1 Przetwarzanie współbieżne, równoległe i rozproszone

1.1 Przetwarzanie współbieżne

- realizacja wielu programów (procesów) w taki sposób, że ich trwanie od momentu rozpoczęcia do momentu zakończenia może się na siebie nakładać
- współbieżność pojawiła się wraz z wielozadaniowymi systemami operacyjnymi (lata 60te, Multics – 1965) i nie wymusza równoległości
- współbieżność związana jest z szeregiem problemów teoretycznych wynikłych z prób realizacji wielozadaniowych systemów operacyjnych
- istnieje wiele mechanizmów niskiego poziomu (systemowych) do rozwiązywania problemów współbieżności

1.2 Obliczenia równoległe

- dwa lub więcej procesów (lub wątków) jednocześnie współpracuje (komunikując się wzajemnie) w celu rozwiązania pojedynczego zadania (najczęściej z określonej dziedziny zastosowań)
- rozwój związany z powstaniem w latach siedemdziesiątych komputerów równoległych
- problemy obliczeń równoległych (poza klasycznymi zagadnieniami współbieżności) są najczęściej związane z konkretną dziedziną zastosowań
- obliczenia równoległe są silnie związane z dziedziną obliczeń wysokiej wydajności (i obliczeniami naukowotechnicznymi)

1.3 Przetwarzanie rozproszone

- realizacja programów na systemach wieloprocesorowych z zasobami lokalnymi (najczęściej odrębnych komputerach połączonych siecią)
- rozwój związany z popularyzacją Internetu w latach osiemdziesiątych i dziewięćdziesiątych
- problem wspólnego korzystania z rozproszonych zasobów
- luźniejsze powiązanie współpracujących procesów niż w przypadku obliczeń równoległych (niekonieczna synchronizacja czasowa, zastępowana przez przyczynowoskutkową)
- wykorzystanie infrastruktury sieciowej (protokoły, bezpieczeństwo)

2 Procesy a wątki

2.1 Proces

Proces to jedno z najbardziej podstawowych pojęć w informatyce. Z definicji jest to po prostu egzemplarz wykonywanego programu. Należy odróżnić jednak proces od wątku - każdy proces posiada własną przestrzeń adresową, natomiast wątki posiadają wspólną sekcję danych. Każdemu procesowi przydzielone zostają zasoby, takie jak:

- procesor,
- pamięć,
- dostęp do urządzeń wejścia-wyjścia,
- pliki.

Każdy proces posiada tzw. „rodzica". W ten sposób tworzy się swego rodzaju drzewo procesów (*process tree, pstree*). Proces może (ale nie musi) mieć swoje procesy potomne. Za zarządzanie procesami odpowiada jądro systemu operacyjnego.

Wykonanie procesu musi przebiegać sekwencyjnie. Może przyjmować kilka stanów:

- działający,
- czekający na udostępnienie przez system operacyjny zasobów,
- przeznaczony do zniszczenia,
- proces zombie,
- właśnie tworzony
- itd.

W skład procesu wchodzi:

- kod programu,
- licznik rozkazów,
- stos,
- sekcja danych.

2.1.1 Tworzenie procesów

- Użytkownik za pomocą powłoki zleca uruchomienie programu, proces wywołujący wykonuje polecenie fork, lub jego pochodną.
- System operacyjny tworzy przestrzeń adresową dla procesu oraz strukturę opisującą nowy proces w następujący sposób:
  - wypełnia strukturę opisującą proces,
  - kopiuje do przestrzeni adresowej procesu dane i kod, zawarte w pliku wykonywalnym,
  - ustawia stan procesu na działający,
  - dołącza nowy proces do kolejki procesów oczekujących na procesor (ustala jego priorytet),
  - zwraca sterowanie do powłoki użytkownika.

2.1.2 Wykonywanie procesów

Dany proces rozpoczyna wykonywanie w momencie przełączenia przez Jądro systemu operacyjnego przestrzeni adresowej na przestrzeń adresową danego procesu oraz takie zaprogramowanie procesora, by wykonywał kod procesu. Wykonujący się proces może żądać pewnych zasobów, np. większej ilości pamięci. Zlecenia takie są na bieżąco realizowane przez system operacyjny.

2.1.3 Kończenie procesów

- Proces wykonuje ostatnią instrukcję - zwraca do systemu operacyjnego kod zakończenia. Jeśli proces zakończył się poprawnie zwraca wartość 0, w przeciwnym wypadku zwraca wartość kodu błędu.
- W momencie zwrotu do systemu operacyjnego kodu zakończenia, system operacyjny ustawia stan procesu na przeznaczony do zniszczenia i rozpoczyna zwalnianie wszystkich zasobów, które w czasie działania procesu zostały temu procesowi przydzielone.

- System operacyjny po kolei kończy wszystkie procesy potomne w stosunku do procesu macierzystego.
- System operacyjny zwalnia przestrzeń adresową procesu. Jest to dosłowna śmierć procesu.
- System operacyjny usuwa proces z kolejki procesów gotowych do uruchomienia i szereguje zadania. Jest to ostatnia czynność wykonywana na rzecz procesu.
- Procesor zostaje przydzielony innemu procesowi.

2.2 Wątek

Wątek (ang. thread) to jednostka wykonawcza w obrębie jednego procesu, będąca kolejnym ciągiem instrukcji wykonywanym w obrębie tych samych danych (w tej samej przestrzeni adresowej). Wątki tego samego procesu korzystają ze wspólnego kodu i danych, mają jednak oddzielne stosy. W systemach wieloprocesorowych, a także w systemach z wywłaszczaniem, wątki mogą być wykonywane równocześnie (współbieżnie). Równoczesny dostęp do wspólnych danych grozi jednak utratą spójności danych i w konsekwencji błędem działania programu. Sposób realizacji wszystkich tych działań jest różny dla różnych systemów operacyjnych.

2.3 Programy

2.3.1 Tworzenie nowego procesu w systemie UNIX przy pomocy *execve()*

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main()
{
  pid_t pid;
  int retVal;
  char *const parmList[] = {"/bin/echo", NULL, "Hello!", NULL};
  char *const envParms[2] = {"SH=/bin/sh", NULL};

  if ((pid = fork()) == -1)
    perror("Error: fork() died.");
  else if (pid == 0)
  {
    retVal = execve("/bin/sh", parmList, envParms);
    if (retVal)
      perror("Error: program returned error code %d",retVal);
  } else
    retVal = wait(&stan);
}
```

2.3.2 Tworzenie nowego wątku w systemie UNIX przy pomocy biblioteki *pthread.h*

```
#include <iostream>
#include <pthread.h>

using namespace std;
#define THNO 10

void* fun(void *arg)
{
```

```
  int thNo = *(int *)arg;
  cout << thNo << endl;
}

int main()
{
  pthread_t thr[THNO];
  for (int i = 0; i < THNO; i++)
  {
    pthread_create(&thr[i],NULL,fun,(void *)&i);
    pthread_join(thr[i],NULL);
  }

  pthread_exit(0);
}
```

2.4 Funkcja pthread create(...)

