Modele programowania równoległego

Programowanie z przekazywaniem komunikatów
Message-Passing Programming

Rafał Walkowiak

MPP - Cechy charakterystyczne 1

- Prywatna - wyłączna przestrzeń adresowa.
- Równoległość definiowana wprost przez programistę.
- Możliwość uzyskania wysokiej efektywności przetwarzania i skalowalności.
- Typowy model dla gron stacji roboczych i systemów wielo-komputerowych o nie-współdzielonej przestrzeni adresowej.

MPP - Cechy charakterystyczne 2

- Elementy danych należą wyłącznie do jednej przestrzeni adresowej – dane rozdzielone wprost.
- Model wzmacnia naturalnie lokalność dostępu do danych.
- Dostęp do danych nielokalnych wymaga współpracy 2 procesów – posiadającego dane i żądającego - konieczność zsynchronizowania procesów.
- Model łatwo implementowalny w różnych architekturach sprzętowych.

MPP - Struktura programów

- Asynchroniczne –całkowita niezależność przetwarzania – trudna analiza kodu ze względu na niedeterminizm powodowany nieokreśloną kolejnością operacji realizowanych współbieżnie.
- Luźno synchroniczne – procesy wykonywane asynchronicznie, lecz grupy procesów synchronizują się co pewien czas dla zrealizowania interakcji. Okresowa synchronizacja pozwala łatwiej wnioskować o przebiegu przetwarzania na podstawie kodu.
- Ogólnie - różne programy dla różnych procesów
- Popularne rozwiązanie - **SPMD** jednakowy kod dla większości procesów, realizacja asynchroniczna bądź luźno-synchroniczna.

Typy podstawowych operacji

- Send , Receive
- Send(void *sendbuf, int nelems, int dest)
- Receive(void *recvbuf, int nelems , int source)

Proces0	Proces1
A=100;	Receive(&A,1,0);
Send(&A,1,1);	printf(‘’%d\n”,A);
A=0;	

Jaką wartość wyświetli Proces1? 100? 0 ?

Blokujące operacje komunikacyjne wstrzymanie nadawcy i/lub odbiorcy

Operacje niebuforowane

- Możliwe znaczne czasy oczekiwania na odbiorcę/nadawcę, który się spóźnia.
- Możliwe zakleszczenie przy jednakowej kolejności operacji wysłanie-później-odbiór (odwrotnie też) w ramach obu procesów - konieczna zmiana kolejności operacji np. w programie SPMD.

Blokujące operacje komunikacyjne

Operacje buforowane

- Szybciej przy nadawaniu
- Niebezpieczeństwo przepełnienia buforów przy braku zsynchronizowanego odbioru danych.
- Możliwe zakleszczenie przy jednakowej kolejności operacji odbiór-później-wysłanie (tylko takiej kolejności!) w ramach obu procesów - konieczna zmiana kolejności operacji np. w programie SPMD.

Blokujące operacje komunikacyjne realizacje

1. Dane buforowane po stronie nadawcy i odbiorcy (realizacja bez udziału procesora)
   – nadawca kontynuuje pracę po przepisaniu danych do bufora, sprzęt kopiuje dane między buforami, odbiorca sprawdza zawartość bufora przy operacji odbioru i kontynuuje pracę po przepisaniu komunikatu do obszaru danych.
2. Dane buforowane po jednej stronie (realizuje procesor)- możliwe realizacje:
   – proces odbiorcy przerywany, dane kopiowane do bufora odbiorcy (wznowienie procesu nadawcy), a następnie kopiowane z bufora do lokacji docelowej po zainicjowaniu operacji odbioru
   - proces nadawcy zapisuje dane do bufora i jest kontynuowany, w momencie gdy odbiorca gotowy następuje przerwanie procesu nadawcy i bezpośrednie kopiowanie do lokacji docelowej.

