Techniki Programowania Równoległego

- OpenMP
- MPI

Programowanie równoległe

- Początki programowania równoległego to lata 1960 –1970
- Obecnie programowanie równoległe oparte jest na:

• znanych od dekad modeli programowania równoległego (MPI, OpenMP),

• nowych paradygmatów programowania równoległego (CUDA, OpenCL),

• „językach przyszłości" tworzonych dla przyszłych architektur komputerowych (X10, Chapel, Fortress).

Czym są obliczenia równoległe?

Obliczenia równoległe to takie, w których wiele operacji obliczeniowych wykonuje się jednocześnie w ramach dostępnych jednostek obliczeniowych (procesorów, rdzeni, węzłów obliczeniowych).

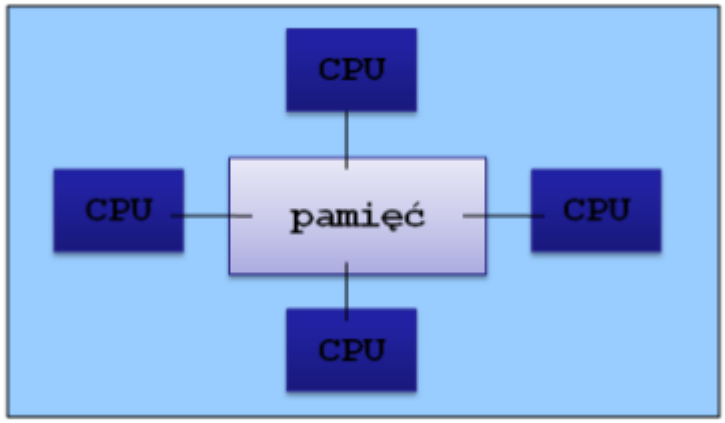
Bardzo często duże problemy obliczeniowe mogą być podzielone na mniejsze podproblemy, które mogą wykonywać się jednocześnie.

Przez wiele lat obliczenia równoległe wykonywane były jedynie w branży HPC (High Performance Computing).

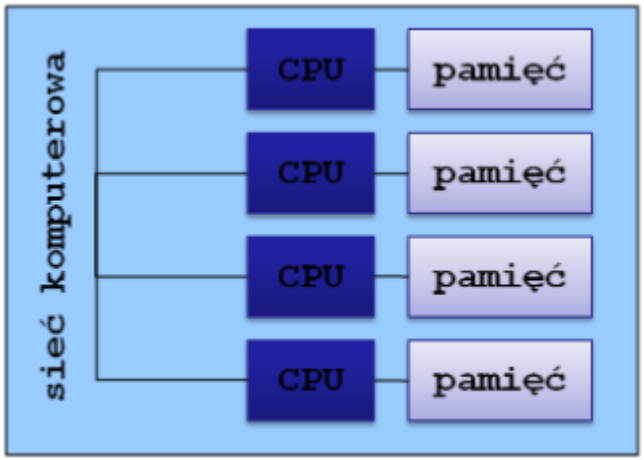
Model pamięci współdzielonej

MODEL OpenMP
- Procesory współdzielą globalną przestrzeń adresową
- Technologia: ograniczona liczba procesorów
- Szybka komunikacja i synchronizacja
- Równoległe typy danych : shared, private
- Ochrona dostępu do danych współdzielonych
- Komunikacja poprzez zmienne współdzielone
- Równoległe wątki

MODEL MPI
- Procesory operują na prywatnej, lokalnej pamięci
- Technologia: duża liczba procesorów
- Mechanizmy komunikacji i synchronizacji
- Operowanie na danych lokalnych
- Komunikacja poprzez połączenie sieciowe
- Koszt komunikacji rośnie z liczbą procesorów
- Równoległe procesy



Model programowania OpenMP



Model programowania MPI

Open MP -Open Multi-processing - dla programowania ze wspólną pamięcią

- Standard definiujący interfejs programowania aplikacji (API - Application Programming Interface) do tworzenia programów równoległych opierających się na modelu wielowątkowym dla systemów z pamięcią wspólną.
- Składa się z:

