1. Analiza istniejących obliczeń uruchamianych w ramach systemów rozproszonych

Paweł Czarnul

Politechnika Gdańska,

Wydział Elektroniki, Telekomunikacji i Informatyki,

Katedra Architektury Systemów Komputerowych

e-mail: pczarnul@eti.pg.gda.pl

Streszczenie

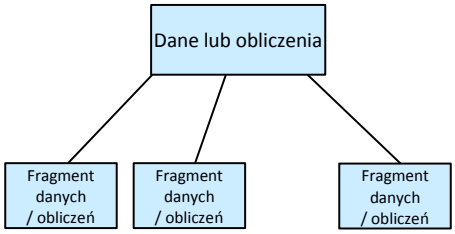
W rozdziale zaprezentowano analizę algorytmów równoległych tradycyjnie uruchamianych w systemach klastrowych wysokiej wydajności, a następnie pokazano charakterystykę algorytmów ze względu na parametry istotne przy implementacji ich rozwiązań w rozproszonym środowisku Comcute. Następnie przedstawiono ocenę możliwości ich przeniesienia do tego środowiska.

Słowa kluczowe: algorytmy równoległe, skalowalność, przyspieszenie obliczeń, zrównoleglanie, klastry obliczeniowe, volunteer computing

1.1. Tradycyjne paradygmaty przetwarzania równoległego – sposób rozpraszania zadań, obliczeń i danych

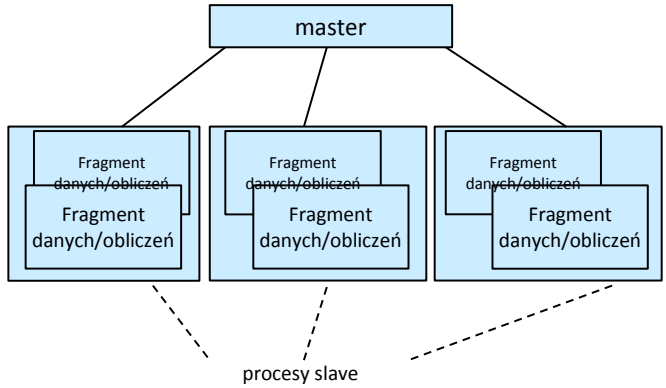
Tradycyjne algorytmy [12-15] uruchamiane w systemach równoległych można podzielić ze względu na paradygmaty przetwarzania, podziału danych i synchronizacji. Wyróżnić można przetwarzanie typu:

- *embarrassingly parallel* – obliczenia lub dane mogą zostać podzielone na niezależne fragmenty i rozproszone przed wykonaniem obliczeń równoległych (rys. 1.1). Wysoka skalowalność.



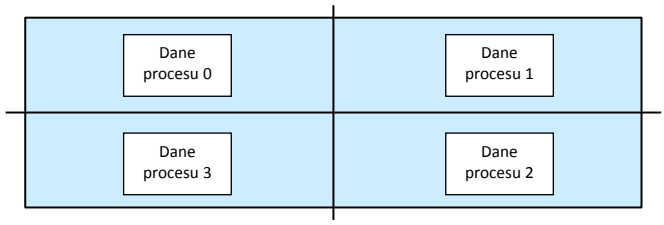
Rys. 1.1. Przetwarzanie typu *embarrassingly parallel*

- *master-slave/task farming* – proces-master dzieli dane wejściowe i/lub obliczenia na fragmenty, których przetworzenie zleca procesom typu *slave* (rys. 1.2). Podział i dystrybucja mogą być wykonane statycznie bądź dynamicznie.



