CD-ROM: PostgreSQL 7.0, GIMP, narzędzia i inne

6/2000

magazyn

Linux & UNIX

WAP - Internet w komórce!

Bezpieczeństwo sieciowe

Power C

Tworzenie bugreportów

Oprogramowanie Pinfo

GUST TeX System

Debian GNU/Linux

CD ROM

NR 6(22)/2000 CZERWIEC CENA 17 ZŁ ISSN 1506-1299

magazyn Linux & UNIX

Adres kontaktowy: magazyn@tao.com.pl

Biznes 8

Debian/GNU z perspektywy administratora - część II 10
Grzegorz Jacek Nalepa

Zabezpieczanie danych w sieciach komputerowych. Nowe streamery dla początkujących i zaawansowanych 17
Volker Langer, Tandberg Data GmbH

Oprogramowanie Pinfo 21
Przemysław Borys

GUST TEX System (TEX Live 5b) 23
Stanisław Wawrykiewicz

Bezpieczeństwo - Aktualności 26

Programowanie

Model obiektu GTK+ 27
Marek Sawerwain

Jak pisać dobre bugreporty 33
Marcin Marszałek

Power C 37
Piotr Koch

Technologie Sieciowe

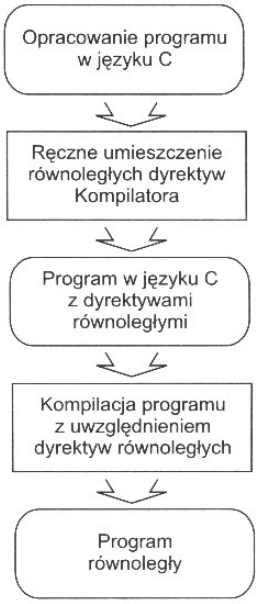
Technologia WAP - Internet w komórce 44
Wojciech Bojdoł

„Komputer sieciowy" a bezpieczeństwo 46
Mariusz Stawowski

Firma TAO Systems Polska, wydawca miesięcznika Linux & UNIX poszukuje osób do testowania i opisywania programów działających w systemie Linux. Zainteresowane osoby prosimy o kontakt z Redaktorem Naczelnym - Marcinem Koprowskim.
tel. 0501-134656, e-mail: marcin.koprowski@tao.com.pl

Power C

Piotr Koch



Rysunek 1. Opracowanie i kompilacja programu w języku Power C.

Power C jest kompilatorem języka C dla platformy SGI Irix, który dokonuje automatycznej analizy sekwencyjnego algorytmu w celu ustalenia możliwości równoległego wykonania zawartych w nim konstrukcji, a następnie generuje wykonywalny kod programu wykorzystujący dostępną równoległość komputera. Power C umożliwia rekompilację istniejących sekwencyjnych programów C tak, aby mogły zostać efektywnie wykonane na architekturach wieloprocesorowych redukując przy tym koszty czasowe ręcznej reorganizacji kodu. W konsekwencji program pozostaje przenośny, a programista nie musi koncentrować się na jego dostosowaniu do specyfiki systemu.

Power C Analyzer (PCA) jest preprocesorem, który dokonuje optymalizacji kodu języka C przez wykrycie w nim potencjalnej równoległości. PCA umieszcza w programie jawne dyrektywy kompilatora umożliwiające równoległe wykonanie fragmentów kodu, w których nie występują zależności między danymi. PCA nie zmienia przy tym algorytmu działania programu pomimo, że może zmodyfikować pewne jego fragmenty w celu poprawy efektywności wykonania. Ponadto, usunięcie bądź zignorowanie dyrektyw Power C czyni taki kod poprawnym programem sekwencyjnym.

Automatyczna generacja równoległego kodu przy użyciu PCA może zostać zastąpiona ręcznym umieszczeniem w programie równoległych dyrektyw kompilatora. Takie podejście jest zalecane przez jego twórców, zwłaszcza przy jednoczesnym wsparciu PCA. Warto również zauważyć, że dyrektywy kompilatora Power C są wzorowane na równoległych dyrektywach Fortranu, zalecanych przez Parallel Computing Forum.