  -dyrektyw kompilatora,

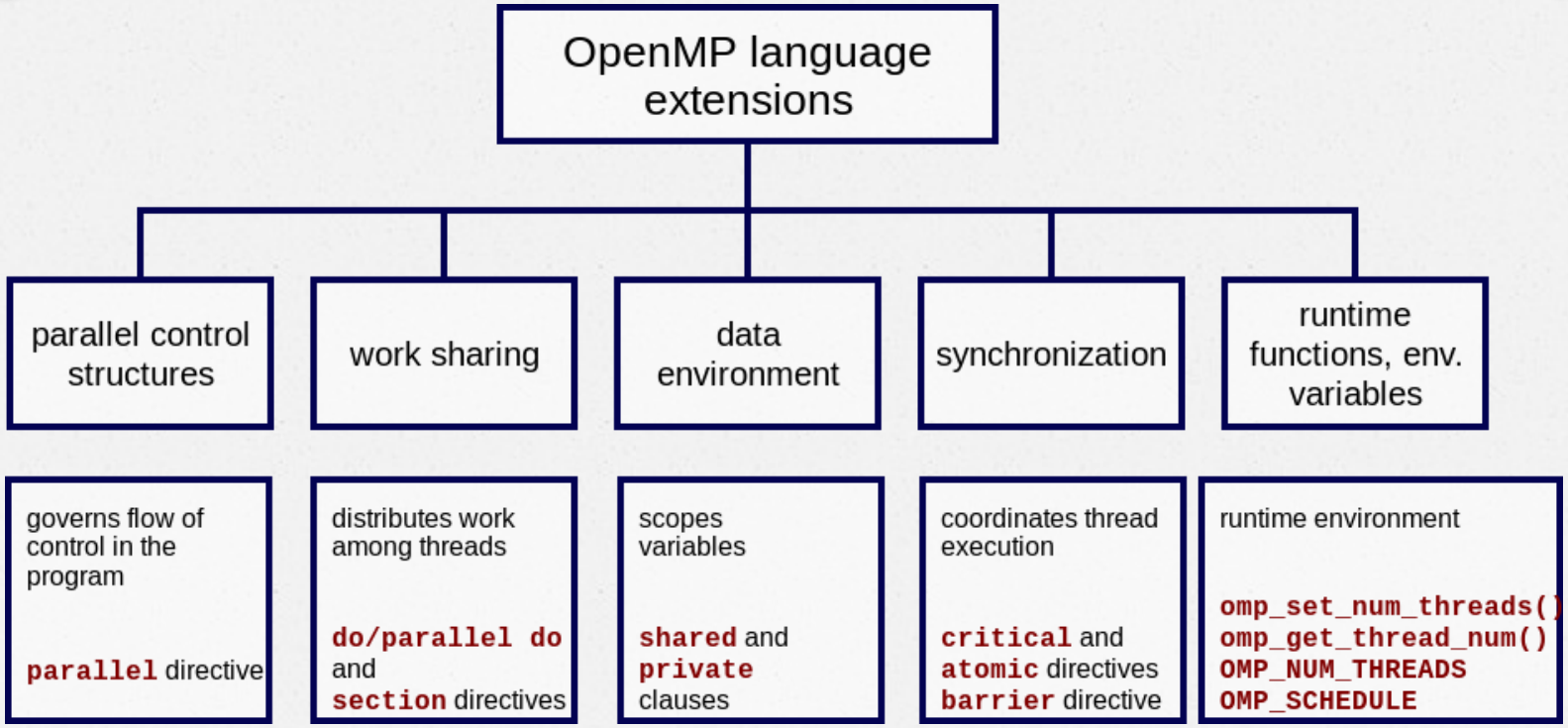
  -funkcji bibliotecznych,

  -zmiennych środowiskowych.
- Dyrektywy kompilatora

Programowanie za pomocą OpenMP –zestaw dyrektyw kompilatora.

Przykład:

```
#pragma omp parallel for
for (i = 0; i < N; i++)
  a[i] = 2 * i;
```



Podstawowymi elementami OpenMP są konstrukcje służące do tworzenia wątków, rozdzielania (współdzielenia) pracy, zarządzania środowiskiem danych, synchronizacji wątków, procedury czasu wykonania (poziomu użytkownika) i zmienne środowiskowe.