```
int pthread_create( pthread_t *thread, pthread_attr_t *attr,
void(*startroutine)(void *), void * arg )
```

Gdzie:

- `pthread_t * thread` – identyfikator wątku (wskaźnik)
- `pthread_attr_t * attr` – struktura zawierająca atrybuty wątku
- `void(*startroutine)(void *)` - procedura do wykonania przez wątek
- `void * arg` – argumenty procedury startroutine

3 Muteks

3.1 Definicja

Muteks – odpowiednik zamka, zmienna używana do sterowania dostępem do sekcji krytycznych (nazwa od mutual exclusion – wzajemne wykluczanie)

3.2 Wzajemne wykluczanie

Zakładamy, że dwa lub więcej procesów chce jednocześnie uzyskać dostęp do zasobu, który nie może być współdzielony – jeśli jeden proces otrzyma dostęp, to powinien wykluczyć inne z możliwości dostępu. Problemem jest rozstrzygnięcie konfliktów przy udzielaniu dostępu i umożliwienie dalszego działania po zakończeniu używania zasobu przez proces.

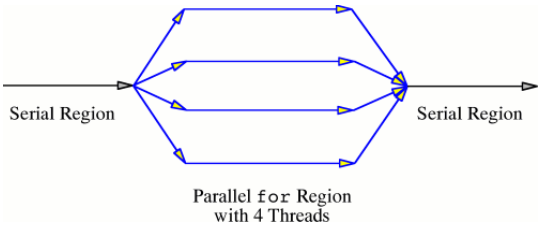
Cechy pożądane - bezpieczeństwo i żywotność

Możliwe błędy: zakleszczenie, zagłodzenie.

3.3 Program

```
#include <iostream>
#include <pthread.h>

using namespace std;

const int THNO = 10;

pthread_mutex_t mutex;
```

```
int thCounter = THNO;
pthread_cond_t ptCond;

// Funkcja tworzaca blokade wewnatrz watku
void bariera()
{
  // Zamykanie mutexa
  pthread_mutex_lock(&mutex);
  // Zmniejszenie licznika
  thCounter--;

  // Jesli nie wszystkie watki przed bariera
  if (thCounter >= 1)
    // Czekaj na spelnienie warunku
    pthread_cond_wait(&ptCond,&mutex);
  else
    // Rozglos spelnienie warunku
    pthread_cond_broadcast(&ptCond);

  // Odblokuj mutex
  pthread_mutex_unlock(&mutex);
}

void * fun(void *arg)
{
  int thNo = *(int*)arg;

  cout << thNo << " przed" << endl;
  bariera();
  cout << thNo << " za" << endl;
}

int main()
{
  pthread_t thr[THNO];
  pthread_cond_init(&ptCond,NULL);

  // Inicjuj mutex
  pthread_mutex_init(&mutex,NULL);

  for (int i = 0; i < THNO; i++)
  {
    int thNo = i;
    pthread_create(&thr[i],NULL,fun,(void*)&thNo);
  }

  for (int i = 0; i < THNO; i++)
    pthread_join(thr[i],NULL);

  pthread_exit(0);
}
```



Rysunek 1: Model *fork-join*

4 Zmienne warunku pthread

Zmienne warunku umożliwiają synchronizację między wątkami poprzez efektywną realizację oczekiwania przez wątki na spełnienie określonego warunku. Służą tylko do identyfikacji konkretnego warunku, same warunki są określane przez programistę (np. za pomocą standardowych instrukcji warunkowych). Spełnienie warunku oznacza jakąś zmianę stanu – ta zmiana musi być dostępna dla wątków, ale tylko w ramach sekcji krytycznej, dlatego każde oczekiwanie na spełnienie warunku jest związane nie tylko ze zmienną warunku ale także muteksem broniącym dostępu do sekcji krytycznej. W programie może być wiele sekcji krytycznych i wiele warunków na spełnienie których oczekiwać mogą wątki, stąd też w programie różne muteksy i różne zmienne warunku.

4.1 pthread cond wait i pthread cond signal

- `int pthread_cond_wait( pthread_cond_t *cond, pthread_mutex_t *mutex)` - oczekiwanie na spełnienie warunku
- `int pthread_cond_signal( pthread_cond_t *cond)` - ogłoszenie spełnienia warunku

4.2 Program

Jw.

5 Model OpenMP

5.1 Działanie metody *fork-join*

Metoda *fork-join* jest modelem obliczeń równoległych używanym przez bibliotekę OpenMP. Po otrzymaniu dyrektywy `parallel`, wątek tworzy szereg wątków potomnych, które współpracują dokonując obliczeń. Po wykonaniu operacji równoległej, wątki łączą się z powrotem (Rys. 1).

5.2 Dyrektywa parallel

Dyrektywa parallel dotyczy zawsze pętli znajdującej się pod nią, nie ma więc potrzeby (choć można) pisania po pętli. Dyrektywa ta tworzy tzw. region równoległy, w którym praca dzielona jest pomiędzy procesory.

Wykonanie programu wygląda następująco: poza regionami równoległymi, kod wykonywany jest przez jeden procesor, tzw. master processor. Gdy natrafiamy na początek regionu równoległego, zatrudniane są pozostałe procesory, które przy kończeniu regionu równoległego są zwalniane. Jest to tzw. model fork-join, schematycznie przedstawiony na poniższym rysunku.

5.3 Określanie ilości wątków

Ilość wątków, na które zostanie podzielony program jest kontrolowana w następujący sposób:

- zmienna środowiskowa `OMPC_NUM_PROCS` oznacza ile jest dostępnych procesorów na danym komputerze. Uruchomionych zostanie tyle wątków. Jeśli nie jest ustawiona sprawdzana jest rzeczywista liczba procesorów.
- zmienna środowiskowa `OMP_NUM_THREADS` oznacza domyślną liczbę wątków jakie mają być utworzone. Jest brana pod uwagę jeśli jej wartość jest większa od wartości `OMPC_NUM_PROCS`
- `funkcja omp_set_num_threads(int num_threads)`

6 Dyrektywy OpenMP

7 Dyrektywy podziału pracy

8 Klauzule współdzielenia zmiennych

8.1 Rodzaje dyrektyw

8.1.1 Dyrektywy podziału pracy (*work-sharing constructs*)

Konstrukcje work-sharing rozdzielają pracę pomiędzy członków teamu. Nie tworzą nowych wątków, brak synchronizacji przy wejściu

- single – obszar wykonywany przez jeden wątek:

```
#pragma omp single lista_klauzul
```

- sections – zbiór obszarów wykonywanych przez pojedyncze wątki:

```
#pragma omp sections lista_klauzul
{
  #pragma omp section
   {....}
  #pragma omp section
   {....}
   /* itd. */
}
```

- for - równoległa pętla for

```
#pragma omp for lista_klauzul
for(.......){........} // pętla w postaci kanonicznej
```

8.1.2 Dyrektywy synchronizacji

- master – podobnie jak single, ale dotyczy konkretnie wątku głównego

```
#pragma omp master znak_nowej_linii {....}
```

- critical – sekcja krytyczna (sekcje o różnych nazwach są traktowane odrębnie)

```
#pragma omp critical nazwa znak_nowej_linii {....}
```

- barrier – klasyczna bariera; dyrektywa barrier niejawnie występuje w wielu miejscach przy realizacji innych dyrektyw