Nie-blokujące operacje komunikacyjne

- Blokujące operacje komunikacyjne zapewniają stan komunikacji zgodny ze swoim znaczeniem - kosztem oczekiwania (bez buforów) lub kosztem obsługi buforów.
- Nie-blokujące operacje wspomagane operacją sprawdzenia statusu - ?status? dla testowania stanu realizacji (ewentualnego stanu niezgodności znaczeniowej) zapoczątkowanej operacji komunikacji. Oczekiwanie na dostarczenie danych może być wykorzystane na przetwarzanie.

Nie-blokujące operacje komunikacyjne

- Nie-blokujące komunikacje nie-buforowane – niepoprawna modyfikacja obszaru danych nadawanych i niepoprawny odczyt danych odbieranych przed zakończeniem transmisji.
- Wspomaganie sprzętowe komunikacji może całkowicie ukryć koszt komunikacji (czas) – procesor liczy gdy dane się pojawią.
- Nie-blokujące komunikacje buforowane – wspomagane sprzętowo – redukcja czasu niedostępności obszarów danych tylko na czas kopiowania danych tylko z/do buforów – realizowane na żądanie procesu (realizacja przesłania).

Praktyczne realizacje

- Message Passing Interface (MPI)
- Parallel Virtual Machine (PVM)
- Implementują blokujące i nie-blokujące operacje komunikacyjne.
- Blokujące – łatwiejsze i bezpieczniejsze programowanie.
- Nie-blokujące – wyższa efektywność poprzez maskowanie kosztów komunikacji, lecz w przypadku błędu niebezpieczeństwo niepoprawnego dostępu do danych komunikujących się procesów.

MPI

- Standardowa biblioteka dla modelu programowania równoległego z przesyłaniem komunikatów.
- http://www.mpi-forum.org/docs/docs.html
- Minimalny zbiór funkcji to:
  - MPI_Init – inicjalizacja MPI
  - MPI_Finalize – zakończenie MPI
  - MPI_Comm_size – uzyskanie liczby procesów
  - MPI_Comm_rank – uzyskanie własnego identyfikatora procesu
  - MPI_Send, MPI_Recv – nadanie, odbiór komunikatu

Cechy MPI

- Funkcje MPI_Init i MPI_Finalize muszą być wywołane przez wszystkie procesy.
- Podział procesów na domeny komunikacyjne – „**komunikator”** mogących się komunikować procesów.
- Podstawowa domena komunikacyjna to MPI_COMM_WORLD
- Adresowanie poprzez identyfikator i nazwę domeny.
- Możliwość etykietowania typu komunikatu i odbierania komunikatu o określonym lub nieokreślonym typie.
- Możliwość odbierania komunikatu z dowolnego źródła w określonej domenie komunikacyjnej.

MPI – komunikacja między parą procesów - blokująca

- Podstawowe: funkcja odbioru blokująca, funkcja nadawania blokująca lub nie w zależności od wersji; zawsze buforowana z bezpiecznym dostępem do obszaru danych nadawanych. Możliwość zakleszczenia przy cyklicznych transmisjach przy wersji blokującej nadawania – konieczność zapewnienia komplementarnej kolejności operacji nadawania i odbioru.
- Możliwość nadawania i odbioru w ramach jednej operacji MPI_Sendrecv.

MPI – komunikacja między parą procesów - nieblokująca

- MPI_Isend, MPI_Irecv
- Konieczność sprawdzenia stanu realizacji operacji – czy dane skopiowano z/do bufora przed rozpoczęciem korzystania z bufora/danych. Zmienna typu MPI_Request parametrem wywołania (tworzona i zwracana).
- MPI_Test – sprawdzanie statusu przy użyciu obiektu zapytania MPI_Request
- MPI_Wait – oczekiwanie na zakończenie operacji przy obiekcie MPI_Request

Operacje kolektywne

MPI posiada w zestawie funkcji operacje kolektywne realizowane równolegle przez procesy na danych rozproszonych w ich pamięciach lokalnych.

