Programowanie współbieżne i rozproszone

WYKŁAD 2

Jan Kazimirski

MPI
1/2

Dlaczego klastry komputerowe?

- Wzrost mocy obliczeniowej jednego jest coraz trudniejszy do uzyskania.
- Koszt dodatkowej mocy obliczeniowej dla jednego komputera jest coraz wyższy.
- Jeden komputer – pojedynczy punkt wystąpienia awarii (SPOF)

Dlaczego klastry komputerowe?

- **Wydajność** – Wzrost mocy obliczeniowej dzięki użyciu wielu węzłów
- **Koszt** – Użycie standardowych komponentów pozwala obniżyć koszty instalacji
- **Elastyczność** – zwiększenie wydajności poprzez dodawanie nowych węzłów
- **Niezawodność** – awaria węzła nie ma wpływu na pracę całego klastra.

Beowulf

- Klaster oferujący dużą wydajność niewielkim kosztem
- Zbudowany zwykle ze standardowych komputerów PC połączonych siecią Ethernet
- Pracuje zwykle pod kontrolą systemu Linux
- Wykorzystuje biblioteki MPI, PVM.
- Często używany do obliczeń naukowych (uczelnie, instytuty)

Klaster typu SSI

- SSI - **Single System Image**
- Klaster widziany jako jedna maszyna. Topologia klastra ukryta przed użytkownikiem
- Nie wymaga specjalnie przygotowanego oprogramowania
- Migracja procesów pozwala na równoważenie obciążenia oraz dynamiczne podłączanie i odłączanie węzłów
- Przykład: **MOSIX**

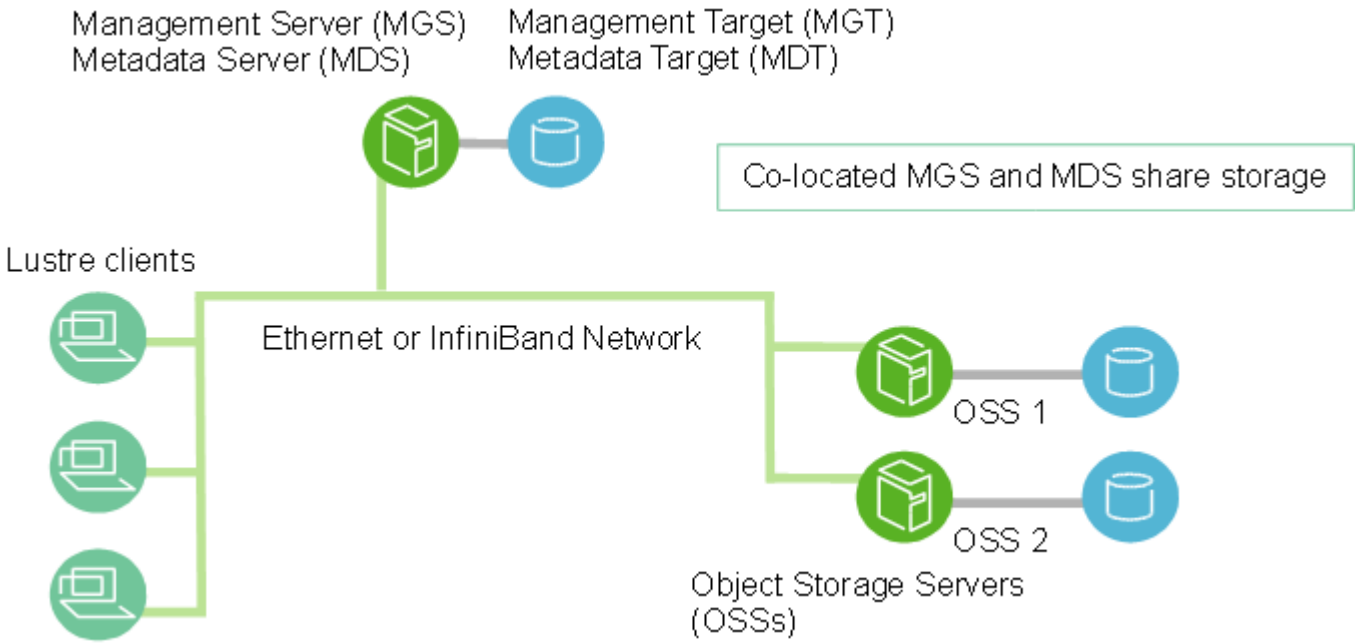
Rozproszony system plików

- Pliki fizycznie zapisywane na wielu serwerach plików (węzłach). Logicznie jeden spójny system plików.
- Zalety
  - Wydajność – podział i równoważenie obciążenia
  - Bezpieczeństwo – brak pojedynczego punktu awarii (SPOF)
  - Wygoda – łatwy dostęp do zasobów dla użytkowników systemu rozproszonego
- Przykład - **Lustre**