Power C nie jest rozszerzeniem notacji języka C, ale zbiorem dyrektyw kompilatora umożliwiających równoległe wykonywanie fragmentów kodu programu (patrz Rysunek 1).

Etapy opracowywania programów równoległych

Istnieje metodologiczne podejście określające zakres elementów uwzględnianych przy opracowywaniu programów równoległych. Zastosowanie tego mechanizmu umożliwia ocenę skuteczności alternatywnych rozwiązań. W metodologii projektowania programów równoległych kwestie niezależne od środowiska sprzętowego rozpatrywane są przed zagadnieniami związanymi ze specyficznymi aspektami architektury docelowej maszyny. Na proces opracowywania programu równoległego w takim ujęciu składają się cztery odrębne etapy: podziału (ang. partitioning), komunikacji (ang. communication), aglomeracji (ang. agglomeration) oraz planowania (ang. Mapping).

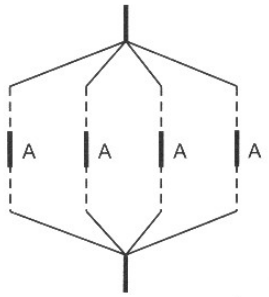
Etap podziału

Etap podziału skupia swoją uwagę na wskazaniu tych wszystkich części problemu, które mogą zostać wykonane w sposób równoległy. W rezultacie uzyskuje się znaczną ilość małych zadań realizujących całościowe zagadnienie. Bez znaczenia pozostaje natomiast liczba procesorów dostępnych na docelowej maszynie wykonawczej. Prawidłowe przeprowadzenie podziału wymaga uwzględnienia zarówno obliczeń związanych z problemem jak i danych, na których te obliczenia operują. Na tym etapie projektowania staramy się uniknąć powielania obliczeń i danych, definiując zadania dzielące obliczenia oraz dane w rozłączne zbiory. Możliwe są dwa sposoby realizacji tego etapu:

- dekompozycja danych (ang. data decomposition) -w pierwszej kolejności rozkładowi poddawane są dane związane z problemem. Podstawą podziału mogą być zarówno dane wejściowe, końcowe wyniki obliczeń jak i pośrednie wartości otrzymywane w programie. Dane najkorzystniej jest rozdzielić na bardzo małe kawałki o mniej więcej równych wielkościach.
- dekompozycja funkcjonalna (ang. functional decomposition) - reprezentuje alternatywny sposób myślenia o strukturze problemu. Podejście to skupia swoją uwagę bardziej na samych obliczeniach niż na manipulowanych przez nie danych.

Etap komunikacji

Uzyskane na etapie podziału zadania mogą wykonywać się jednocześnie lecz niekoniecznie niezależnie. Obliczenia dokonywane w jednym zadaniu często



Rysunek 2. Działanie dyrektywy #pragma parallel.

wymagają przesłania danych przydzielonych do innego zadania. Przepływ informacji jest analizowany na etapie komunikacji.

Etap aglomeracji

Dwa pierwsze etapy budowy programu równoległego dzieliły obliczenia na małe zbiory zadań i wprowadzały między nimi komunikację. W rezultacie otrzymany algorytm wciąż pozostawał abstrakcją nie uwzględniającą specyficznych właściwości komputera równoległego. Może to prowadzić do niskiej wydajności programu, gdyż na przykład duża liczba zadań konkurujących o cykle niewielu procesorów powoduje znaczne straty czasowe spowodowane przełączaniem kontekstu.