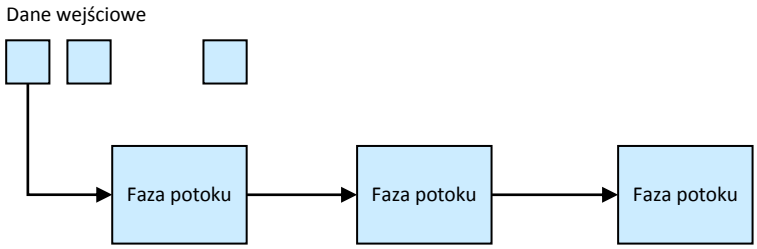
Rys. 1.2. Przetwarzanie typu master-*slave*

- *single program multiple data (SPMD)/geometric parallelism* – wiele procesów lub wątków aplikacji wykonuje te same obliczenia (stąd określenie *Single Program*) na różnych danych (stąd *Multiple Data*) – rys. 1.3 Zwykle obliczenia tego typu polegają na podziale dużej przestrzeni danych wejściowych na procesy/wątki i rozwiązywanie fragmentów równolegle z następującą zaraz później synchronizacją danych. Przykładem może być wiele problemów fizycznych, takich jak aplikacje symulujące zmiany pogodowe w danym fragmencie przestrzeni, zjawiska zachodzące np. w organizmie ludzkim, rozchodzenie się fal w przestrzeni, rozprzestrzenianie się skażeń promieniotwórczych lub biologicznych, czy inne aplikacje, w których przestrzeń modelowana jest przez podział na wiele komórek zależnych od komórek sąsiednich. Wydajne zrównoleglanie wymaga tutaj w szczególności krótkich opóźnień komunikacyjnych, w szczególności dla uzyskania wysokiej skalowalności dla dużej liczby procesorów. W paradygmacie tym następuje podział przestrzeni na fragmenty przydzielone różnym procesom. Procesy komunikują się ze sobą. Symulacja często polega na pętli zawierającej przeplatające się obliczenia i komunikację



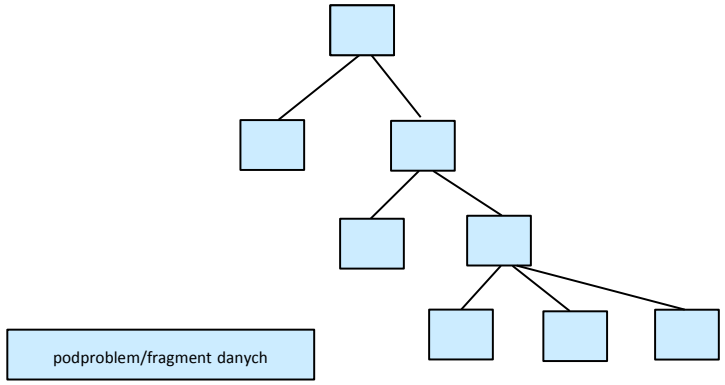
Rys. 1.3. Przetwarzanie typu SPMD

- przetwarzanie potokowe – wyróżnia się niezależne fragmenty przetwarzania (polegające zwykle na uruchomieniu innego kodu), które wykonywane są po kolei na danych wejściowych. Dane wejściowe to zwykle wiele paczek danych, które są kolejno podawane do kolejnych punktów przetwarzania w potoku (rys. 1.4).



Rys. 1.4. Przetwarzanie potokowe

- dziel-i-rządź – początkowy problem dzielony jest na podproblemy (często dzielone są przy tym dane wejściowe). Dalej następuje rekurencyjny podział danych/problemu na fragmenty/podproblemy (rys. 1.5).



Rys. 1.5. Przetwarzanie typu dziel i rządź

1.2. Cele równoległego uruchomienia w kontekście środowiska Comcute

Systemy równoległe takie jak superkomputery, wysokowydajne klastry czy lokalne sieci komputerowe wykorzystywane są do uruchamiania aplikacji równoległych lub sekwencyjnych operujących na niezależnych fragmentach danych w celu:

- Zmniejszenia czasu rozwiązania problemu. Prawo Amdahla określa przyspieszenie obliczeń możliwe do uzyskania w takim systemie rozpatrując fragment rozwiązania, który można zrównoleglić oraz fragment sekwencyjny. Przyspieszeniem obliczeń określa się stosunek czasu

wykonania w systemie z jednym procesorem i czasu wykonania w systemie równoległym. Podział danych na fragmenty, które mogą być przetworzone równolegle przez różne jednostki systemu równoległego pozwala na skrócenie czasu wykonania obliczeń.

