzone na podstawie tej relacji zawierają stany SCGS, oraz stany wypełniające wspomniane interwały pomiędzy kolejnymi SCGS. Takie podejście wykazywałoby dwie zasadnicze różnice względem stosowania SCGS opisanego w tej pracy: pod uwagę brane by były stany potencjalnie, ale niekoniecznie zaistniałe podczas wykonania aplikacji, oraz obowiązywałyby inne zasady (algorytmy i koszty) konstrukcji stanów spójnych.

Wspominana potencjalna możliwość uzyskania sterowania poprawnego przez konstrukcję, dzięki oparciu tego sterowania o predykaty globalne - zwykle wykorzystywane do weryfikacji poprawności działania aplikacji - to kolejny interesujący temat dalszych prac.

W pracy skoncentrowaliśmy się na opracowaniu samej metody sterowania oraz na zbadaniu jej charakterystyk. Zastosowania proponowanej metody szukaliśmy głównie w programach obliczeniowych. Sterowanie przez predykaty globalne w programach obliczeniowych to nowy pomysł. Jednak warto byłoby przyjrzeć się też innym, bardziej klasycznym zastosowaniom. Proponowane skojarzenie decyzji podejmowanych na podstawie wartości predykatów globalnych z asynchronicznym uruchamianem czynności sterujących w procesach można uznać za formę programowania reaktywnego. Programowanie reaktywne stosowane jest często do sterowania procesami przemysłowymi, czy liniami technologicznymi. Czujniki zbierają dane, na których podstawie buduje się stany spójne i podejmuje decyzje, przekazywane do elementów wykonawczych. Wykorzystanie proponowanej metody sterowania w tego typu zastosowaniach to istotny temat dalszych prac.

Użyte przez nas stany silnie spójne oparte są o pojęcie czasu rzeczywistego. Pozwala to stosować oparte na nich predykaty biorące pod uwagę czas. Np. czy pomiędzy kolejnymi zdarzeniami minął określony interwał czasu? Można też zapisywać ciąg obserwowanych wartości w czasie i na podstawie takich przebiegów czasowych podejmować decyzje. Są to zagadnienia bliskie dziedzinie systemów czasu rzeczywistego. Potencjalne zastosowanie sterowania opartego o obserwację silnie spójnych stanów aplikacji właśnie w systemach czasu rzeczywistego to kolejny temat dalszych badań.

Bibliografia

[1] http://www.cs.wmich.edu/ parint/.

[2] Lista 500 najszybszych komputerów na swiecie. www.top500.org.

[3] OMNeT++ discreet event simulation package. http://www.omnet.org.

[4] F. Adelstein and M. Singhal. Priority ethernets: Multimedia support on local area networks. In B. Fuhrt, editor, *Proceedings of the IASTED/ISMM International Conference. Distributed Multimedia Systems and Applications*, pages 45–48, 1994.

[5] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating long messages into the LogP model - one step closer towards a realistic model for parallel computation. In *7th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '95)*, pages 95–105, June 1995.

[6] I. P. Androulakis and C. A. Floudas. Distributed branch and bound algorithms for global optimization. *The IMA Volumes in Mathemathics and its Applications*, 106, "Parallel Processig of Discreet Problems":1–37, 1999.

[7] M. Apte, S. Chakravarthi, J. Padmanabhan, and A. Skjellum. A synchronized real-time linux based myrinet cluster for deterministic high performance computing and MPI/RT. In *9th Intl Workshop on Parallel and Distributed Real- Time Systems WPDRTS'01*, San Francisco, CA, April 2001.

[8] I. T. Association. InfiniBand architecture specification, release 1.0, October 24 2000.

[9] O. Babaoglu and K. Marzullo. *Distributed Systems*, chapter Consistent global states of distributed systems: fundamental concepts and mechanisms. Addison-Wesley, 1995.

[10] M. Baker and H. Ong. A quantitative study on the communication performance of myrinet nework interfaces. Technical Report TR20020301ONGH, DSG/University of Portsmouth, March 2002.

[11] R. Baldoni and G. Melideo. K-dependency vectors: A scalable causality-tracking protocol. In *Proc. Of 11th Euromicro Conf. On Parallel, Distributed and Network-Based Processing PDP'03, Genova, Italy*, pages 219–227. IEEE, 2003.