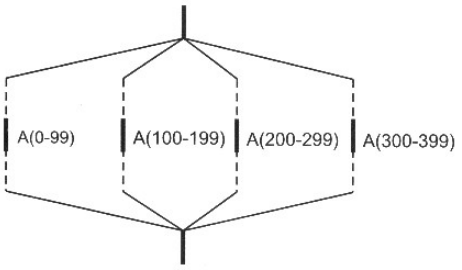
W trzeciej fazie projektowania programu równoległego, jakim jest aglomeracja, ponownej rozwadze zostają poddane decyzje podjęte na etapach podziału i komunikacji tak, aby poprawić efektywność algorytmu realizowanego na pewnej klasie komputera równoległego. W szczególności rozwadze zostaje poddana użyteczność połączenia (lub inaczej aglomeracji) zadań tak, aby uzyskać mniejszą ich liczbę lecz o zwiększonej wielkości. Należy się również przyjrzeć korzyściom płynącym z powielania danych i obliczeń.

Etap planowania

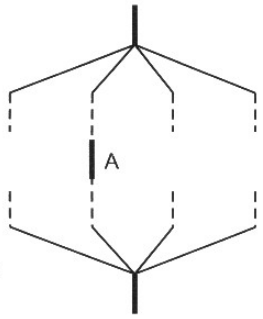
W ostatnim etapie procesu projektowania algorytmu równoległego zostaje określone miejsce wykonania każdego zadania. Problem przydziału procesora nie powstaje w systemach jednoprocesorowych, w których taką odpowiedzialnością obciąża się system operacyjny. W przypadku istnienia wielu procesorów problem planowania ich pracy odpowiednio się komplikuje. Celem stosowania algorytmów planowania jest minimalizacja całkowitego czasu wykonania programu. Możliwe są dwie strategie realizacji tego celu:

- umieszczenie zadań możliwych do jednoczesnego wykonania na różnych procesorach, tak by zwiększyć współbieżność.
- umieszczenie zadań często komunikujących się na tym samym procesorze, tak by zwiększyć lokalność.

Obie strategie mogą czasami pozostawać w konflikcie wymagającym dokonania optymalnego wyboru. Dodatkowo limity zasobów mogą ograniczać liczbę możliwych zadań przydzielanych jednemu procesorowi.



Rysunek 4. Ilustracja działania przykładu z pętlą równoległą.



Rysunek 3. Działanie dyrektywy #pragma one processor.

Podział

W języku Power C pojęcie obszaru równoległego (ang. parallel region) określa fragment programu, w którym pętle i niezależne fragmenty kodu mogą zostać wykonane równolegle. Stworzenie programu złożonego z rozległych obszarów równoległych może przyczynić się do znacznej poprawy wydajności kodu w sposób niemożliwy do osiągnięcia przez kolejne wykonanie oddzielnych pętli równoległych. W szczególności obszar równoległy oszczędza programiście kłopotu związanego z każdorazowym przygotowaniem kolejnych fragmentów programu do równoległego wykonania.

Obszar równoległy składa się z wielu konstrukcji wspólnie realizujących całość problemu. Najprostszy obszar równoległy może zawierać wyłącznie jedną konstrukcję, którą przeważnie jest pętla równoległa. Objęcie obszarem równoległym wyrażeń wymagających sekwencyjnego wykonania w oczywisty sposób obniża wydajność takiego programu, głównie przez konieczność realizacji synchronizacji. Równie nieekonomicznym posunięciem jest tworzenie obszaru równoległego zawierającego wyłącznie jeden niezależny kod programu.