MPI -Message Passing Interface – dla typowego klastra

- Protokół komunikacyjny będący standardem przesyłania komunikatów pomiędzy procesami programów równoległych działających na jednym lub więcej komputerach.
- Celami MPI są wysoka jakość, skalowalność oraz przenośność.
- Transfer danych pomiędzy poszczególnymi procesami programu wykonywanymi na procesorach maszyn będących węzłami klastra odbywa się za pośrednictwem sieci.
- Program w MPI składa się z niezależnych procesów operujących na różnych danych (MIMD).

Główne własności MPI

- umożliwia efektywną komunikację bez obciążania procesora operacjami kopiowania pamięci,
- udostępnia funkcje dla języków C/C++, Fortran oraz Ada,
- specyfikacja udostępnia hermetyczny interfejs programistyczny, co pozwala na skupienie się na samej komunikacji, bez wnikania w szczegóły implementacji biblioteki i obsługi błędów,
- definiowany interfejs zbliżony do standardów takich jak: PVM, NX czy Express,
- udostępnia mechanizmy komunikacji punkt – punkt oraz grupowej,
- może być używany na wielu platformach, tak równoległych jak i skalarnych, bez większych zmian w sposobie działania.

Wykorzystanie biblioteki MPICH

- MPICH to ogólnodostępna, darmowa i przenośna implementacja standardu MPI. Pozwala na przekazywanie komunikatów pomiędzy aplikacjami działającymi równolegle. Nadaje się do stosowania na małych klastrach.
- MPICH jest implementacją biblioteki sterowania procesem obliczeń równoległych na maszyny klasy PC. Protokół ten przeznaczony jest do sterowania procesem obliczeń równoległych w sieciach rozproszonych. Biblioteka procedur MPICH jest dostępna bezpłatnie. Najnowsza wersja tej biblioteki MPICH2 poza zapewnieniem bardziej wydajnych mechanizmów komunikacji posiada dodatkowo:

