

# Parallel sorting by regular sampling

SRIR

## 1. Opis algorytmu

Algorytm „Równoległego sortowania przez regularne dzielenie” został wymyślony przez Li.

Oryginalnie algorytm składa się z czterech faz:

### 1. Faza I

Sortowanie swojego zestawu danych przez każdy proces

### 2. Faza II

Jeden z procesów zbiera od innych procesów i siebie wartości równe liczbie procesów, które są wybierane w sposób regularny w każdym z procesów. Kolejno sortuje te elementy i wybiera „liczba procesów - 1” elementów, które przesyłane są do kolejnych procesów. Następnie każdy proces partycjonuje swój zbiór danych na odpowiednią liczbę podzbiorów w oparciu o otrzymane *pivoty* od serwera.

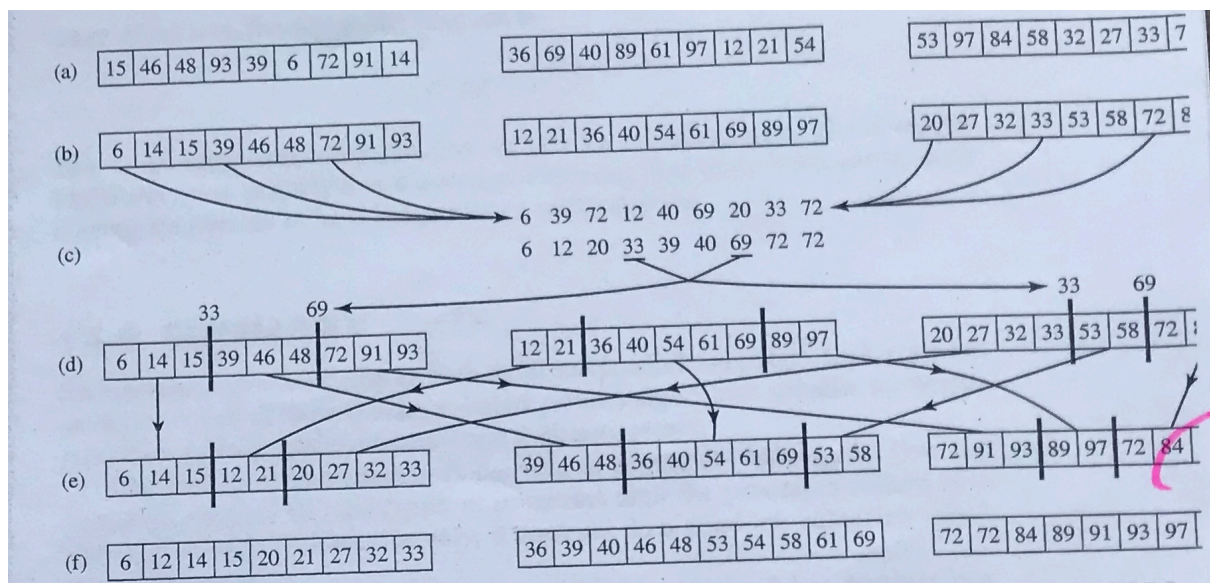
### 3. Faza III

W tej fazie, każdy z procesów zachowuje „swoją” (przypisaną do danego procesu) porcję danych i przesyła pozostałe dane do reszty procesów.

### 4. Faza IV

Każdy z procesów łączy zebrane dane z każdego procesu w jedną listę i je sortuje (*multimerge*).

W fazie tej maksymalna liczba elementów, które dany proces musi złączyć wynosi  $2 * \text{liczba elementów} / \text{liczba procesów}$



Rys. 1 Diagram przedstawiający przejście algorytmu

Wydajność algorytmu:

p - liczba procesów

n - liczba elementów

Dla obliczeń

$$O[(n/p) * \log(n/p) + p^2 * \log p + (n/p) * \log(p)]$$

$$\text{Dla } n \gg p: O[(n/p) * (\log(n) + \log(p))]$$

Dla komunikacji:

$$O(\log(p) + n/p)$$

$$\text{Dla } n \gg p: O(n/p)$$

Funkcja skalowalności:

$$p^C / p = p^{(C-1)}$$

Algorytm ten posiada trzy zalety względem algorytmu

*hyperquicksort*:

- Przechowuje listę rozmiarów w bardziej zbalansowany sposób pomiędzy procesami
- Pomija niepotrzebną komunikację pomiędzy kluczami
- Nie wymaga, aby liczba procesów była potęgą liczby 2

Ponadto algorytm umożliwia wprowadzenie prostej komunikacji *all-to-all*, która może być zaimplementowana w taki sposób, aby każdy element był przenoszony tylko raz.

## 2. Opis struktury projektu

Projekt składa się z 5 plików:

- utilities.h + utilities.c
- profiles.h + profiles.c
- PSRC.c

W plikach utilities znajdują się metody wykorzystywane w algorytmie do wypisywania macierzy, porównywania oraz łączenia tablic (multimerge).

W plikach profiles występują metody MPI, nadpisane, tak aby zrealizować profilowanie.

Plik PSRC.c to właściwy kod realizujący algorytm, w komentarzach w kodzie mamy wydzielone fazy, które są opisane w punkcie 3.

