Programowanie współbieżne i rozproszone

WYKŁAD 3

Jan Kazimirski

Literatura

1. Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, Jack Dongarra, **MPI—The Complete Reference Volume 1, The MPI Core**, Second Edition, MIT Press, 1999
2. George Em Karniadakis, Robert M. Kirby II, **Parallel Scientific Computing in C++ and MPI**, Cambridge University Press, 2003
3. William Gropp, Ewing Lusk, Rajeev Thakur, **Using MPI-2. Advanced Features of the Message-Passing Interface**, MIT Press, 1999
4. Marcin Jaźnicki, **Model programowania równoległego z wykorzystaniem biblioteki MPI**, praca inżynierska, Wydział Informatyki Wyższej Szkoły Menedżerskiej w Warszawie, 2006

MPI
2/2

Komunikacja blokująca

- Funkcja MPI_Recv działa w trybie blokującym
- Powrót z MPI_Recv nastąpi dopiero po odebraniu komunikatu
- Brak pasującego komunikatu spowoduje zawieszenie programu
- Sposób realizacji funkcji MPI_Recv gwarantuje odebranie komunikatu

MPI_Send, MPI_Recv

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

using namespace std;

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

  int id;
  int data;
  MPI_Status stat;

  MPI_Init(&argc,&argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &id);

  if(id==0) {
    MPI_Recv(&data,1,MPI_INT,MPI_ANY_SOURCE,1,MPI_COMM_WORLD,&stat);
    cout << "Otrzymałem\n";
  } else {
    MPI_Send(&id,1,MPI_INT,0,1,MPI_COMM_WORLD);
    cout << "Wysłałem\n";
  };

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

MPI_Recv

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

using namespace std;

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

  int id;
  int data;
  MPI_Status stat;

  MPI_Init(&argc,&argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &id);

  if(id==0) {
    MPI_Recv(&data,1,MPI_INT,MPI_ANY_SOURCE,1,MPI_COMM_WORLD,&stat);
    cout << "Otrzymałem\n";
  } else {
    [redacted]
  };

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

Program nie zakończy się

MPI_Send

- Działanie funkcji MPI_Send zależy od trybu
  - Tryb **buforowany** – powrót z funkcji następuje po skopiowaniu wysyłanych danych do wewnętrznego bufora MPI
  - Tryb **niebuforowany** – powrót z funkcji następuje dopiero po wysłaniu danych
- W trybie buforowanym powrót z funkcji MPI_Send nie gwarantuje wysłania komunikatu.

MPI_Send c.d.

- Wybór trybu komunikacji (buforowana lub nie) należy do MPI.
- Programista może narzucić tryb komunikacji zastępując funkcję MPI_Send funkcją:
  - **MPI_Ssend** – nie korzysta z buforowania
  - **MPI_Bsend** – wykorzysta bufor, lub zwróci błąd jeżeli bufor jest zbyt mały

MPI_Ssend

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

using namespace std;

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

  int id;
  int data;
  MPI_Status stat;

  MPI_Init(&argc,&argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &id);

  if(id==0) {
    [redacted]
  } else {
    MPI_Ssend(&id,1,MPI_INT,0,1,MPI_COMM_WORLD);
    cout << "Wysłałem\n";
  };

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

Program nie zakończy się

MPI_Bsend

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

using namespace std;

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

  int id;
  int data;
  MPI_Status stat;

  MPI_Init(&argc,&argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &id);

  if(id==0) {
    [redacted]
    cout << "Otrzymałem\n";
  } else {
    MPI_Bsend(&id,1,MPI_INT,0,1,MPI_COMM_WORLD);
    cout << "Wysłałem\n";
  };

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

Program zakończy się mimo tego, że komunikat nie został odebrany

Komunikacja asynchroniczna

- Funkcje MPI_Recv i MPI_Send (MPI_Bsend, MPI_Ssend) działają w trybie synchronicznym.
- Tryb synchroniczny zwiększa bezpieczeństwo komunikacji, ale powoduje dodatkowe narzuty czasowe i zwiększa ryzyko zakleszczeń
- Poza komunikacją synchroniczną MPI pozwala na realizację komunikacji w trybie asynchronicznym.
- Do przekazywania komunikatów w trybie asynchronicznym służą funkcje: **MPI_Isend** i **MPI_Irecv**.

MPI_Isend

```
int MPI_Isend(void *buf,
              int count,
              MPI_Datatype dtype,
              int dest,
              int tag,
              MPI_Comm comm,
              MPI_Request *request);
```

MPI_Irecv

```
int MPI_Irecv(void *buf,
              int count,
              MPI_Datatype dtype,
              int source,
              int tag,
              MPI_Comm comm,
              MPI_Request *request);
```

MPI_Isend/MPI_Irecv

- Parametry funkcji są analogiczne do ich wersji synchronicznych
- Parametr *request* identyfikuje operację
- Po wywołaniu funkcji sterowanie zwracane jest natychmiast do programu.
- **Powrót z funkcji nie oznacza wykonania operacji przesłania danych**

MPI_Wait / MPI_Test

- Komunikacja asynchroniczna pozwala programowi wykorzystać czas oczekiwania na zakończenie operacji przesłania.
- Czasami trzeba określić czy operacja się zakończyła, lub poczekać na jej zakończenie.
- **MPI_Wait** – blokuje wykonanie programu do czasu zakończenia podanej operacji
- **MPI_Test** – sprawdza czy dana operacja się zakończyła

MPI_Wait / MPI_Test

```
int MPI_Wait( MPI_Request *request,
              MPI_Status *status );
```

```
int MPI_Test( MPI_Request *request,
              int *flag,
              MPI_Status *status );
```

MPI Wait

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

using namespace std;

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

  int id;
  int data;
  MPI_Status stat;
  MPI_Request req;

  MPI_Init(&argc,&argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &id);

  if(id==0) {
    MPI_Irecv(&data,1,MPI_INT,MPI_ANY_SOURCE,1,MPI_COMM_WORLD,&req);
    MPI_Wait(&req,&stat);
    cout << "Otrzymałem\n";
  } else {
    MPI_Isend(&id,1,MPI_INT,0,1,MPI_COMM_WORLD,&req);
    MPI_Wait(&req,&stat);
    cout << "Wysłałem\n";
  };

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

MPI_Test

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

using namespace std;

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

  int id;
  int data;
  MPI_Status stat;
  MPI_Request req;
  int flag;

  MPI_Init(&argc,&argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &id);

  if(id==0) {
    MPI_Irecv(&data,1,MPI_INT,MPI_ANY_SOURCE,1,MPI_COMM_WORLD,&req);
    while(1) {
      MPI_Test(&req,&flag,&stat);
      if(flag) break;
      cout << "Czekam\n";
      sleep(1);
    }
    cout << "Otrzymałem\n";
  } else {
    sleep(5);
    MPI_Isend(&id,1,MPI_INT,0,1,MPI_COMM_WORLD,&req);
    MPI_Wait(&req,&stat);
    cout << "Wysłałem\n";
  };

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

```
Czekam
Czekam
Czekam
Czekam
Czekam
Wysłałem
Otrzymałem
```

MPI_Probe / MPI_Iprobe

- Pozwalają na sprawdzenie czy komunikat może być odebrany bez faktycznego jego odebrania
- **MPI_Probe** – Działa w trybie blokującym
- **MPI_Iprobe** – Działa w trybie nieblokującym
- Po sprawdzeniu czy komunikat oczekuje na odbiór (został wysłany) program może zdecydować się na jego odebranie lub anulowanie.

MPI_Probe / MPI_Iprobe

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

using namespace std;

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

  int id;
  int data;
  MPI_Status stat;
  MPI_Request req;
  int flag;

  MPI_Init(&argc,&argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &id);

  if(id==0) {
    while(1) {
      MPI_Iprobe(MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD,&flag,&stat);
      if(flag) {
        cout << "Komunikat czeka na odebranie: źródło:" << stat.MPI_SOURCE
             << " tag: " << stat.MPI_TAG << endl;
        MPI_Recv(&data,1,MPI_INT,MPI_ANY_SOURCE,1,MPI_COMM_WORLD,&stat);
        break;
      } else {
        cout << "Czekam\n";
        sleep(1);
      }
    }
    cout << "Otrzymałem\n";
  } else {
    sleep(5);
    MPI_Isend(&id,1,MPI_INT,0,1,MPI_COMM_WORLD,&req);
    MPI_Wait(&req,&stat);
    cout << "Wysłałem\n";
  };

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

```
Czekam
Czekam
Czekam
Czekam
Czekam
Wysłałem
Czekam
Komunikat czeka na odebranie: Źródło:1 tag: 1
Otrzymałem
```

Własne typy danych

- Poza predefiniowanymi typami danych, MPI pozwala na definiowanie własnych typów.
- Możliwości MPI tworzenia własnych typów są rozbudowane
- Typy złożone (struktury, tablice)
- Możliwość pakowania i rozpakowywania danych rozproszonych w pamięci.