- Zwiększenia niezawodności przetwarzania poprzez uruchomienie wielu instancji rozwiązania dla tych samych danych. W przypadku awarii, czy też zwrócenia różnych wyników przez różne instancje, mechanizm głosowania może zadecydować o wyniku bądź konieczności powtórzenia obliczeń.
- Weryfikacji wyników poprzez równoległe uruchomienie różnych algorytmów rozwiązujących danych problem dla tych samych danych wejściowych. Porównanie wyników pozwala ocenić ich wiarygodność.
- Możliwości rozwiązania problemu wymagającego przetworzenia danych dużych rozmiarów. W wielu przypadkach problem dla danych dużych rozmiarów nie może zostać rozwiązany na pojedynczym komputerze, gdyż system nie jest w stanie wydajnie przetworzyć danych dużych rozmiarów. Oczywiście, wydajność zależy również od sposobu zarządzania danymi w samym algorytmie.

Wszystkie powyższe cele mogą być również pożądane w kontekście uruchomienia danego problemu w rozproszonym środowisku Comcute pod warunkiem, że system będzie w stanie wydajnie dany algorytm uruchomić (patrz punkt 1.3 poniżej).

1.3. Istotne parametry przy migracji tradycyjnych obliczeń na środowisko Comcute

Bazując na ww. prawie Amdahla, z punktu widzenia wydajności równoległego czy rozproszonego wykonania algorytmu istotne są wartości związane z:

- czasem przetwarzania,
- czasem komunikacji,
- kosztem i częstotliwością synchronizacji.

Parametry te będą determinowały skalowalność algorytmów w środowiskach rozproszonych, w których czas komunikacji w stosunku do czasu obliczeń będzie znacznie większy niż dla tego samego rozwiązania uruchomionego w systemach klastrowych.

Z tego punktu widzenia, istotne parametry sieci komunikacyjnej to:

- przepustowość (ang. *bandwidth*) – zdolność łącza komunikacyjnego do przesłania określonej ilości danych w jednostce czasu,
- *startup time* – czas niezbędny na zainicjowanie połączenia.

Typowe wartości uzyskiwane dla klastrów bazujących na sieci Gigabit Ethernet to 100 MB/s oraz opóźnienia rzędu 50-80μs, zaś dla wydajnych klastrów z siecią Infiniband 1000 MB/s oraz 1-5 μs.

Z kolei, przepustowości dla komunikacji klient-serwer w Internecie mają zwykle wartości od kilku KB/s do około 10-20 Mbit/s dla szerokopasmowych łącz i na wybranych fragmentach sieci.

Stąd też, analiza możliwości przeniesienia tradycyjnych algorytmów z systemów klastrowych do systemu Comcute powinna uwzględniać:

1. paradygmaty przetwarzania:
   - *embarrassingly parallel* [11],
   - *master-slave* [11],
   - *single program multiple data* [5],
   - *pipeline* [11],
   - *divide-and-conquer* [3-4,6].

   Ze względu na proponowaną architekturę systemu algorytmy, które będą wstępnie predysponowane do zrównoleglenia powinny działać w jednym w paradygmatów: *embarrassingly parallel*, *master-slave* lub *divide-and-conquer*.

2. typowe rozmiary danych wejściowych,

3. typowe rozmiary danych wyjściowych,

4. typowa liczba procesorów/rdzeni, na których uruchamia się algorytm

5. stosunek czasu obliczeń do czasu komunikacji w algorytmie - przy danym rozmiarze danych wejściowych jest to zwykle zależne od liczby procesorów, na których się uruchamia daną aplikację przy zadanym rozmiarze danych wejściowych. Dla dużej liczby procesorów coraz większe znaczenie odgrywa opóźnienie komunikacyjne, w szczególności dla paradygmatów takich jak SPMD. Tradycyjnie, zrównoleglając zadania, stosunek ten powinien być możliwie duży. W innym przypadku, zrównoleglanie może być nieopłacalne w porównaniu do uruchomienia na pojedynczej maszynie. W tym przypadku, rozpatrzmy możliwość zrównoleglania krótkotrwałych zadań pomiędzy użytkowników – zadań, dla których czas komunikacji będzie znaczący. Z jednej strony, przy bardzo dużej liczbie klientów daje to możliwość rozproszenia obliczeń i nawet długi czas komunikacji może być akceptowalny przy bardzo dużym rozmiarze danych oraz bardzo dużej liczbie klientów. Z drugiej, powodować może nadmierne obciążenie serwera/serwerów rozpraszających, które będą musiały obsługiwać bardzo dużą liczbę zgłoszeń po zadania oraz z wynikami od bardzo dużej liczby klientów w zadanym przedziale czasowym.