[12] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and K. Moore. HeNCE: A heterogeneous network computing environment. Technical Report Technical Report UT-CS-93-205, University of Texas, 1993.

[13] G. Berry and G. Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, Nov. 1992.

[14] J. Borkowski. Towards more and flexible synchronization primitives. In *Int. Conf. On Parallel Processing in Electrical Engineering, PARELEC 2000*, pages 18–22, Trois Rivieres, Canada, Aug/Sept 2000. IEEE PR00759.

[15] J. Borkowski. Interrupt and cancellation as synchronization methods. In *Parallel Processing and Applied Mathematics PPAM 2001*, pages 2–8, Naleczow, Poland, Sept 2001. Springer LNCS 2328.

[16] J. Borkowski. Region-based petri nets for modeling interrupts and cancellations. In *Parallel Processing in Electrical Engineering PARELEC 2002*, pages 67–71, Warsaw, Poland, Sept. 2002. IEEE.

[17] J. Borkowski. Global predicates for on-line control of distributed applications. In *International Conference on Parallel Processing and Applied Mathematics PPAM 2003*, pages 269–277, Czestochowa, Poland, Sept 2003. Springer LNCS 3019.

[18] J. Borkowski. Strongly consistent global states detection using relative clock errors. In *ISPDC*, pages 43–49, 2003.

[19] J. Borkowski. Hierarchical detection of strongly consistent global states. In *ISPDC/HeteroPar*, pages 256–261. IEEE, 2004.

[20] J. Borkowski. Parallel program control based on hierarchically detected consistent global states. In *International Conference on Parallel Computing in Electrical Engineering (PARELEC 2004)*, pages 328–333. IEEE, 2004.

[21] J. Borkowski. Strongly consistent global state detection for on-line control of distributed applications. In *12-Th Euromicro Conference on Parallel Distributed and Network-Based Processing, PDP 2004*, pages 126–133, La Coruna, Spain, Feb 2004. IEEE.

[22] J. Borkowski. Measuring and improving quality of parallel application monitoring based on global states. In *Proc. Of the International Symposium on Parallel and Distributed Systems ISPDC'05*, Lille, France, 2005. IEEE.

[23] J. Borkowski. Performance features of global states based application control. In *PDP*, pages 276–279, 2006.

[24] J. Borkowski, D. Kopanski, and M. Tudruj. Adding advanced synchronization to processes in GRADE. In *Int. Conference on Parallel Computing in Electrical Engineering, PARELEC 2002*, number ISBN 0-7695-1730-7, pages 318–322, Warsaw, Poland, Sept 2002. IEEE.

[25] J. Borkowski, D. Kopañski, and M. Tudruj. Dynamic workflow implementation based on synchronizers. In *Submited to Euromicro Web Application, Dubrovnik, Croatia.* IEEE, 2006.

[26] J. Borkowski, D. Kopanski, and M. Tudruj. Parallel irregular computations control based on global predicate monitoring. In *Proc. Of International Symposium on Parallel Computing in Electrical Engineering PARELEC'06*, pages 233–238, Bialystok, Poland, 2006. IEEE.

[27] J. Borkowski, D. Kopanski, and M. Tudruj. Implementing control in parallel programs by synchronization-driven activation and cancellation. In *11-Th Euromicro Conference on Parallel Distributed and Network Based Processing Euro PDP 2003*, pages 316–323, Genova, Italy, Feb. 2003. IEEE Computer Society Press.

[28] J. Borkowski, M. Tudruj, and D. Kopanski. Graphical design tool for parallel programs with execution control. In *Second International Symposium on Parallel and Distributed Computing ISPDC 2003*, pages 37–42, Ljubljana, Slovenia, October 2003. IEEE.

[29] J. Borkowski, M. Tudruj, and D. Kopanski. Parallel program design tool with application control methods based on global states. In *PPAM*, pages 338–343, 2003.

[30] P. Brinch-Hansen. The programming language concurrent pascal. In P. Brinch-Hansen, editor, *The Search for Simplicity: Essays in Parallel Programming*, pages 58–79. IEEE Computer Society, Los Alamitos, California, 1996. chapter 7.

[31] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–459, Apr. 1989.