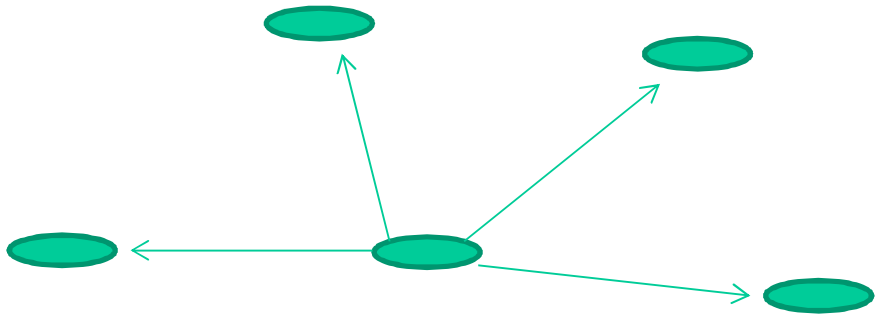
Wszystkie uczestniczące w operacji procesy wywołują ją.

Komunikacja niezbędna do zrealizowania operacji jest realizowana niewprost (bez specyfikacji przez programistę). Wszystkie procesy komunikatora wywołują funkcję operacji kolektywnej.

Komunikacja kolektywna – rozgłaszanie

MPI_Bcast(

void* wiadomość, int liczba_danych,
MPI_Datatype, int identyf_źródła,
MPI_Comm)- rozsyłanie tej samej informacji



Komunikacja kolektywna – redukcja

int MPI_Reduce (

void* dane, void* wyniki, int liczba_danych, MPI_Datatype typ_danych, MPI_op operacja, int korzeń, MPI_Comm komunikator)

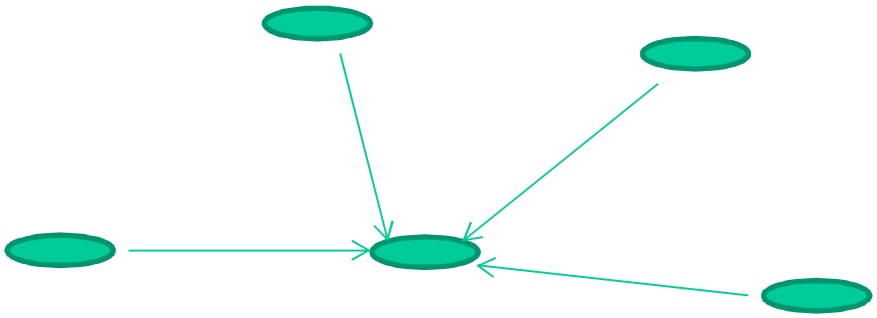
Dane i wyniki umieszczone w ciągłych obszarach

Możliwość określenia ilości zbiorów obiektów podlegających redukcji – liczba_danych.

Typy używanych danych zależne od operacji i zastosowania.

WERSJA: Funkcja MPI_Allreduce – wyniki operacji są rozgłaszane do wszystkich procesów – nie ma korzenia.

- Redukcja wszyscy-do-jednego: połączenie elementów lokalnych za pomocą podanej operacji i przesłanie wyniku do podanego odbiorcy (lub wszystkich) - operacje: minimum; maksimum; suma; iloczyn; log/bitowe and, or, xor; pozycja minimum, pozycja maksimum;



OPERACJA NA ZEBRANYCH DANYCH

Tworzenie komunikatorów – operacja kolektywna

MPI_Comm_split – podział komunikatora

Int MPI_split (MPI_comm istniejący_komunitkator,

int klucz_podziału, int numer_procesu, MPI_Comm *nowy_komunikator)

Tworzone są nowe komunikatory w liczbie równej liczbie różnych wartości klucza. Te procesy, które mają jednakowy klucz wchodzą w skład tego samego komunikatora z identyfikatorem numer_procesu. Każdy proces wywołuje funkcję, lecz może być wyłączony z tworzonych komunikatorów (specyficzna wartość klucza podziału).