Lustre

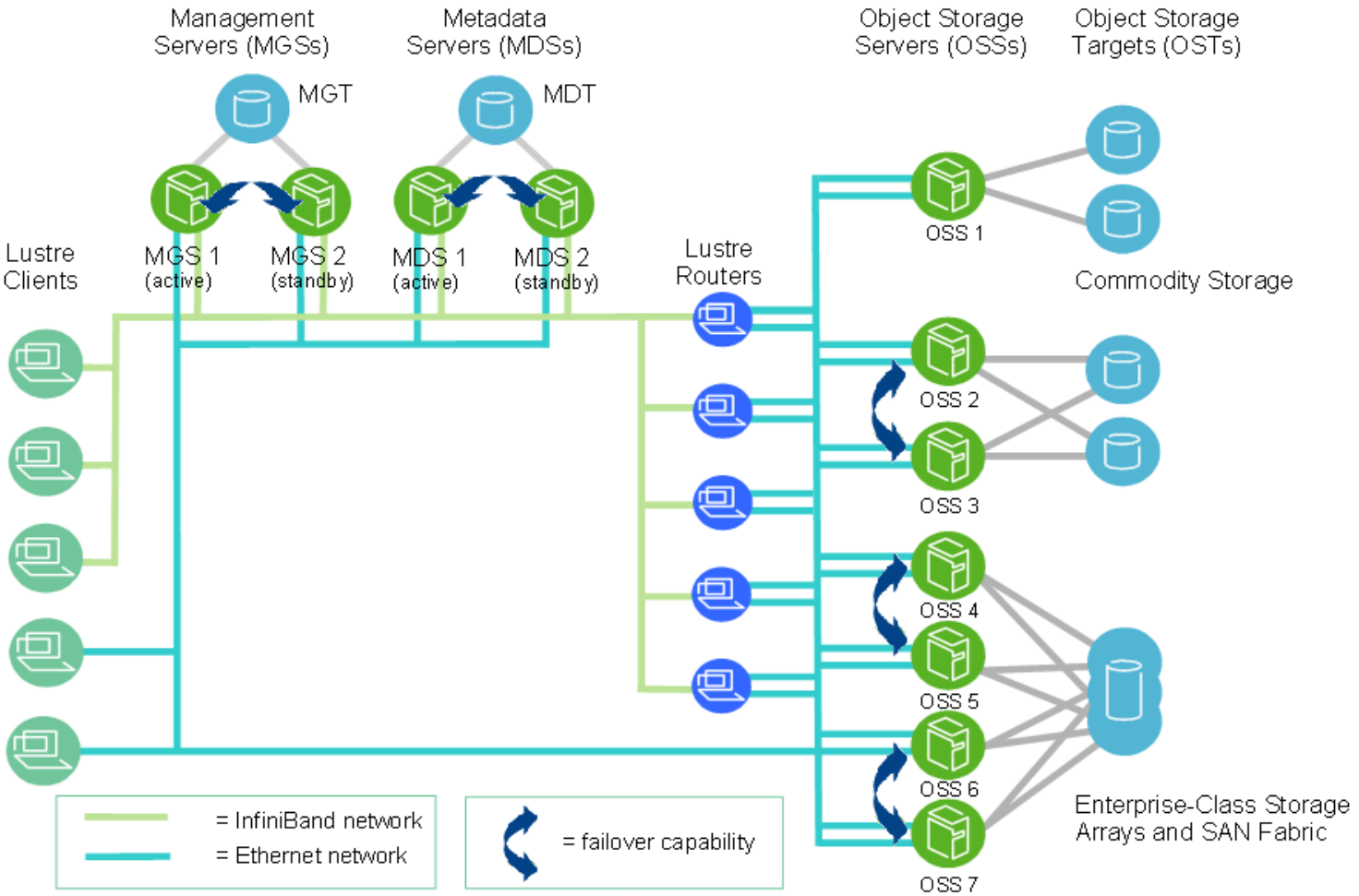
- Sieciowy system plików dla sieci o dużej liczbie węzłów, na licencji Open Source
- Pliki i katalogi traktowane są jako obiekty. Ich atrybuty (metadane) przechowywane są na serwerach metadanych (MDS/MDT).
- Dane przechowywane są na serwerach obiektów (OSS/OST)
- Komponent MGS (Management Server) przechowuje dane konfiguracyjne systemu plików.
- Maszyny korzystające z Lustre wykorzystują oprogramowanie klienta Lustre (komponenty MGC,MDC,OSC – dla poszczególnych OSS).

Lustre - architektura



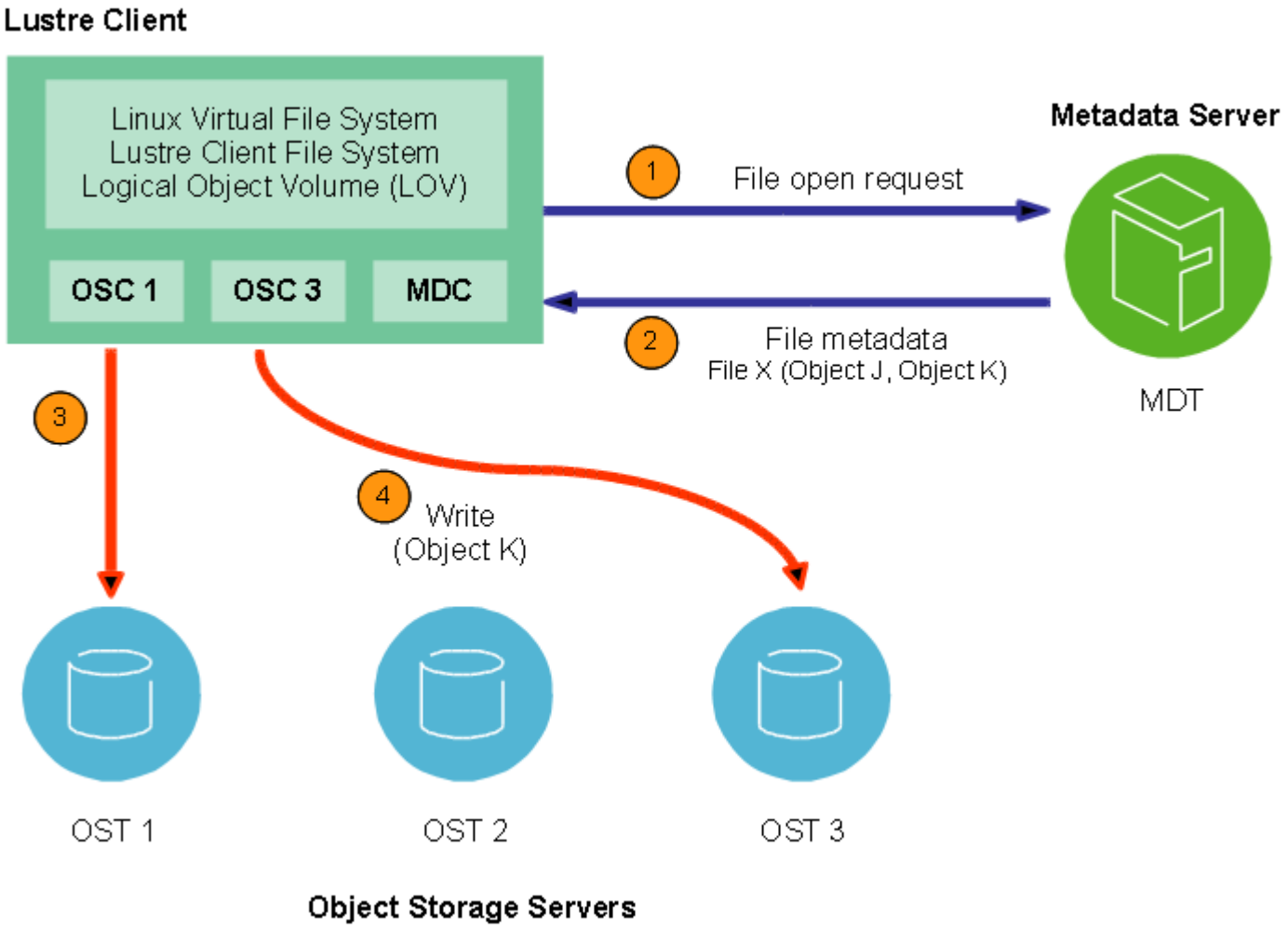
Źródło: http://wiki.lustre.org

Lustre – architektura c.d.