[32] S. Chakravarthi, A. Pillai, J. Padmanabhan, M. Apte, and A. Skjellum. A fine-grain clock synchronization mechanism for QoS based communication on myrinet. Technical Report MSU000917, 2001.

[33] M. Chittajullu and B. McMillin. History clipping in event-driven distributed systems. In *Proc. Of Parallel and Distributed Computing and Systems*. Actapress, 2000. http://citeseer.ist.psu.edu/390439.html.

[34] F. Christian. Probabilistic clock synchronization. *Distributed Computing*, 3:146–158, 1989.

[35] J. Clausen. Pushing the limits of solvable QAP problems using parallel processing - is nugent30 within reach? *The IMA Volumes in Mathemathics and its Applications*, vol 106:59–74, 1999. ISBN 0-387-98664-2.

[36] R. Cooper and K. Marzullo. Consistent detection of global predicates. In *Proceedings ACM/ONR Workshop on Parallel Distributed Debugging*, pages 163–173, 1991.

[37] A. L. Cox, S. Dwarkadas, H. L. P. Keleher, R. Rajamony, and W. Zwaenepoel. Software versus hardware shared-memory implementation: A case study. In *Proc. Of the 21th Annual International Symposium on Computer Architecture (ISCA'94)*, pages 106–117, 1994.

[38] D. E. Culler, R. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. V. Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings of the 4th SIGPLAN Symp. On Principles and Practices of Parallel Programming*. ACM, May 1993.

[39] E. Denti, A. Natali, and A. Omicini. Expressive power of the ACLT reaction specification language. Technical Report DEIS-LIA-97-009, DEIS - University di Bologna - L I A - Laboratorio d'Informatica Avanzata, 1997.

[40] W. V. der Aalst et al. Workflow patterns. *Distributed and Parallel Databases*, 14 (1):5–51. http://is.tm.tue.nl/research/patterns/.

[41] E. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

[42] E. Dijkstra and C. Scholten. Termination detection for diffusing computations. *Inf. Proc. Lett.*, 11,1:1–4, Aug. 1980.

[43] E. W. Dijkstra. *Cooperating Sequential Processes*, pages 43–112. Academic Press, 1968.

[44] J. Elson, L. Girod, and D. Estrin. Fine-grained network time synchronization using reference broadcasts. In *ACM SIGOPS Operating Systems Review Vol 36: Proc. Of the 5th Symposium on Operating System Design and Implementation (OSDI 2002)*, pages 147–163. ACM Press, Dec 2002.

[45] H. F. et Al. Architecture and performance of the hitachi SR2201 massively parallel processor system. In *11-Th Int. Parallel Processing Symposium, Geneva, Switzerland*, pages 233–241, April 1997.

[46] N. T. et Al. Low latency communication on DIMMnet-1 network interface plugged into a DIMM slot. In *Proceedings of the Int. Conf. On Parallel Computing in Electrical Eng., Warsaw,*, pages 9–14. IEEE, 2002.

[47] C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proc. 11th Australian Computer Science Conference*, pages 55–66, University of Queensland, 1988.

[48] E. Fromentin and M. Raynal. Characterizing and detecting the set of global states seen by all observers of a distributed computation. In *Proceedings of the Fifteenth International Conference on Distributed Computing Systems*, pages 431–438, 1995.

[49] V. Garg and B. Waldecker. Detection of weak unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):299–307, March 1994.

[50] V. K. Garg and N. Mittal. On slicing a distributed computation. In *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS), Phoenix, Arizona, April 2001*, pages 322–329, 2001.

[51] V. K. Garg and B. Waldecker. Detection of strong unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 7, No. 12:1323–1333, December 1996.

[52] A. Geist. Advanced tutorial on PVM 3.4 new features and capabilities. Tutorial presented at EuroPVM-MPI'97, http://www.csm.ornl.gov/pvm/EuroPVM97/, 1997.

[53] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine*. MIT Press, 1994. http://www.netlib.org/pvm3/book/pvm-book.html.

[54] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the (MPI) message passing interface standard. *Parallel Computing*, 22, 1996.

[55] W. Gu, G. Eisenhauer, K. Schwan, and J. Vetter. Falcon: On-line monitoring and steering of large-scale parallel programs. *Concurrency: Practice and Experience*, 6(2), April-June 1998.

[56] R. Gupta. The fuzzy barrier: A mechanism for high speed synchronization of processors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, pages 54–63, Boston, Massachusetts, 1989. ACM Press.