```
#pragma omp barrier znak_nowej_linii
```

- atomic – atomowa zmiana wartości zmiennej (możliwa realizacja za pomocą sekcji krytycznej lub rozkazów sprzętowych)

  ```
  #pragma omp atomic znak_nowej_linii
  ```

- flush – ukończenie wszelkich operacji na zmiennych, ustalenie jednolitej dla wszystkich wątków wartości zmiennych (np. przepisanie do pamięci z rejestrów, buforów, itp.); dyrektywa flush niejawnie występuje w wielu miejscach przy realizacji innych dyrektyw

  ```
  #pragma omp flush lista_zmiennych znak_nowej_linii
  ```

- ordered - wykonanie iteracji otaczającej pętli for uporządkowane jak w wersji sekwencyjnej

8.1.3 Klauzule współdzielenia zmiennych

- shared – zmienna wspólna wątków
- private – zmienna lokalna wątków
- firstprivate – zmienna lokalna wątków z kopiowaną wartością początkową
- lastprivate – zmienna lokalna wątków z wartością końcową równą wartości jaka byłaby przy wykonaniu sekwencyjnym

8.1.4 Klauzula reduction

Klauzula reduction(op, listazmiennych) przeprowadza redukcję za pomocą określonego operatora dla określonych zmiennych, np.

```
#pragma omp parallel for reduction(+:a)
for(i=0;i<n;i++) a += b[i];
```

- możliwe operatory (op): `+, *, ,&, ^, |`
- możliwe działania w pętli: x = x op wyrażenie, x++, x op= wyrażenie
- na początku pętli zmienna jest odpowiednio inicjowana
- zmienna jest traktowana jak prywatna dla poszczególnych wątków
- wynik po zakończeniu pętli równoległej ma być taki sam jak byłby po wykonaniu sekwencyjnym
- szczegóły realizacji są pozostawione implementacji

8.1.5 Traktowanie zmiennych

Zmienna jest wspólna (dostępna wszystkim wątkom) jeśli:

- istnieje przed wejściem do obszaru równoległego i nie występuje w dyrektywach i klauzulach czyniących ją prywatną
- została zdefiniowana wewnątrz obszaru równoległego jako zmienna statyczna
- zmienna jest prywatna (lokalna dla wątku) jeśli
- została zadeklarowana dyrektywą threadprivate
- została umieszczona w klauzuli private lub podobnej (firstprivate, lastprivate, reduction)
- została zdefiniowana wewnątrz obszaru równoległego jako zmienna automatyczna
- jest zmienną sterującą równoległej pętli for

9 Zależności między instrukcjami

Zależności danych

- zależności wyjścia (zapis po zapisie, write-after-write) - odpowiednia analiza kodu może doprowadzić do wniosku o możliwej eliminacji instrukcji lub przemianowaniu zmiennych

```
a := ......
a := ......
```

- antyzależności (zapis po odczycie, write-after-read) - także tutaj odpowiednie przemianowanie zmiennych może zlikwidować problem

```
... := a
a := ......
```

- zależności rzeczywiste (odczyt po zapisie, readafterwrite) - uniemożliwiają zrównoleglenie algorytmu – konieczne jest jego przeformułowanie w celu uzyskania wersji możliwej do zrównoleglenia

```
a := ......
... := a
```

Zależności danych przenoszone przez pętle

- zależności wyjścia (output dependencies) - możliwe zrównoleglenie działania musi zachować kolejność zapisu danych

```
for( i=0; i<N; i++ )
{
  A[i] = .... ;
  A[i+2] = ....;
}
```

- antyzależności (antidependencies) - możliwe zrównoleglenie działania poprzez przemianowanie

```
for( i=0; i<N; i++ )
{
  A[i] = ...;
  ... = A[i+2] ;
}
```

  - Przemianowanie:

```
for( i=0; i<N; i++ ) A0[i] = A[i];
for( i=0; i<N; i++ ) ... = A0[i+2];
for( i=0; i<N; i++ ) A[i] = ...;
```

- zależności rzeczywiste (true dependencies) - w przypadku użycia wskaźników automatycznie zrównoleglające kompilatory mogą podejrzewać występowanie utożsamienia (zamienności) nazw (aliasing)

```
for( i=0; i<N; i++ )
{
  A[i] = A[i1]
  + ...;
}
```

10 Model równoległy z przesyłaniem komunikatów

Program w modelu przesyłania komunikatów zawiera polecenia wysyłania i odbierania danych („komunikatów"). Dla każdego wysłania musi istnieć odpowiadające odebranie. Jeżeli w kodzie znajduje się polecenie wysłania danych i proces właśnie je realizuje to:

- albo istnieje inny proces wykonywany jednocześnie i realizujący inny program, który jest gotowy odebrać te dane – model MPMD
- albo polecenie odebrania jest realizowane przez inny proces wykonujący ten sam program – model SPMD (w modelu SPMD konieczne jest istnienie odpowiednich struktur sterowania, identyfikujących procesory wykonujące kod i określających, który procesor jaką operację realizuje)

Pisząc program z przesyłaniem komunikatów w modelu SPMD zakładamy, że jednocześnie kilka procesów realizuje ten program. Stosując model MPMD piszemy kilka programów i ustalamy wzajemną komunikację między nimi. Konkretne środowiska programowania z przesyłaniem komunikatów zawierają narzędzia uruchamiania programów (często także kompilowania i debugowania). W modelu programowania z przesyłaniem komunikatów przy kodowaniu dominuje perspektywa lokalna – rozważanie pojedynczego procesu i uzgadnianie jego realizacji z realizacją innych procesów. Perspektywa globalna jest konieczna przy analizie algorytmu – czy skoordynowane działanie wszystkich procesów prowadzi do rozwiązania problemu.

10.1 Model przesyłania komunikatów

- `send( cel, identyfikator, dane, typ, rozmiar )`
- `receive( źródło, identyfikator, dane, typ, rozmiar )`

gdzie:

- cel i źródło – definiowane w ramach przyjętej konwencji
- identyfikator – związany z konkretnym komunikatem lub typem komunikatów (ułatwia sterowanie procesami przesyłania wielu komunikatów)
- dane – treść komunikatu
- typ, rozmiar – charakterystyki przesyłanego komunikatu

10.2 Przesyłanie blokujące a nieblokujące

- Przesyłanie blokujące – procedura nie przekazuje sterowania dalej dopóki operacje komunikacji nie zostaną ukończone i nie będzie można bezpiecznie wykorzystywać ponownie buforów (zmiennych) będących argumentami operacji.
- Przesyłanie nieblokujące – procedura natychmiast przekazuje sterowanie dalszym instrukcjom programu; dla prawidłowego funkcjonowania może to wymagać dodatkowych procedur sprawdzających

10.3 MPI Wait

`int MPI_Wait(MPI_Request *preq, MPI_Status *pstat)` – oczekiwanie na zakończenie operacji (nieblokującego wysyłania lub odbierania). Czeka, aż operacja wysyłania lub odbioru zidentyfikowana przez zmienną request zostanie zakończona; Wejściowe parametry:

- request - kontekst wiadomości do sprawdzenia (uchwyt)

Wyjściowe parametry:

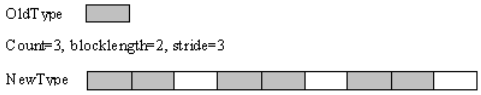
- status - status obiektu (Status);

11 Podstawowe procedury MPI

- `int MPI_Init( int *pargc, char ***pargv)` – inicjacja środowiska, wywoływana jako pierwsza procedura MPI
- `int MPI_Comm_size(MPI_Comm comm, int *psize)` – zwrot rozmiaru komunikatora (liczby procesów tworzących komunikator), `comm` jest uchwytem do nieprzenikalnego obiektu, często będziemy zamiennie mowić o obiekcie nieprzenikalnym i uchwycie do niego
- `int MPI_Comm_rank(MPI_Comm comm, int *prank)` – podanie rangi procesu wywołującego w ramach komunikatora `comm`
- `int MPI_Send(void *msg, int count, MPI_Datatype dt, int dst, int tag, MPI_Comm comm);` - wysyła komunikat typu `dt` do procesu numer `dst` oznaczony znacznikiem `tag` w obrębie komunikatora `comm`. Typ komunikatu jest zawarty w zmiennej datatype i może to być któryś z predefiniowanych typów takich jak `MPI_INT`, `MPI_FLOAT`, `MPI_DOUBLE`, `MPI_CHAR`, lub inne, jak i typy zdefiniowane przez użytkownika. Tag jest liczbą w zakresie [0..`MPI_TAG_UB`] i określa dodatkowy typ komunikatu wykorzystywany przy selektywnym odbiorze funkcją `MPI_Recv`.
- `int MPI_Recv(void *msg, int count, MPI_Datatype dt, int src, int tag, MPI_Comm comm, MPI_Status *status);` - funkcja odczytuje z kolejki komunikatora `comm` (z ewentualnym blokowaniem do czasu nadejścia) pierwszy komunikat od procesu `src` oznaczony znacznikiem `tag` typu `dt`. Wynik umieszczany jest w buforze `msg` a status operacji w zmiennej `status`. Jeżeli proces ustawi `src == MPI_ANY_SOURCE` to odczytany będzie pierwszy komunikat od dowolnego procesu. Podobnie, dla `tag == MPI_ANY_TAG` nie będzie sprawdzany znacznik typu wiadomości. Bufor stanu `status` musi zostać uprzednio stworzony przez programistę. Dla C jego pojedynczy element składa się z trzech liczb całkowitych: `MPI_SOURCE`, `MPI_TAG` oraz `MPI_STATUS`. W ogólnym przypadku bowiem (przy odbieraniu komunikatów w których `count > 1`) tablica `status` określa nam źródło i typ każdego komunikatu z osobna
- `int MPI_Finalize(void)` – zakończenie pracy środowiska MPI, najlepiej jako ostatnia wywoływana procedura programu

11.1 Program