Źródło: http://wiki.lustre.org

Lustre – dostęp do danych



Źródło: http://wiki.lustre.org

Lustre – chatakterystyka

- Liczba klientów: 100-100 000
- Liczba OSS: 4-500 (do 4000 OST)
- Liczba MDS 1+1 (failover)
- Możliwość dostępu wielu użytkowników do tego samego pliku
- Dodawanie OSS w trakcie pracy systemu
- Podział danych (pliki, katalogi) na różne OSS („stripping”)
- Wbudowane wsparcie dla MPI
- System plików: ext4fs zmodyfikowany dla potrzeb Lustre

Programowanie klastra

- Programowanie klastra wymaga wykorzystania specjalnych bibliotek, np.. MPI, PVM, Linda lub innych
- Obliczenia na klastrach mogą wymagać specyficznych algorytmów.
- Wydajność klastra silnie zależy od problemu
  - Dla niektórych problemów prawie liniowy wzrost wydajności ze wzrostem liczby węzłów
  - Dla specyficznych problemów wydajność może być gorsza niż dla pojedynczej maszyny

Linda

- Model komunikacji między procesami w programach równoległych.
- Linda koncentruje się na komunikacji odseparowanej od obliczeń (ortogonalność)
- Komunikacja za pomocą abstrakcyjnej przestrzeni pamięci dzielonej (przestrzeń krotek)

Linda – przestrzeń krotek

- Przestrzeń krotek jest wspólną przestrzenią dla wszystkich procesów.
- Procesy mogą umieszczać i usuwać krotki z przestrzeni krotek
- Krotki „żywe” - są aktualnie wykonywane przez równoległe procesy
- Krotki danych – przechowują dane lub rezultaty działania krotek żywych.

Linda - operacje

- **out** – umieszczenie krotki w przestrzeni krotek
- **in** – wyszukanie i pobranie krotki z przestrzeni krotek
- **rd** – wyszukanie i odczytanie krotki z przestrzeni krotek
- **eval** – utworzenie „żywej” krotki, tzn. operacji która zostanie wykonana przez niezależny proces.

Linda – operacje c.d.

- Przestrzeń krotek jest pamięcią typu asocjacyjnego (brak adresów).
- Przykłady podstawowych operacji:
  - **out(„sum”,2,3)** – utworzenie krotki
  - **in(„sum”,?i,?j)** – odczytanie krotki, przypisanie wartości do zmiennych
  - **eval(„ab”,-6,abs(-6))** – utworzenie żywej krotki. Po wykonaniu operacji krotka uzyska formę („ab”,-6,6) i będzie mogła być odczyta przez inny proces

Linda - własności

- Wartości w krotkach zależą od implementacji, zwykle są to typy proste
- W założeniu Linda jest zintegrowana z sekwencyjnym językiem programowania (host language)
- Procesy komunikują się wyłącznie poprzez przestrzeń krotek.

Linda - implementacje

- Dostępne implementacje dla wielu języków programowania, m.in.:
- C : TCP-Linda, LinuxTuples
- C++ : CppLinda
- Java : JavaSpaces, TSpaces, LIME
- Lisp, Prolog, Python, Ruby ...

PVM – Parallel Virtual Machine

- Projekt rozpoczęty w 1989 r. przez grupę badawczą w Oak Ridge National Lab.
- W założeniu miał być rozproszonym systemem operacyjnym (maszyna wirtualna)
- Demon PVM (pvmd) kontroluje pracę aplikacji w środowisku PVM.
- PVM zakłada dynamiczną konfigurację klastra (dodawanie i usuwanie węzłów) – możliwość równoważenia obciążenia, migracji procesów, odporność na awarie.

PVM - filozofia

- Zasoby klastra widoczne dla użytkownika jako maszyna wirtualna
- Struktura klastra ukryta przed użytkownikiem
- Możliwość przenoszenia na różne platformy

Jak pracuje PVM

- Po uruchomieniu PVM identyfikuje swoją maszynę wirtualną
- Na każdym komputerze uruchamiany jest demon pvmd
- PVM dostarcza funkcje pozwalające na komunikację między węzłami
- Cała komunikacja między węzłami realizowana jest poprzez demony pvmd