Ponadto w projekcie umieszczono katalog z przykładowymi danymi: *sample*, gdzie input to dane wejściowe do posortowania, a output wynik oczekiwany

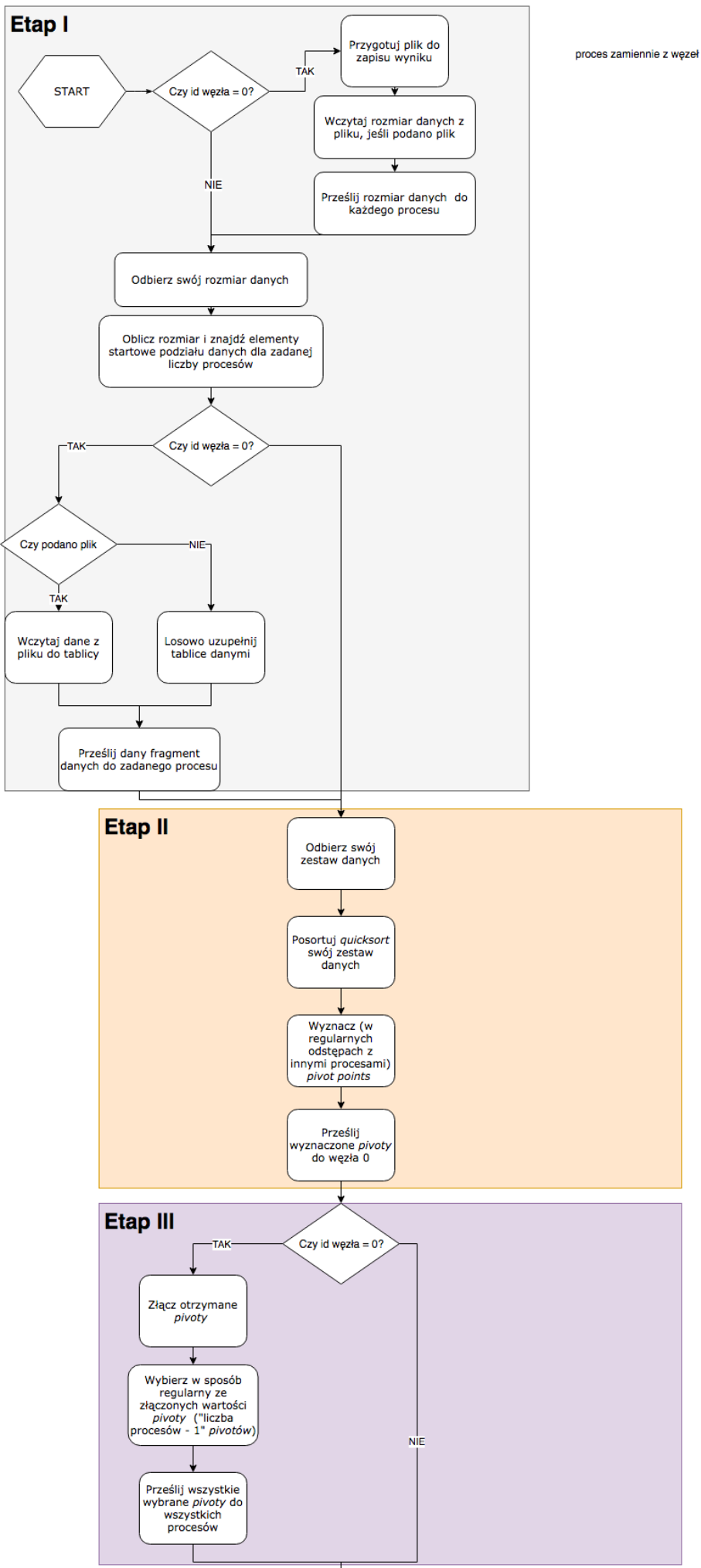
W projekcie znajduje się również katalog doc z dokumentacją.

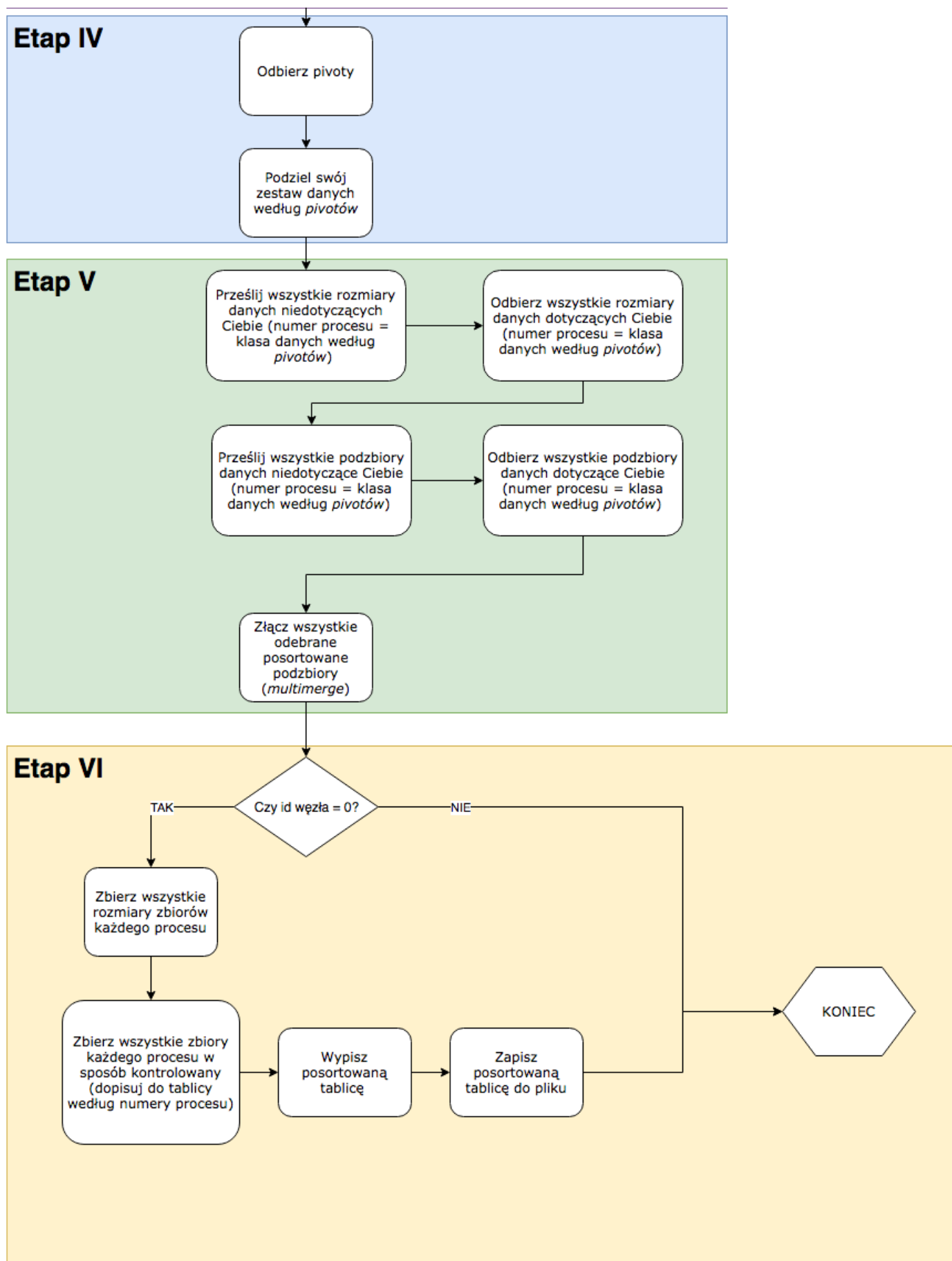
Ponadto do projektu dołączono makefile, którego metody opisane zostały w punkcie 4