```
#include "mpi.h"
int main( int argc, char** argv )
{
  int rank, ranksent, size, source, dest, tag, i; MPI_Status status;
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  MPI_Comm_size( MPI_COMM_WORLD, &size );
  if( rank != 0 )
  {
    dest=0; tag=0;
    MPI_Send( &rank, 1, MPI_INT, dest, tag, MPI_COMM_WORLD );
  } else {
    for( i=1; i<size; i++ )
    {
      MPI_Recv( &ranksent, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,

MPI_COMM_WORLD, &status );
      printf(\Dane od procesu o randze: %d (%d)\n", ranksent, status.MPI_SOURCE );
    }
```



Rysunek 2: Typ *vector*

```
  }
  MPI_Finalize();
  return(0);
}
```

12 Typy danych MPI

Podobnie jak w językach programowania istnieją elementarne typy danych (predefiniowane obiekty typu `MPI_Datatype`), takie jak `MPI_INT, MPI_DOUBLE, MPI_CHAR, MPI_BYTE`. Podobnie jak w językach programowania istnieją utworzone typy danych (odpowiedniki rekordów czy struktur), które dają możliwość prostego odnoszenia się do całości złożonych z wielu zmiennych elementarnych. Definicja typu danych określa sposób przechowywania w pamięci zmiennych typów elementarnych tworzących zmienną danego typu. Definicję typu można przedstawić w postaci tzw. mapy typu postaci: $(typ_1, odstep_1), \cdots, (typ_n, odstep_n)$.

12.1 Rodzaje typów danych

- `int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)` - zmienna złożona z count zmiennych typu oldtype występujacych w pamięci bezpośrednio po sobie
- `int MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype` - zmienna złożona z count bloków zmiennych typu oldtype, każdy z bloków o długości blocklength i o początku w pamięci w odległości stride zmiennych typu oldtype od początku bloku poprzedzającego (Rys. 2)

13 Sieci połączeń

14 Sieci statyczne

Istota architektury systemów wieloprocesorowych i wielokomputerowych – sieć połączeń

Rodzaje sieci połączeń:

- podział ze względu na łączone elementy:
  - połączenia procesory-pamięć (moduły pamięci)
  - połączenia międzyprocesorowe (międzywęzłowe)
- podział ze względu na charakterystyki łączenia:
  - sieci statyczne – zbiór połączeń dwupunktowych
  - sieci dynamiczne – przełączniki o wielu dostępach

14.1 Połączenia dynamiczne

- magistrala (bus) – prostota konstrukcji, niski koszt, słaba skalowalność przy rozszerzeniach. Ddla zwiększenia skalowalności wprowadza sie pamięć podręczną co jednak powoduje problemy z jej spójnością. Najczęściej powiązana z systemami SMP i UMA
- krata przełączników (crossbar switch; przełącznica krzyżowa) – efektywność, wysoki koszt rozszerzeń (zazwyczaj liczba przełączników proporcjonalna do kwadratu liczby procesorów)
- sieć wielostopniowa (multistage network) – rozwiązanie pośrednie
- sieć lokalna (LAN) – dedykowana lub ogólnego przeznaczenia. Topologie zbliżone do sieci statycznych

14.2 Połączenia statyczne

- Sieć w pełni połączona - parametry: średnica = 1, połączalność = $p-1$, szerokość = $p^2/4$, koszt = $p(p-1)/2$.
- Gwiazda: parametry: 2, 1, 1, $p-1$
- Rząd: $p-1$, 1, 1, $p-1$
- Kraty: 1D, 2D, 3D
  - Krata 2D bez zawijania: $2\sqrt{p-1}$, 2, $\sqrt{p}$, $2(p-\sqrt{p})$
  - Torus 2D: $2\sqrt{p}/2$, 4, $2\sqrt{p}$, $2p$
- . Drzewa: zwykłe lub tłuste
  - Drzewo binarne zupełne: $2log(p/2+1/2)$, 1, 1, $p-1$
- Hiperkostki: 1D, 2D, 3D itd..
  - Hiperkostka: $logp$, $logp$, $p/2$, $p(logp)/2$

14.3 Parametry

- Średnica – maksymalna odległość dwóch węzłów
- Połączalność krawędziowa (arch connectivity) – minimalna liczba krawędzi koniecznych do usunięcia dla podziału sieci na dwie sieci rozłączne (mierzy wrażliwość na przepełnienie łączy, także odporność na uszkodzenia)
- Szerokość połowienia (bisection width) – minimalna liczba krawędzi koniecznych do usunięcia dla podziału sieci na dwie równe sieci rozłączne (razem z przepustowością pojedynczego kanału daje przepustowość przepołowienia – (bisection bandwidth)
- Koszt – np. szacowany liczbą drutów, kart sieciowych itp.

14.4 Hiperkostki

Hiperkostka, kostka (angielskie hypercube) - system wieloprocesorowy o architekturze przełączanej, który topologicznie tworzy n-wymiarowy sześcian. Wychodząc od sześcianu trójwymiarowego, w którym każdy wierzchołek oznacza procesor, a każda krawędź symbolizuje połączenie między dwoma procesorami, hiperkostka rzędu 4 buduje się przez połączenie bliźniaczych wierzchołków obu sześcianów. Istnieją hiperkostki zawierające 1024, a nawet 16 384 procesory. Ilość procesorów dla n-wymiarowej hiperkostki = 2^n.

15 Grupowe przesyłanie komunikatów

16 MPI Reduce

Schematy grupowego przesyłania komunikatów:

- rozgłaszanie jeden do wszystkich (singlenode broadcast)
- rozpraszanie jeden do wszystkich (singlenode scatter)
- zbieranie wszyscy do jednego (singlenode gather)
- redukcja, gromadzenie wszyscy do jednego (reduction, singlenode accumulation)
- rozgłaszanie wszyscy do wszystkich (multinode broadcast) (równoważne zbieraniu wszyscy do wszystkich)
- gromadzenie (redukcja) wszyscy do wszystkich (multinode accumulation)
- wymiana wszyscy do wszystkich (total exchange) (równoważna rozpraszaniu wszyscy do wszystkich)

16.1 Przykładowe procedury MPI

16.1.1 Rozgłaszanie jeden do wszystkich