[57] A. Gursoy and L. A. Kale. Dagger: Combining benefits of synchronous nad asynchronous communication styles. PPS94, 1994.

[58] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, Sept. 1991.

[59] J. Hicks, D. Chiou, B. S. Ang, and Arvind. Performance studies of id on the monsoon dataflow system. Technical Report CSG Memo 345-3, MIT, Computation Structeres Group, Lab. for Computer Science, http://www.csg.lcs.mit.edu:8001/monsoon/monsoon-performance/monsoon-performance.html.

[60] C. Hoare. *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science, 1985. ISBN 0-13-153271-5.

[61] D. Hoechtl and U. Schmidt. Long-term evaluation of GPS timing receiver failures. In *Proceedings of the 29th IEEE Precise Time and Time Interval Systems and Application Meeting (PTTI'97)*, pages 165–180, Long Beach, California, Dec 1997.

[62] H. Horauer. *Clock Synchronization in Distributed Systems - Architecture and Evaluation of Ethernet-Based Network Interfaces with Support for Precision Clock Synchronization*. PhD thesis, Vienna University of Technology, 2004.

[63] R. Horst and H. Tuy. *Global Optimization. Deterministic Appoaches*. Springer, 1996. 3rd edition.

[64] H.Trienekens. Parallel branch and bound and anomalies. Technical Report EUR-FEW-CS-89-01, Erasmus University Rotterdam, 1989.

[65] M. Hurfin, M. Mizuno, M. Raynal, and M. Singhal. Efficient distributed detection of conjunctions of local predicates. *IEEE Trans. on Software Eng*, Vol. 24, No. 8:664–677, August 1998.

[66] P. Johansson. IPv4 over IEEE 1394, 1999. RFC 2734.

[67] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach. Crl: Highperformance all-software distributed shared memory. In *ACM Symp. On Operating Systems Principles*, 1995.

[68] A. Jungbauer and B. Czyżewska. Rozszerzenie interfejsu edycji graficznej programów równoległych systemu p-GRADE dla nowych mechanizmów synchronizujšcych, 2004. inżynierska praca dyplomowa w PJWSTK.

[69] P. Kacsuk, G. Dózsa, and T. Fadgyas. GRADE: A graphical programming environment for PVM applications. In *Proc. Of the 5th Euromicro Workshop on Parallel and Distributed Processing*, pages 358–365, London, 1997.

[70] P. Kacsuk, G. Dózsa, and R. Lovas. *Parallel Program Development for Cluster Computing: Methodology, Tools and Integrated Environments*, chapter The GRADE Graphical Parallel Programming Environment (chapter 10), pages 231–247. Nova Science Publishers New York, 2001.

[71] K.Chandy and L.Lamport. Distributed snapshots - determining global states of distributed systems. *ACM Transactions on Computer Systems*, vol 3. No 1:63–75, 1985.

[72] R. B. Kieburtz. Reactive functional programming. In D. Gries and W.-P. de Roever, editors, *Programming Concepts and Methods (PROCOMET 98)*. Chapman-Hall, 1998.

[73] D. Kopański. *Instrukcja Systemu PS-GRADE*. PJWSTK, 2005.

[74] D. Kopański. Wspomaganie systemu graficznej edycji programów równoległych p-GRADE nowymi mechanizmami synchronizacji procesów. Master's thesis, PJWSTK, 2005.

[75] D. Kopanski, J. Borkowski, and M. Tudruj. Co-ordination of parallel grid applications using synchronizers. In *International Conference on Parallel Computing in Electrical Engineering PARELEC 2004*, pages 323–327, Dresden, Germany, 2004. IEEE.

[76] D. Kopański, J. Borkowski, and M. Tudruj. Application control on grid using predicates defined on states of cooperating parallel programs. In *Proc. Of. Euromicro Parallel and Distributed Processing PDP'06*, pages 248–255, Montbeliard, France, 2006. IEEE.

[77] A. Koubaa, A. Jarraya, and Y. Song. SBM protocol for providing real-time QoS in ethernet LANs. In *1st Intl. Workshop on Real-Time LANs in the Internet Age, Satellite Event to 14th Euromicro Conference on Real-Time Systems*, Technical University of Vienna, Austria, June 2002.

