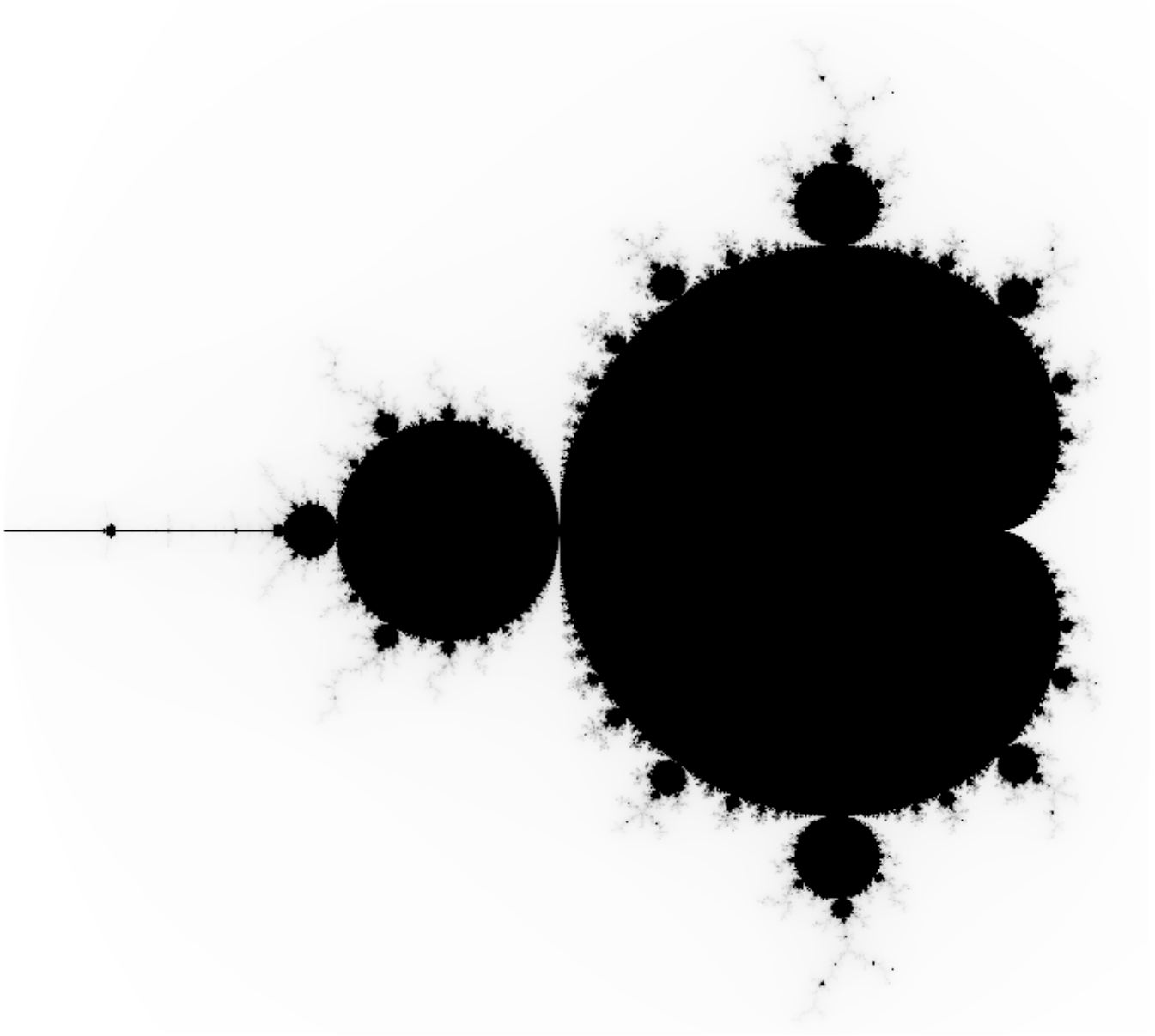
Przykład MPI: zbiór Mandelbrota



Zbiór Mandelbrota

- Zbiór punktów na płaszczyźnie zespolonej, które są quasi-stabilne, kiedy są iterowane funkcją:

$$z_{k+1} = z_k^2 + c$$

gdzie z_{k+1} wartość zespolona w k+1 iteracji, z_k wartość w poprzedniej iteracji (z_0=0), c badany punkt na płaszczyźnie zespolone.