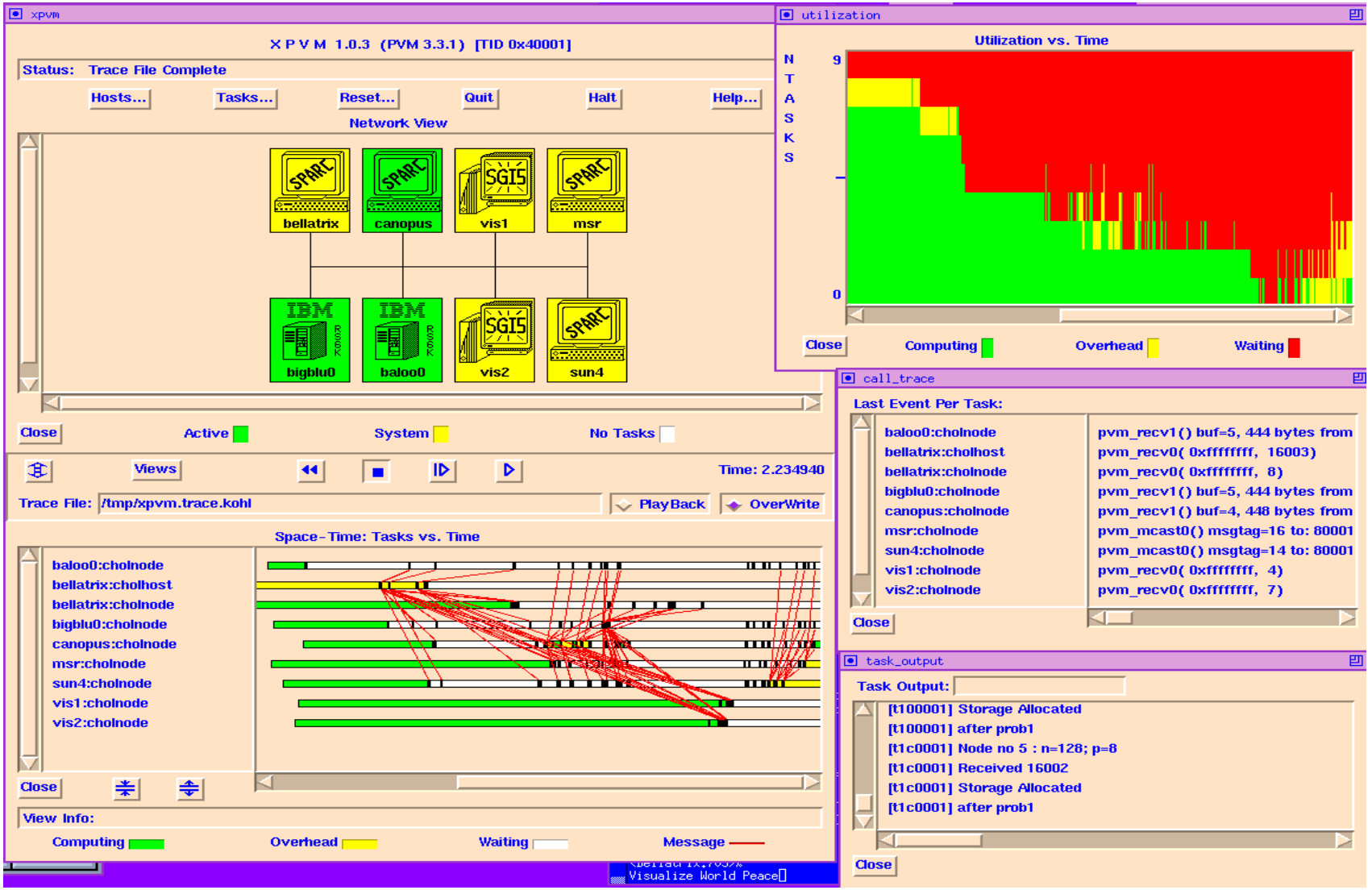
Demon pvmd

- Zapewnia punkt komunikacyjny węzła
- Zapewnia autentykację – poszczególne zadania PVM nie przeszkadzają sobie wzajemnie
- Uruchamia programy na węźle
- Monitoruje węzeł na wypadek awarii
- Buforuje wysyłane komunikaty

Konsola PVM

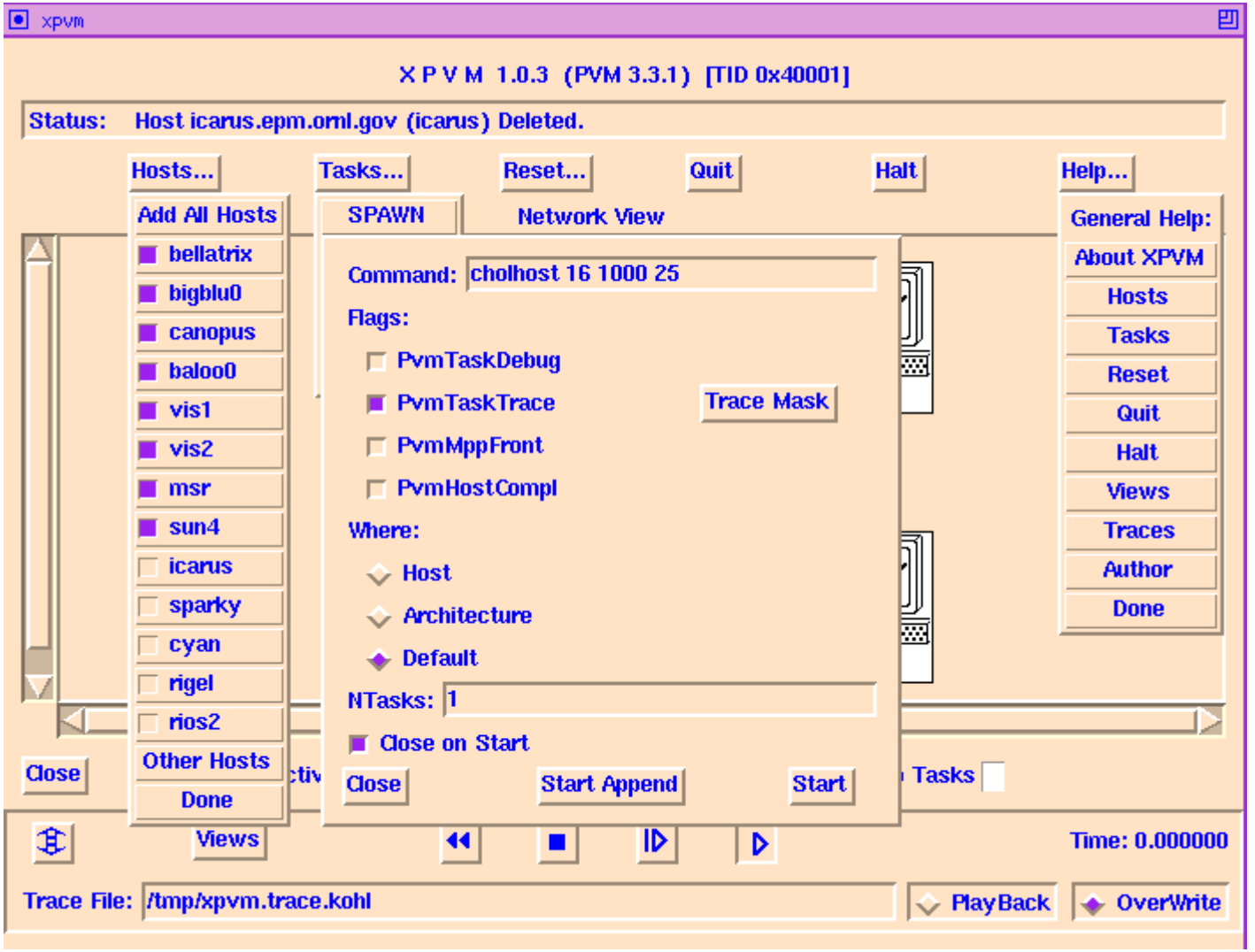
- Umożliwia sterowanie maszyną wirtualną
- Uruchamia lokalnego demona PVM
- Dodaje i usuwa węzły maszyny wirtualnej
- Uruchamia i monitoruje aplikacje działające na maszynie wirtualnej
- Zatrzymuje maszynę wirtualną