[78] A. D. Kshemkalyani. A fine-grained modality classification for global predicates. *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, No.8:807–816, Aug. 2003.

[79] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[80] D. Lee, R. Attias, A. Pur, R. Sengupta, S. Tripakis, and P. Varaiya. A wireless token ring protocol for intelligent transportation systems. In *Proc of the IEEE 4th International Conference on Intelligent Transportation Systems*, August 2001.

[81] J. Liu, W. Jiang, P. Wyckoff, D. Panda, D. Ashton, D. Buntinas, W. Gropp, and B. Toonen. Design and implementation of mpich2 over infiniband with rdma support. In *Int'l Parallel and Distributed Processing Symposium (IPDPS 04)*, 2004.

[82] J. Loeser and H. Haertig. Low-latency hard real-time communication over switched ethernet. In *16th Euromicro Conference on Real-Time Systems*, Catania, Sicily, July 2004.

[83] R. Lovas, G. Dózsa, P. Kacsuk, N. Podhorszki, and D. Drótos. Workflow support for complex grid applications: Integrated and portal solutions. In *Proceedings of 2nd European Across Grids Conference*, Nicosia, Cyprus, 2004.

[84] H. Ltd. *HI-UX/MPP - Remote DMA -C- User's Guide Manual Number: 6A20-3-021-10(E), Second Edition*, January 1997.

[85] K. Marzullo and G. Neiger. Detection of global state predicates. In S. Toueg, P. G. Spirakis, and L. M. Kirousis, editors, *Distributed Algorithms, 5th International Workshop, WDAG '91, Delphi, Greece, LNCS 579*. Springer, 1991.

[86] K. Marzullo and M. D. Wood. Tools for constructing distributed reactive systems. Technical Report 14853, Cornell Univerity, Department of Computer Science, Ithaca, New York, Feb. 1991. http://citeseer.nj.nec.com/145302.html.

[87] F. Mattern. Virtual time and global states in distributed systems. In M. C. et Al., editor, *Proc. Workshop on Parallel and Distributed Algorithms, Chateau de Bonas*, pages 215–226. Elsevier / North Holland, 1989.

[88] J. Mayo. *Global State Predicates in Rough Real Time*. PhD thesis, Department of Computer Science, The College of William and Mary in Virginia, 1997.

[89] Message Passing Interface Forum, http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html. *MPI: A Message Passing Interface Standard Ver. 1.1*, 1995.

[90] Message Passing Interface Forum, http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html. *MPI-2: Extensions to the Message-Passing Interface*, 1997.

[91] D. Mills. RFC-1305: Network time protocol (version 3): Specification, implementation and analysis, 1992.

[92] M. Minas. Detecting quantified global predicates in parallel programs. In *Proceedings of Europar 95, LNCS, Vol. 966*, pages 403–414. Springer, 1995.

[93] N. Mittal and V. K. Garg. On detecting global predicates in distributed computations. In *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS), Phoenix, Arizona,*, pages 3–10. IEEE, April 2001.

[94] M. Okamoto, K. Aida, M. Miyazawa, H. Honda, and H. Kasahara. A hierarchical macro-dataflow computation scheme for OSCAR multi-grain compiler. *Transactions of Information Processing*, 35(4):513–521, Apr. 1994.

[95] J. Ostrowiński. Opracowanie biblioteki funkcji wewnêtrznych p-GRADE dla nowych mechanizmów synchronizacji procesów. Master's thesis, PJWSTK, 2004.

[96] G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.

[97] P.Kacsuk and M.Amamiya. Prolog on the multithreaded datarol-II machine based on the logicflow execution model. In *, Proceedings of SPDP'96*, New Orleans, 1996.

[98] J. Protic, M. Tomasevic, and V. Milutinovic. *Distributed Shared Memory: Concepts and Systems*. IEEE Computer Soc. Press, Los Alamitos, Calif, 1998.

[99] Z. Radovic and E. Hagersten. Efficient synchronization for nonuniform communication architectures. In *Supercomputing 2002, Baltimore, USA*, 2002. www.supercomp.org/sc2002/paperpdfs/pap.pap221.pdf.

[100] M. Ramachandran and M. Singhal. Distributed semaphores. Technical Report OSU-CISRC-6/94-TR34, Ohio State Univ., 1994.