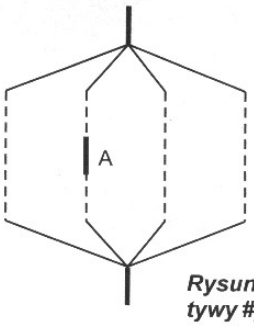
Obszar równoległy

Użycie w pewnym punkcie programu dyrektywy #pragma parallel informuje kompilator o początku obszaru równoległego. Kod programu stanowiący obszar równoległy jest ograniczony klamrami i występuje bezpośrednio po tej dyrektywie. Składnia dyrektywy wskazującej na równoległy obszar programu jest następująca:

```
#pragma parallel shared (variables) byvalue (variables)
#pragma local (variables) opcjonalne modyfikatory
```

Dyrektywa #pragma parallel posiada sześć modyfikatorów:

- shared(variable names) - wskazuje kompilatorowi nazwy wszystkich zmiennych współdzielonych przez wątki.
- byvalue(variable names) - wskazuje kompilatorowi nazwy zmiennych, które powinny być współdzielone przez wątki, ale współdzielenie powinno odbywać się przez wartość, nie przez referencje. Modyfikator ten powinno stosować się ze szczególną ostrożnością. Niewłaściwe użycie modyfi-



Rysunek 5. Działanie dyrektywy #pragma independent.

katora byvalue może skutkować niedeterministycznym zachowaniem się programu, prowadzić do trudnych do wykrycia błędów w programie oraz nieoczekiwanych wyników jego działania. Na liście zmiennych modyfikatora byvalue mogą się znaleźć zmienne, które są skalarem, nie były wcześniej użyte na liście modyfikatora shared i będą używane wyłącznie do odczytu.

- local (variable names) - wskazuje kompilatorowi nazwy wszystkich zmiennych, które muszą pozostać prywatne dla każdego wątku (modyfikacje tej zmiennej w jednym wątku nie wpływają na zmianę jej wartości w innym, nazwa zmiennej jest wykorzystywana lokalnie).
- if(integer valued expr) - ustanawia warunek, który określa czy obszar równoległy zostanie wykonany równolegle czy sekwencyjnie. Ponieważ podczas kompilacji nie zawsze istnieje możliwość określenia wielkości kodu wykonywanego w obszarze równoległym (z tego względu, że np. liczba iteracji pętli równoległej może zależeć od wartości innych obliczeń) dlatego, aby zmniejszyć koszt powoływania nieefektywnej liczby równoległych wątków można uzależnić utworzenie obszaru równoległego od modyfikatora if.
- [no]ifinline - umożliwia kompilatorowi dokonanie optymalizacji kodu w sytuacji, gdy został użyty modyfikator if. Użycie modyfikatora ifinline powoduje niewielki zwiększenie wielkości kodu, lecz jednocześnie wzrost szybkości jego wykonania. Działanie modyfikatora noifinline jest odwrotne.

```
numthreads(min = expr1; max = expr2)
numthreads(percent = expr)
numthreads(expr)
```

Wskazuje kompilatorowi liczbę dostępnych wątków do wykonania obszaru równoległego. Człon min nakazuje kompilatorowi równoległe wykonanie tej części kodu jeżeli co najmniej expr1 wątków jest dostępnych. Jednocześnie człon max wskazuje, że co najwyżej expr2 wątków powinno zostać użytych do wykonania bloku równoległego. W innej postaci modyfikatora numthreads(percent = expr) człon percent nakazuje kompilatorowi użycie expr procent dostępnych wątków. Jeżeli programista pragnie wykonać pętlę równoległą jednocześnie z innym kodem, wówczas powinien wykorzystać ten modyfikator do wskazania, aby kompilator użył jedynie pewnej części dostępnych procesorów.

Działanie dyrektywy #pragma parallel ilustruje poniższy przykład 1 oraz rysunek 2.

```
...
#pragma parallel local ...
{
{
...          A
}
}
```

Przykład 1. Użycie dyrektywy #pragma parallel.

Istnieje możliwość zawarcie w obszarze równoległym dyrektywy #pragma one processor wskazującej kompilatorowi, że następujące po niej wyrażenie powinno zostać wykonane wyłącznie przez jeden wątek. Użycie tej pragmy jest następujące (rysunek 3):

```
...
#pragma parallel local ...
{ ...
#pragma one processor
{
...          A
}
}
```

Przykład 2. Użycie dyrektywy #pragma one processor.