XPVM – graficzna konsola PVM



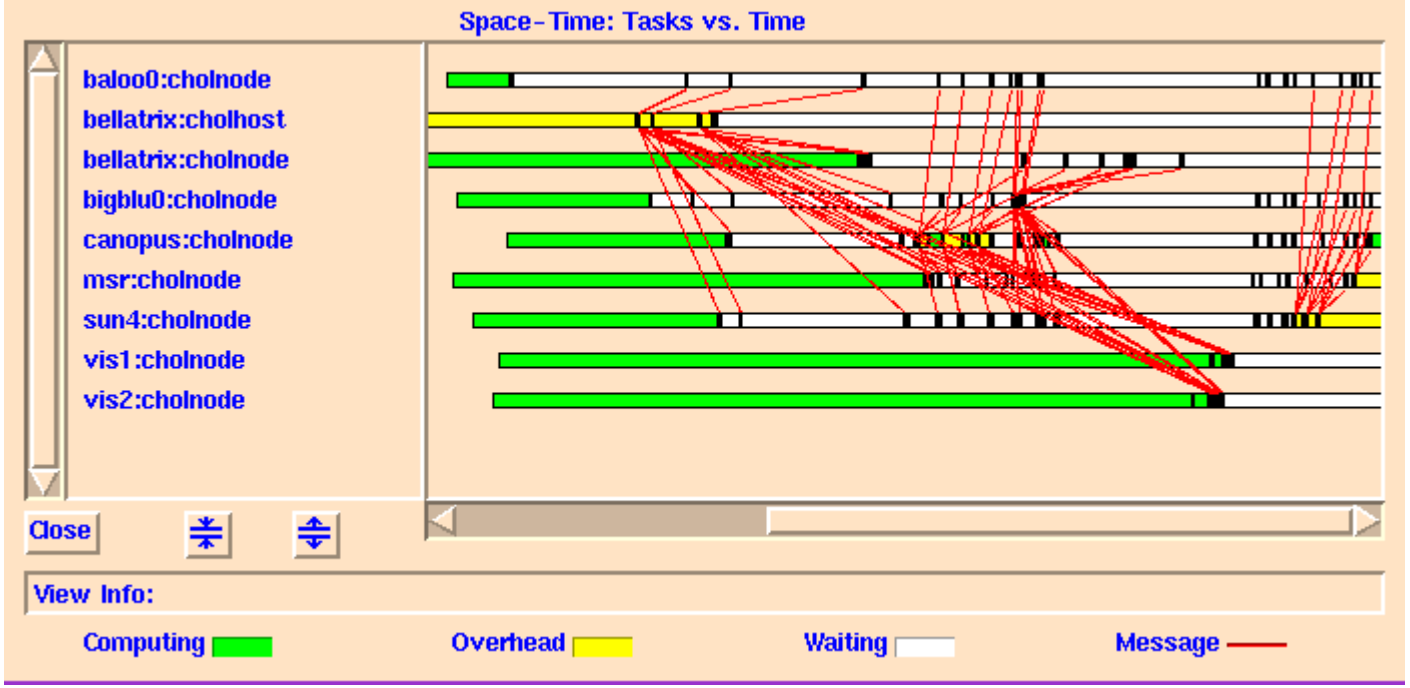
Źródło: http://www.netlib.org/utk/icl/xpvm/xpvm.html

XPVM c.d.



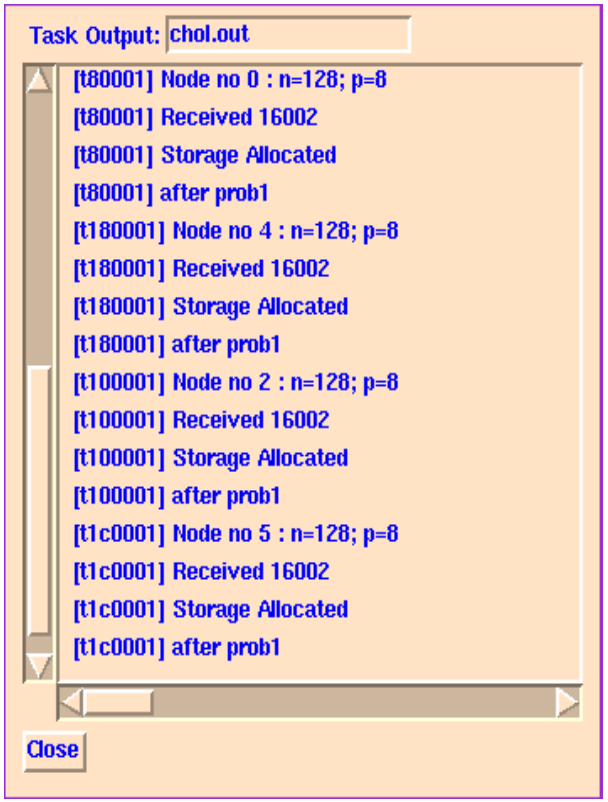
Źródło: http://www.netlib.org/utk/icl/xpvm/xpvm.html

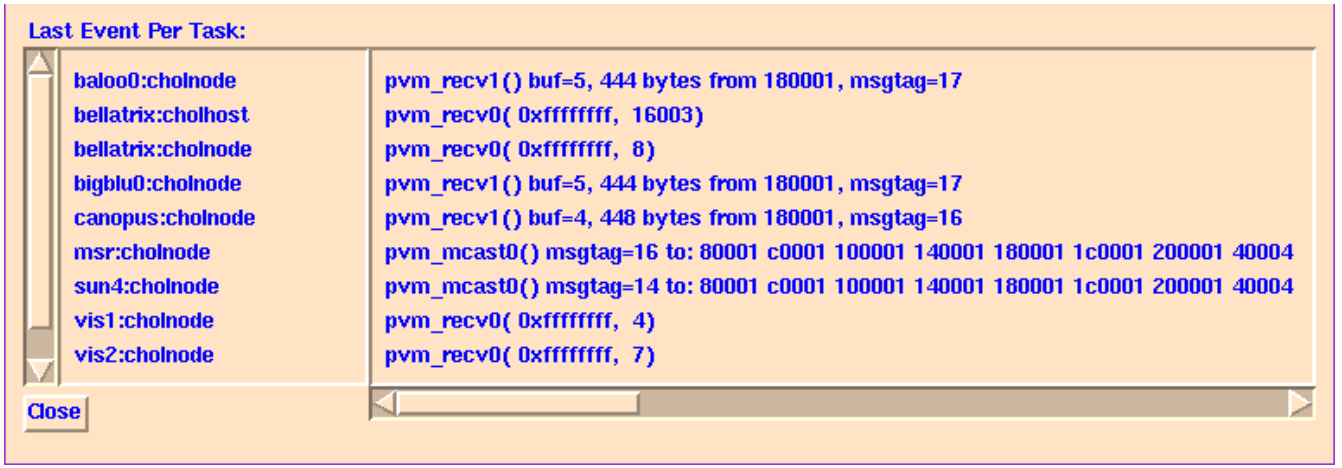
XPVM c.d.