### **3. Opis działania**

Opis działania przedstawiony został na diagramie Diag. 1, Diag. 2

Diagram został również umieszczony w folderze doc/





Diag.2

Aplikacja składa się z 6 etapów, które pokrywają się z etapami algorytmu przedstawionymi w punkcie 1.

1. ETAP1 - Początkowo aplikacja przygotowuje dane do posortowania
2. ETAP 2 - W kolejnej fazie, dane są rozdzielane na procesy i wybierane są wartości, będące *pivotami*
3. ETAP 3 - Następnie pivoty są przesłane do servera, który je łączy i wybiera „liczba procesów - 1” pivotów. Pivoty te są przesyłane do każdego procesu.
4. ETAP 4 - Kolejno pivoty te są wykorzystywane przez każdy proces do podziału swoich danych na partycje
5. ETAP 5 - W następnym kroku każdy proces pozyskuje tylko dane z zadanej przez identyfikator partycji od każdego procesu i je łączy.
6. ETAP 6 - Na końcu dane te są przesyłane do serwera, który je łączy i wypisuje do pliku

## 4. Opis obsługi programu

Kompilacja programu i jego uruchomienie odbywa się przy wykorzystaniu makefile'a.

Udostępnia on 4 reguły:

- *make all* zbudowanie projektu
- *make clean* przywrócenie folderu do stanu początkowego (usunięcie plików .o oraz .out)
- *make rebuild* wyczyszczenie i zbudowanie projektu

- *make run N=<liczba procesów> FILE==<ścieżka do pliku>*

*NODES=<nodes>*

- uruchomienie aplikacji podając odpowiednio parametry: N oraz opcjonalnie FILE i NODES, gdzie

N - liczba procesów (domyślnie 4),

FILE to nazwa pliku do wczytania do procesowania,

NODES to wskazanie na plik z adresami węzłów

np: *make run N=4 FILE==sample/input\_50.data NODES=nodes*

- uruchomi program na 4 procesach i na tylu węzłach ile w pliku nodes, wczytując dane z pliku o ścieżce sample/input\_50.data

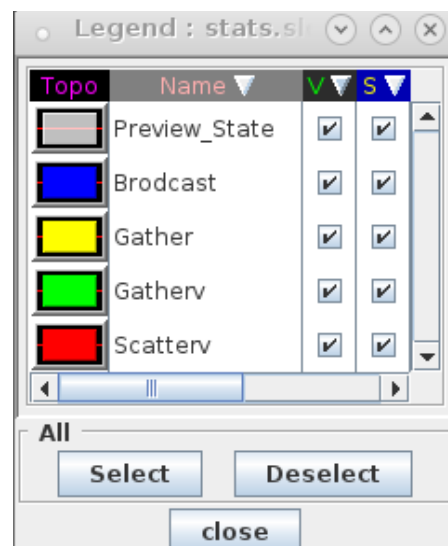
Kroki do skompilowania i uruchomienia na jednym węźle:

1. Przejdź do katalogu z projektem i plikiem make: MPI\_PSRS
2. Wykonaj *make clean*
3. Wykonaj *make run N=<liczba procesów> FILE=<ścieżka do pliku>*