`int MPI_Bcast( void *buff, int count, MPI_Datatype datatype, int root, MPI_Comm comm)` – rozesłanie count zmiennych typu datatype przechowywanych w pamięci procesu o randze root pod adresem buff do wszystkich procesów tworzących komunikator comm ; po rozesłaniu zmienne znajdują się pod adresem buff w pamięci wszystkich procesów

16.1.2 Zbieranie wszyscy do jednego

`int MPI_Gather(void *sbuf, int scount, MPI_Datatype sdtype, void *rbuf, int rcount, MPI_Datatype rdtype, int root, MPI_Comm comm)` – przesłanie przez każdy z procesów tworzących komunikator comm scount zmiennych typu sdtype przechowywanych w pamięci pod adresem sbuff do procesu o randze root, który zbiera wszystkie zmienne, w kolejności rang wysyłajacych procesów) w buforze o typie rdtype (z rozmiarem na pojedynczy komunikat rcount) pod adresem rbuff

16.1.3 Rozpraszanie jeden do wszystkich

`int MPI_Scatter(void *sbuf, int scount, MPI_Datatype sdtype, void *rbuf, int rcount, MPI_Datatype rdtype, int root, MPI_Comm comm)` - przesłanie przez proces o randze root do każdego z procesów tworzących komunikator comm scount zmiennych typu sdtype przechowywanych w pamięci począwszy od adresu sbuff ; każdy z procesów otrzymuje sobie przynależną część danych i umieszcza je w buforze o typie rdtype i rozmiarze rcount zmiennych pod adresem rbuff

16.1.4 Redukcja wszyscy do jednego

`int MPI_Reduce(void *sbuf, void *rbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)` – przesłanie przez wszystkie procesy tworzące komunikator comm count zmiennych typu datatype przechowywanych w pamięci począwszy od adresu sbuf do procesu o randze root, który wykonuje na każdej kolejnej otrzymanej grupie odpowiadającej kolejnej zmiennej operację op i przechowuje wynik w kolejnych zmiennych w buforze rbuf

Operacje stosowane przy realizacji redukcji:

- predefiniowane operacje – uchwyty do obiektów typu `MPI_Op` (każda z operacji ma dozwolone typy argumentów):
  - `MPI_MAX` – maksimum
  - `MPI_MIN` – minimum
  - `MPI_SUM` – suma
  - `MPI_PROD` – iloczyn
- operacje maksimum i minimum ze zwróceniem indeksów
- operacje logiczne i bitowe
- operacje definiowane przez użytkownika za pomocą procedury: `int MPI_Op_create(MPI_User_function *func, int commute, MPI_Op *pop);`

17 Gniazda - transmisja połączeniowa

18 Gniazda - transmisja bezpołączeniowa

Interfejs gniazd Unixa zaprojektowany został do realizacji modelu przetwarzania klient-serwer. Serwer jest programem oczekującym na żądania klientów i realizującym różnorodne usługi. Klient jest programem, który zgłasza serwerowi żądanie i następnie odbiera rezultaty działania serwera. Istotą relacji klient-serwer jest duża niezależność występujących programów, które mogą być wykonywane na różnych systemach komputerowych, o różnych architekturach i systemach operacyjnych. W programowaniu dla modelu klient-serwer istnieją dwa istotne problemy techniczne: wyszukiwanie serwera (usługodawcy) przez klienta oraz przesyłanie danych pomiędzy klientem i serwerem.

W interfejsie gniazd problem wyszukiwania serwera rozwiązywany jest przy użyciu mechanizmów internetowych – identyfikacji programu poprzez adres internetowy węzła sieci (host) i numer portu jednoznacznie powiązany z konkretnym programem. Problem przesyłania danych pomiędzy klientem i serwerem jest rozwiązany poprzez zastosowanie protokołów internetowych: TCP dla transmisji danych w postaci strumieni bajtów – model połączeniowy oraz UDP dla transmisji datagramów - model bezpołączeniowy. Gniazdo można utożsamiać z tzw. półasocjacją czyli trójką (protokół, adres węzła, numer portu).

Nawiązanie połączenia do komunikacji wymaga powiązania dwóch gniazd (używających tego samego protokołu) – powstaje wtedy pełna asocjacja. W uzyskaniu adresu i numeru portu dla konkretnego serwera przydatne są np. funkcje systemowe: `gethostbyname` (obecnie `getipnodebyname`) i `getservbyname` (lub ich warianty). Przesyłanie danych odbywa się – zgodnie z Unixową tradycją – analogicznie jak zapisywanie do pliku i odczytywanie z pliku (rolę deskryptorów plików pełnią identyfikatory połączonych gniazd). Szczegóły realizacji komunikacji zależą od tego czy program jest klientem czy serwerem oraz od tego czy transmisja danych jest połączeniowa czy bezpołączeniowa.

18.1 Specyfikacja interfejsu gniazd

`int socket(int domain, int type, int protocol);` - procedura tworzy pojedyncze gniazdo i zwraca identyfikator (deskryptor) gniazda

Parametry:

- dziedzina (dla rodziny protokołów internetowych oznaczona przez `AF_INET`) – gniazda mogą obsługiwać różne typy komunikacji międzyprocesowej (np. za pomocą plików, tzw. gniazda w dziedzinie Unixa – oznaczanej przez `AF_UNIX`)
- typ: połączeniowy (`SOCK_STREAM`) lub bezpołączeniowy (`SOCK_DGRAM`); możliwe są też inne typy gniazd

- protokół: w dziedzinie internetowej `SOCK_STREAM` jest domyślnie kojarzony z TCP, a `SOCK_DGRAM` z UDP; można protokoły określać jawnie (także inne niż dwa powyższe)

`int bind(int sockfd, struct sockaddr *addr, socklen_t addrlen)` - procedura wiąże gniazdo z adresem lokalnym i zwraca kod sukcesu lub błędu. Proces, w ramach którego następuje wywołanie procedury bind informuje system, że będzie obsługiwał komunikaty przesłane na wskazany adres