Źródło: http://www.netlib.org/utk/icl/xpvm/xpvm.html

XPVM c.d.





Źródło: http://www.netlib.org/utk/icl/xpvm/xpvm.html

Model MPI

- **MPI – Message Passing Interface** – protokół komunikacyjny stosowany w programach równoległych
- Podstawowe założenia:
  - Program składa się ze zbioru procesów, każdy proces ma własny obszar pamięci.
  - Procesy komunikują się wymieniając komunikaty
  - Model MIMD, procesy mogą wykonywać różne operacje na innych danych

MPI c.d.

- **MPI-1** – Starsza, ale ciągle stosowana wersja protokołu.
  - Statyczna struktura procesów
  - Prosty model komunikacji
- **MPI-2** – Nowa wersja protokołu.
  - Możliwość dynamicznej konfiguracji (tworzenie, dołączanie procesów)
  - Rozszerzenia związane z lepszą obsługą pamięci współdzielonej, I/O ...

MPI c.d.

- MPI jest **standardem**. Właściwa implementacja zależy od platformy i języka programowania
- Dostępne są implementacje MPI dla języków C, C++, Fortran, Python, Java i innych.
- Pomimo wprowadzenia standardu MPI-2 w dalszym ciągu szeroko wykorzystywany jest standard MPI-1.

MPI vs. PVM

- PVM rozwijany był początkowo przez niewielką grupę badawczą do eksperymentów. MPI jest standardem „przemysłowym”.
- PVM – dynamiczna kontrola zasobów, MPI – statyczne zasoby (MPI 1.0).
- MPI – niewielka odporność na błędy. PVM – możliwość obsługi sytuacji awaryjnych.

Główne elementy MPI

- Funkcje do inicjowania, obsługi i zakończenia komunikacji między procesami.
- Funkcje do przesyłania komunikatów między parami procesów (point-to-point).
- Funkcje do przesyłania komunikatów między grupami procesów
- Funkcje do tworzenia specyficznych typów danych.

Pierwszy program

```
#include<iostream>
#include<stdio.h>
#include<mpi.h>
using namespace std;
int main(int argc,char* argv[]) {
  MPI_Init(&argc,&argv);
  cout << "Hello" << endl;
  MPI_Finalize();
  return 0;
};
```

Pierwszy program

```
#include<iostream>
#include<stdio.h>
#include<mpi.h>
using namespace std;
int main(int argc,char* argv[]) {
  MPI_Init(&argc,&argv);
  cout << "Hello" << endl;
  MPI_Finalize();
  return 0;
};
```

Plik nagłówkowy

Pierwszy program

```
#include<iostream>
#include<stdio.h>
#include<mpi.h>
using namespace std;
int main(int argc,char* argv[]) {
  MPI_Init(&argc,&argv);
   cout << "Hello" << endl;
   MPI_Finalize();
   return 0;
};
```

Inicjalizuje równoległe wykonanie

Pierwszy program

```
#include<iostream>
#include<stdio.h>
#include<mpi.h>
using namespace std;
int main(int argc,char* argv[]) {
  MPI_Init(&argc,&argv);
  cout << "Hello" << endl;
  MPI_Finalize();
  return 0;
};
```

Kończy równoległe wykonanie

Kompilacja programu z MPI

- Różne implementacje mogą się różnić sposobem kompilacji i uruchomiania programu z MPI
- Kompilacja dla Ubuntu/OpenMPI:

mpiCC ex01.C -Wall -o ex01

ex01.C jest nazwą pliku z kodem źródłowym, pozostałe opcje są zgodne z kompilatorem GNU C++

Uruchomienie programu z MPI

- Aby uruchomić program na klastrze należy przygotować plik z konfiguracją węzłów.
- W przypadku jednej maszyny wystarczy wykonać polecenie:

mpirun -np 4 ex01

ex01 jest nazwą skompilowanego programu, opcja -np określa liczbę uruchamianych kopii.

Pierwszy program „w akcji”

```
$ mpiCC ex01.C -Wall -o ex01
$ mpirun -np 4 ex01
Hello
Hello
Hello
Hello
```

Określanie otoczenia procesu

- Wykonanie N razy identycznych działań nie jest zbyt praktyczne.
- Aby stworzyć użyteczny program należy uzyskać dwie informacje:
  - Ile jest procesów?
  - Którym z nich jest aktualnie działająca kopia?
- Każda kopia programu MPI ma swój unikalny numer.

Otoczenie procesu

- MPI definiuje dwie funkcje pozwalające określić liczbę procesów i aktualny identyfikator
- **MPI_Comm_size** – podaje liczbę procesów
- **MPI_Comm_rank** – podaje identyfikator danego procesu (numer w zakresie od 0 do liczby procesów – 1)

Otoczenie procesu c.d.

```
MPI_Init(&argc,&argv);
int np,id;
MPI_Comm_rank(MPI_COMM_WORLD, &id);
MPI_Comm_size(MPI_COMM_WORLD, &np);
cout << "Jestem numer " << id << " z " << np
     << " procesów" << endl;
MPI_Finalize();
```

(*) UWAGA! Listing nie jest kompletnym programem.

Otoczenie procesu c.d.

- Kompilacja i wykonanie programu.

$ ***mpiCC ex02.C -Wall -o ex02***

$ ***mpirun -np 4 ex02***

Jestem numer 0 z 4 procesów

Jestem numer 1 z 4 procesów

Jestem numer 3 z 4 procesów

Jestem numer 2 z 4 procesów

Model master-slave

- Korzystając z funkcji identyfikujących otoczenie działającego procesu możemy zaimplementować prosty model komunikacji master-slave
- Struktura programu:
  - Węzeł nadrzędny (master) – nadzoruje pracę programu, przydziela zadania itd..
  - Węzły podrzędne (slaves) – wykonują konkretne zadania.

Master - Slave

```
MPI_Init(&argc,&argv);
int np,id;
MPI_Comm_rank(MPI_COMM_WORLD, &id);
MPI_Comm_size(MPI_COMM_WORLD, &np);

if(id==0) cout << "Master" << endl;
else cout << "Slave" << endl;

MPI_Finalize();
```

(*) UWAGA! Listing nie jest kompletnym programem.

Master – Slave c.d.

$ **mpiCC ex03.C -Wall -o ex03**

$ **mpirun -np 4 ex03**

Slave

Slave

Slave

Master

$ **mpirun -np 4 ex03**

Slave

Slave

Master

Slave

Master – Slave c.d.