[101] M. Raynal and J. Helary. *Synchronization and Control of Distributed Systems and Programs.* Wiley, England, 1990. ISBN 0-471-92453-9.

[102] J. R. Rice. A metalgorithm for adaptive quadrature. *Journal of the Association for Computing Machinery*, 22:61–82, 1975.

[103] U. Schmid, J. Klasek, T. Mandl, H. Nachtnebel, G. Cadek, and N. Kero. A network time interface m-module for distributing GPS-time over LANs. *Journal of Real-Time Systems*, 18,1:24–57, Jan. 2000.

[104] R. Schürer. Optimal communication interval for parallel adaptive integration. *Parallel and Distributed Computing Practices*, 5 (3):269–278, 2002.

[105] R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 7:149–174, 1994.

[106] M. Scott, L. Michael, and M. Maged. The topological barrier: A synchronization abstraction for regularly-structured parallel applications. Technical Report TR605, URCSD, 1996.

[107] K. G. Shin and P. Ramanathan. Clock synchronization of a large mutliprocessor system in the presence of faults. *IEEE transactions on Computers*, C-36(1):2–12, Jan. 1987.

[108] J. Silc, B. Robic, and T. Ungerer. Asynchrony in parallel computing: From dataflow to multithreading. *Parallel and Distributed Computing Practices*, Vol.1, No.1, March 1998.

[109] M. Singhal and A. Kshemkalyani. An efficient implementation of vector clocks. *Information Processing Letters*, 43:47–52, Aug. 1992.

[110] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.

[111] A. Smyk and M. Tudruj. RDMA control support for fine-grain parallel computations. In *12th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP'04)*, pages 208–214, La Coruna, Spain, 2004. IEEE.

[112] S. D. Stoller. Detecting global predicates in distributed systems with clocks. *Distributed Computing*, vol. 13, issue 2:85–98, 2000.

[113] S. D. Stoller, L. Unnikrishnan, and Y. A. Liu. Efficient detection of global properties in distributed systems using partial-order methods. In *In Proceedings of the 12th International Conference on Computer-Aided Verification (CAV), LNCS 1855*, pages 264–279. Springer, July 2000.

[114] H. Tanaka, Y. Muraoka, M. Amamiya, N. Saito, and S. Tomita. *Massively Parallel Processing System Jump-1*, volume ISBN: 9051992629. IOS Press, 1996.

[115] A. Tarafdar and V. Garg. Predicate control for active debugging of distributed programs. In *IPPS/Symposium on Distributed and Parallel Debugging.* IEEE, 1998.

[116] M. Tudruj. Fine-grained global control constructs for parallel programming environments. In A. Bakkers, editor, *Parallel Programming and Java: WoTUG-20.* IOS, 1997.

[117] M. Tudruj, J. Borkowski, and D. Kopanski. Graphical design of parallel programs with control based on global applications states using an extended p-grade system. In *DAPSYS*, pages 113–120, 2004.

[118] M. Tudruj, J. Borkowski, and D. Kopanski. Load balancing with migration based on synchronizers in PS-GRADE graphical tool. In *Proc. Of the International Symposium on Parallel and Distributed Computing ISPDC'05*, Lille, France, 2005. IEEE.

[119] M. Tudruj and P. Kacsuk. Extending GRADE towards explicit process synchronization in parallel programs. *Computers and Artificial Intelligence*, 17, No. 5:507–516, 1998.

[120] M. Tudruj and L. Masko. Task scheduling for dynamically configurable multiple SMP clusters based on extended DSC approach. In R. Wyrzykowski, J. Dongarra, M. Paprzycki, and J. Waniewski, editors, *Parallel Processing and Applied Mathematics : 4th International Conference, PPAM 2001*, volume 2328 of *LNCS*, Naleczow, Poland, 2003. Springer-Verlag.

[121] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: A mechanism for integrated communication and computation. In *19th International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, 1992.

[122] S. Walukiewicz. *Integer Programming.* PWN and Kluwer Academic Publishers, 1991.

[123] J. Wu. *Distributed System Design.* CRC Press, 1999. ISBN 0-8493-3178-1.

[124] R. Zanny, K. Kaugars, and E. de Doncker. Scalability of branch-and-bound and adaptive integration. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'01)*, pages 674–680, 2001.