- Stosując zasady mnożenia liczb zespolonych dostajemy wzory na update liczby z:

$$z_{real} = z_{real}^2 - z_{imag}^2 + c_{real}$$
$$z_{imag} = 2z_{real}\, z_{imag} + c_{imag}$$

gdzie $_{imag}$ i $_{real}$ oznaczają część całkowitą i urojoną liczby zespolonej. Iteracje prowadzi się do momentu, gdy:

$$z_{real}^2 + z_{imag}^2 > 4$$

- co oznacza że |z| rozbiegnie się do ∞, lub gdy osiągniemy maksymalną liczbę iteracji.
- Jasność piksela równa 255 - liczba iteracji

Zapis obrazu na dysk

```
void WritePGM(char *fname,int Width,int Height)
{
   FILE *file=fopen(fname,"wt");
   if (fname==NULL) return;
   fprintf(file,"P5 %d %d 255\n",Width,Height);
   for(int i=0;i<Height;i++)
      fwrite(Image[i],sizeof(char),Width,file);
   fclose(file);
}
```

- Format .pgm (portable graymap – obraz szary) jeden najprostszych formatów graficznych.
- Nagłówek postaci **`P5 szerokość wysokość maks_natężenie`** ; po nim bajty z których każdy z nich specyfikuje szarość jednego piksela.
- Standard MPI-2 zawiera własny interfejs do równoległego wejścia-wyjścia, być może o tym później.

Funkcja main

```
void CalcImage(int Width,int Height)
{
   for(int i=0;i<Height;i++)
      CalcImgRow(Image[i],i,Width,Height);
}

const int imgWidth=4096;
const int imgHeight=4096;

int main()
{
   AllocMem(imgWidth,imgHeight);
   CalcImage(imgWidth,imgHeight);
   WritePGM("mandel.pgm",imgWidth,imgHeight);
   FreeMem(imgWidth,imgHeight);
   return 0;
}
```

- Po kolei obliczamy wartości natężenia dla wszystkich wierszy.

Alokacja i dealokacja pamięci

```
// dynamicznie alokowana tablica wskaźników
char **Image;

void AllocMem(int Width,int Height)
{
   Image=new char *[Height];
   for(int i=0;i<Height;i++)
      Image[i]=new char[Width];
}
// Analogicznie funkcja FreeMem zwalnia pamięć
```

- Obraz jest tablicą wskaźników o rozmiarze równym wysokości obrazu.
- i-ty element tablicy jest wskaźnikiem wskazującym na dane i-tego wiersza.
- i-ty wiersz to tablica liczb typu char o rozmiarze równym szerokości obrazu.
- W efekcie `Image[i][j]` zawiera jasność piksela o współrzędnych (i,j).

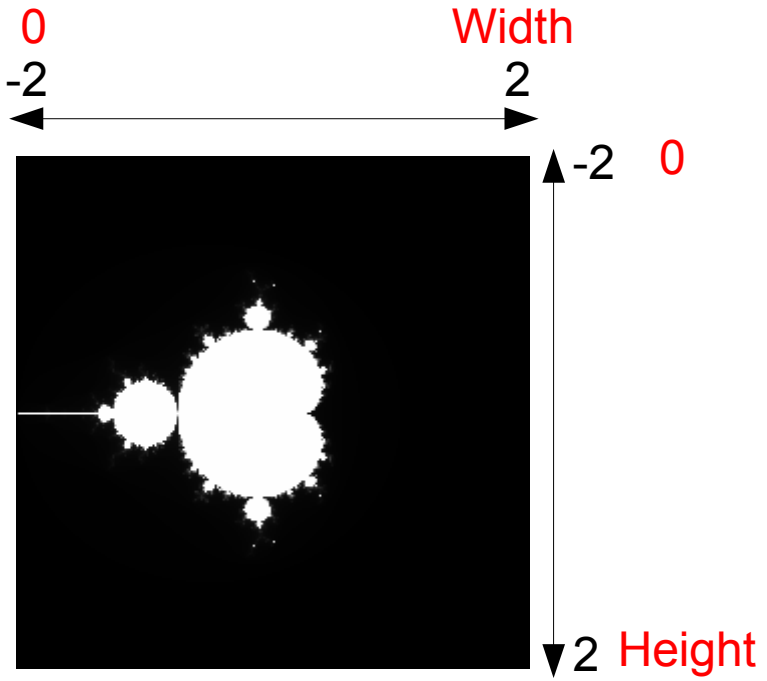
Obliczenia wartości piksela

```
int CalcPixel (double x,double y)
{
   double zre,zim,zoldre=0.0,zoldim=0.0;
   int i;
   for(i=0;i<255;i++) {
      zre=zoldre*zoldre-zoldim*zoldim+x;
      zim=2*zoldre*zoldim+y;
      if (zim*zim+zre*zre>=4.0) return i;
      zoldre=zre; zoldim=zim;
   }
   return i;
}
```