6. synchronizacja w algorytmie:

   - globalna,
   - lokalna (partycje synchronizujące się lokalnie),
   - brak (podział danych wejściowych i zebranie wyników),

   Najprawdopodobniej ze względu na ograniczenia technologiczne (domyślne ograniczenia do komunikacji klienta z serwerem, z którego pobrany został kod klienta) synchronizacja musiałaby odbywać się poprzez scentralizowaną bądź hierarchicznie skonfigurowaną część serwerową systemu Comcute.

W zależności od liczby klientów, liczby uruchomionych aplikacji mogłaby powodować zbyt duże obciążenie systemu i uniemożliwiać wykonanie.

7. złożoność algorytmu (ew. NP-zupełny) algorytm:
   - optymalny – w dużym systemie takim jak Comcute z tysiącem, dziesiątkami lub setkami tysięcy klientów rozwiązanie problemu nawet NP-zupełnego dla dużych zbiorów danych może być realne. Co więcej, dla paradygmatów przetwarzania *embarrassingly parallel*, *master-slave* lub *divide-and-conquer* bez interakcji pomiędzy problemami, może się dobrze skalować.
   - heurystyczny - rozwiązanie przybliżone stosowane jest zwykle ze względu na to, żeby skrócić czas wykonania algorytmu. Ze względu na potencjalnie dużą liczbę klientów, można rozważyć zastosowanie prostego, dobrze skalującego się rozwiązania optymalnego. Zależne od algorytmu.
   - losowe rozwiązania – w tak dużym systemie możliwe jest szukanie rozwiązań np. problemów kombinatorycznych poprzez losowe generowanie rozwiązań przez potencjalnie bardzo dużą liczbę klientów i porównywanie wyników. Bardziej dokładną alternatywą może być podział przestrzeni danych wejściowych na fragmenty i przydzielenie do poszczególnych klientów.
8. typowe rozmiary danych przesyłanych pomiędzy węzłami (klastra) i częstotliwość synchronizacji - istotne w kontekście pytania ilu klientów może się synchronizować w tym momencie przez system Comcute i czy z powodu rozmiaru danych i ew. dużej częstotliwości nie stanie się to wąskim gardłem systemu.
9. typowy czas działania algorytmu – zwykle algorytmy działające dłużej (ze względu na swoją złożoność bądź zwykle stosowane rozmiary danych wejściowych) będą prezentowały większy potencjał zrównoleglania w środowisku Comcute niż algorytmy działające krócej. Wynika to z potencjalnie dużych czasów komunikacji w rozproszonym środowisku Comcute.

1.4. Charakterystyki istniejących systemów i standardów obliczeniowych – w kontekście migracji algorytmów do systemu Comcute

Potencjalne przeniesienie algorytmów z tradycyjnych systemów klastrowych na rozproszony system Comcute wiąże się bezpośrednio z technologią kodowania, kompilacji i uruchomienia tego typu aplikacji. Tradycyjnie wykorzystuje się niskopoziomowe programowanie równoległe na klastrach – m.in. następujące modele i interfejsy programistyczne oraz środowiska:

1. programowanie w modelu z pamięcią współdzieloną – poszczególne procesy aplikacji bądź wątki działające w obrębie procesu mają dostęp do wspólnej przestrzeni w pamięci. Synchronizacja może następować przez zapis i odczyt w komórkach pamięci współdzielonej z dodatkowymi

mechanizmami synchronizacji takimi jak monitory, blokady, zmienne warunkowe, zasypianie i budzenie wątków etc. Przykładami technologii implementującymi ten model są np.:

- Pthreads – API wspierające wielowątkowość w języku C.
- Java Threads - API wspierające wielowątkowość w języku Java.
- OpenMP – możliwość rozszerzania programów o sekcje wykonywane równolegle poprzez ich specyfikację za pomocą specjalnych dyrektyw i deklaracji.