Własne typy danych c.d.

```
01: #include<iostream>
02: #include<stdio.h>
03: #include<mpi.h>
04:
05: using namespace std;
06:
07: int main(int argc,char* argv[]) {
08:
09:   MPI_Init(&argc,&argv);
10:
11:   struct MyData {
12:     int i;
13:     float f[2];
14:   } data;
15:
16:   int dlen[2] = {1,2};
17:   MPI_Datatype dtyp[2] = {MPI_INT,MPI_FLOAT};
18:   MPI_Aint dloc[2];
19:   MPI_Aint base;
20:
21:   MPI_Address(&data,&base);
22:   MPI_Address(&data.i,&dloc[0]);
23:   dloc[0]-=base;
24:   MPI_Address(&data.f,&dloc[1]);
25:   dloc[1]-=base;
26:
```

Własne typy danych c.d.

```
27:   MPI_Datatype MPI_MyType;
28:
29:   MPI_Type_struct(2,dlen,dloc,dtyp,&MPI_MyType);
30:   MPI_Type_commit(&MPI_MyType);
31:
32:   int id;
33:   MPI_Status stat;
34:   MPI_Comm_rank(MPI_COMM_WORLD, &id);
35:
36:   if(id==0) {
37:     MPI_Recv(&data,1,MPI_MyType,MPI_ANY_SOURCE,1,MPI_COMM_WORLD,&stat);
38:     cout << "Otrzymałem: i=" << data.i << " f[0]=" << data.f[0]
39:          << " f[1]=" << data.f[1] << endl;
40:   } else {
41:     data.i = 1; data.f[0] = 2.34; data.f[1] = 5.67;
42:     MPI_Send(&data,1,MPI_MyType,0,1,MPI_COMM_WORLD);
43:     cout << "Wysłałem\n";
44:   };
45:
46:   MPI_Finalize();
47:   return 0;
48: };
```

Własne typy danych c.d.

- Zdefiniowanie klasycznej struktury C

```
11:   struct MyData {
12:     int i;
13:     float f[2];
14:   } data;
```

Własne typy danych c.d.

- Definicja struktury definiowanej zmiennej
  - *dlen* – długości bloków
  - *dtyp* – typy bloków
  - *dloc*, *base* – przesunięcia bloków

```
16:    int dlen[2] = {1,2};
17:    MPI_Datatype dtyp[2] = {MPI_INT,MPI_FLOAT};
18:    MPI_Aint dloc[2];
19:    MPI_Aint base;
```

Własne typy danych c.d.

- Wyznaczamy początek struktury (base) oraz początki bloków (dloc).
- Do określenia adresu używamy funkcji MPI_Address

```
21:   MPI_Address(&data,&base);
22:   MPI_Address(&data.i,&dloc[0]);
23:   dloc[0]-=base;
24:   MPI_Address(&data.f,&dloc[1]);
25:   dloc[1]-=base;
```

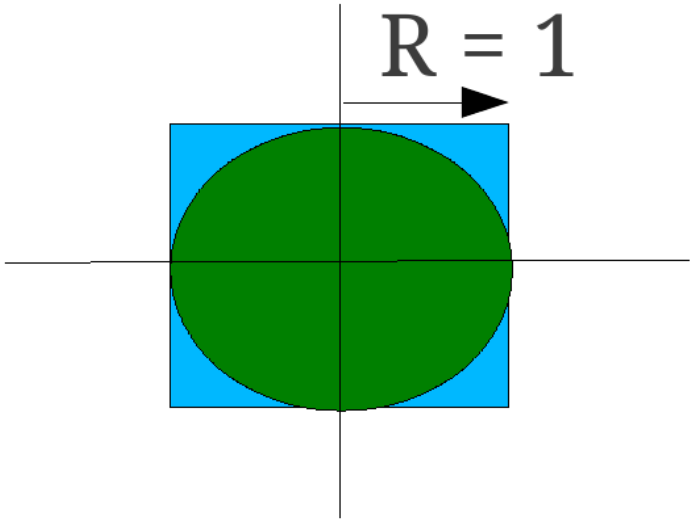
Własne typy danych c.d.

- Funkcja MPI_Type_struct – tworzy nowy typ danych na podstawie przekazanych parametrów
- Funkcja MPI_Type_commit – kończy definiowanie typu
- Utworzony typ może być wykorzystany do przesyłania komunikatów lub tworzenia nowych typów

```
27:   MPI_Datatype MPI_MyType;
28:
29:   MPI_Type_struct(2,dlen,dloc,dtyp,&MPI_MyType);
30:   MPI_Type_commit(&MPI_MyType);
```

Przykład projektu MPI „PI”

Projekt MC-PI - Algorytm



¼ $P_{kw} = R*R$

¼ $P_{kola} = ¼ * PI * R * R$

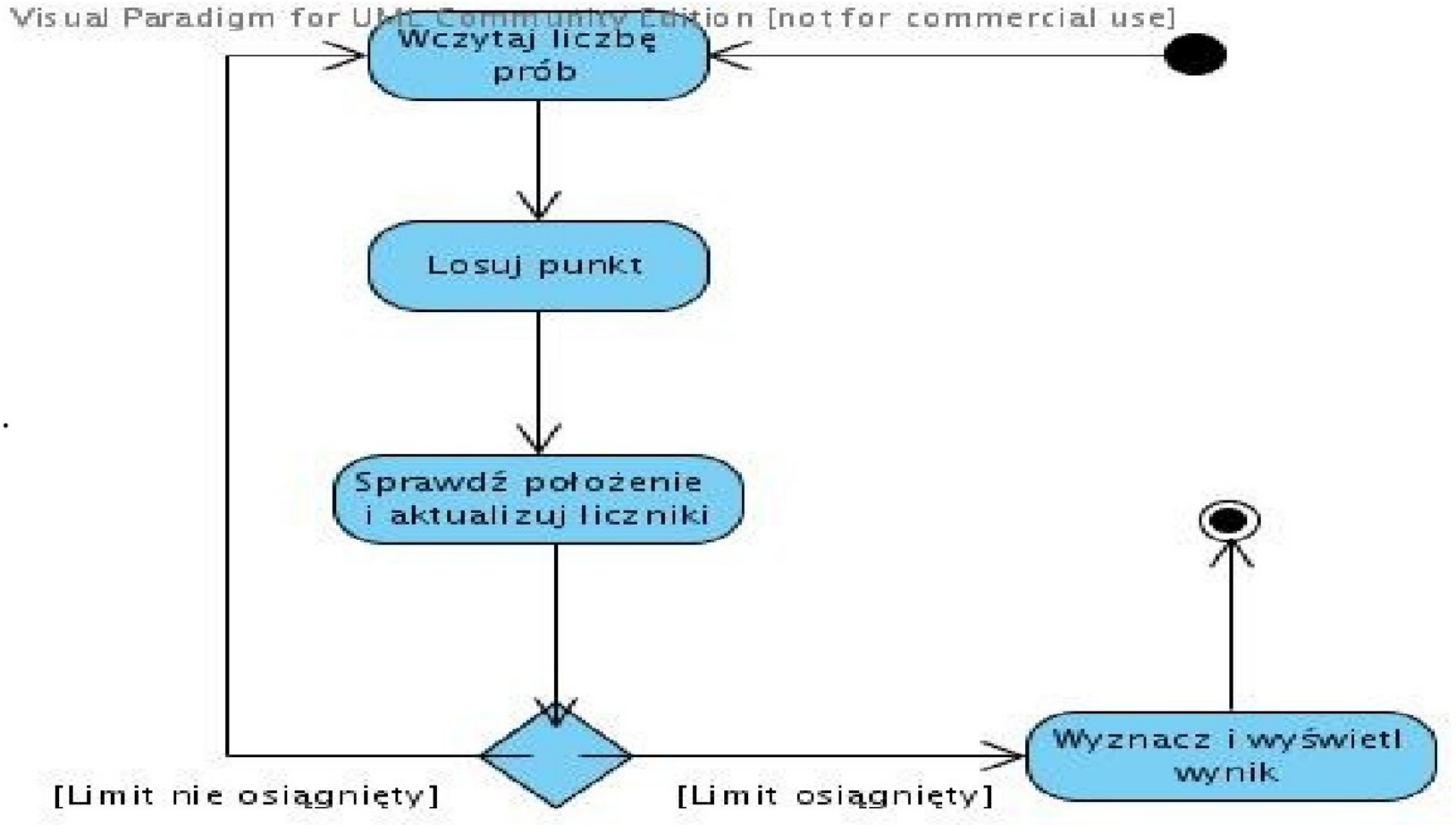
$P_{kola}/P_{kw} = ¼ * PI$

$PI = 4 * P_{kola}/P_{kw}$

Projekt MC-PI – implementacja szeregowa

- Implementacja szeregowa algorytmu jest bardzo prosta
  - Losujemy parę liczb x,y z przedziału 0.0 … 1.0
  - Sprawdzamy czy długość wektora (x,y) jest mniejsza od 1.0
  - Jeśli tak to punkt leży w wycinku koła – rejestrujemy „trafienie”
  - Rejestrujemy „rzut”
  - Czynność powtarzamy zadaną liczbę razy
  - Po zakończeniu wyliczamy Pi ze stosunku trafień do rzutów

Projekt MC-PI c.d.