## 4. Profilowanie

W ramach projektu zrealizowano profilowanie

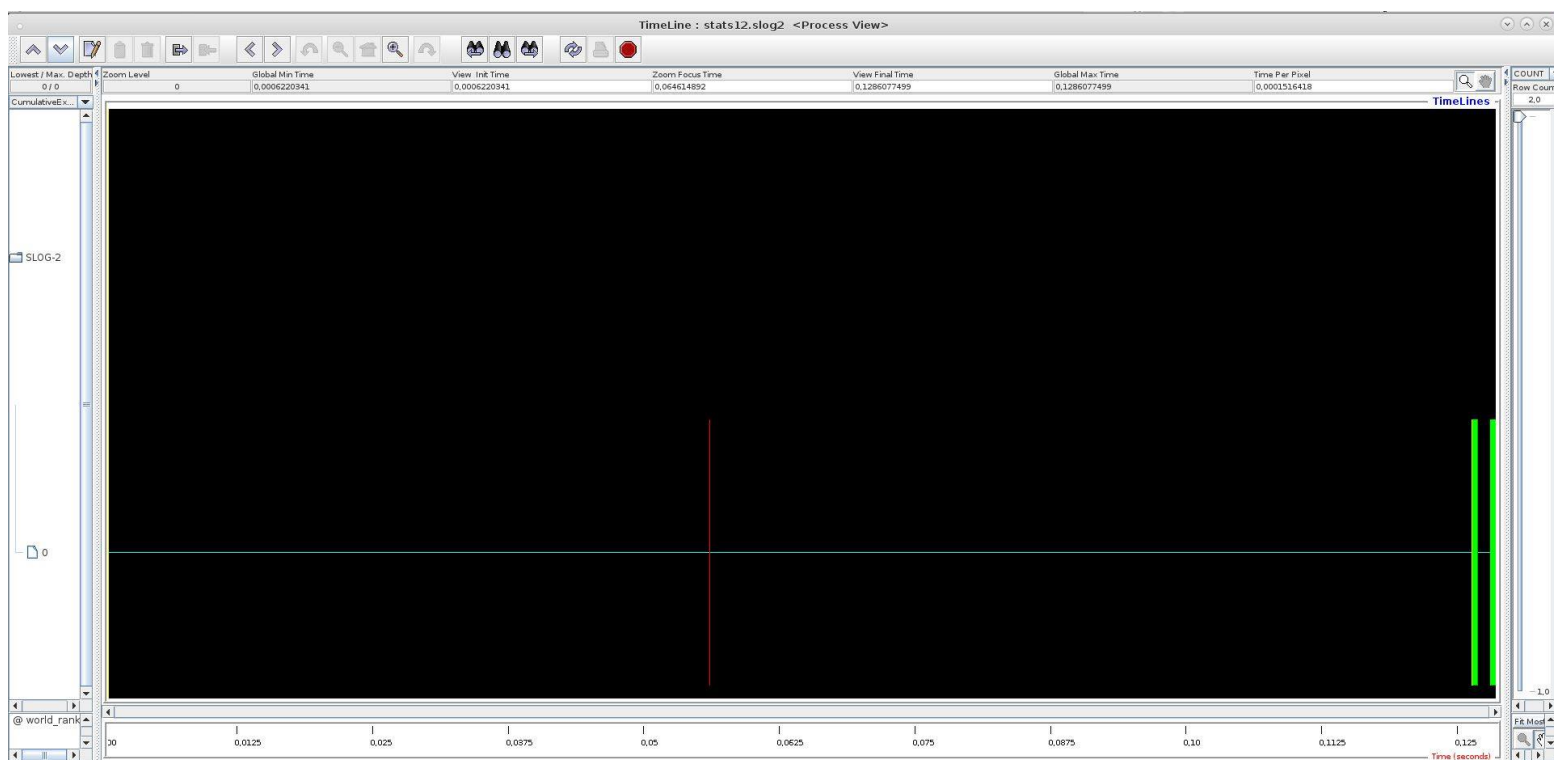
Kolory wykorzystane w wykresach przedstawia grafika Rys 2:



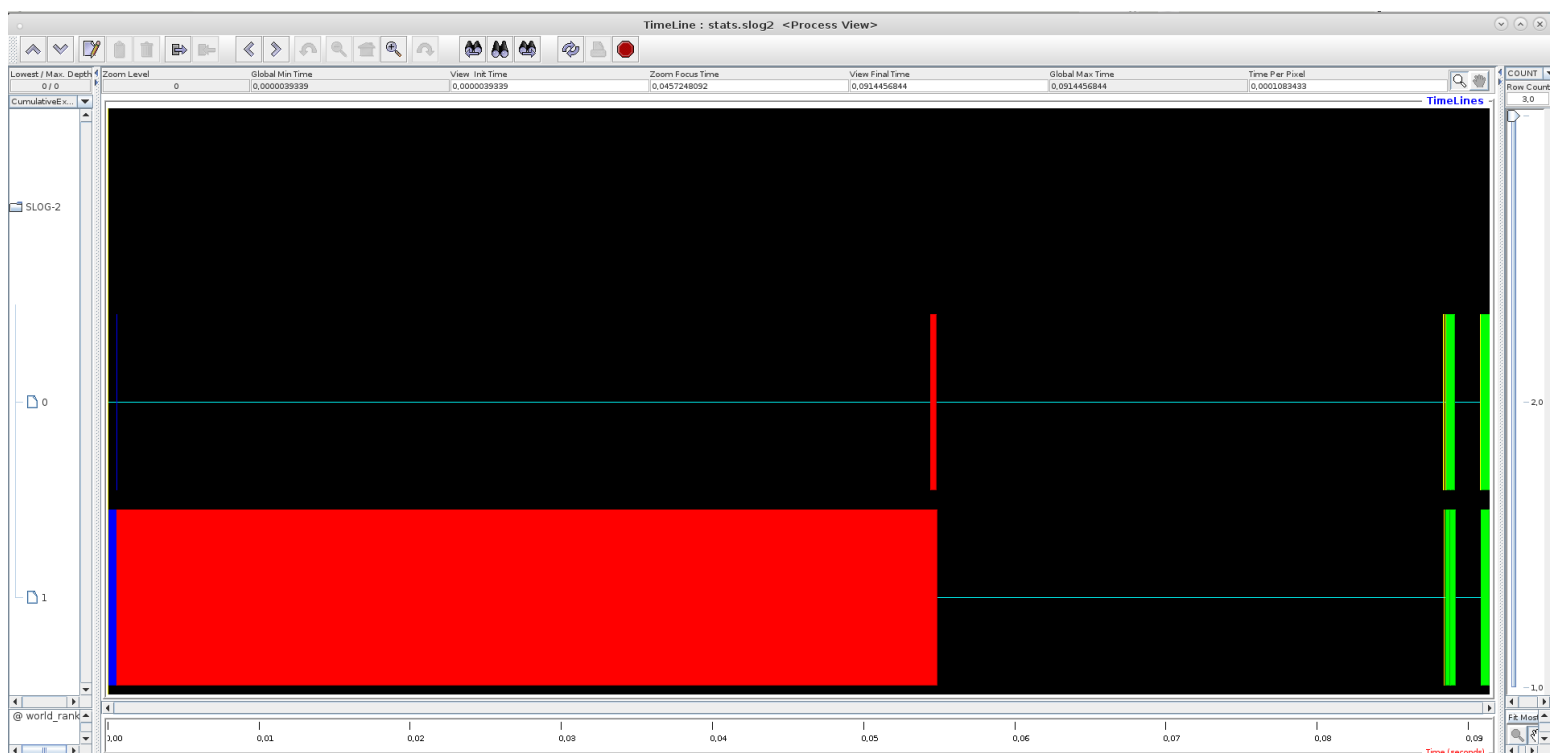
Rys. 2 Funkcje i ich kolory



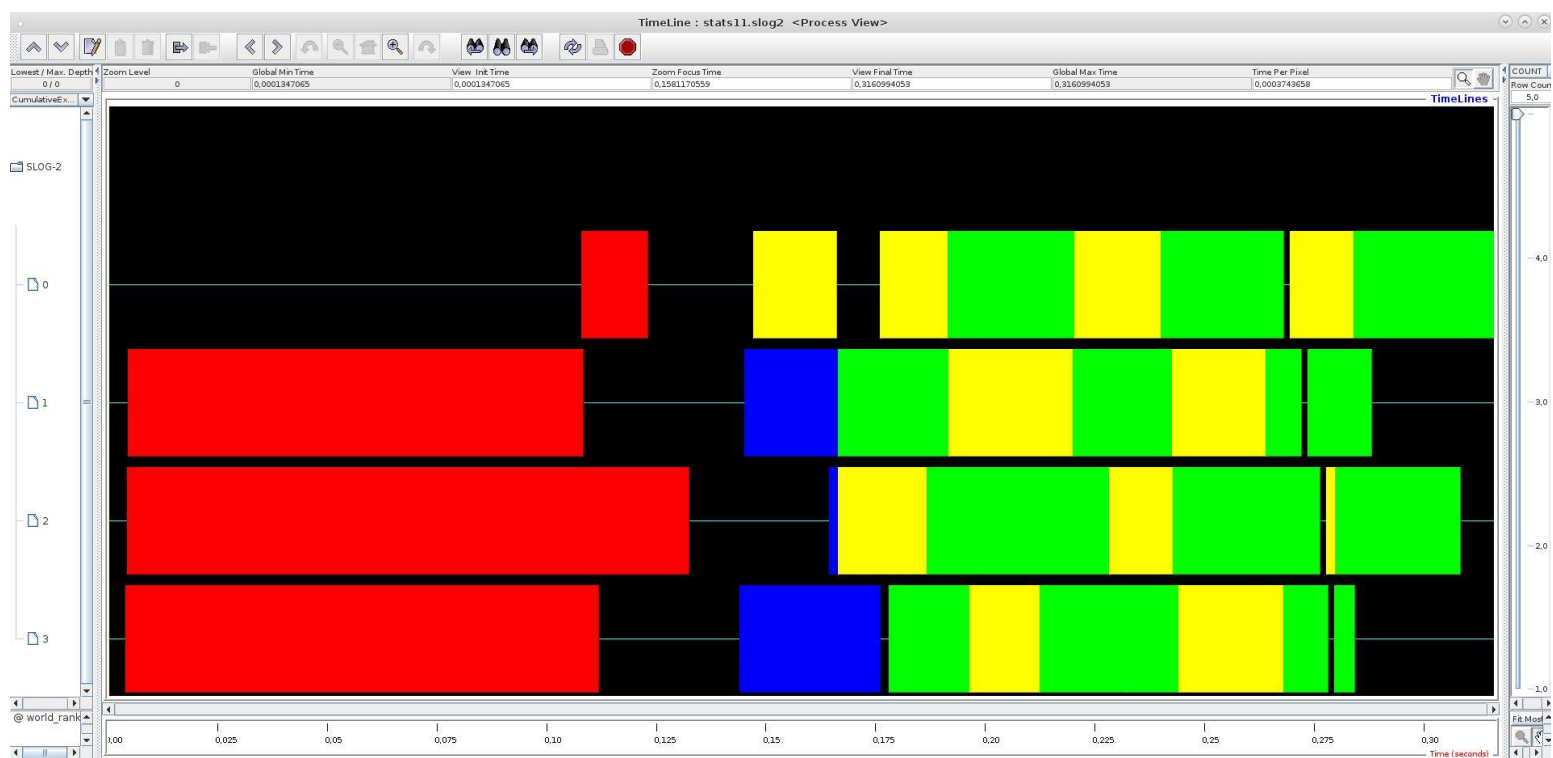
Profilowanie zrealizowano na tej samej ilości danych dla 1, 2, 4 i 8 procesów co przedstawiono na rysunkach Rys. 3, Rys. 4, Rys. 5, Rys. 6