- Wybór węzła 0 na węzeł nadrzędny pozwala gwarantuje że niezależnie od liczby węzłów zawsze jeden z nich będzie typu master.
- Poprzedni slajd ilustruje niezwykle istotną cechę działania programów rozproszonych:

  KOLEJNOŚĆ WYKONYWANIA POSZCZEGÓLNYCH PROCESÓW JEST PRZYPADKOWA

  Oznacza to, że projektując program w środowisku rozproszonym nie można przyjmować **żadnych** założeń co do kolejności wykonania poszczególnych procesów

Komunikacja między węzłami

- Najprostszym modelem komunikacji w MPI jest blokująca komunikacja typu point-to-point
- Jeden węzeł wysyła komunikat, inny węzeł odbiera ten komunikat.
- Węzeł wysyłający po zainicjowaniu wysyłania czeka na odebranie komunikatu – jest w tym czasie zablokowany
- Węzeł odbierający po zainicjowaniu odbierania oczekuje na komunikat i jest zablokowany do czasu nadejścia komunikatu

MPI_Send / MPI_Recv

- Do realizacji synchronicznej komunikacji point-to-point służą funkcje: **MPI_Send** i **MPI_Recv**
- Ze względu na specyfikę MPI sposób ich wywołania jest dosyć skomplikowany. Ograniczymy się teraz do omówienia tylko podstawowych elementów

MPI_Send / MPI_Recv

```
int MPI_Send(      void* message,
                   int count,
                   MPI_Datatype datatype,
                   int dest,
                   int tag,
                   MPI_Comm comm)

int MPI_Recv(      void* message,
                   int count,
                   MPI_Datatype datatype,
                   int source,
                   int tag,
                   MPI_Comm comm,
                   MPI_Status* status)
```

MPI_Send / MPI_Recv

Definicja komunikatu

void* message - treść komunikatu

int count - liczba zmiennych

MPI_Datatype datatype - typ zmiennych

MPI_Send / MPI_Recv

Predefiniowane typy danych

MPI_CHAR	**MPI_SHORT**
MPI_INT	**MPI_LONG**
MPI_UNSIGNED_CHAR	**MPI_UNSIGNED_SHORT**
MPI_UNSIGNED	**MPI_UNSIGNED_LONG**
MPI_FLOAT	**MPI_DOUBLE**
MPI_LONG_DOUBLE	

MPI_Send / MPI_Recv

Nadawca i odbiorca

int dest - numer odbiorcy (przy nadawaniu)

int source - numer nadawcy (przy odbieraniu)

MPI_ANY_SOURCE - maska pasująca do dowolnego numeru nadawcy

MPI_Send / MPI_Recv

Identyfikator komunikatu określa rodzaj komunikatu (np. dane, wyniki, statystyki, diagnostyka) – pozwala uzgodnić rodzaj komunikatu między nadawcą i odbiorcą.

int tag	-	identyfikator komunikatu
MPI_ANY_TAG	-	maska pasująca do dowolnego identyfikatora komunikatu (tylko odbiór)

MPI_Send / MPI_Recv

Przestrzeń komunikatów pozwala odseparować niezależne rodzaje komunikacji (np. program równoległy korzysta z biblioteki, która też jest równoległa).

MPI_Comm comm - komunikator (przestrzeń komunikatów).

MPI_COMM_WORLD - predefiniowana przestrzeń obejmująca wszystkie procesy

MPI_Send / MPI_Recv

Status operacji

MPI_Status* stat - informacje o odebranym komunikacie.

MPI_Status.MPI_SOURCE - nadawca komunikatu

MPI_Status.MPI_TAG - identyfikator komunikatu

Przykład MPI_Send/MPI_Recv 1/2

```
#include<iostream>
#include<stdio.h>
#include<mpi.h>

using namespace std;

int main(int argc,char* argv[]) {

  int np,id;
  int data;
  MPI_Status stat;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &id);
  MPI_Comm_size(MPI_COMM_WORLD, &np);
```

Przykład MPI_Send/MPI_Recv 1/2

```
  if(id==0) {
    // Master
    for(int i=1;i<np;i++) {
      MPI_Recv(&data,1,MPI_INT,MPI_ANY_SOURCE,1,
               MPI_COMM_WORLD,&stat);
      cout << "Od węzła " << stat.MPI_SOURCE
           << " otrzymałem " << data << endl;
    };
  } else {
    // Slave
    MPI_Send(&id,1,MPI_INT,0,1,MPI_COMM_WORLD);
  };

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

Przykład MPI_Send/MPI_Recv

$ **mpiCC ex04.C -Wall -o ex04**

$ **mpirun -np 6 ex04**

Od węzła 1 otrzymałem 1

Od węzła 2 otrzymałem 2

Od węzła 3 otrzymałem 3

Od węzła 4 otrzymałem 4

Od węzła 5 otrzymałem 5

Krótkie podsumowanie

- MPI posiada ponad 100 różnych funkcji. Dotychczas poznaliśmy 6 funkcji
  - MPI_Init
  - MPI_Finalize
  - MPI_Comm_size
  - MPI_Comm_rank
  - MPI_Send
  - MPI_Recv
- Funkcje te wystarczą aby napisać kompletny program działający w środowisku rozproszonym.

Komunikacja zbiorcza - MPI_Bcast

```
int MPI_Bcast(    void* message,
                  int count,
                  MPI_Datatype datatype,
                  int root,
                  MPI_Comm comm)
```

Wysyłanie komunikatu z procesu root do wszystkich pozostałych w danej przestrzeni komunikatów.

Musi być wywołana przez wszystkie procesy uczestniczące w komunikacji.

MPI_Bcast – przykład 1/2

```
#include<iostream>
#include<stdio.h>
#include<mpi.h>
using namespace std;

int main(int argc,char* argv[]) {

  int np,id;
  int l = 0;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &id);
  MPI_Comm_size(MPI_COMM_WORLD, &np);
```

MPI_Bcast – przykład 2/2

```
  if(id==0) {
    cout << "Podaj liczbe ";
    cin >> l;
  };

  MPI_Bcast(&l,1,MPI_INT,0,MPI_COMM_WORLD);

  if(id!=0) {
    cout << "Proces " << id << " liczba: " << l << endl;
  };

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

MPI_Bcast – wykonanie

$ **mpiCC ex05.C -Wall -o ex05**

$ **mpirun -np 6 ex05**

Podaj liczbe *16*

Proces 1 liczba: 16

Proces 2 liczba: 16

Proces 5 liczba: 16

Proces 4 liczba: 16

Proces 3 liczba: 16

MPI_Reduce

```
int MPI_Reduce(     void* operand,
                    void* result,
                    int count,
                    MPI_Datatype datatype,
                    MPI_Op op,
                    int root,
                    MPI_Comm comm)
```

Łączy operandy ze wszystkich procesów za pomocą operacji op i umieszcza rezultat w buforze *result* procesu root.