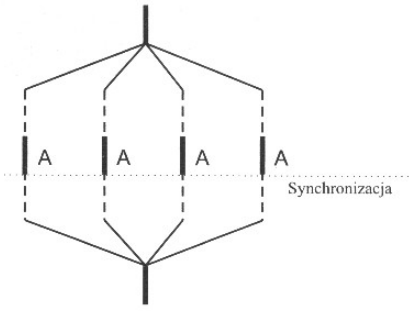
Pętle równoległe

Dyrektywa #pragma pfor wykonuje równolegle pętlę for, gdy spełnione są poniższe warunki:

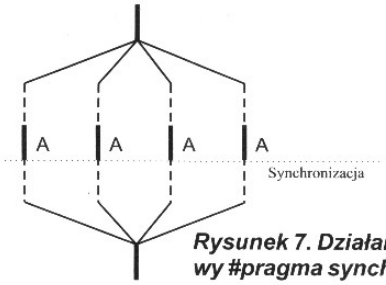
- wszystkie wartości zmiennej indeksowej są niezależne od obliczeń w poszczególnych iteracjach
- wszystkie iteracje są niezależne od siebie, tzn. nie występują między nimi zależności danych
- znana jest liczba iteracji potrzebnych do wykonywania pętli
- `#pragma pfor` jest zawarta w obszarze równoległym

Składnia dyrektywy #pragma pfor jest następująca:

```
#pragma pfor iterate () opcjonalne modyfikatory
{code}
```



Rysunek. 6. Działanie dyrektywy #pragma critical.



Rysunek 7. Działanie dyrektywy #pragma synchronize.

Poniższy przykład ilustruje działanie dyrektywy (rysunek 4):

```
...
#pragma parallel local (i)...
{
#pragma pfor iterate (i=0;400;1)
for(i=0;i<400;i++)
{
...                A
}
}...
```

Przykład 3. Użycie dyrektywy #pragma pfor.

Pętla równoległa posiada trzy modyfikatory:

- iterate(index variable = expr2; expr2; expr3) - dostarcza kompilatorowi informacji o liczbie iteracji pętli i umożliwia dokonania ich podziału między poszczególne wątki. W większości wypadków index variable identyfikuje zmienną indeksową, expr1 ustanawia początkową wartość indeksu pętli, expr2 określa liczbę iteracji zaś expr3 informuje o inkrementacji wartości zmiennej indeksowej.
- schedtype(type) - informuje kompilator w jaki sposób powinien rozdzielić iteracje pętli między dostępne procesory. Możliwe są strategie: simple, runtime, dynamic, interleave i gss.
- chunksize(expr) - informuje kompilator jak dużych porcji iteracji powinien użyć dla strategii dynamic i interleave ich rozdziału do procesorów.

Niezależne segmenty kodu

Dyrektywa #pragma independent nakazuje kompilatorowi uruchomienie objętego nią fragmentu programu równolegle w stosunku do reszty kodu wewnątrz obszaru równoległego. Działanie tej dyrektywy polega zatem na utworzeniu niezależnego fragmentu kodu wykonywanego przez jeden tylko wątek, co pozwala na jednoczesną realizację heterogenicznych obliczeń w programie. Składnia dyrektywy #pragma independent jest następująca:

```
#pragma independent
{code}
```

zaś jej działanie ilustruje poniższy przykład:

```
#pragma parallel local ...
{
#pragma independent
{
...          A
}
}...
```

Przykład 4. Użycie dyrektywy #pragma independent.