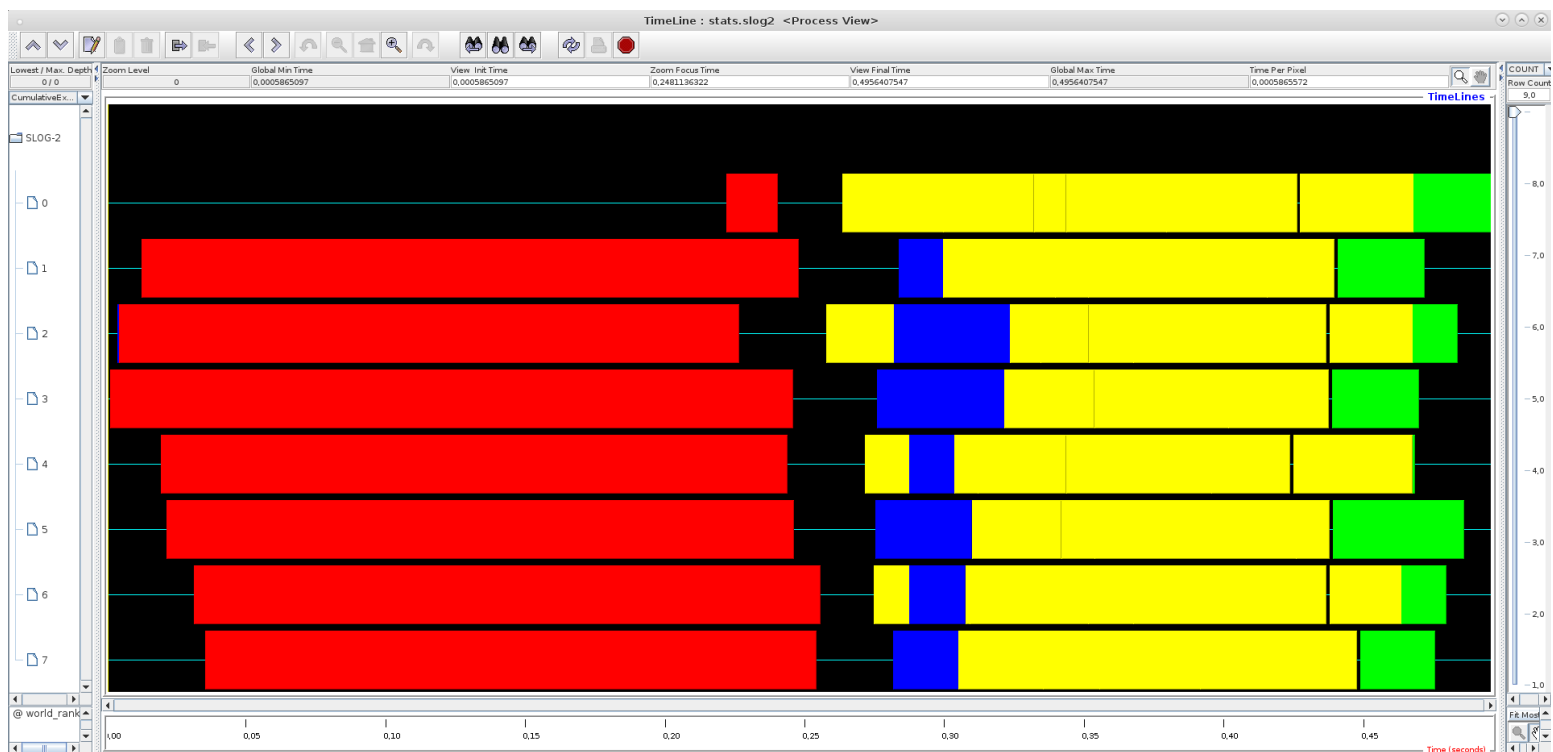
Rys.3 Wykres czasowy aplikacji dla 1 procesu