Parametry:

- sockfd – deskryptor gniazda
- addr – wskaźnik do struktury zawierającej adres
- addrlen – długość w bajtach struktury adresu

18.2 Transmisja bezpołączeniowa

W przeciwieństwie do komunikacji połączeniowej komunikacja bezpołączeniowa zakłada większą symetrię klienta i serwera. I klient, i serwer jawnie wiążą tworzone gniazda z adresami. Postać procedur przesyłania danych (sendto, recvfrom zamiast write, read) uwzględnia różnice w odniesieniu do komunikacji połączeniowej.

18.3 Transmisja połączeniowa

Procedury nawiązywania połączenia między serwerem i klientem – strona serwera:

`int listen(int sockfd, int max_dl_kol)` - zgłoszenie gotowości odbierania żądań od klientów dla gniazda sockfd. `max_dl_kol` określa maksymalną dopuszczalną długość kolejki żądań klientów oczekujących na realizację

`int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);` - utworzenie połączenia dzięki gniazdu sockfd z klientem o adresie zwracanym za pomocą dwóch ostatnich argumentów procedury. Procedura zwraca deskryptor nowego gniazda, które będzie służyło do właściwej wymiany danych (np. będzie argumentem procedur read i write)

Procedury nawiązywania połączenia między serwerem i klientem – strona klienta:

Utworzenie gniazda po stronie klienta za pomocą procedury socket (gniazdo nie jest łączone jawnie z konkretnym adresem, choć w rzeczywistości system nadaje mu indywidualny adres)

`int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen);` - połączenie klienta z serwerem (utworzenie pełnej asocjacji). sockfd jest deskryptorem gniazda klienta utworzonego przez socket. Dwa ostatnie argumenty procedury określają adres gniazda serwera z którym następuje połączenie (struktura `serv_addr` musi zostać utworzona za pomocą odpowiednich omówionych wcześniej narzędzi). Deskryptor sockfd może być dalej wykorzystywany do właściwej wymiany danych (np. jako argument procedur read i write)

18.4 Demony UNIX

Demon Unixowy jest procesem działającym w sposób ciagły w tle, bez powiązania z terminalem i bez udziału użytkownika uruchamiającego demona (demony często uruchamiane są przy starcie systemu). Tworzenie demonów Unixowych jako serwerów wykorzystujących mechanizm gniazd do odbierania żądań od klientów wymaga poza opisanymi poprzednio krokami dodatkowych operacji związanych z odbieraniem sygnałów, standardowymi plikami, obsługą plików i katalogów, itp. Przykładem często stosowanego demona jest demon Internetu – inetd (lub xinetd), który może służyć jako superserwer dla innych serwerów Internetowych. Zadaniem inetd jest zmniejszenie liczby procesów oczekujących na żądania klientów i zajmujących zasoby komputera pracując w tle, poprzez zastąpienie ich jednym oczekującym procesem. inetd jest uruchamiany przy starcie sytemu i nasłuchuje czy nie są zgłaszane żądania dla dowolnego z serwerów zarejestrowanych w pliku konfiguracyjnym (/etc/inetd.conf lub /etc/xinetd.conf). W przypadku nadejścia żądania inetd uruchamia odpowiedni serwer i kontynuuje nasłuchiwanie.

19 Sun RPC

Wewnętrzny mechanizm realizacji wywołań odleg.ych jest złożony i wielostopniowy:

- z programu klienta wywoływana jest procedura w standardowy sposob
- lokalny system zamiast procedury realizującej usługę wywołuje tzw. namiastkę procedury klienta (client stub)
- namiastka procedury u klienta wykorzystuje mechanizmy sieciowe (ukrywane przed programistą klienta) do wysłania żądania do odpowiedniego serwera i odebrania od niego wynikow realizacji usługi
- namiastka dokonuje także odpowiednich przekształceń argumentow i wyników pomiędzy formą języka programowania programu klienta i formą komunikacji sieciowej (tzw. przetaczanie - marshalling)

Pozostaje problem jak namiastka u klienta ma wyszukać namiastkę po stronie serwera - można zorganizować centralny system, gdzie każda namiastka u klienta kontaktuje się z systemem zarządzającym o znanym adresie, ktory kieruje żądania w odpowiednie miejsce, można także wprowadzić w każdym komputerze realizującym wywołania zdalne lokalny system kierujący nadchodzące żądania do odpowiednich procesów (portmapper).

Tworzenie programu serwera:

- kompilacja pliku z interfejsem (identycznego lub tego samego jak dla programu klienta), ktora tworzy:
  - plik namiastki po stronie serwera
  - plik nagłówkowy dla zdalnie wywo.ywanej procedury w języku C
  - program do przekształcania argumentow
  - ewentualnie szkielet programu klienta w języku C do uzupełnienia przez programistę i wzor pliku Makefile
- napisanie kodu klienta z odpowiednimi wywołaniami zgodnymi z plikiem nagłówkowym
- utworzenie (kompilacja i konsolidacja) programu serwera

Tworzenie programu klienta:

- określenie interfejsu zdalnie wywoływanej procedury (w ramach XDR)
- kompilacja pliku z interfejsem, ktora tworzy:
  - plik namiastki po stronie klienta
  - plik nagłówkowy dla zdalnie wywoływanej procedury w języku C
  - program do przekształcania argumentow
  - ewentualnie szkielet programu klienta w języku C do uzupełnienia przez programistę i wzor pliku Makefile
- napisanie kodu klienta z odpowiednimi wywołaniami zgodnymi z plikiem nagłówkowym
- utworzenie (kompilacja i konsolidacja) programu klienta

19.1 Plik XDR dla serwera