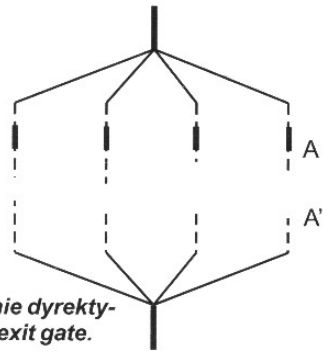
Komunikacja

Równoległe wątki języka Power C komunikują się między sobą przez współdzielone struktury danych. W szczególności użycie przy deklaracji obszaru równoległego modyfikatora shared określa listę zmiennych współdzielonych przez wszystkie utworzone wątki. Kompilator Power C nie sprawdza i nie przestrzega programisty przed naruszeniem zależności między danymi. Power C pozostawia to zadanie analizatorowi PCA, dlatego umieszczając ręcznie dyrektywy kompilatora w celu zrównoleglenia sekwencyjnego kodu programista samodzielnie powinien zadbać o poprawność programu przez zapewnienie prawidłowego dostępu do współdzielonych danych.

Atomowość

Użycie omówionej na etapie podziału dyrektywy #pragma one processor powoduje niepodzielne wykonanie objętego nią kodu przez jeden tylko procesor. Pierwszy z wątków, który podczas realizacji obszaru równoległego dojdzie do punktu w programie, w którym znajduje się dyrektywa #pragma one processor podejmie wykonanie zawartych w niej wyrażeń. Każdy inny wątek, który następnie dojdzie do tego punktu musi oczekiwać dopóki nie zostaną w całości wykonane wyrażenia pierwszego wątku, po czym ominie ten fragment programu i ponownie podejmie swój przebieg. Jeżeli jeden z wątków zakończy realizację bloku #pragma one processor zanim pozostałe wątki natkną się na ten fragment kodu, wówczas takie wątki bez jakiegokolwiek oczekiwania będą mogły kontynuować swój przebieg, oczywiście nie podejmując wykonania bloku #pragma one processor.

Czasami, szczególnie w przypadku pętli równoległej, homogeniczne obliczenia muszą zostać wykonane jednocześnie przez wszystkie równoległe wątki programu. Gdy w realizowanych przez nie wy-



Rysunek 8. Działanie dyrektywy #pragma enter/exit gate.

rażeniach występują zależności między danymi, których przeniesienie poza obszar równoległy nie jest możliwe, wówczas dla zapewnienia niepodzielnego wykonania sekcji krytycznej należy użyć dyrektywy #pragma critical. Objęty nią blok programu zostanie w danej chwili wykonany wyłącznie przez jeden tylko wątek. Jeżeli pewien wątek realizuje w danej chwili operacje na wyrażeniach chronionych dostępem atomowym, wówczas dowolny inny wątek, który pragnie wykonać te wyrażenia musi oczekiwać na zwolnienie sekcji krytycznej kodu. Składnia dyrektywy #pragma critical jest następująca:

```
#pragma critical
{code}
```

zaś jej działanie można zilustrować następująco (rysunek 5):

```
...
#pragma parallel ...
{ ...
#pragma critical
{
...          A
}
}
```

Przykład 5. Użycie dyrektywy #pragma critical.

Nie należy zapominać, że operacje atomowe redukują równoległość kodu a przez to i efektywność całego programu. Stąd programista powinien dążyć do redukcji wielkości kodu wykonywanego atomowo.

Synchronizacja

W Power C dyrektywami kompilatora, które umożliwiają koordynację wykonania równoległych wątków są: #pragma synchronize oraz #pragma enter/exit gate.

#pragma synchronize jest klasyczną konstrukcją bariery. Umieszczenie w pewnym punkcie obszaru równoległego tej dyrektywy sprawia, że wątek nie może kontynuować wykonania następujących po nim wyrażeń dopóki wszystkie wątki nie dotrą do tego miejsca programu. Składnia dyrektywy #pragma synchronize jest następująca:

```
#pragma synchronize
```

zaś jej działanie ilustruje poniższy przykład (rysunek 6):

```
...
#pragma parallel ...
{
#pragma synchronize        A
...
}...
```

Przykład 6. Użycie dyrektywy #pragma synchronize.