Rys. 4 Wykres czasowy aplikacji dla 2 procesów



Rys. 5 Wykres czasowy aplikacji dla 4 procesów

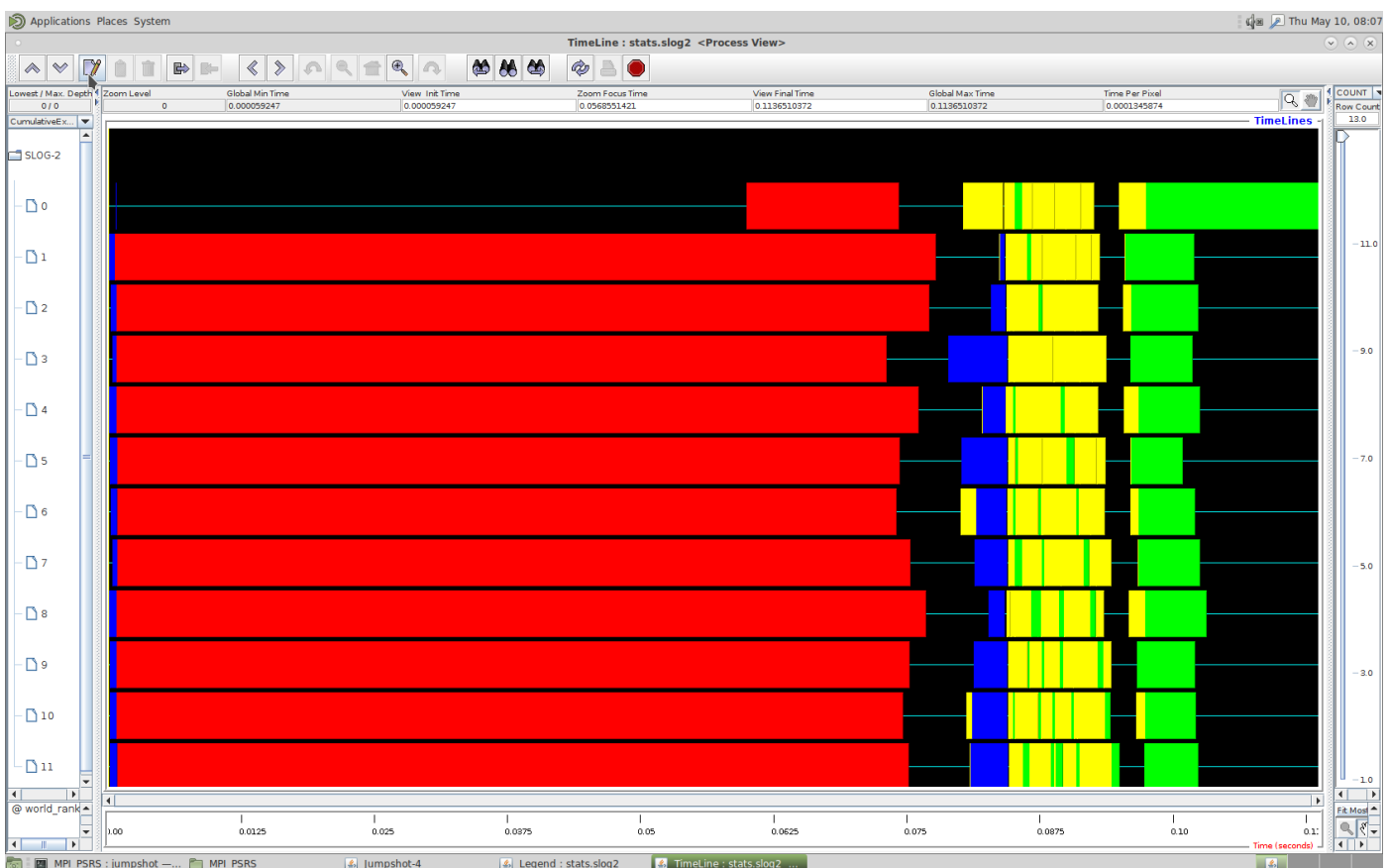


Rys. 6 Wykres czasowy aplikacji dla 8 procesów

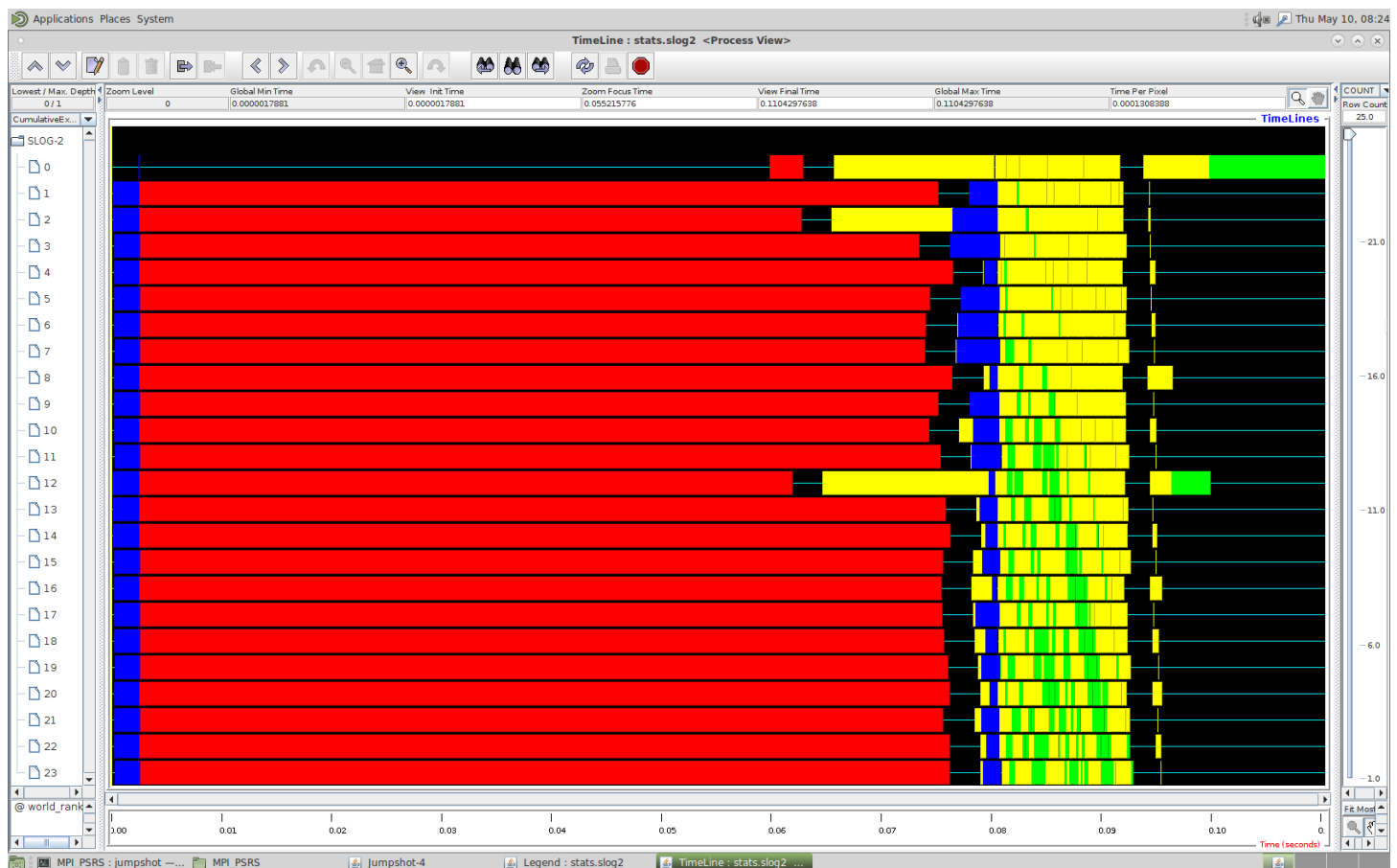
Na rysunkach 3,4, 5 oraz 6 można zaobserwować, że najlepszy wynik został osiągnięty dla liczby procesów równej 2. Wynika to z najbardziej optymalnego zestawu ilości przesyłań komunikatów pomiędzy procesami, a ilością rdzeni procesora.

Dla jednego procesu narzutu komunikacyjnego nie obserwujemy, ale samo sortowanie trwa długo. Z kolei dla 8 procesów same obliczenia trwają krótko, jednak komunikacja pomiędzy wszystkimi procesami trwa na tyle długo, że wynik jest gorszy niż sortowanie lokalnie. Podobnie w przypadku 4 procesów.

Zrealizowano profilowanie również dla kilku węzłów, co przedstawiono na rysunkach Rys. 7, Rys. 8



Rys. 7 Wykres czasowy aplikacji dla 12 procesów uruchomionych na 12 węzłach



Rys. 8 Wykres czasowy aplikacji dla 24 procesów  
na 12 węzłach

Odpowiedni dobór ilości procesów/węzłów dla zadania zależy od ilości danych do posortowania, specyfikacji sprzętu i znajomości infrastruktury klastra.

Problem sortowania jest wrażliwy i wybierając odpowiednie parametry należy pamiętać o wszystkich elementach, które mogą opóźnić proces sortowania równoległego. Niejednokrotnie może okazać się korzystniej sortować dane lokalnie na jednym węźle niż próbować takie sortowanie zrównoleglać.

Dodatkowe wyniki zostały przedstawione w rozdziale 5.