```
struct typ_danych_1{ double p; int j; };
typedef struct typ_danych_1 typ_danych_1;

struct typ_wyniku_1{ int i; double q; };
typedef struct typ_wyniku_1 typ_wyniku_1;

program ZROB_COS_ZDALNIE
{
  version ZROB_COS_ZDALNIE_WERSJA
  {
    typ_wyniku_1 ZDALNA_PROCEDURA_1(typ_danych_1) = 1;
    typ_wyniku_2 ZDALNA_PROCEDURA_2(typ_danych_2) = 2;
    ......
  } = 1;
} = 12345;
```

20 Architektura SOA

Architektura usługowa (Service Oriented Architecture SOA) – sposób organizacji systemów rozproszonych, w którym rozważa się każdy system poprzez pryzmat świadczonych przez niego usług. Termin Service Oriented Architecture jest określeniem architektury systemow rozproszonych, w ktorych wyrożnia się:

- usługobiorcę - klienta korzystającego z usług
- dostawcę usług - usługą jest realizacja pewnego przetwarzania z wykorzystaniem dostarczonych danych
- rejestr usług - miejsce, gdzie klient uzyskuje informacje o potrzebnych mu usługach

Podstawą SOA jest wykorzystanie przesyłania komunikatow do wymiany informacji między uczestnikami przetwarzania. Architektury usługowe odróżnia się od architektur obiektowych i architektur komponentowych.

Dostarczając usługę w ramach architektury usługowej należy w ogolnie znanym i dostępnym rejestrze dokonać zgłoszenia:

- miejsca dostępności usługi
- sposobu korzystania z usługi

Klient chcący korzystać z określonej usługi wyszukuje w ogolnie znanych i dostępnych rejestrach opisy świadczonych usług, wybiera najbardziej mu odpowiadającą, a następnie kontaktuje się z serwerem usługi w celu realizacji przetwarzania.

Sposób korzystania z usługi jest równoważny interfejsowi oprogramowania realizującego usługę. Interfejs taki powinien być napisany w sposób niezależny od:

- języka programowania
- systemu operacyjnego
- sprzętu realizujacego obliczenia

Sposób powiązania usługodawcy z usługobiorcą za pomoca tak określonych interfejsów określa się jako luźny (loosely coupled). Usługi zgłoszone w systemach SOA mogą być jednocześnie klientami innych usług. W ten sposób można tworzyć złożone schematy przetwarzania (workflows) składające

się z wielu, odpowiednio zorganizowanych usług. Architektura usługowa jest w tym ujęciu przeciwstawiona architekturze komponentowej, gdzie ten sam cel (ponowne wykorzystanie oprogramowania) jest realizowany za pomocą organizacji usług sieciowych (zamiast lokalnego komponowania dostarczonych fragmentów oprogramowania).

20.1 Web services

Nazwą Web Services określa się dziś jedną z możliwych realizacji modelu SOA, umożliwiającą wykorzystanie istniejącej infrastruktury internetowej (protokół HTTP, przeglądarki). Web Services wykorzystują następujące elementy, aby przystosować przetwarzanie rozproszone do standardowego środowiska internetowego:

- XML - język, ktory może być przetwarzany przez maszyny
- SOAP - protokół przesyłu danych wykorzystujący koperty XML (XML envelopes)
- WSDL - język opisu usług wykorzystuj.cy XML

20.2 WSDL

Dokument WSDL (napisany w XML) zawiera następujące elementy:

- `<interface>` - odpowiednik interfejsu IDL lub definicji klasy abstrakcyjnej w obiektowych językach programowania; element `<interface>` zawiera elementy `<operation>` opisujące pojedyncze procedury
- `<message>` - opisuje wymianę komunikatów związaną z wywołaniem procedury (argumenty wejścia i wyjścia)
- `<types>` - typy danych użytych jako argumenty
- `<binding>` - sposób realizacji wywołania, użyty protokół, itp.
- `<service>` - połączenie interfejsu, sposobu realizacji wywołania i adresu internetowego dla uzyskania ostatecznej definicji usługi

21 Gridy

W systemach gridowych rozważa się dostęp do zasobów i ich wykorzystanie poprzez pewien jednolity system. Jednolitość systemu ma być zapewniona poprzez wirtualizację zasobów, tzn. każdy zasób jest przedstawiany jako pewien możliwy do zrealizowania zestaw usług. Ze względu na oparcie systemów gridowych na usługach ostatnie specyfikacje GGF wskazują na Web Services jako podstawę realizacji tych systemów. Zasoby informatyczne tworzące systemy gridowe są ze swej natury heterogeniczne i rozproszone. Stanowią je:

- urządzenia (procesory, dyski, drukarki, urządzenia pomiarowe, wyposażenie laboratoriów, urządzenia sieciowe, itp.)
- programy, bazy danych, pliki

Ideą jest tworzenie systemów gridowych i współdzielenie zasobów przez różne podmioty, geograficznie rozproszone, tworzące tzw. wirtualne organizacje (VO). Systemy gridowe z założenia mają być budowane w oparciu o standardy. Liczba tych standardów ciągle rośnie, wraz z rozszerzaniem obszarów stosowania systemów gridowych. Specyfikacje systemów gridowych opracowywane są przez organizacje (głównie grupy robocze GGF – Global Grid Forum) w postaci rekomendacji. Rekomendacje GGF ukierunkowane są na wszystkie obszary zastosowań. Podstawowym standardem GGF jest OGSA (Open Grid Software Architecture) – specyfikacja określająca ogólna budowę systemów gridowych.

22 Właściwości systemów rozproszonych

Co tu napisać?

23 Taksonomia Flynna

Taksonomia Flynna jest klasyfikacją architektur komputerowych, zaproponowaną w latach sześćdziesiątych XX wieku przez Michaela Flynna, opierająca się na liczbie przetwarzanych strumieni danych i strumieni rozkazów.

W taksonomii tej wyróżnia się cztery grupy:

- SISD (Single Instruction, Single Data) - przetwarzany jest jeden strumień danych przez jeden wykonywany program - komputery skalarne (sekwencyjne).
- SIMD (Single Instruction, Multiple Data) - przetwarzanych jest wiele strumieni danych przez jeden wykonywany program - tzw. komputery wektorowe.
- MISD (Multiple Instruction, Single Data) - wiele równolegle wykonywanych programów przetwarza jednocześnie jeden wspólny strumień danych. W zasadzie jedynym zastosowaniem są systemy wykorzystujące redundancję (wielokrotne wykonywanie tych samych obliczeń) do minimalizacji błędów.
- MIMD (Multiple Instruction, Multiple Data) - równolegle wykonywanych jest wiele programów, z których każdy przetwarza własne strumienie danych - przykładem mogą być komputery wieloprocesorowe, a także klastry i gridy.

A teraz będzie chwilę po angielsku, bo nie miałem siły tłumaczyć...

23.1 UMA

Uniform Memory Access (UMA) is a computer memory architecture used in parallel computers having multiple processors and probably multiple memory chips.

All the processors in the UMA model share the physical memory uniformly. Peripherals are also shared. Cache memory may be private for each processor. In a UMA architecture, accessing time to a memory location is independent from which processor makes the request or which memory chip contains the target memory data. It is used in symmetric multiprocessing (SMP). A typical example of different memory architectures used in parallel computer is Non-Uniform Memory Access (NUMA).

23.2 ccNUMA

Nearly all CPU architectures use a small amount of very fast non-shared memory known as cache to exploit locality of reference in memory accesses. With NUMA, maintaining cache coherence across shared memory has a significant overhead.

Although simpler to design and build, non-cache-coherent NUMA systems become prohibitively complex to program in the standard von Neumann architecture programming model. As a result, all NUMA computers sold to the market use special-purpose hardware to maintain cache coherence, and thus class as "cache-coherent NUMA", or ccNUMA.