-wsparcie dla komunikacji jednostronnej

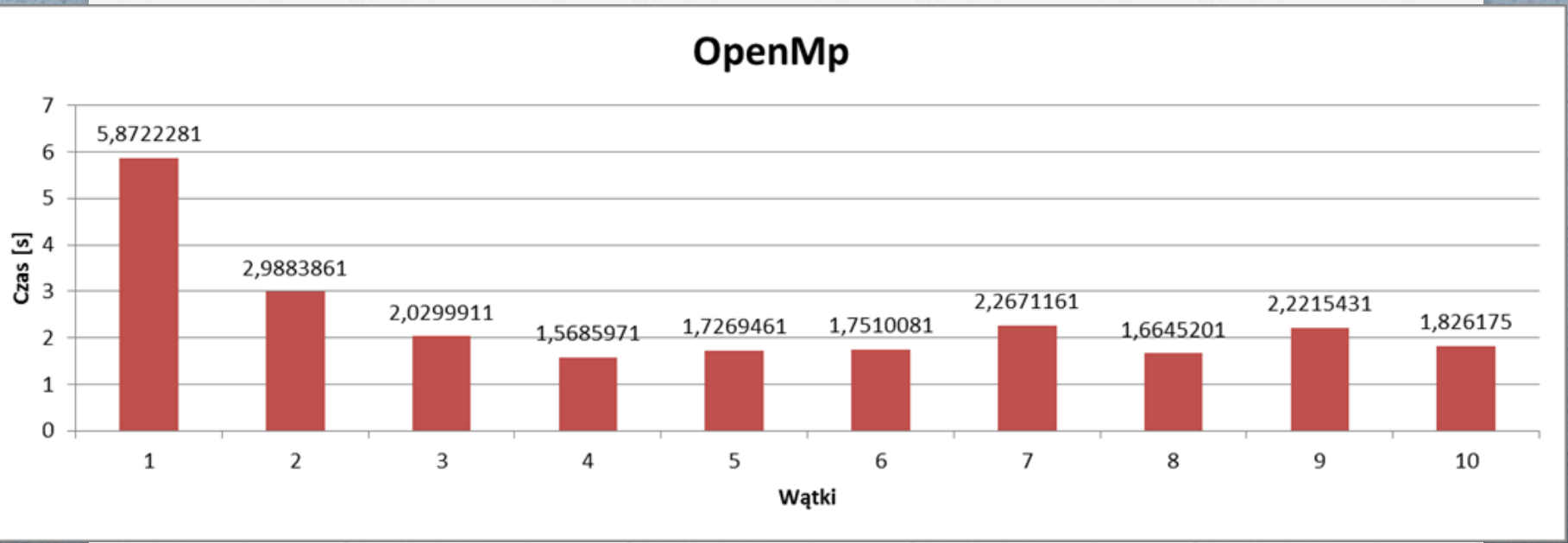
-rozszerzoną funkcjonalność MPI-IO

Program OpenMp

```
#include <stdlib.h>
#include <omp.h>
#include <stdio.h>
#include <time.h>
#include <stdbool.h>

int main(int argc, char **argv) {
   int i, j, k;
   if (argc < 2) {
      printf("Usage: %s [size] [threads]\n", *argv);
      return 1;
   }

   int size = atoi(argv[1]);
   int threads = atoi(argv[2]);

   float **arr_a = (float **) malloc(size * sizeof(float *));
   for (i = 0; i < size; i++)
      arr_a[i] = (float *) malloc(size * sizeof(float));

   float **arr_b = (float **) malloc(size * sizeof(float *));
   for (i = 0; i < size; i++)
      arr_b[i] = (float *) malloc(size * sizeof(float));

   float **arr_result = (float **) malloc(size * sizeof(float *));
   for (i = 0; i < size; i++)
      arr_result[i] = (float *) malloc(size * sizeof(float));

srand(time(NULL));
```

```
for (i = 0; i < size; i++)
      for (j = 0; j < size; j++)
         arr_a[i][j] = rand();
   for (i = 0; i < size; i++)
      for (j = 0; j < size; j++)
         arr_b[i][j] = rand();
   for (i = 0; i < size; i++)
      for (j = 0; j < size; j++)
         arr_result[i][j] = 0.0;

   double start = omp_get_wtime();
#pragma omp parallel for default(none) num_threads(threads) shared(arr_a, arr_b,
arr_result, size) private(i, j, k)

   for (i = 0; i < size; i++)
      for (j = 0; j < size; j++)
         for (k = 0; k < size; k++)
            arr_result[i][j] += arr_a[i][k] * arr_b[k][j];
   free(arr_a);
   free(arr_b);
   free(arr_result);
   double end = omp_get_wtime();
   printf("%d,%d,%f\n", size, threads, (end - start));
   return 0;
}
```

1	5,8722281
2	2,9883861
3	2,0299911
4	1,5685971
5	1,7269461
6	1,7510081
7	2,2671161
8	1,6645201
9	2,2215431
10	1,826175