- Funkcja sprawdza czy liczba zespolona x+y*i należy do zbioru Mandelbrota. Maksymalna liczba iteracji 255; jeżeli jest osiągnięta to przyjmujemy, że liczba jest elementem zbioru.
- Uwaga 1 : mając dane x i y ***nie wiemy, ile będziemy potrzebować iteracji***.
- Uwaga 2: widać, że ***obliczenia dla jednego punktu wykonywane są niezależnie od jego sąsiadów***.

Obliczenie wartości linii pikseli

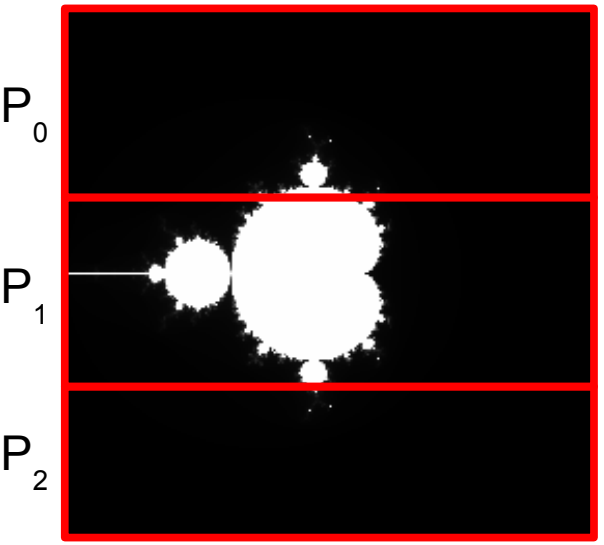
```
void CalcImgRow(char *Row,int num,int Width,int Height)
{
   double y=-2.0+4.0*num/Height;
   for(int i=0;i<Width;i++) {
      double x=-2.0+4.0*i/Width;
      // Jasność piksela proporcjonalna do liczby iteracji pętli
      Row[i]=(char)(255-CalcPixel(x,y));
   }
}
```



- Przeskalowanie współrzędnych
  - pł. rzeczywista [0,Width]> -> [-2,2]
  - pł. urojona [0,Height]> -> [-2,2]

Pierwsze podejście do algorytmu równoległego

- Zauważmy, że wartość każdego piksela jest obliczana niezależnie od pozostałych.
- Jeden proces nadrzędny zbiera dane od procesów obliczeniowych - podrzędnych.
- Podzielmy obraz wierszami na liczbę części równą liczbie procesów podrzędnych.
- Każdy proces podrzędny przesyła swoje dane procesowi nadrzędnemu, który zapisuje wyniki do pliku.
- Ale uwaga: Liczba iteracji funkcji CalcPixel zależy od natężenia piksela.
- W przypadku podziału pracy obok P_1 musi wykonać dużo więcej obliczeń niż P_0 i P_2. - jest znacznie więcej jasnych pikseli.
- Po zakończeniu pracy P_0 i P_2 czekają bezproduktywnie na P_1. Na pewno nie uda się osiągnąć trzykrotnego przyspieszenia.