2. programowanie w modelu z pamięcią rozproszoną. Najczęściej stosowany jest model z przekazywaniem wiadomości (*message passing*).

- MPI [2] – popularna specyfikacja z przekazywaniem wiadomości, również wsparciem dla wielowątkowości, API dla języków C/C++ i Fortran.
- PVM [1] – środowisko przetwarzania równoległego i rozproszonego dla języka C/C++ i Fortran.

Zwykle kompilacja tego typu zadań wykonywana jest przez programistę-użytkownika, który następnie uruchamia aplikację równoległą na dedykowanej maszynie wirtualnej lub wykorzystuje do uruchomienia systemy kolejkowe takie jak PBS, LSF itp.

W literaturze wymienia się wysokopoziomowe oprogramowanie i wzorce do automatycznego zrównoleglania pewnych klas algorytmów takich jak:

1. *divide-and-conquer* – np. Cilk dla języka C, Satin [9] dla języka Java, DAMPVM/DAC [4] dla języka C++.
2. *single program multiple data (SPMD)* – ParMETIS, Zoltan.

Aplikacje równoległe mogą być także uruchamiane na wielu klastrach za pomocą narzędzi takich jak MPICH-G2, PACX-MPI lub BC-MPI.

Z racji tego, że wyżej wymienione narzędzia wymagają specjalistycznej wiedzy, powstało wiele systemów pozwalających na uruchamianie aplikacji równoległych (przygotowanych często dla użytkownika) dla zadanych danych z wykorzystaniem łatwego w użyciu interfejsu. Zasoby wykorzystywane przez tego typu systemy gridowe są ukryte przed użytkownikiem, który jedynie zleca zadania do wykonania i oczekuje na wyniki. System dokonuje wyboru zasobów spełniających wymagania tj. np. architektura procesora, na której może być uruchamiany plik wykonywalny, rozmiar dostępnej pamięci RAM oraz przestrzeni dyskowej, po czym dokonuje rezerwacji zasobów, kopiuje podane przez użytkownika dane wejściowe, uruchamia aplikację (sekwencyjnie bądź równolegle), z powrotem przesyła dane wyjściowe do użytkownika. Tego typu systemy wykorzystują często tzw. warstwę pośrednią grid jak np. Globus Toolkit, Unicore etc.

Pewnym rozwinięciem tego typu przetwarzania jest *cloud computing* gdzie dostawca oferuje użytkownikowi całą platformę, infrastrukturę bądź oprogramowanie w zadanej konfiguracji, na określony czas, za określoną kwotę. Użytkownik może dostosowywać system do swoich potrzeb kontrolując

jednocześnie koszt zasobów i oprogramowania. Nie musi wówczas utrzymywać całego środowiska samodzielnie.

Z kolei przetwarzanie typu *volunteer computing* wykorzystuje komputery internautów do podzielenia danych wejściowych dużego problemu (*master-slave*) na fragmenty i zlecenia ich przetwarzania użytkownikom. Kwestie z tym związane dotyczą:

1. wiarygodności przetwarzania – obliczenia muszą być zlecane niezależnym klientom i wyniki porównywane,
2. tylko pewna klasa algorytmów (możliwa do uruchomienia w paradygmacie *master-slave*) może być efektywnie zrównoleglona,
3. ograniczeń po stronie klienta – użytkownik musi jawnie zainstalować kod zarządzający i aplikacji na swoim komputerze, wyrazić zgodę na uruchomienie i określić warunki wykorzystania,
4. użytkownicy zwykle nie są wynagradzani finansowo za uczestnictwo w projekcie, mają natomiast świadomość współuczestnictwa w ważnych projektach.

Bardzo często implementacje algorytmów wykorzystują zewnętrzne biblioteki do wykonania pewnych operacji lub części algorytmów – np. sortowanie, operacje na macierzach, przetwarzanie obrazów etc. Powstaje kwestia możliwości wykorzystania tego typu bibliotek po stronie klienta w aplikacji systemu Comcute. Konkretna technologia może ograniczyć możliwość wykorzystania danej biblioteki bądź wymusić implementację w innym języku (np. migrację do języka Java jeśli strona klienta systemu Comcute wykorzysta technologię apletów Javy itp.). Bardzo wiele programów równoległych implementowanych jest w językach C/C++ lub Fortranie. Istnieje wiele często wykorzystywanych bibliotek zaimplementowanych w tych językach.