Program MPI

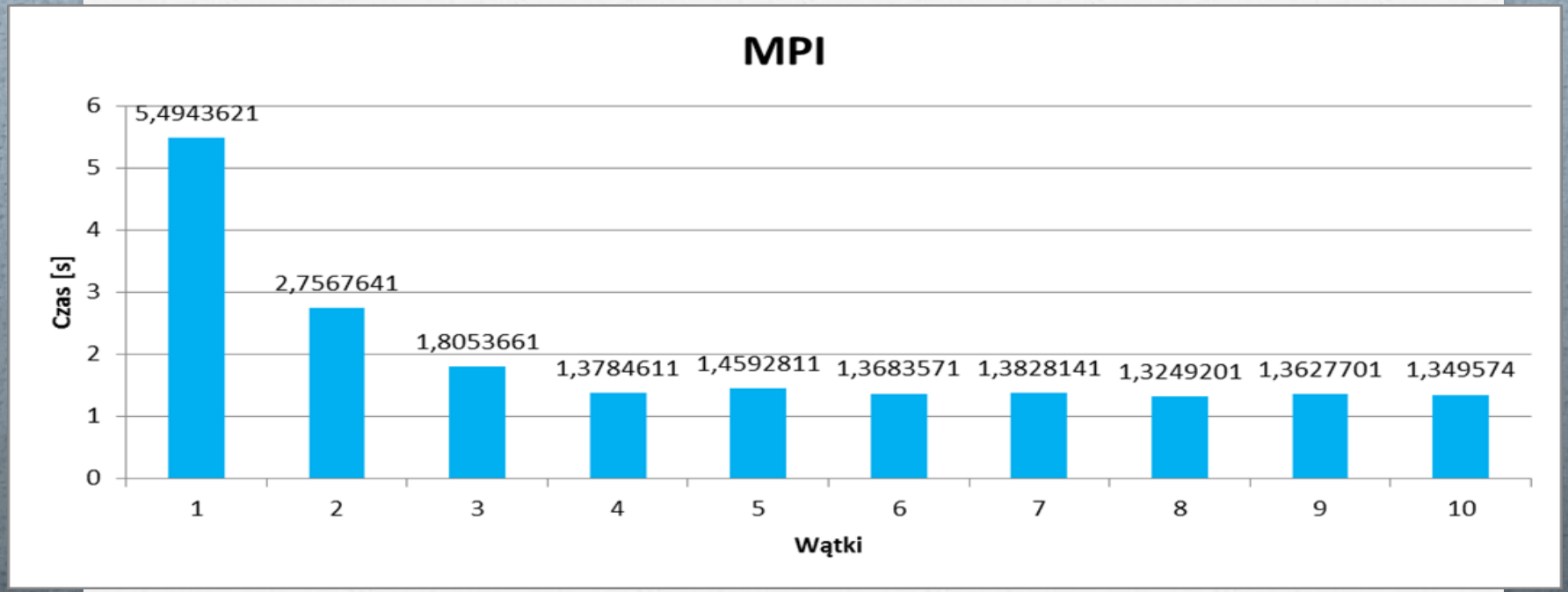
```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main(int argc, char *argv[]) {
   double **arr_a, **arr_b, **arr_c, *temp, start, end;
   int elements, offset, size, rank, nodes, n, i, j, k;
   int tag = 13;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &nodes);
   if (rank == 0) {
      start = MPI_Wtime();  }
   n = atoi(argv[1]);
   if (rank == 0) {
      temp = (double *) malloc(sizeof(double) * n * n);
      arr_a = (double **) malloc(sizeof(double *) * n);
      for (i = 0; i < n; i++) {
         arr_a[i] = &temp[i * n];    }
   } else {
      temp = (double *) malloc(sizeof(double) * n * n / nodes);
      arr_a = (double **) malloc(sizeof(double *) * n / nodes);
      for (i = 0; i < n / nodes; i++) {
         arr_a[i] = &temp[i * n];
      }
   }
```