Funkcja main - identyczna dla obydwu rozwiązań

```
int main(int argc,char *argv[])
{
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   MPI_Comm_size(MPI_COMM_WORLD,&size);
   if (size<2) {
      printf("At least 2 processes are needed\n");
      MPI_Finalize();
      exit(-1);
   }
   if (rank==0) {
      AllocMem(imgWidth,imgHeight);
      Master();
      WritePGM("mandel.pgm",imgWidth,imgHeight);
      FreeMem(imgWidth,imgHeight);
   } else
     Slave();
   MPI_Finalize();
}
```

- Proces nadrzędny wykonuje funkcje `Master` po czym zapisuje obraz na dysk
- Procesy podrzędne wykonują funkcję `Slave`.

Proces podrzędny – alokacja statyczna

```
void Slave()
{
   int Slaves=size-1;
   int Slave=rank-1;
   int NRows=imgHeight/Slaves;
   int StartRow=Slave*NRows;
   int Remainder=imgHeight%Slaves;
   if (Slave<Remainder) {
      NRows++;
      StartRow+=Slave;
   } else
      StartRow+=Remainder;
   for(int i=StartRow;i<StartRow+NRows;i++) {
      CalcImgRow(Row,i,imgWidth,imgHeight);
      MPI_Send(Row,imgWidth,MPI_CHAR,0,i,MPI_COMM_WORLD);
   }
}
```

- Równomierny podział wierszy na procesy podrzędne.
- Dodatkowe korekty na wypadek gdy liczba wierszy nie dzieli się przez liczbę procesów podrzędnych.
- Po obliczeniu danych proces podrzędny wysyła je do procesu nadrzędnego, komunikat ten otrzymuje etykietę równą numerowi wiersza – tak aby proces nadrzędny wiedział, który wiersz otrzymuje.

Proces nadrzędny - alokacja statyczna

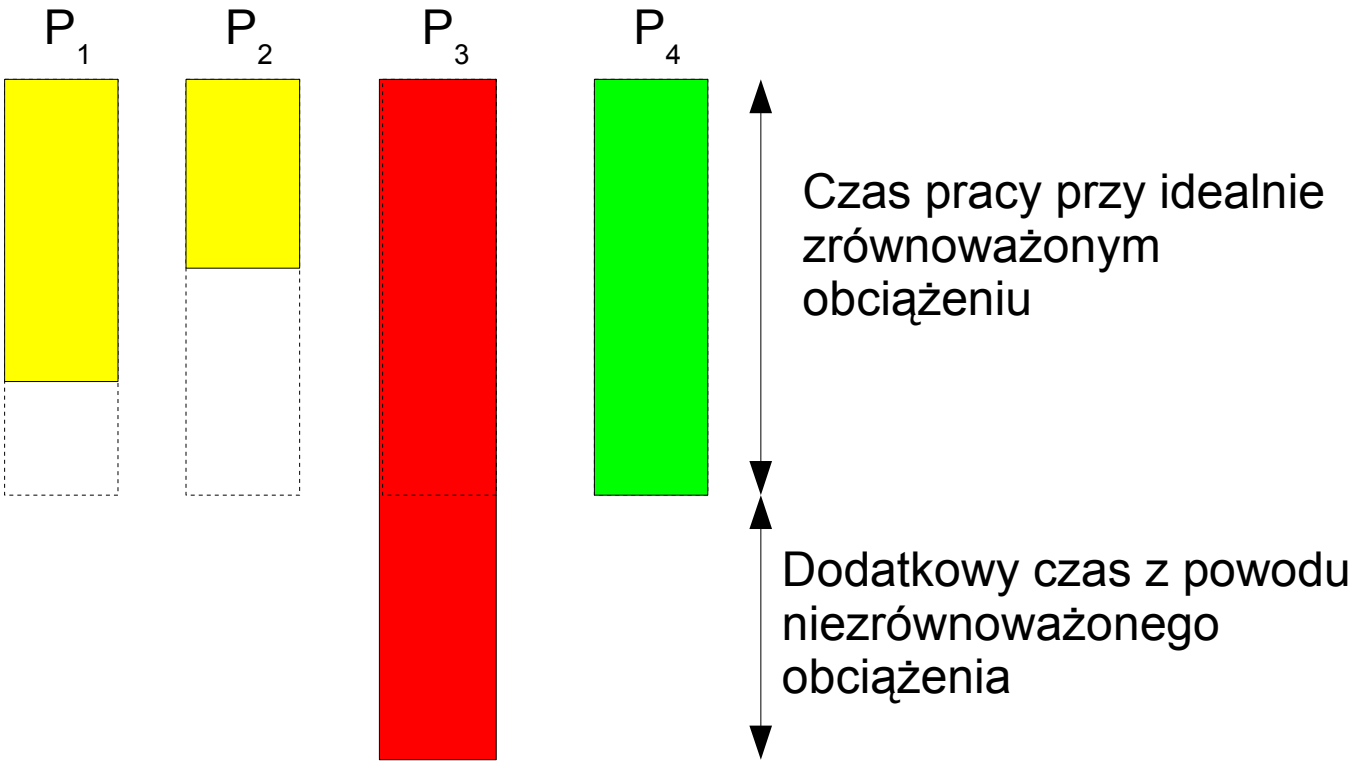
```
void Master(){
   MPI_Status Status;
   int Slaves=size-1;
   for (int i=1;i<size;i++) {
      int Slave=i-1;
      int NRows=imgHeight/Slaves;
      int StartRow=Slave*NRows;
      int Remainder=imgHeight%Slaves;
      if (Slave<Remainder) {
         NRows++;
         StartRow+=Slave;
      } else
      StartRow+=Remainder;

      for(int j=0;j<NRows;j++) {
         MPI_Recv(Row,imgWidth,MPI_CHAR,i,MPI_ANY_TAG,MPI_COMM_WORLD,
         &Status);
         int ReceivedRow=Status.MPI_TAG;
         memcpy(Image[ReceivedRow],Row,sizeof(char)*imgWidth);
      }
   }
}
```