Inne funkcje kolektywne

MPI_Gather – odbieranie przez jeden proces danych z różnych procesów

MPI_Allgather – odbieranie przez wszystkie procesy danych z różnych procesów

MPI_Scatter – rozsyłanie różnych danych z jednego do wielu procesów

MPI_Barrier(MPI_comm komunikator) – wstrzymywanie przetwarzania do momentu osiągnięcia wywołania przez wszystkie wątki

Przykład wykorzystania operacji kolektywnych dla sortowania n liczb z tablicy a[n] - nxn procesów

int **MPI_Gather**(const void *sendbuf, int sendcount,
MPI_Datatype sendtype, void *recvbuf, int recvcount,
MPI_Datatype recvtype, int root, MPI_Comm comm)

- **sendbuf -** starting address of send buffer (choice)
- **sendcount -** number of elements in send buffer (integer)
- **sendtype -** data type of send buffer elements (handle)
- **recvcount -** number of elements for any single receive (integer, significant only at root)
- **recvtype -** data type of recv buffer elements (significant only at root) (handle)
- **root -** rank of receiving process (integer)
- **comm -** communicator (handle)

sortowanie n liczb w tablicy a[n] – nxn procesów

```
i= id/N; j=id %N; index=0; // zmienne pomocnicze
if (a[i]>a[j]) || (a[i]==a[j] && (i>j))) w=1; else w=0; // porównanie pary liczb
MPI_Comm_split(MPI_COMM_WORLD,i,j,&kom_wiersza);
MPI_Reduce(&w ,&index,1,MPI_INT,MPI_SUM,0,kom_wiersza);
//na której pozycji jest moja liczba
MPI_Comm_split(MPI_COMM_WORLD,j,index , &kom_kolumny);
If (j==0)  // wynik w kolumnie 0 procesie o id=0 w tym komunikatorze
MPI_Gather(&a[i],1,MPI_FLOAT,wynik,1,MPI_FLOAT,0,kom_kolumny);
// ustaw liczbę na właściwej pozycji
```

3412 - wyliczony jako suma w nr procesu w komunikatorze kolumny

0011 2 - liczba 3 jest większa od 2 innych (będzie na 2 pozycji licząc od 0)

1011 3 komunikator kolumny - zbieranie wynik w procesie 0 (2,0)

0000 0 1,2,3,4

0010 1

sortowanie n liczb w tablicy a[n] – nxn procesów
krok pierwszy

```
i= id/N; // tą liczbą się zajmę

j=id %N; // z tą liczbą ją porównam

index=0; // zmienna pomocnicza

if (a[i]>a[j]) || (a[i]==a[j] && (i>j))) w=1; else w=0;
// porównanie pary liczb

//wątki o identyfikatorach od 0 do N-1 zajmują
się pierwszą liczbą – analogicznie następne
```

sortowanie n liczb w tablicy a[n] – nxn procesów krok drugi

```
// utwórz komunikator
MPI_Comm_split(MPI_COMM_WORLD,i,j,&kom_wiersza);
// każdy zerowy proces komunikatora odbiera pozycję swojej liczby
MPI_Reduce(&w ,&index,1,MPI_INT,MPI_SUM,
0,kom_wiersza);
```

sortowanie n liczb w tablicy a[n] – nxn procesów
krok trzeci

```
// utwórz komunikatory – potrzebny tylko jeden
MPI_Comm_split(MPI_COMM_WORLD,j,index ,
&kom_kolumny);

If (j==0)  // ja odbiorę wynik - tablicę
    MPI_Gather(&a[i],1,MPI_FLOAT,wynik,1,MPI_FLOA
    T,0,kom_kolumny);
//zbieranie wyników tylko w jednym
komunikatorze
```

Powtórka na przykładzie

3412Liczby do posortowanie

W	indeks	wynik
0011	2	1
1011	3	2
0000	0	3
0010	1	4