Musi być wywołana przez wszystkie procesy uczestniczące w komunikacji.

MPI_Reduce – predefiniowane operacje

MPI_MAX	-	maksimum
MPI_MIN	-	minimum
MPI_SUM	-	suma
MPI_PROD	-	iloczyn
MPI_LAND	-	logiczne AND
MPI_BAND	-	bitowe AND
MPI_LOR	-	logiczne OR
MPI_BOR	-	bitowe OR
MPI_LXOR	-	logiczne XOR
MPI_BXOR	-	bitowe XOR
MPI_MAXLOC	-	maksimum i jego położenie
MPI_MINLOC	-	minimum i jego położenie

Użytkownik może definiować własne operacje.

MPI_Reduce – przykład 1/2

```
#include<iostream>
#include<stdio.h>
#include<mpi.h>
using namespace std;

int main(int argc,char* argv[]) {

  int np,id;
  int n,sum;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &id);
  MPI_Comm_size(MPI_COMM_WORLD, &np);
```

MPI_Reduce – przykład 2/2

```
  n=10;
  cout << "Proces " << id << " n=" << n << endl;

  MPI_Reduce(&n,&sum,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);

  if(id==0) {
    cout << "Suma: " << sum << endl;
  };

  MPI_Finalize();

  return 0;
};
```

MPI_Reduce – wykonanie

$ **mpiCC ex06.C -Wall -o ex06**

$ **mpirun -np 6 ex06**

Proces 0 n=10

Proces 2 n=10

Proces 1 n=10

Proces 3 n=10

Proces 4 n=10

Proces 5 n=10

Suma: 60

MPI_Scatter

Wysyła dane z jednego procesu do pozostałych (rozdzielanie danych).

```
int MPI_Scatter(    void *sbuf,
                    int scount,
                    MPI_Datatype sdtype,
                    void *rbuf,
                    int rcount,
                    MPI_Datatype rdtype,
                    int root,
                    MPI_Comm comm)
```

MPI_Scatter c.d.

WEJŚCIE

sbuf	-	adres bufora do rozesłania (istotne tylko dla procesu root).
scount	-	liczba elementów do wysłania do każdego procesu (istotne tylko dla procesu root).
sdtype	-	typ danych do wysłania (istotne tylko dla Procesu root).
rcount	-	liczba odbieranych elementów.
rdtype	-	typ odbieranych elementów.
root	-	Id rozsyłającego procesu.
comm	-	przestrzeń komunikatów.

WYJŚCIE

rbuf	-	adres bufora dla odbieranych elementów

MPI_Scatter – przykład 1/2

```
#include<iostream>
#include<stdio.h>
#include<mpi.h>

using namespace std;

int main(int argc,char* argv[]) {

  int np,id;
  int nlist[10]={1,2,3,4,5,6,7,8,9,10};
  int n;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &id);
  MPI_Comm_size(MPI_COMM_WORLD, &np);
```

MPI_Scatter – przykład 2/2

```
  MPI_Scatter(nlist,1,MPI_INT,&n,1,MPI_INT,0,MPI_COMM_WORLD);

  cout << "Proces " << id << "  dostal " << n << endl;


  MPI_Finalize();

  return 0;
};
```

MPI_Scatter – uruchomienie

$ **mpiCC ex07.C -Wall -o ex07**

$ **mpirun -np 6 ex07**

Proces 0 dostal 1

Proces 2 dostal 3

Proces 4 dostal 5

Proces 1 dostal 2

Proces 3 dostal 4

Proces 5 dostal 6

MPI_Gather

Zbiera dane z grupy procesów.

```
int MPI_Gather(      void *sbuf,
                     int scount,
                     MPI_Datatype sdtype,
                     void *rbuf,
                     int rcount,
                     MPI_Datatype rdtype,
                     int root,
                     MPI_Comm comm)
```

MPI_Gather

WEJŚCIE

- **sbuf** - adres bufora z danymi do wysłania.
- **scount** - liczba wysyłanych elementów.
- **sdtype** - typ wysyłanych elementów.
- **rcount** - liczba elementów odbieranych od każdego procesu (istotne tylko dla procesu root).
- **rdtype** - typ odbieranych elementów (istotne tylko dla procesu root).
- **root** - id odbierającego procesu.
- **comm** - przestrzeń komunikatów

WYJŚCIE

- **rbuf** - adres bufora dla odbieranych danych (istotny tylko dla procesu root).

MPI_Gather – przykład 1/2

```
#include<iostream>
#include<stdio.h>
#include<mpi.h>

using namespace std;

int main(int argc,char* argv[]) {

  int np,id;
  int nlist[10]={0,0,0,0,0,0,0,0,0,0};
  int n;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &id);
  MPI_Comm_size(MPI_COMM_WORLD, &np);
```

MPI_Gather – przykład 2/2

```
  n=id;

  MPI_Gather(&n,1,MPI_INT,nlist,1,MPI_INT,0,MPI_COMM_WORLD);

  if(id==0) {
    for(int i=0;i<10;i++)
      cout << "Element " << i << "  wartosc: " << nlist[i] << endl;
  };

  MPI_Finalize();

  return 0;
};
```

MPI_Gather – uruchomienie

$ **mpiCC ex08.C -Wall -o ex08**
$ **mpirun -np 6 ex08**
Element 0 wartosc: 0
Element 1 wartosc: 1
Element 2 wartosc: 2
Element 3 wartosc: 3
Element 4 wartosc: 4
Element 5 wartosc: 5
Element 6 wartosc: 0
Element 7 wartosc: 0
Element 8 wartosc: 0
Element 9 wartosc: 0

Podsumowanie

- Poznaliśmy podstawowe modele komunikacji MPI
  - **Send/Recv** – Komunikacja point-to-point między węzłami
  - **Bcast/Reduce** – Możliwość rozesłania danych wyjściowych do węzłów i połączenia wyników obliczeń
  - **Scatter/Gather** – Podział danych wyjściowych między węzłami a następnie odebranie wyników obliczeń.

Praca domowa

Zaimplementować program wyznaczający liczbę PI metodą Monte Carlo wykorzystujący bibliotekę MPI.