- Proces nadrzędny odpiera wiersze obrazu od kolejnych wierszy nadrzędnych.
- Każdy wiersz jest przesyłany odrębnym komunikatem

Alokacja statyczna - uwagi

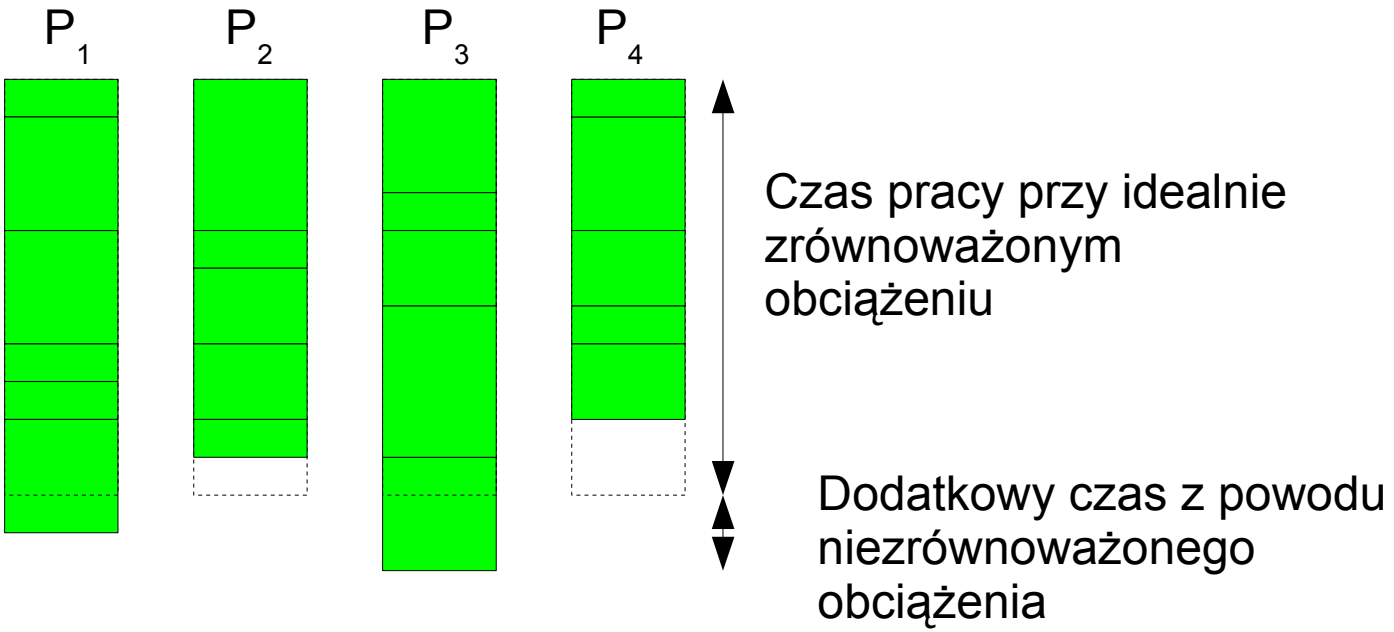
- Rozwiązanie można znacznie zoptymalizować.
- Przesyłanie bloku wierszy jednym komunikatem.
- Proces nadrzędny również biorący udział w obliczeniach (obecnie zajmuje się wyłącznie odbiorem komunikatów).
- Obecna forma programu ma na celu najbardziej bezpośrednie porównanie z alokacją dynamiczną.