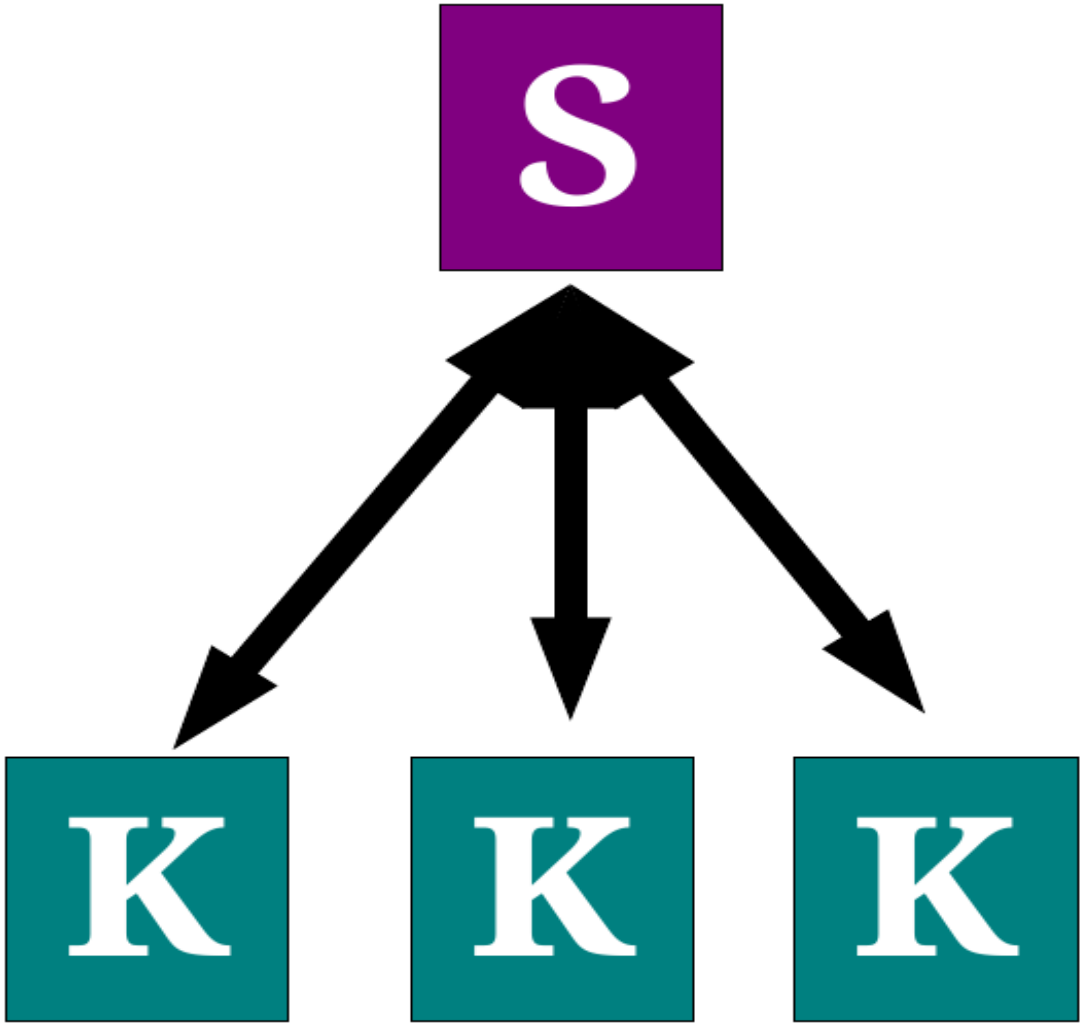
Implementacja równoległa

- Implementacja „**drobnoziarnista**” (fine-grain)
  - Niewielkie zadania (jedno losowanie)
  - Dobre równoważenie obciążenia
  - Duży narzut na przesyłanie danych
- Implementacja „**gruboziarnista**” (coarse-grain)
  - Duże zadania (wiele losowań)
  - Problem z równoważeniem obciążenia
  - Niewielki narzut na przesyłanie danych

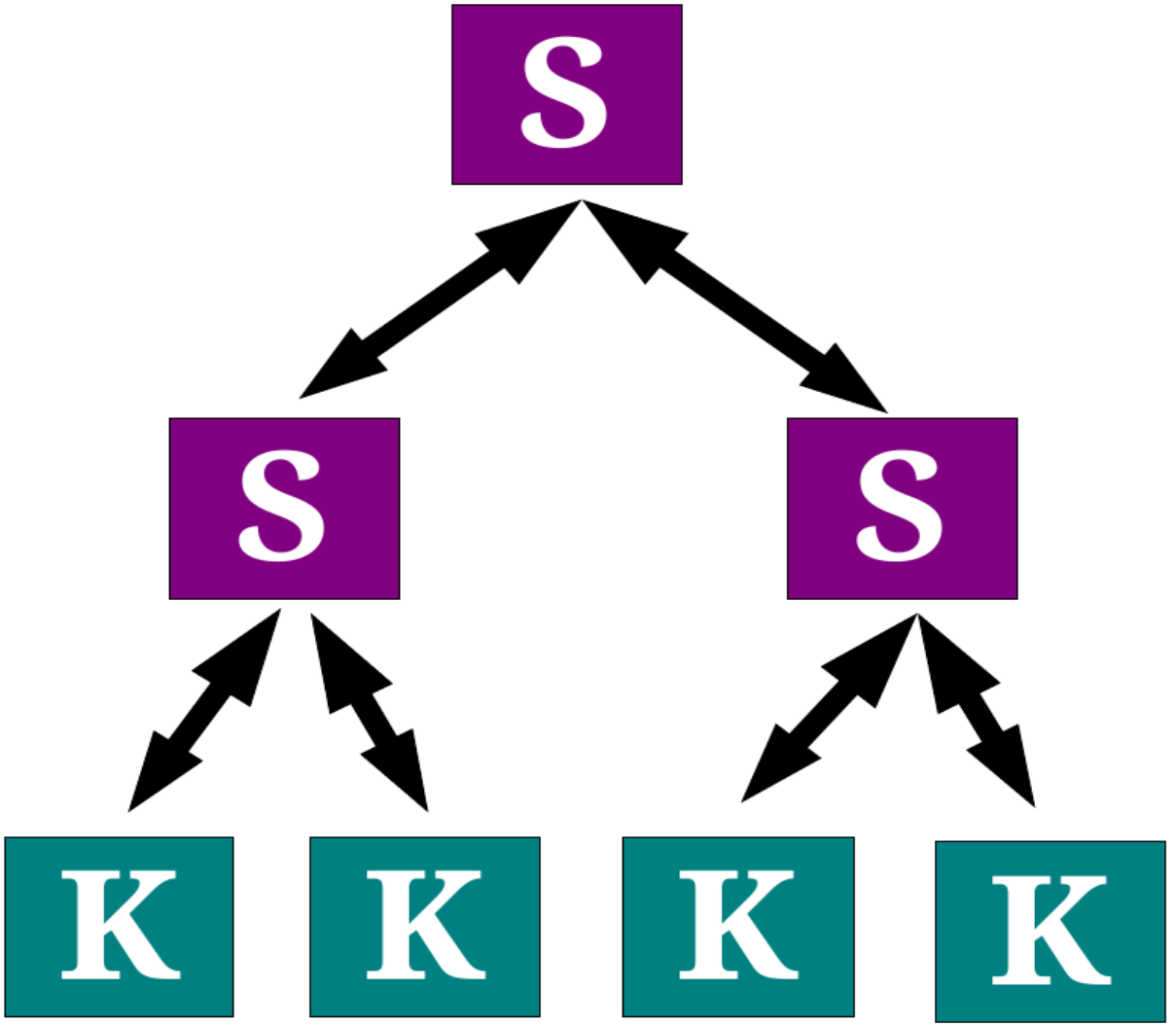
Topologia procesów

- Model **master-slave**
  - Prosta topologia – węzeł zarządzający, węzły wykonujące obliczenia
  - „Wąskie gardło” - komunikacja między węzłem master i węzłami slave
  - Ryzyko awarii – Single Point Of Failure (SPOF)
- Bardziej zaawansowane topologie mogą wykorzystywać kilka węzłów typu master, węzły pośredniczące, lub całkowitą równorzędność wezłów

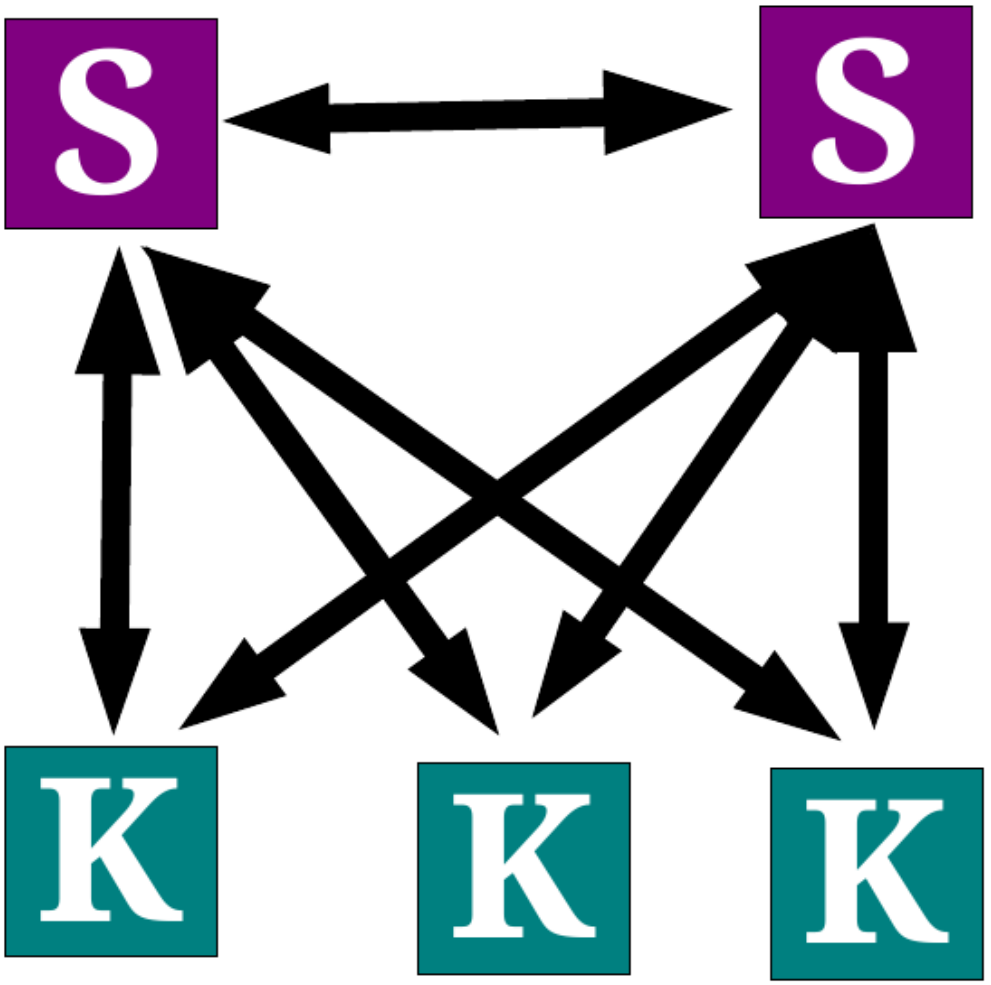
Topologia procesów c.d.