Typically, this takes place by using inter-processor communication between cache controllers to keep a consistent memory image when more than one cache stores the same memory location. For this reason, ccNUMA performs poorly when multiple processors attempt to access the same memory area in rapid succession. Operating-system support for NUMA attempts to reduce the frequency of this kind of access by allocating processors and memory in NUMA-friendly ways and by avoiding scheduling and locking algorithms that make NUMA-unfriendly accesses necessary.

Current implementations of ccNUMA systems are multi processor systems based on the Intel Itanium and AMD Opteron processor. Earlier ccNUMA approaches were systems based on the MIPS processor in SGI systems and Alpha processor EV7 of Digital Equipment Corporation(DEC).

23.3 SMP

Symmetric multiprocessing, or SMP, is a multiprocessor computer architecture where two or more identical processors are connected to a single shared main memory. Most common multiprocessor systems today use an SMP architecture.

SMP systems allow any processor to work on any task no matter where the data for that task are located in memory; with proper operating system support, SMP systems can easily move tasks between processors to balance the workload efficiently.

SMP is one of many styles of multiprocessor machine achitecture; others include NUMA (Non-Uniform Memory Access) which dedicates different memory banks to different processors allowing them to access memory in parallel. This can dramatically improve memory throughput as long as the data is localized to specific processes (and thus processors). On the downside, NUMA makes the cost of moving data from one processor to another, as in workload balancing, more expensive. The benefits of NUMA are limited to particular workloads, notably on servers where the data is often associated strongly with certain tasks or users.

Other systems include asymmetric multiprocessing (ASMP), in which separate specialized processors are used for specific tasks, and computer clustered multiprocessing (e.g. Beowulf), in which not all memory is available to all processors.

The former is not widely used or supported (though the high-powered 3D chipsets in modern videocards could be considered a form of asymmetric multiprocessing) while the latter is used fairly extensively to build very large supercomputers. In this discussion a single processor is denoted as a uni processor (UN).

24 Sposoby tworzenia programów równoległych

W procesie tworzenia programów równoległych istnieją dwa kroki o zasadniczym znaczeniu:

- wykrycie dostępnej współbieżności w trakcie realizacji programu
- określenie koniecznej synchronizacji lub wymiany komunikatów pomiędzy procesami lub wątkami realizującymi program

Pierwszy z tych kroków ma charakter bardziej twórczy, drugi bardziej techniczny (zakłada znajomość modelu programowania). Sposób realizacji tych kroków może zależeć od organizacji procesu tworzenia programu równoległego.

24.1 Algorytm sekwencyjny

Jeżeli do dyspozycji mamy algorytm lub program sekwencyjny, a nie mamy możliwości lub chęci analizy samego algorytmu korzystamy najczęściej z modelu programowania z pamięcią wspólną np. dla środowiska OpenMP: dokonujemy analizy pętli wybierając do zrównoleglenia te, które dają możliwość efektywnego wykonania równoległego i mają znaczący udział w czasie wykonania programu, przeprowadzamy poprawne i efektywne zrównoleglenie pętli poprzez:

- użycie odpowiednich dyrektyw kompilatora
- właściwe przyporządkowanie zmiennych jako wspólnych lub prywatnych (w rozmaitych wariantach, np. reduction)
- optymalną organizację obliczeń (zarządzanie podziałem iteracji między wątki, użycie sekcji krytycznych, itp.)

24.2 Problem

Dokonujemy analizy problemu starając się podzielić zadanie obliczeniowe na podzadania możliwe do wykonania rownoległego

- podział funkcji (functional decomposition)
- podział struktury danych (data decomposition)
- podział obszaru problemowego (domain decomposition)

Określamy konieczną wymianę informacji między podzadaniami w celu poprawnej i efektywnej realizacji obliczeń oraz wybieramy model programowania:

- rownoległości danych
- rownoległości na poziomie pętli
- rownoległości na poziomie zadań

Dobieramy stosowne narzędzia (środowiska programowania - HPF, OpenMP, MPI lub inne). Implementując problem dokonujemy ostatecznego odwzorowania obliczeń na architekturę systemu komputerowego. Klasycznym sposobem podzia.u zadania na podzadania jest dokonanie dekompozycji w dziedzinie problemu lub dekompozycji struktury danych i stosowanie zasady właściciel oblicza (owner computes).

25 Przyspieszenie obliczeń

$$S(p) = T_s/T_p(p)$$

gdzie:

- T_s – czas rozwiązania zadania najlepszym algorytmem sekwencyjnym na pojedynczym procesorze (w praktyce często zamiast T_s używa się $T_p(1)$)
- $T_p(p)$ – czas wykonania zadania rozważanym algorytmem równoległym na p procesorach

25.1 Efektywność zrównoleglenia

Efektywność zrównoleglenia:

$$E(p) = S(p)/p$$

Ideałem jest uzyskanie liniowego przyspieszenia i 100% efektywności zrównoleglenia.

26 Analiza Amdahla, analiza Gustafsona

Każdy program składa się z części sekwencyjnej (nie dającej się zrównoleglić), o czasie wykonania T^s, i części dającej się zrównoleglić idealnie, o czasie wykonania sekwencyjnego T^p. Czas wykonania równoległego jest więc równy:

$$T_p(p) = T^s + T^p/p$$

co prowadzi do przyspieszenia obliczeń:

$$S(p) = (T^s + T^p)/(T^s + T^p/p)$$

I efektywności:

$$E(p) = (T^s + T^p)/(pT^s + T^p)$$

26.1 Prawo Amdahla

Przy liczbie procesorów zmierzającej do nieskończoności czas rozwiązania określonego zadania nie może zmaleć poniżej czasu wykonania części sekwencyjnej programu, przyspieszenie nie może przekroczyć wartości $(T^s + T^p)/T^s$, a efektywność zrównoleglenia zmierza do zera. Prawo Amdahla stwierdza istotne ograniczenie wydajności programów równoległych (co historycznie przyczyniło się do zmniejszenia zainteresowania badaniami w tej dziedzinie). Prawo Amdahla kładzie nacisk na istnienie fragmentów sekwencyjnych w programach, nie uwzględnia natomiast komunikacji, która dodatkowo spowalnia obliczenia równoległe.

26.2 Analiza Gustafsona

Prawo Amdahla dotyczy równoległego rozwiązania z coraz większą liczbą procesorów tego samego zadania (analiza przy stałym rozmiarze zadania). Możliwe jest odwrócenie zagadnienia i zamiast pytać jaki jest czas równoległego rozwiązania danego zadania, zapytać jaki ma być ciąg zadań (rozmiarów zadań) aby uzyskać stały czas wykonania przy rosnącej liczbie procesorów (analiza przy stałym czasie wykonania). Rozwiązaniem jest np. ciąg zadań o stałym rozmiarze części sekwencyjnej i rozmiarze części równoległej proporcjonalnym do liczby procesorów, taki że czas rozwiązania równoległego wynosi: $T_p(p) = T^s + (pT^0)/p$

27 Składniki czasu wykonania rzeczywistych programów równoległych

- Czas rzeczywistych obliczeń T^{obl}
- Czas przeznaczony na komunikację T^{kom}
- Czas jałowy, związany np. z oczekiwaniem na jeden z procesów T^{jaowy}

Optymalizacja czasu wykonania dokonywana jest przez:

- równoważenie obciążenia (load balancing)
- redukcja czasu komunikacji:
  - uproszczony model czasowy
  - zmniejszanie liczby komunikatow
  - zmniejszanie całkowitego rozmiaru transferu
  - unikanie przepełnienia sieci komunikacyjnej
- minimalizowanie dodatkowych obliczeń