## 5. Przykład wyjścia z programu

Na rys. 9 przedstawiono wyjście z programu dla danych z pliku przekazywanego jako parametr make'a oraz dla 4 procesów.

```
Starting
Number of processes 4
Size of whole data to process: 50
Table values to sort:
7 16 3 84 20 17 47 40 3 69 92 96 74 32 53 43 81 73 31 13 91 88 21 98 54 52 36 67 49 78 95 84 75 46 88 41 76 2 83 4 8 77 49 93 40 80 97 18 26 36
Lengths of data to be send to processes...
[process-0] myDataLength=12, myDataStarts=0
[process-1] myDataLength=12, myDataStarts=12
[process-2] myDataLength=12, myDataStarts=24
[process-3] myDataLength=14, myDataStarts=36
[process-0] [class-0] class start=0, class length=5
[process-0] [class-1] class start=5, class length=3
[process-0] [class-2] class start=8, class length=1
[process-0] [class-3] class start=9, class length=3
[process-1] [class-0] class start=0, class length=1
[process-1] [class-1] class start=1, class length=4
[process-1] [class-2] class start=5, class length=3
[process-1] [class-3] class start=8, class length=4
[process-2] [class-0] class start=0, class length=0
[process-2] [class-1] class start=0, class length=4
[process-2] [class-2] class start=4, class length=5
[process-2] [class-3] class start=9, class length=3
[process-3] [class-0] class start=0, class length=4
[process-3] [class-1] class start=4, class length=4
[process-3] [class-2] class start=8, class length=3
[process-3] [class-3] class start=11, class length=3
Sorted data:
2 3 3 4 7 8 13 16 17 18 20 21 26 31 32 36 36 40 40 41 43 46 47 49 49 52 53 54 67 69 73 74 75 76 77 78 80 81 83 84 84 88 88 91 92 93 95 96 97 98
Clock time (seconds) = 0.000820

The end
```

Rys. 9 Przykładowe wyjście z

make run N=4 FILE==sample/input\_50.data

Wyjście z programu prezentuje:

- liczbę procesów,
- rozmiar danych do sortowania
- tablice wejściową z danymi
- rozmiar danych do posortowania na danym procesie wraz z indeksem startowym
- podział danych na danym procesie ze względu na pivoty wraz z długością danych w danej partycji

- posortowane dane
- czas działania algorytmu

Na rysunkach rys. 10, rys. 11 oraz rys. 12 niektóre komentarze zostały wyłączone dla większej ilości danych w celu zwiększenia czytelności.

```
ignacy@debian:~/MPI_PSR$ make run N=1 ARGS="-f sample/input_500000.data"
mpicc PSRS.o utilities.o profiles.o -o PSRS.out -I/opt/nfs/mpe2-2.4.9b/include -L/opt/nfs/mpe2-2.4.9b/lib -std=c11 -lmpe -lm -lpthread
/opt/nfs/mpich-3.2/bin/mpirun -n 1 ./PSRS.out -f sample/input_500000.data
Starting
Number of processes 1
Size of whole data to process: 500000
Sorted data:

Clock time (seconds) = 0.071986

The end
Enabling the Default clock synchronization...
ignacy@debian:~/MPI_PSR$ make run N=2 ARGS="-f sample/input_500000.data"
mpicc PSRS.o utilities.o profiles.o -o PSRS.out -I/opt/nfs/mpe2-2.4.9b/include -L/opt/nfs/mpe2-2.4.9b/lib -std=c11 -lmpe -lm -lpthread
/opt/nfs/mpich-3.2/bin/mpirun -n 2 ./PSRS.out -f sample/input_500000.data
Starting
Number of processes 2
Size of whole data to process: 500000
Sorted data:

Clock time (seconds) = 0.036937

The end
Enabling the Default clock synchronization...
ignacy@debian:~/MPI_PSR$
```

Rys. 10 Przykładowe wyjścia z programu

```

4migas@stud206-011:~/MPI_PSR$ make run N=$(( 2 * $(cat nodes | wc -l) )) FILE=sample/input_500000.data NODES=nodes
/opt/nfs/mpich-3.2/bin/mpirun -n 24 -f nodes ./PSRS.out -f sample/input_500000.data

Starting
Number of processes 24
Size of whole data to process: 500000

The end
Lengths of data to be send to processes...
Sorted data:

Clock time (seconds) = 0.050891

The end
Enabling the Default clock synchronization...
4migas@stud206-011:~/MPI_PSR$ make run N=$(( 1 * $(cat nodes | wc -l) )) FILE=sample/input_500000.data NODES=nodes
/opt/nfs/mpich-3.2/bin/mpirun -n 12 -f nodes ./PSRS.out -f sample/input_500000.data

Starting
Number of processes 12
Size of whole data to process: 500000

The end
Lengths of data to be send to processes...
Sorted data:

Clock time (seconds) = 0.052119

The end
Enabling the Default clock synchronization...

```

Rys. 11 Przykładowe wyjścia z programu

```

4migas@stud206-011:~/MPI_PSR$ make run N=$(( 4 * $(cat nodes | wc -l) )) FILE=sample/input_500000.data NODES=nodes
/opt/nfs/mpich-3.2/bin/mpirun -n 48 -f nodes ./PSRS.out -f sample/input_500000.data

Starting
Number of processes 48
Size of whole data to process: 500000

The end
Lengths of data to be send to processes...
Sorted data:

Clock time (seconds) = 0.119132

The end
Enabling the Default clock synchronization...
4migas@stud206-011:~/MPI_PSR$ make run N=1 FILE=sample/input_500000.data NODES=nodes
/opt/nfs/mpich-3.2/bin/mpirun -n 1 -f nodes ./PSRS.out -f sample/input_500000.data

Starting
Number of processes 1
Size of whole data to process: 500000

The end
Lengths of data to be send to processes...
Sorted data:

Clock time (seconds) = 0.089736

The end

```

Rys. 12 Przykładowe wyjścia z programu

Na rysunkach rys. 10, rys. 11 oraz rys. 12 najważniejszy jest czas wykonania programu, ilość węzłów/procesów.

## 6. Źródła

*Quinn, zad. 15.5, Grama, Gupta, rozdz. 13*

<http://cswb.cs.wfu.edu/bigiron/LittleFE-PSRS/build/html/PSRSalgorithm.html>

<http://cswb.cs.wfu.edu/bigiron/LittleFE-PSRS/build/html/PSRSAppendixI.html>