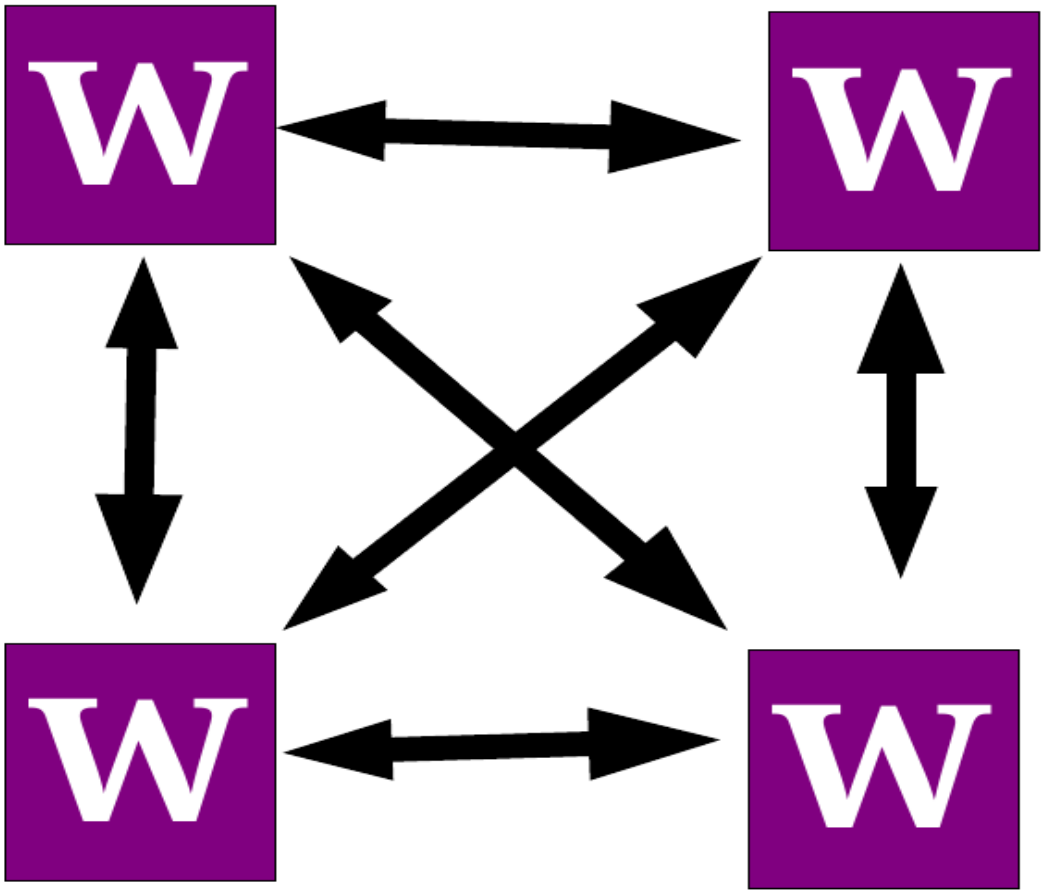
Topologia procesów c.d.



Topologia procesów c.d.



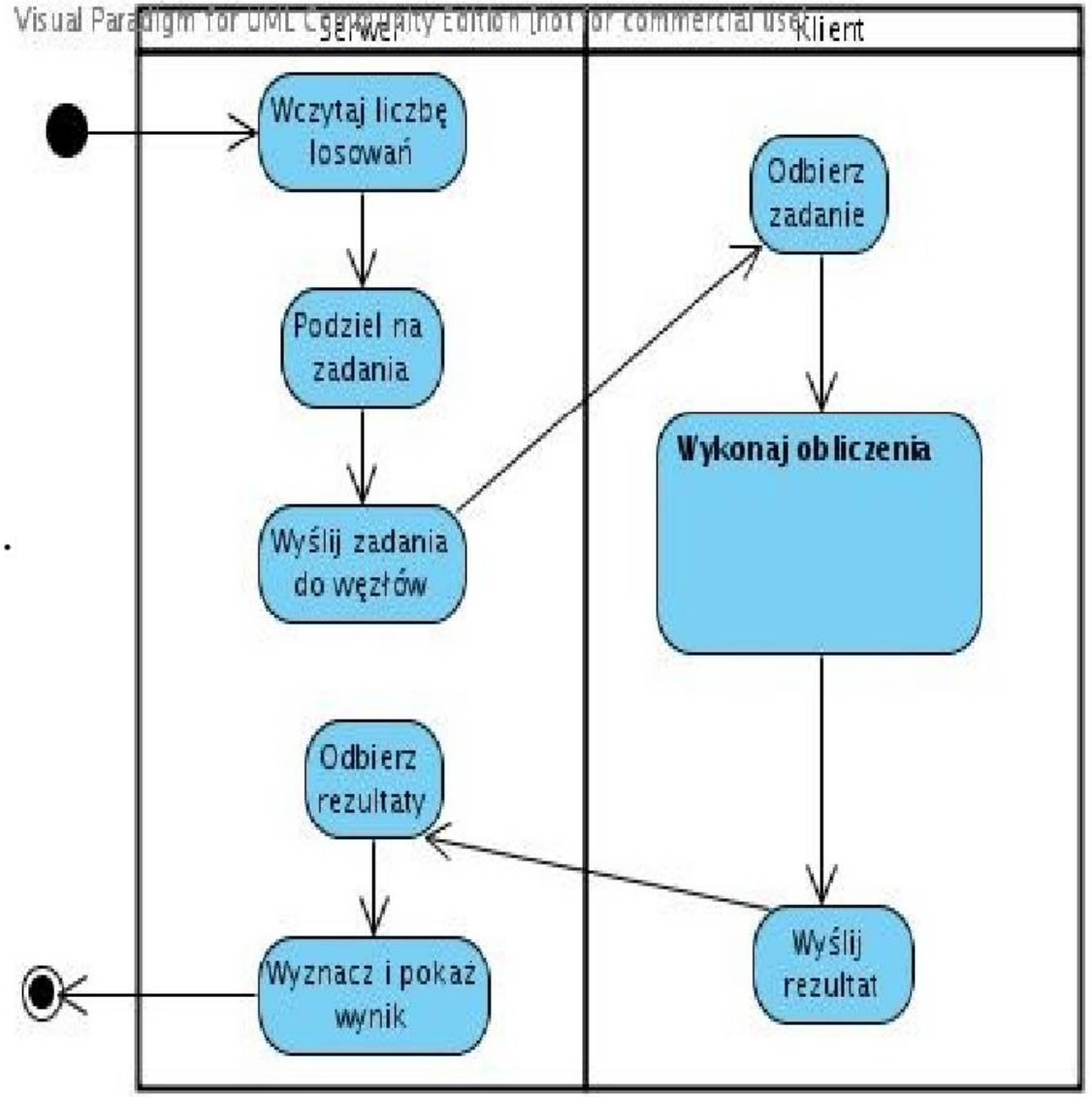
Topologia procesów c.d.



MC-PI implementacja c.d.