W kontekście systemu Comcute, wymaga to albo nowego sposobu kodowania algorytmów z uwzględnieniem języka wykorzystywanego przez daną technologię bądź możliwości uruchomienia kodu po stronie klienta, albo specjalnych nakładek na istniejące API, co wydaje się kłopotliwe.

Problemy w Comcute:

1. podział danych – jaki – statyczny/dynamiczny,
2. język kodowania algorytmów,
3. repozytorium algorytmów,
4. technologia uruchamiania.

Na ile łatwo zmigrować już istniejący kod do tego typu środowiska? Implementacje aplikacji z systemów typu BOINC [16] do konkretnej technologii przez przeglądarkę będą możliwe do przeniesienia z uzyskaniem istotnego przyspieszenia obliczeń. Będzie to możliwe dla aplikacji w paradygmatach *embarrassingly parallel* oraz *master-slave* z dużym stosunkiem czasu obliczeń do komunikacji i synchronizacji, w mniejszym stopniu dla aplikacji dziel-i-rządź. Aplikacje SPMD oraz potokowe nie będą pracowały wydajnie w środowisku Comcute chyba, że system zostanie wykorzystany do uruchamiania całych instancji z różnymi danymi wejściowymi

u różnych internautów. W takim przypadku Comcute pozwoli na równoległe obliczenie wielu scenariuszy z różnymi danymi wejściowymi.

1.5. Charakterystyka wybranych algorytmów

W rozdziale przedstawiono charakterystykę wybranych i często używanych algorytmów równoległych (uruchamianych do tej pory na klastrach, sieciach LAN), pod kątem możliwości uruchomienia w środowisku rozproszonym (tj. takim gdzie koszty komunikacji są relatywnie większe niż na klastrach), a więc możliwości uruchomienia w środowisku Comcute.

1.5.1. Równoległe symulacje SPMD

Zwykle równoległe aplikacje rozwiązujące układ równań różniczkowych – sprowadzone do liniowych równań rozwiązywanych w czasie równolegle.

Paradygmat przetwarzania

single program multiple data

Dane wejściowe

zwykle przestrzeń 2D lub 3D podzielona na fragmenty – możliwy podział statyczny lub dynamiczny

Wynik

wartości danych w przestrzeni o rozmiarze takim jak dane wejściowe

Typowe rozmiary danych wejściowych

np. 1000x1000x1000x kilka zmiennych

Typowe rozmiary danych wyjściowych

jak dane wejściowe

Typowa liczba procesorów/ rdzeni, na których uruchamia się algorytm

1-256

Stosunek czasu obliczeń do czasu komunikacji w algorytmie

taki, który pozwala na wydajność rzędu 0,5-0,8 na 32-64 procesorach (przetwarzanie z pamięcią rozproszoną)

Synchronizacja w algorytmie

Zwykle lokalna, może być globalna np. co pewną liczbę iteracji

Liczba synchronizacji

synchronizacja co 1 iterację algorytmu

Złożoność algorytmu

przybliżone rozwiązanie

Algorytm

heurystyczny

Typowe rozmiary danych przesyłanych pomiędzy węzłami

zależy od liczby procesorów, ale rzędu rozmiaru wymiaru przestrzeni bazowej

Typowy czas działania algorytmu

zależy od dokładności, może być od kilku godzin do wielu dni

Znane implementacje algorytmu

publikacje w literaturze raportu

Istniejące implementacje - licencja + język programowania

oprogramowanie ułatwiające implementację + algorytmy zrównoleglania np. Zoltan, ParMETIS

1.5.2. Dziel-i-rządź

Schemat pozwalający na złożone i często nieregularne obliczenia np. przeszukiwanie drzew w grach etc.