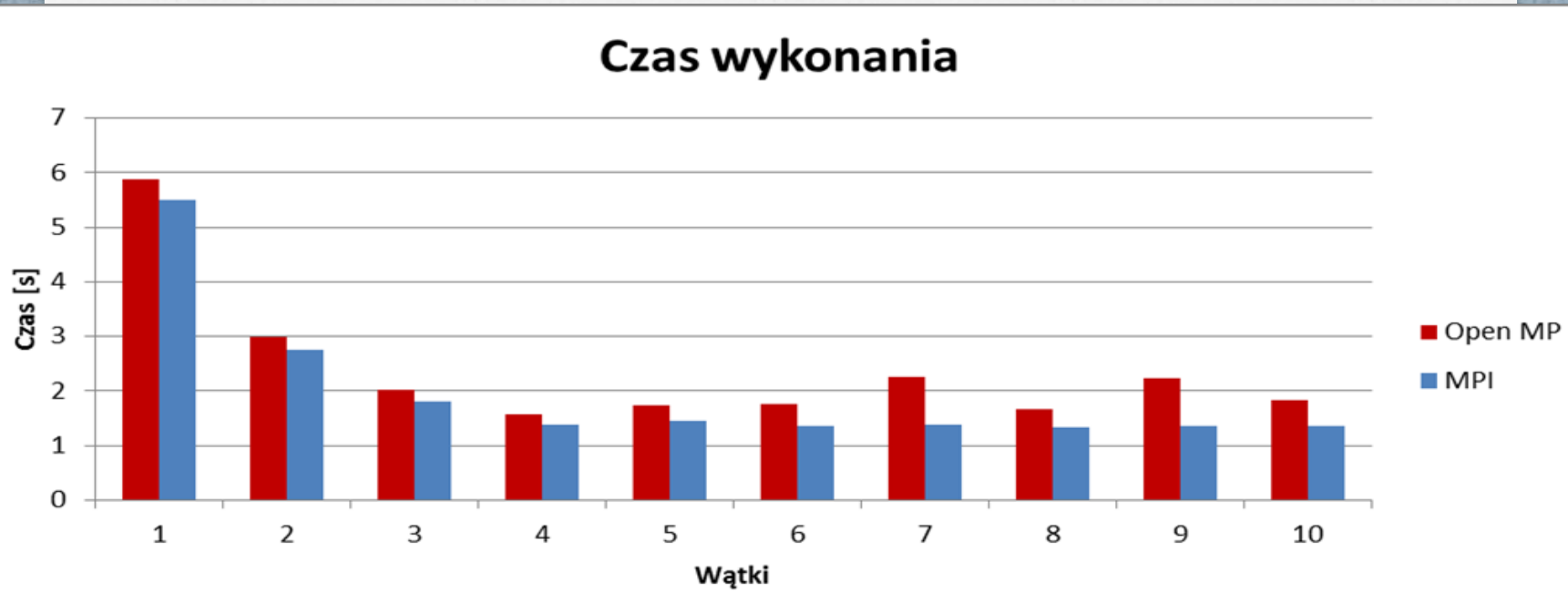
```
temp = (double *) malloc(sizeof(double) * n * n);
arr_b = (double **) malloc(sizeof(double *) * n);
for (i = 0; i < n; i++) {
   arr_b[i] = &temp[i * n];
}
if (rank == 0) {
   temp = (double *) malloc(sizeof(double) * n * n);
   arr_c = (double **) malloc(sizeof(double *) * n);
   for (i = 0; i < n; i++) {
      arr_c[i] = &temp[i * n];
   }
} else {
   temp = (double *) malloc(sizeof(double) * n * n / nodes);
   arr_c = (double **) malloc(sizeof(double *) * n / nodes);
   for (i = 0; i < n / nodes; i++) {
      arr_c[i] = &temp[i * n]; }
}
srand(time(NULL));
if (rank == 0) {
   for (i = 0; i < n; i++) {
      for (j = 0; j < n; j++) {
         arr_a[i][j] = rand();
         arr_b[i][j] = rand();
      }
   }
}
```

```
size = n / nodes;
  elements = size * n;

  MPI_Datatype row;
  MPI_Type_contiguous(elements, MPI_DOUBLE, &row);
  MPI_Type_commit(&row);

  if (rank == 0) {
     offset = size;
     for (i = 1; i < nodes; i++) {
        MPI_Send(arr_a[offset], 1, row, i, tag, MPI_COMM_WORLD);
        offset += size;
     }
  } else {
     MPI_Recv(arr_a[0], 1, row, 0, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  }

  MPI_Datatype b_cast;
  MPI_Type_contiguous(n * n, MPI_DOUBLE, &b_cast);
  MPI_Type_commit(&b_cast);

  MPI_Bcast(arr_b[0], 1, b_cast, 0, MPI_COMM_WORLD);

  for (i = 0; i < size; i++) {
     for (j = 0; j < n; j++) {
        arr_c[i][j] = 0.0;
     }
  }
```

```
for (i = 0; i < size; i++) {
      for (j = 0; j < n; j++) {
         for (k = 0; k < n; k++) {
            arr_c[i][j] += arr_a[i][k] * arr_b[k][j];
         }
      }
   }

   if (rank == 0) {
      offset = size;
      elements = size * n;
      for (i = 1; i < nodes; i++) {
         MPI_Recv(arr_c[offset], 1, row, i, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
         offset += size;
      }
   } else {
      MPI_Send(arr_c[0], 1, row, 0, tag, MPI_COMM_WORLD);
   }
   if (rank == 0) {
      end = MPI_Wtime();
      printf("%d,%d,%f\n", n, nodes, end - start);
   }

   MPI_Finalize();
   return 0;
}
```

1	5,4943621
2	2,7567641
3	1,8053661
4	1,3784611
5	1,4592811
6	1,3683571
7	1,3828141
8	1,3249201
9	1,3627701
10	1,349574





Dane obliczenia wykonywane na 4-rdzeniowym procesorze, więc czasy obliczeń spadają do momentu uruchomienia 4 procesów, a przy większej ilości nie widać już zysku.

DZIĘKUJEMY ZA UWAGĘ!