- Przykładowa implementacja równoległa:
  - Serwer wczytuje liczbę losowań
  - Serwer określa liczbę losowań każdego klienta
  - Serwer rozsyła zadania do klientów
  - Klient wykonuje obliczenia
  - Klient wysyła rezultat (liczba trafień) do serwera
  - Serwer wylicza i wyświetla ostateczny rezultat

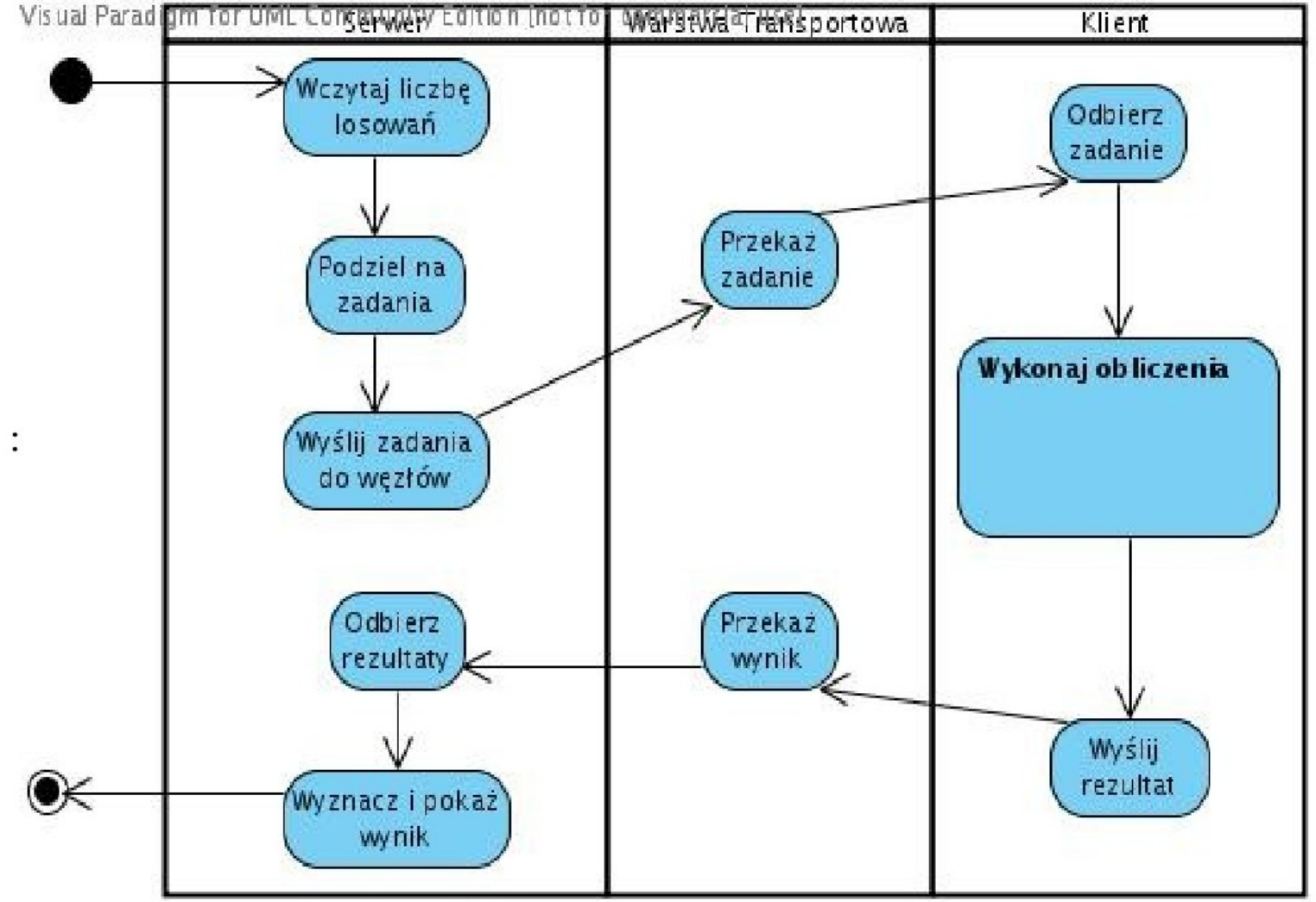
MC-PI architektura 2-warstwowa



MC-PI architektura 3-warstwowa

- Wydzielenie dodatkowej warstwy odpowiedzialnej za komunikację
  - Możliwość wykorzystania podejścia obiektowego
  - Separacja kodu „obliczeniowego” i „komunikacyjnego”
  - Łatwiejsze testowanie
  - Możliwość łatwej zmiany warstwy transportowej

MC-PI architektura 3-warstwowa c.d.



MC-PI architektura 3-warstwowa c.d.

- Serwer wczytuje liczbę prób
- Serwer dzieli zadanie na zadania dla klientów
- Serwer przekazuje zadania do warstwy transportowej (WT)
- WT przekazuje zadanie do klienta
- Klient odbiera zadanie od WT
- Klient wykonuje obliczenia
- Klient przekazuje wynik do WT
- WT przekazuje wynik do Serwera
- Serwer odbiera zadania od WT
- Serwer wylicza i wyświetla ostateczny wynik

MC-PI architektura 3-warstwowa c.d.

- Analiza architektury systemu w oparciu o analizę specyfikacji
  - **Rzeczowniki** – identyfikacja klas
  - **Czasowniki** – identyfikacja metod
- Analiza pozwala na wstępne zaprojektowanie klas i metod zgodnie z paradygmatem projektowania obiektowego

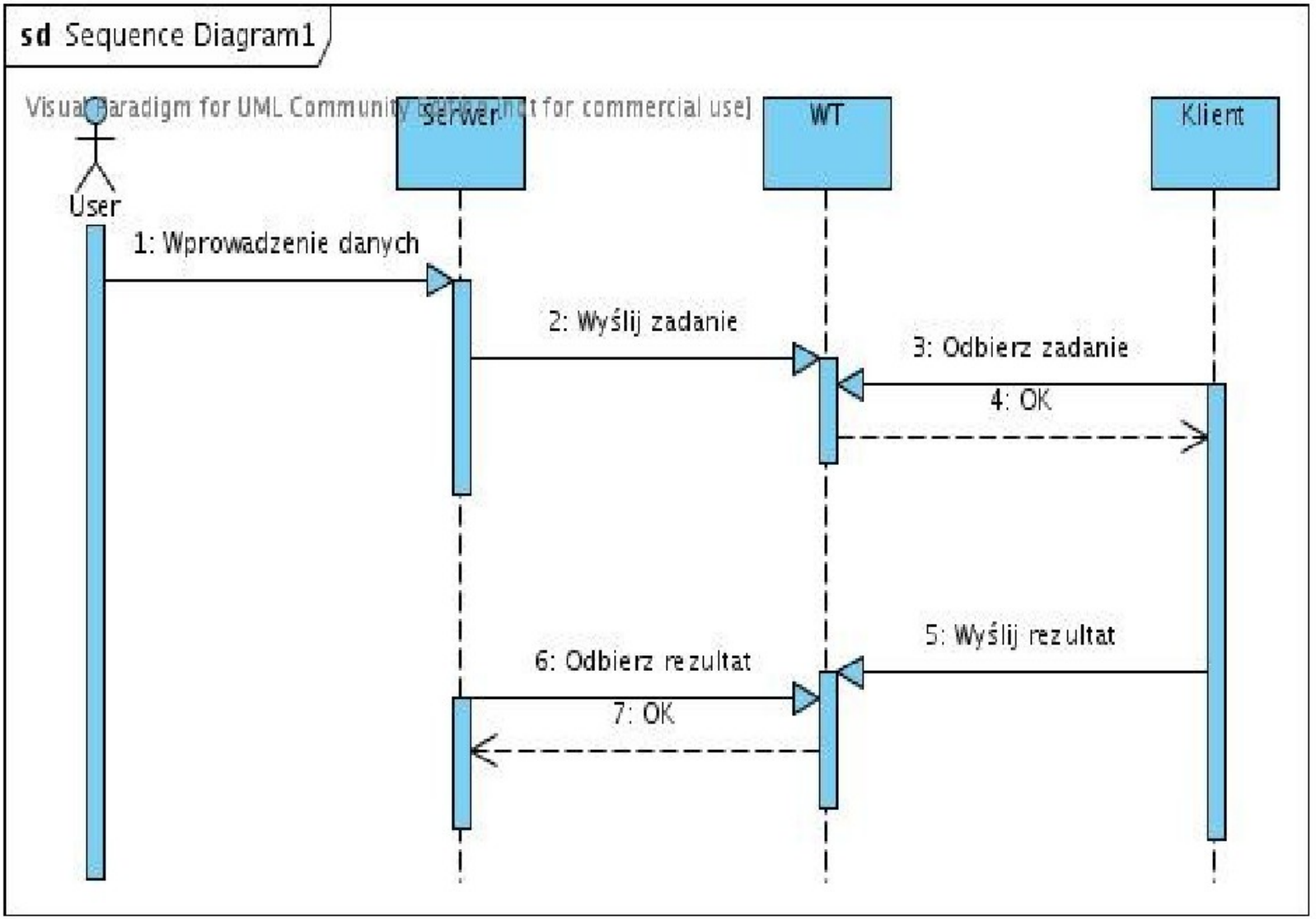
MC-PI architektura 3-warstwowa c.d.

- **Serwer wczytuje zadanie** (liczbę prób)
- **Serwer dzieli zadanie** na **zadania** dla **klientów**
- **Serwer przekazuje zadania** do **warstwy transportowej (WT)**
- **WT przekazuje zadania** dla **klientów**
- **Klient odbiera** swoje **zadanie** od **WT**
- **Klient wykonuje** obliczenia
- **Klient przekazuje rezultat** do **WT**
- **WT przekazuje rezultaty** do **serwera**
- **Serwer odbiera rezultaty** od **WT**
- **Serwer wylicza** i **wyświetla** ostateczny **wynik**

MC-PI architektura 3-warstwowa c.d.