Paradygmat przetwarzania

divide-and-conquer

Dane wejściowe

może to być pojedynczy wektor rozmiaru n (np. sortowanie) i/lub pewne globalne dane – np. szachownica do przeszukiwania ruchów

Wynik

zwykle wartości danych w przestrzeni o rozmiarze takim jak dane wejściowe

Typowe rozmiary danych wejściowych

od kilkunastu-kilkuset zmiennych do dziesiątek setek tysięcy

Typowe rozmiary danych wyjściowych

jak dane wejściowe

Typowa liczba procesorów/ rdzeni, na których uruchamia się algorytm

1-256

Stosunek czasu obliczeń do czasu komunikacji w algorytmie

zależy od algorytmu – głównie zależy od czasu podziału/ scalania danych oraz obliczeń wykonywanych w węzłach, duży dla np. rozwiązywania gier typu szachy/ warcaby

Synchronizacja w algorytmie

Zwykle brak, ale może też być okazjonalna synchronizacja lokalna i globalna

Liczba synchronizacji

zwykle podział na podproblemy i zebranie wyników, może być okazjonalna synchronizacja globalna (np. równoległy algorytm alfa-beta)

Złożoność algorytmu

zwykle pełne przeszukiwanie przestrzeni rozwiązań, mogą być odcięcia jak w algorytmach alfa-beta

Algorytm

optymalny

Typowe rozmiary danych przesyłanych pomiędzy węzłami

od kilkunastu / kilkuset zmiennych do dziesiątek setek tysięcy, ale stosunkowo rzadka komunikacja o ile czas obliczeń w węzłach wystarczająco długi, problem przy skalowaniu gdy czasy te są krótkie i jeszcze dodatkowo nieznane z góry

Typowy czas działania algorytmu

zależy od problemu, może być od kilku godzin do wielu dni

Znane implementacje algorytmu

publikacje w literaturze raportu

Istniejące implementacje - licencja + język programowania

różne frameworki dla różnych języków programowania – do automatycznego zrównoleglania również np. Satin, Cilk, DAMPVM/DAC

1.5.3. Sortowanie *quick-sort*

Często wykorzystywany algorytm sortowania oparty na porównywaniu.

Paradygmat przetwarzania

divide-and-conquer

Dane wejściowe

wektor rozmiaru n

Wynik

wektor rozmiaru n

Typowe rozmiary danych wejściowych

od kilkunastu do wielu tysięcy lub więcej zmiennych

Typowe rozmiary danych wyjściowych

od kilkunastu do wielu tysięcy lub więcej zmiennych

Typowa liczba procesorów/ rdzeni, na których uruchamia się algorytm

Kilka-kilkadziesiąt

Stosunek czasu obliczeń do czasu komunikacji w algorytmie (przy danym rozmiarze danych wejściowych)

duży przy dużym wektorze wejściowym, algorytm naturalnie dzieli problem wejściowy na podproblemy (dziel i rządź)

Synchronizacja w algorytmie

brak w dziel i rządź

Liczba synchronizacji

brak pośrednich, podział danych i zebranie wyników

Złożoność algorytmu

O(nlogn)

Algorytm

optymalny

Typowe rozmiary danych przesyłanych pomiędzy węzłami

O(n)

Typowy czas działania algorytmu

bardzo szybki algorytm, sekundy, minuty

Znane implementacje algorytmu

książki o tematyce HPC

Istniejące implementacje - licencja + język programowania

wiele implementacji dostępnych, np. w bibliotekach do C, wiele implementacji dla różnych języków

1.5.4. Wyszukiwanie wzorca tekstowego w pliku/plikach

Wyszukiwanie wzorca w jednym lub wielu danych plikach.

Paradygmat przetwarzania

single program multiple data, pipeline

Dane wejściowe

pliki do przeszukania i wzorzec (zwykle znacznie mniejszy)

Wynik

fragmenty plików bądź też indeksy

Typowe rozmiary danych wejściowych

wzorce rzędu kilku, kilkuset, kilku tysięcy bajtów, pliki o rozmiarach megabajtów