Programista może użyć dwóch dodatkowych dyrektyw kompilatora do koordynacji realizacji kodu wewnątrz obszaru równoległego: #pragma enter gate i #pragma exit gate. Implementują one w programie specjalną barierę zwaną bramą. Jej działanie polega na tym, że żaden wątek nie może opuścić bramy dopóki wszystkie wątki w nią nie wejdą. Ta konstrukcja dostarcza programiście większej elastyczności w zarządzaniu zależnościami między wątkami wewnątrz obszaru równoległego. Składnia obu dyrektyw jest następująca:

```
#pragma enter gate (name)    #pragma exit gate (name)
```

zaś jej działanie ilustruje poniższy przykład (rysunek 7):

```
#pragma parallel ...
{ ...
#pragma enter gate (x) A
...
#pragma exit gate (x)          A'
...
}...
```

Przykład 7. Użycie dyrektywy #pragma enter gate i #pragma exit gate.

Użycie bramy zamiast dyrektywy #pragma synchronize może przyczynić się do poprawy efektywności programu równoległego. Dla przykłady, jeżeli w poniższym fragmencie programu:

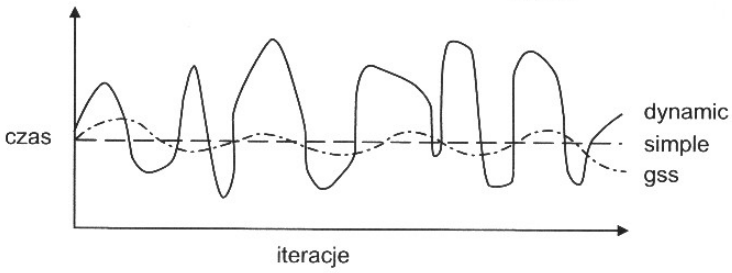
```
#pragma parallel ...
{
  A
  B
  C
  D
  E
}
```

istnieje zależność między wyrażeniami B i E, wówczas żaden z wątków nie powinien wykonać wyrażenia E dopóki B nie zostanie zrealizowane przez wszystkie wątki. Jednym ze sposobów jest umieszczenie dyrektywy #pragma synchronize przed E. Jest to jednak posunięcie wysoce nieefektywne, ponieważ wątki oczekujące przy barierze nie mogą kontynuować swojego wykonania, nawet jeżeli wszystkie pozostałe wątki zakończyły realizację B i aktualnie wykonują wyrażenia C i D. Wszystkie najszybsze wątki będą zatem oczekiwały na zakończenie przez najwolniejszy wątek wyrażeń C i D. Umieszczenie dyrektyw #pragma enter gate po B i #pragma exit gate przed E odpowiednio informuje system o zakończeniu realizacji wyrażenia B przez wszystkie wątki, po czym dla oczekujących wątków blokadę przed E.

Aglomeracja

Aglomeracja w języku Power C polega na łączeniu kilku obszarów równoległych w jeden większy obszar, dzięki czemu zyskuje się znaczne oszczędności polegające na ograniczeniu operacji każdorazowego przygotowania kolejnych fragmentów programu do równoległego wykonania. Łączenie obszarów równoległych

przyczynia się do wzrostu wydajności programów równoległych również poprzez redukcję działań synchronizacji wątków. Należy bowiem pamiętać, że na końcu każdego obszaru równoległego istnieje bariera, do której muszą dotrzeć wszystkie wątki zanim rozpoczęta zostanie realizacja sekwencyjnego fragmentu programu. Zmniejszenie liczby obszarów równoległych (przez ich aglomerację, nie likwidację) spowoduje w konsekwencji ograniczenie punktów synchronizacyjnych takiego programu i dalej prawdopodobny wzrost jego wydajności. Jednak na programiście spoczywa ciężar konstrukcji algorytmu równoległego, optymalizującego liczbę obszarów równoległych kodu. Jakkolwiek często pomocne może okazać się wsparcie PCA.



Rysunek 9. Czas wykonania pętli równoległej w zależności od liczby iteracji dla różnych strategii planowania.