- **Klasy** i ich **odpowiedzialność**
  - **Serwer**: wczytuje (zadanie), dzieli (zadanie), przekazuje (zadanie), odbiera (rezultaty), wylicza i wyświetla (wynik)
  - **Klient**: odbiera (zadanie), wykonuje (zadanie), wysyła (wynik)
  - **WT**: przekazuje (zadanie, wynik)

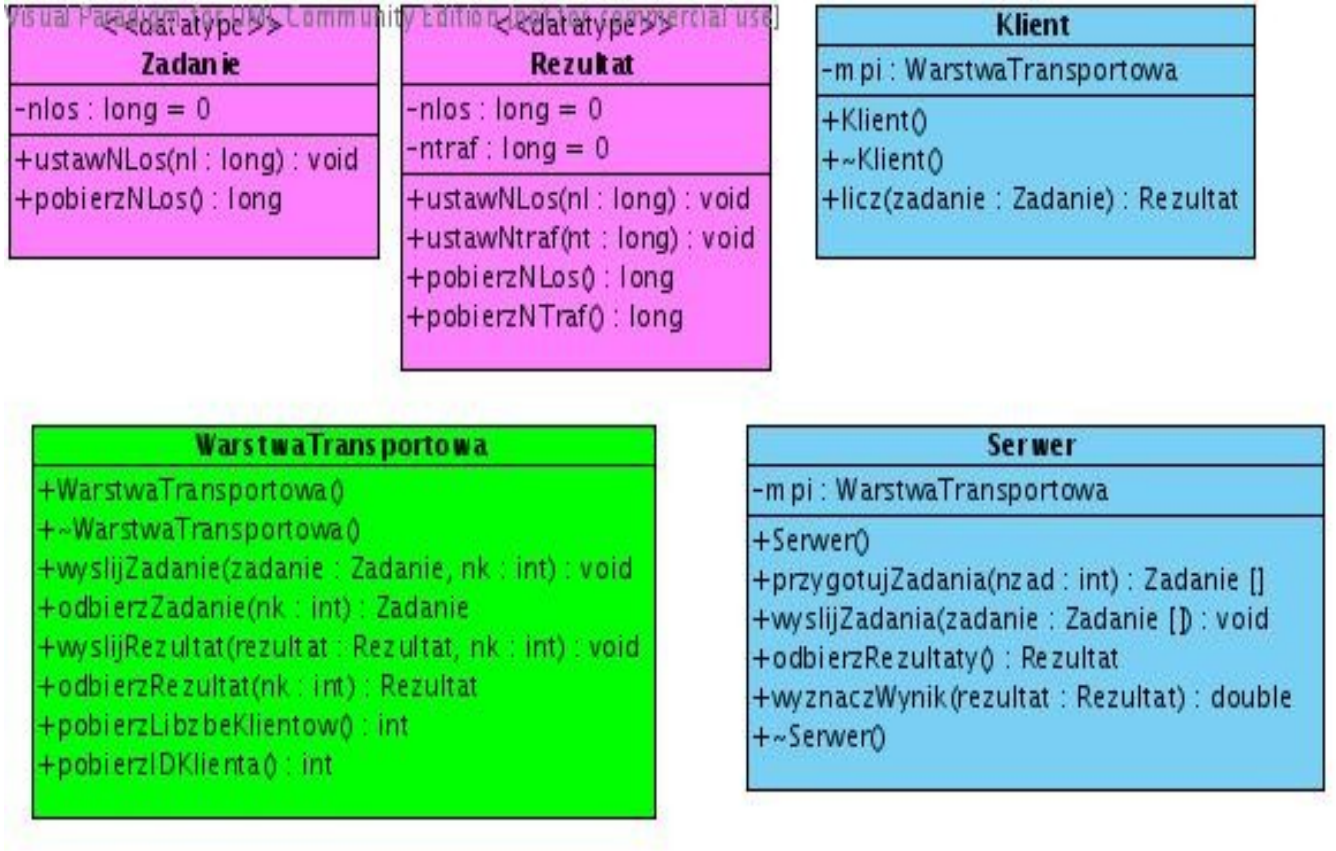
MC-PI architektura 3-warstwowa c.d.



MC-PI architektura 3-warstwowa c.d.

- Wstępna analiza architektury aplikacji pozwala na łatwe określenie potrzebnych klas oraz ich metod
- Każda klasa może być zaimplementowana i przetestowana osobno (TDD)
- Dodatkowe decyzje dotyczące sposobu implementacji metod zależą od decyzji programisty

MC-PI – Propozycja klas

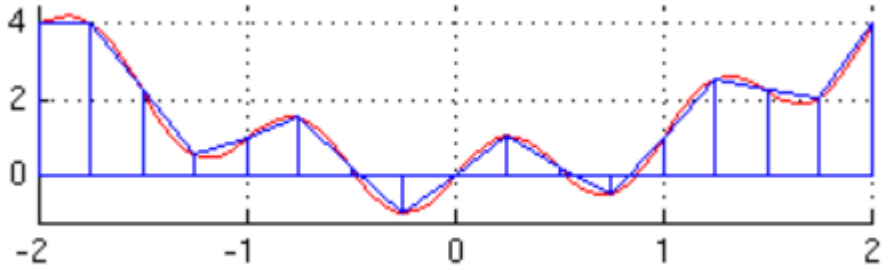


Przykład projektu MPI CAŁKA

Problem – obliczanie całki metodą trapezów

Całka funkcji jednej zmiennej. Użytkownik podaje przedział całkowania (początek i koniec) oraz krok całkowania. Całkowana funkcja jest “wpisana” w program na stałe. Cały przedział dzielony jest na mniejsze przedziały i rozsyłany do węzłów obliczeniowych. Każdy węzeł obliczeniowy liczy swój przedział i odsyła wynik. Wyniki są łączone i całkowity rezultat wyświetlany jest na ekran.

Algorytm



$$\int_a^b f(x)dx \approx h\left(\frac{y_1}{2} + y_2 + y_3 + \ldots + y_n + \frac{y_{n+1}}{2}\right) = \frac{h}{2}\sum_{i=1}^{n}(f(x_i) + f(x_{i+1}))$$

Źródło: Wikipedia

Opis aplikacji

Serwer czyta zadanie. Serwer dzieli zadanie na fragmenty (lista zadań). Serwer przekazuje listę zadań do Warstwy Transportowej (WT). WT rozsyła zadania do klientów. Klient odbiera zadanie od WT. Klient liczy całkę. Klient przekazuje rezultat do WT. WT przesyła rezultat do serwera. Serwer odbiera wyniki od WT. Serwer oblicza końcowy rezultat. Serwer wyświetla rezultat.

Opis aplikacji – rzeczowniki i czasowniki

Serwer czyta zadanie. Serwer dzieli zadanie na fragmenty (lista zadań). Serwer przekazuje listę zadań do Warstwy Transportowej (WT). WT rozsyła zadania do klientów. Klient odbiera zadanie od WT. Klient liczy rezultat. Klient przekazuje rezultat do WT. WT przesyła rezultat do serwera. Serwer odbiera wyniki od WT. Serwer oblicza końcowy rezultat. Serwer wyświetla końcowy rezultat.

Obiekty i operacje

- **Serwer – czyta** zadanie, **dzieli**, **przekazuje** do WT, **odbiera** wynik z WT, **oblicza** końcowy rezultat, **wyświetla** końcowy rezultat.
- **Zadanie, Lista zadań**
- **Warstwa Transportowa – przesyła** zadanie, **przesyła** wynik
- **Klient** – **odbiera** zadanie od WT, **liczy** wynik, **przekazuje** wynik do WT
- **Wynik, końcowy rezultat**

Klasa Zadanie

```
#ifndef ZADANIE_H
#define ZADANIE_H

class Zadanie {
    private:
        double _poczatek;
        double _koniec;
        double _krok;
    public:
        Zadanie();
        ~Zadanie();
        int ustawKrok(double kr);
        int ustawZakres(double p,double k);
        double poczatek();
        double koniec();
        double krok();
};
#endif
```

Klasa Warstwa Transportowa

```
#ifndef WARSTWATRANSPORTOWA_H
#define WARSTWATRANSPORTOWA_H

#include "Zadanie.h"

class WarstwaTransportowa {
      public:
            WarstwaTransportowa();
            ~WarstwaTransportowa();
            int wyslijZadanie(Zadanie zad);
            Zadanie odbierzZadanie();
            int wyslijRezultat(double rez);
            double odbierzRezultat();
};
#endif
```

Klasa Klient

```
#ifndef KLIENT_H
#define KLIENT_H
#include "Zadanie.h"
#include "WarstwaTransportowa.h"

class Klient {
      public:
            Klient();
            ~Klient();
            int odbierzZadanie(WarstwaTransportowa wt);
            void liczZadanie();
            void wyslijWynik(WarstwaTransportowa wt);
      private:
            Zadanie zad;
            double rez;
};
#endif
```

Klasa Serwer

```
#ifndef SERVER_H
#define SERVER_H
#include <vector>
#include "Zadanie.h"
#include "WarstwaTransportowa.h"

typedef std::vector<Zadanie> ListaZadan;
typedef std::vector<double> ListaRezultatow;

class Serwer {
    public:
        Serwer();
        ~Serwer();
        Zadanie czytajDane();
        ListaZadan dzielZadanie(Zadanie zad,int N);
        void wyslijListeZadan(WarstwaTransportowa wt);
        ListaRezultatow odbierzListeRezultatoe(WarstwaTransportowa wt);
        double sumujRezultaty(ListaRezultatow lr);
        void wyswietlWynik(double rez);
};
#endif
```

Dokumentacja

- Dokumentacja jest ważnym elementem każdego nietrywialnego projektu, szczególnie projektów które będą w przyszłości rozwijane/modyfikowane.
- W utrzymywaniu kompletnej i aktualnej dokumentacji deweloperskiej bardzo pomagają programy takie jak Doxygen
- Doxygen generuje dokumentację na podstawie komentarzy w kodzie – eliminuje konieczność utrzymywania osobno kodu i dokumentacji

Testy

- Projektowanie inicjowane testami (TDD) – testy tworzone są jeszcze przed tworzeniem samego kodu.
- Moduły testujące – automatyzacja procesu testowania.
- Bardziej przejrzysty projekt (trzeba z góry wiedzieć co testować).
- Większa niezawodność.
- Znacznie większa swoboda usprawniania i modyfikacji implementacji bez obawy o zniszczenie czy zmianę funkcjonalności poszczególnych elementów projektu.