Statyczne równoważenie obciążenia



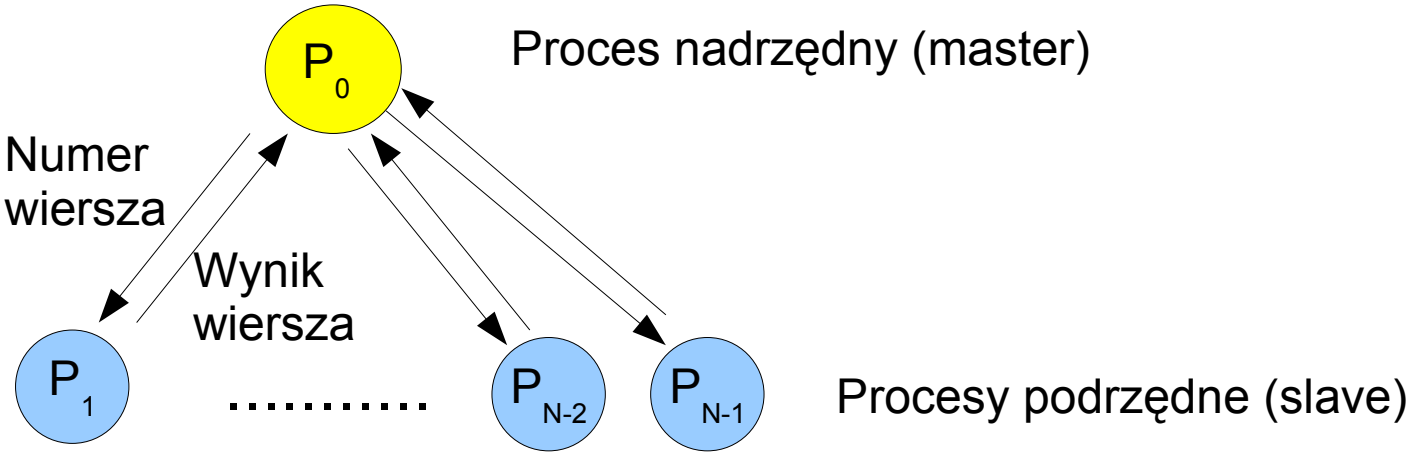
- Cztery procesory, podział zadania obliczeniowego ***jeszcze przed przystąpieniem do pracy***.
- Podział na cztery różne części.
- Czas obliczeń większy niż wynika z czterokrotnego przyspieszenia (trzy procesory czekają na najbardziej obciążony)

Dynamiczne równoważenie obciążenia



- Dynamiczny podział zadania obliczeniowego ***w trakcie jego wykonywania***.
  - Każdy procesor, gdy staje się wolny zgłasza zapotrzebowanie i otrzymuje niewielki fragment pracy. Dzieje się tak do momentu gdy zadanie zostanie wykonane.
- Podobnie jak w poprzednim wypadku nie znamy rozmiaru fragmentu, ale ponieważ fragmenty są mniejsze to i mniejsze jest niezrównoważenie.

Model master-slave



- Aplikacja MPI składa się z N procesów, (co najmniej dwóch)
- Proces o numerze 0 (master) – zajmuje się rozsyłaniem numerów wierszy do procesów podrzędnych a następnie odebraniem od nich danych wiersza.
  - Wada: proces master nie zajmuje się obliczeniami -> Maksymalne przyspieszenie dla trzech procesów: 2 , dla czterech: 3.
- Procesy o numerach 1,...,N-2,N-1 zajmują się obliczaniem pikseli wierszy, których numery przesłał im proces nadrzędny.