Typowe rozmiary danych wyjściowych

indeksy znalezionych fragmentów bądź fragmenty plików

Typowa liczba procesorów/ rdzeni, na których uruchamia się algorytm

1-256

Stosunek czasu obliczeń do czasu komunikacji w algorytmie

zwykle duży – dobrze się zrównolegla

Synchronizacja w algorytmie

brak

Liczba synchronizacji

niewiele, podział danych i zebranie wyników, może być dynamiczny *master-slave*

Złożoność algorytmu

pełne przeszukiwanie

Algorytm

optymalny

Typowe rozmiary danych przesyłanych pomiędzy węzłami

rzędu rozmiarów plików wejściowych, ale dane mogą być przesyłane bądź na początku i na końcu działania aplikacji bądź też dosyłane fragmentami – dynamiczny *master-slave* – większa możliwość nakładania obliczeń i komunikacji

Typowy czas działania algorytmu

kilka minut – kilka dni

Znane implementacje algorytmu

książki o tematyce HPC

Istniejące implementacje - licencja + język programowania

programy grep i inne, możliwe wykorzystanie do równoległej implementacji

1.6. Wykaz literatury

1. Geist A., Beguelin A., Dongarra J., Jiang W., Mancheck R., Sunderam V.: *PVM Parallel Virtual Machine. A Users Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, 1994. http://www.epm.ornl.gov/pvm/.
2. Pacheco P.: *Parallel programming with MPI*. Morgan Kaufmann. 1996
3. Czarnul P.: *Dynamic Process Partitioning and Migration for Irregular Applications*.. accepted for International Conference on Parallel Computing in Electrical Engineering PARELEC'2002, Warsaw, Poland, 2002

4. Czarnul P, Tomko K., Krawczyk H.: *Dynamic Partitioning of the Divide-and-Conquer Scheme with Migration in PVM Environment*. In Recent Advances in Parallel Virtual Machine and Message Passing Interface, no 2131 in Lecture Notes in Computer Science, pp. 174-182. Springer-Verlag, 8th European PVM/MPI Users' Group Meeting, Santorini/Thera, Greece, September 23-26, 2001
5. Sarris C.D., Tomko K., Czarnul P., Shih-HaoHung, Robertson R.L., Donghoon Chun, Davidson E. S., Katehi L.P.B.: *Multiresolution Time Domain Modeling for Large Scale Wireless Communication Problems*. In Proceedings of the 2001 IEEE AP-S International Symposium on Antennas and Propagation, volume 3, pages 557–560, 2001.
6. Horowitz E., Zorat A.: *Divide-and-conquer for Parallel Processing*. IEEE Transactions on Computers, C-32(6):582-585, June 1983.
7. Schloegel K., Karypis G. Kumar V.: *Graph Partitioning for High Performance Scientic Simulations, CRPC Parallel Computing Handbook,* Morgan Kaufmann, http://citeseer.nj.nec.com/schloegel00graph.html, 2000
8. Krawczyk H. Saif J.: *Parallel Image Matching on PC Cluster*. In Recent Advances in Parallel Virtual Machine and Message Passing Interface, number 2131 in Lecture Notes in Computer Science, pages 312-318. Springer-Verlag, 8th European PVM/MPI Users' Group Meeting, Santorini/Thera, Greece, September 23-26, 2001
9. Nieuwpoort R.V. van, Kielmann T., Bal H.E.: *Satin: Efficient Parallel Divide-and-Conquer in Java*. In Euro-Par 2000 Parallel Processing, Proceedings of the 6th International Euro-Par Conference, no 1900 in LNCS, pp. 690-699, 2000.
10. Czarnul P.: *Programming, Tuning and Automatic Parallelization of Irregular Divide-and-Conquer Applications in DAMPVM/DAC*. International Journal of High Performance Computing Applications 17/2003, pp. 77-93, 2003
11. Wilkinson B., Allen M.: *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. Prentice Hall, 1999
12. Akl S. G.: *The design and analysis of parallel algorithms*. Prentice Hall, 1989
13. Foster I.: *Designing and Building Parallel Programs*. Addison-Wesley, http://www -unix.mcs.anl.gov/dbpp/, 1995
14. Rajkumar Buyya (ed): *High Performance Cluster Computing, Programming and Applications*. Prentice Hall, 1999.
15. Kirk D., Wen-mei Hwu: *Programming Massively Parallel Processors*. Morgan Kaufmann, 2010
16. Anderson D. P.. *BOINC: A system for public-resource computing and storage.* 5th IEEE ACM International Workshop on Grid Computing, 2004