Planowanie

Ilość równoległych wątków obszaru równoległego języka Power C nie powinna w zasadzie przekraczać liczby dostępnych procesorów komputera wykonawczego. Dlatego programista tworząc obszar równoległy powinien określać modyfikator numthreads przy użyciu zmiennej środowiskowej MPC_NUM_THREADS informującej kompilator o liczbie dostępnych wątków. Power C domyślnie tworzy tyle wątków kontroli, aby tworzyły z procesorami relację typu jeden do jeden. Stąd też modyfikator numthreads należy raczej rozumieć jako liczbę procesorów, którym zostaną przydzielone równoległe wątki.

Politykę planowania jednoczesnych wątków pętli równoległej utworzonej za pomocą dyrektywy #pragma pfor realizują jej modyfikatory schedtype oraz chunksize. Modyfikator schedtype informuje kompilator w jaki sposób powinien rozdzielić iteracje pętli między dostępne procesory. Zależnie od wykorzystywanego systemu (liczby dostępnych procesorów) programista może ustalić jeden z poniższych typów planowania zadań do procesorów:

- simple - domyślny typ, który nakazuje planiście równomiernie podzielić iteracje między wszystkie dostępne procesory

T1	T2	T3	T4

- runtime - planowanie zadań do procesorów zostanie określone automatycznie podczas wykonania programu przy pomocy zmiennej środowiskowej MP_SCHEDTYPE
- dynamic - zależnie od przetwarzanych danych niektóre iteracje pętli mogą wymagać dłuższego czasu obliczeń, przez co pewne wątki zakończą swoje wykonanie znacznie wcześniej niż inne. Przy strategii simple takie wątki muszą oczekiwać na pozostałe wątki. Natomiast rozmieszczenie danych do procesorów typu dynamic spowoduje utworzenie wielu wątków, każdy wykonujący stałą liczbę chunksize iteracji. Wartość chunksize powinna być mniejsza niż (liczba wszystkich iteracji)/(liczba procesorów). Jeżeli jakiś proces zakończy przydzieloną mu część obliczeń, wówczas do wolnego procesora zostanie przydzielony inny wątek dokonujący obliczeń do kolejnych iteracji pętli. Przewagą strategii dynamic nad simple jest to, że dynamic umożliwia bardziej równomierne rozłożenie pracy między dostępne procesory.

T1	T2	T3	T4	T2	T1	T2	T4	T3	T1	T4	T3	T2	T4	T1	T3

- interleave - tworzy wiele wątków, każdemu przydzielając stałą liczbę chunksize iteracji pętli. Przydzielenie poszczególnych wątków do procesorów jest następnie realizowane przez przeplot porcji chunksize iteracji pętli.

T1	T2	T3	T4	T1	T2	T3	T4	T1	T2	T3	T4	T1	T2	T3	T4

- gss - nakazuje planiście, aby przydzielić do każdego procesora zmienną liczbę iteracji pętli. Strategia gss jest podobna do dynamic, lecz zamiast stałej wartości chunksize liczba iteracji przydzielanych dla każdego wątku rozpoczyna się od dużych kawałków a kończy na małych.

T1	T2	T3	T4	T1	T4	T3	T2	T1	T4	T3	T2

Modyfikator chunksize dostarcza informacji o wielkości porcji iteracji tworzonych wątków dla strategii planowania dynamic oraz interleave. Programista powinien pamiętać, że zbyt wielkie rozdrobnienie iteracji może skutkować jednak zmniejszeniem wydajności obliczeń (patrz Rysunek 9).

Artykuł powstał dzięki nieocenionej pomocy Pana mgr inż. Krzysztofa Kraski pracownika Wydziału Informatyki na Politechnice Szczecińskiej oraz wykorzystano w nim materiały z książki W. Ferguson, IRIS POWER C (tm) User's Guide, Silicon Graphics, Inc., Mountain View, California 1991-1993.