Testy – Co testować?

- Klasa Zadanie
  - Wartości początkowe
  - Akcesory (ustaw/czytaj wartości).
  - Błędy – zły zakres (koniec<początek), zły krok (krok<=0)
- Klasa Warstwa Transportowa
  - Inicjalizacja biblioteki MPI (w konstruktorze)
  - Finalizacja biblioteki MPI (w destruktorze)
  - Przesyłanie zadań (wyślij/odbierz)
  - Przesyłanie wyników (wyślij/odbierz)

Kodowanie

- Najnudniejsza część pracy.
- Zwykle wykonywana przez młodszych programistów / koderów (studentów :-))
- Wykorzystanie modułów testujących pozwala na testowanie kodu i śledzenie jego rozwoju na dowolnym jego etapie.
- Każdy komponent może być rozwijany niezależnie przez innego programistę (dokumentacja i moduły testowe ściśle definiują moduł).

Zakończony projekt

- Wszystkie moduły osobno przechodzą wszystkie testy.
- Skonsolidowane moduły przechodzą testy funkcjonalne (czy system jako całość robi to, co powinien).
- **SUKCES!!! GRATULACJE!**
- **Oddanie gotowego projektu i obowiązkowa impreza**

Analiza wydajności MPI

Wydajność MPI

- Badanie wydajności MPI było przedmiotem pracy dyplomowej:

Marcin Jaźnicki

Praca inżynierska

Wyższa Szkoła Menedżerska w Warszawie

(WSM)

Wydajność MPI c.d.

- Cele pracy:
  - Analiza wydajności MPI dla kilku wybranych problemów obliczeniowych
  - Wykorzystanie klastra o rozmiarze od 1 do 16 komputerów
  - Problemy:
    - **Problem 1** - Obliczanie całki metodą trapezów
    - **Problem 2** - Wyznaczanie wartości tangensa hiperbolicznego dla zadanej wartości

Problem 1

- Problem bardzo łatwy do implementacji równoległej („embarrassingly parallel”)
- Podział zadania na duże bloki („coarse-grain”) redukuje narzut na komunikację
- Problem doskonale nadaje się do rozwiązania za pomocą klastra komputerów

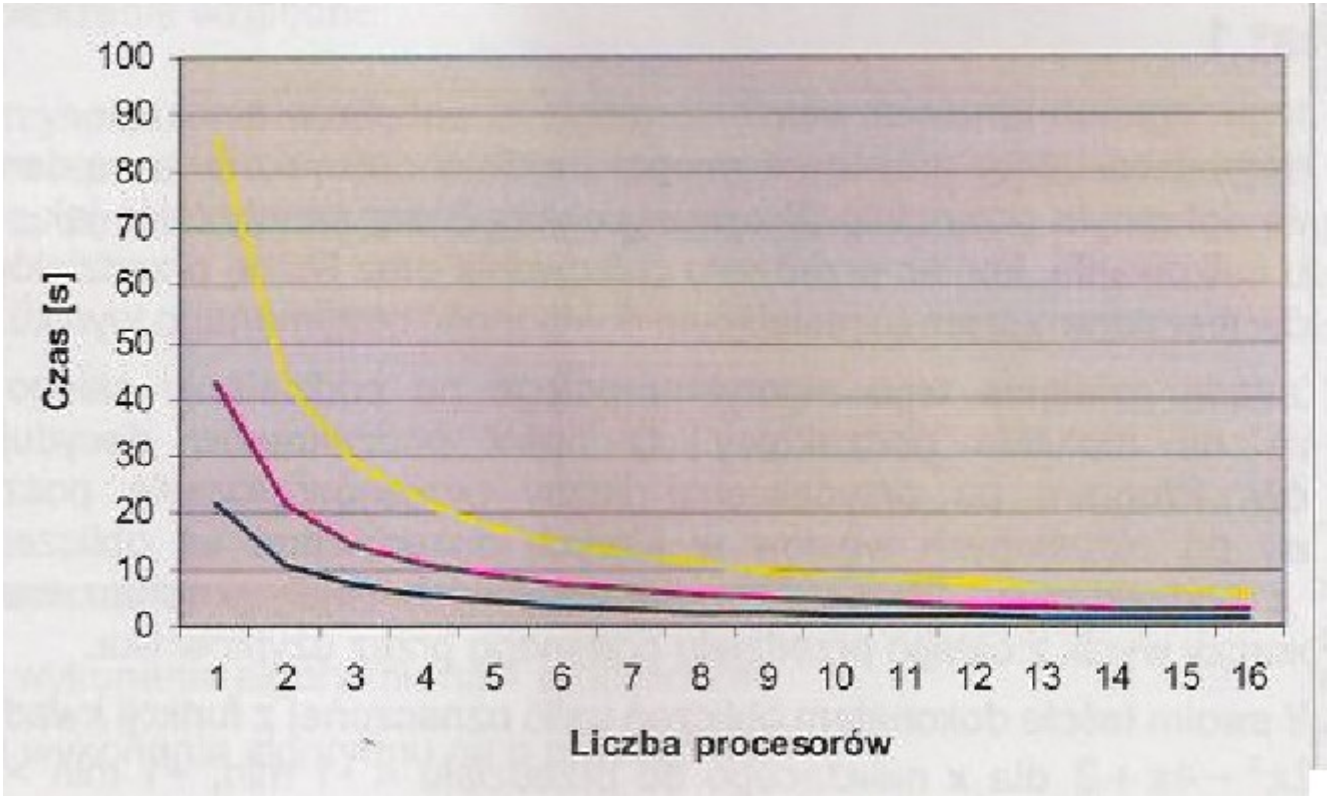
Problem 2

- Bardziej złożony algorytm, wymagający częstej wymiany danych między węzłami.
- Zwiększanie liczby węzłów zwiększa narzut na komunikację – słaba skalowalność
- Środowisko klastra słabo sprawdza się do tego typu problemów, znacznie lepsza byłaby architektura wieloprocesorowa ze wspólną pamięcią.

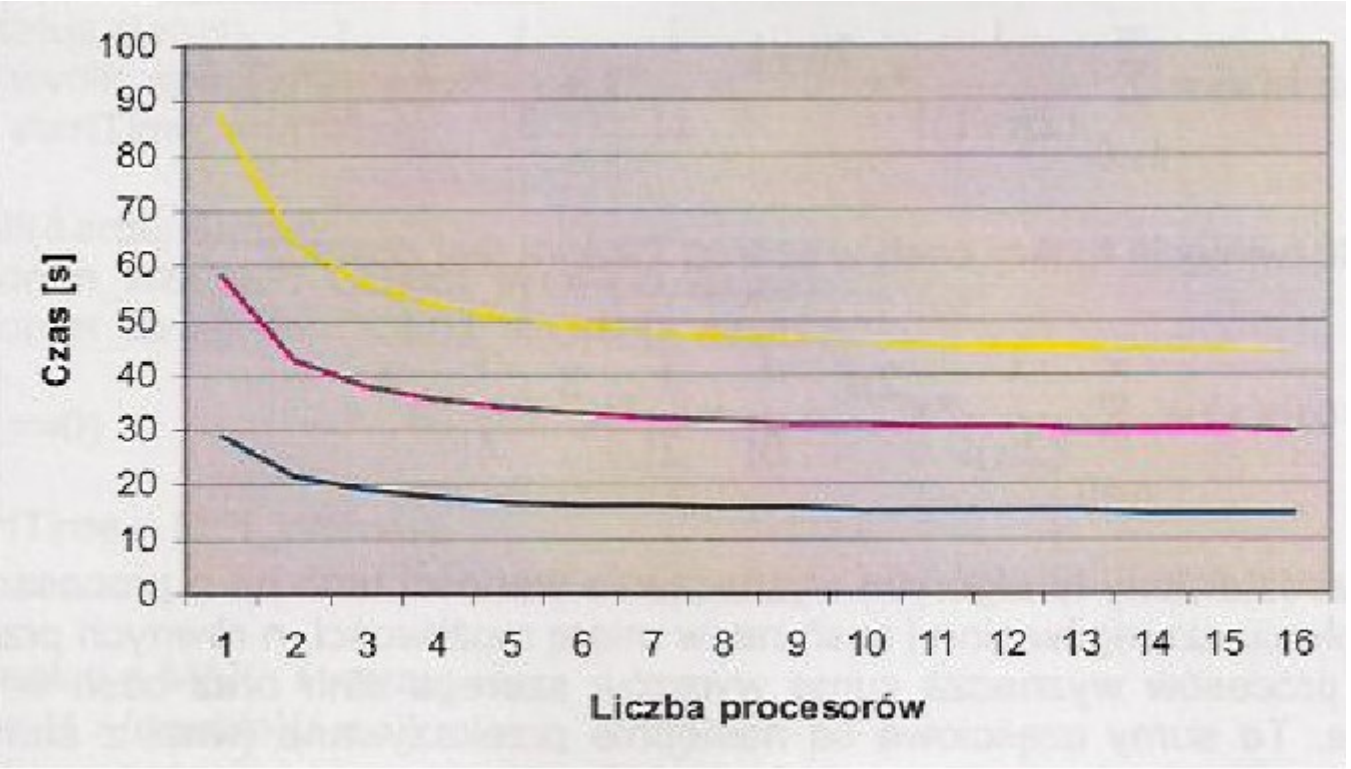
Badane parametry

- **Czas wykonania programu** – czas w sekundach od rozpoczęcia do zakończenia programu.
- **Przyspieszenie względne** – zależność między czasem wykonania na N węzłach, od czasu wykonania na jednym węźle (np. dla N=2, p=2.0 oznacza dwukrotnie szybsze wykonanie programu.
- **Efektywność** – przyspieszenie względne podzielone przez liczbę węzłów. Wartością idealną jest 1.0, wartości rzeczywiste będą mniejsze.

Czas wykonania – problem 1



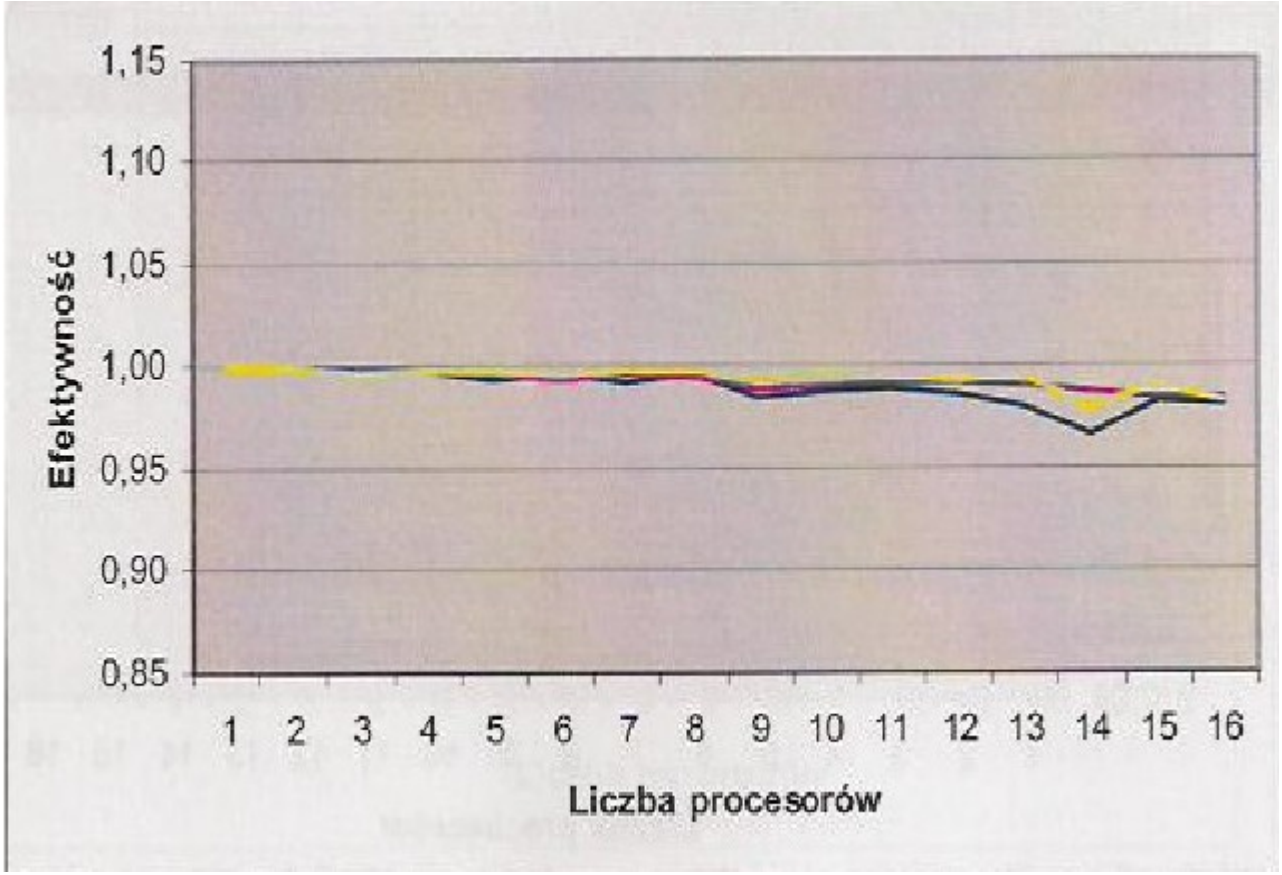
Czas wykonania – problem 2



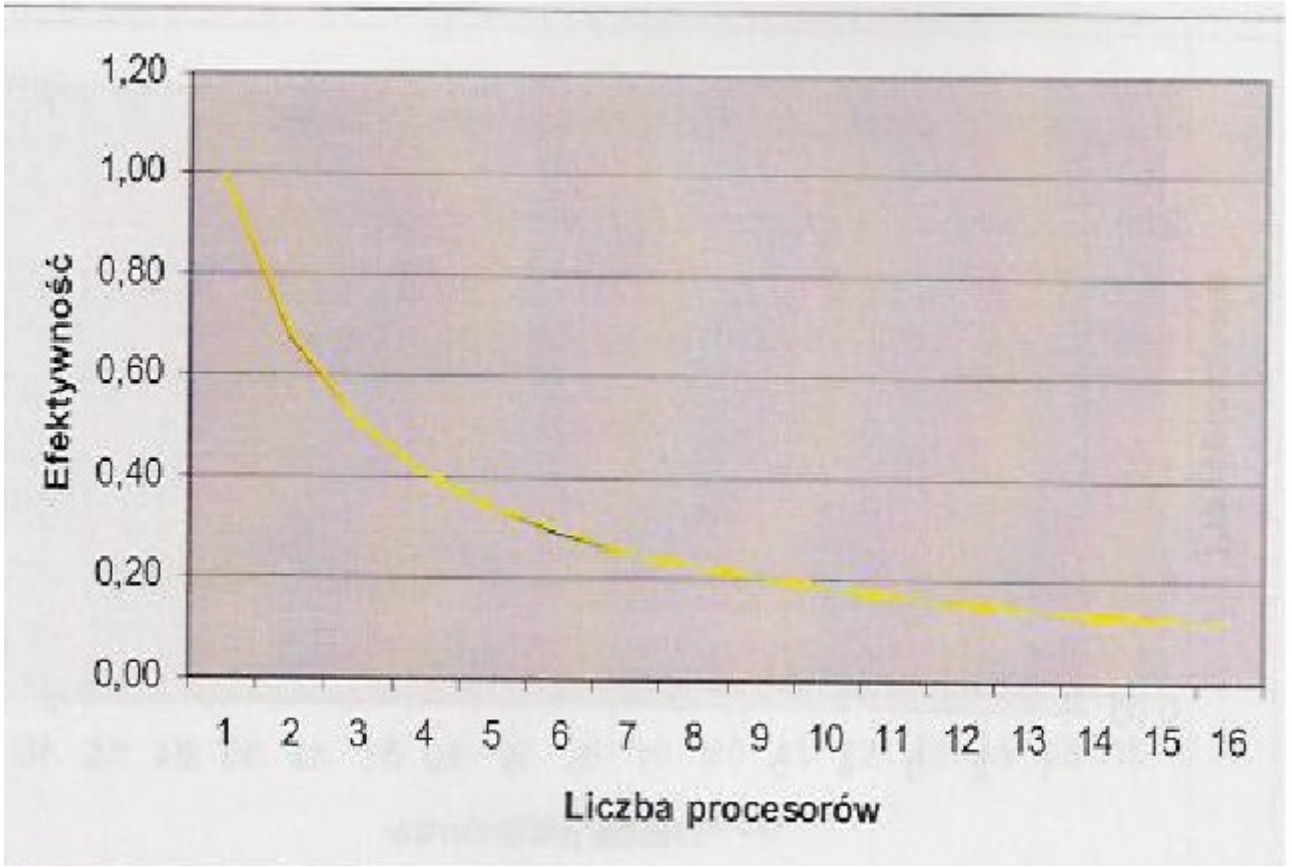
Czas wykonania - rezultaty

- **Problem 1**
  - dodawanie węzłów zdecydowanie obniża czas wykonania programu
- **Problem 2**
  - dodawanie węzłów w niewielkim stopniu wpływa na czas obliczeń
  - użycie więcej niż 5 węzłów jest praktycznie nieopłacalne

Efektywność – problem 1



Efektywność – problem 2



Efektywność - rezultaty

- **Problem 1**
  - Dla liczby węzłów N<13 efektywność utrzymuje się praktycznie na idealnym poziomie
  - Dla N>13 efektywność nieznacznie się pogarsza
- **Problem 2**
  - Już dodanie drugiego węzła dramatycznie pogarsza efektywność (obliczenia przyspieszają tylko o 40%).
  - Użycie 16 węzłów powoduje jedynie 2-krotne przyspieszenie obliczeń

Wnioski

- Wydajność klastra bardzo zależy od rodzaju problemu
- Problemy, które łatwo podzielić na duże pod-zadania, nie wymagające komunikacji (CPU bound,coarse-grain) – efektywne wykorzystanie klastra
- Problemy wymagające komunikacji i podziału na dużo małych zadań (network bound, fine-grain) – efektywność klastra nie będzie duża. Problemy tego typu wymagają komputerów wieloprocesorowych ze wspólną pamięcią.