Proces podrzędny – alokacja dynamiczna

```
void Slave()
{
   int RowNum;
   MPI_Status Status;
   while(1) {
      MPI_Recv(&RowNum,1,MPI_INT,0,0,MPI_COMM_WORLD,&Status);
      if (RowNum==-1)
         return;
      CalcImgRow(Row,RowNum,imgWidth,imgHeight);
      MPI_Send(Row,imgWidth,MPI_CHAR,0,RowNum,MPI_COMM_WORLD);
   }
}
```

- Proces podrzędny odbiera od procesu nadrzędnego numer wiersza.
- Numer wiersza równy -1 oznacza koniec pracy.
- Po obliczeniu danych wiersza wysyła je do procesu nadrzędnego, komunikat ten otrzymuje etykietę równą numerowi wiersza – tak aby proces nadrzędny wiedział, który wiersz otrzymuje.

Proces nadrzędny - problemy

- Na początku należy zatrudnić wszystkie procesy podrzędne – każdemu z nich wysyłamy numer wiersza. Procesy o numerach 1,..,N-1 (numer 0 to master) otrzymują numery wierszy 0,..,N-2.
  - Założenie: Liczba procesów nadrzędnych jest mniejsza niż liczba wierszy.
- Następnie odbieramy od procesu nadrzędnego (dowolnego) dane wiersza i
  - Jeżeli są nowe wiersze to wysyłamy kolejny
  - Jeżeli nie ma nowych wierszy to wysyłamy numer -1 oznaczający koniec pracy
- Należy zadbać aby wszystkie procesy podrzędne otrzymały sygnał -1.
  - W tym celu w zmiennej `ActiveSlaves` zliczamy aktywne procesy podrzędne

Proces nadrzędny - alokacja dynamiczna (1)

```
void Master()
{
   int NextRow,ActiveSlaves;
   MPI_Status Status;

   // rozesłanie wszystkim slave'om początkowych numerów wierszy
   for(int i=0;i<size-1;i++)
      MPI_Send(&i,1,MPI_INT,i+1,0,MPI_COMM_WORLD);
   // Liczba zatrudnionych procesów podrzędnych
   ActiveSlaves=size-1;
   // Numer następnego wiersza do wysłania
   NextRow=rank;
   do {
      // Odbierz wiersz danych od slave'a
      //(z dowolnego źródła i o dowolnej etykiecie)
      MPI_Recv(Row,imgWidth,MPI_CHAR,MPI_ANY_SOURCE,MPI_ANY_TAG,
      MPI_COMM_WORLD,&Status);
      ActiveSlaves--;
      // Numer odebranego wiersza to etykieta komunikatu
      int ReceivedRow=Status.MPI_TAG;
```

Proces nadrzędny - alokacja dynamiczna (2)

```
      // Który slave przyszłał wiersz ?
      int Slave=Status.MPI_SOURCE;
      // skopiuj wiersz do macierzy Image
      memcpy(Image[ReceivedRow],Row,sizeof(char)*imgWidth);
      // Czy są jeszcze wiersze do przesłania ?
      if (NextRow<imgHeight) {
         MPI_Send(&NextRow,1,MPI_INT,Slave,0,MPI_COMM_WORLD);
         ActiveSlaves++;
         NextRow++;
      } else {
      // Nie, wyślij -1 tzn sygnał zakończenia procesu podrzędnego
         int minusOne=-1;
         MPI_Send(&minusOne,1,MPI_INT,Slave,0,MPI_COMM_WORLD);
      }
   // Powtarzaj jak długo są akwywne procesy podrzędne
   } while (ActiveSlaves>0);
}
```

Szczegółowa analiza (1)

- *r* - jest liczbą wierszy, *u* - liczbą pikseli w wierszu, *p* liczbą procesów, t_w - średnim czasem generowania jednego wiersza. Przyjmujemy model czasu przesłania komunikatu pokazany na drugim wykładzie.
- Każdy numer wiersza musi być wysłany do procesora nadrzędnego, co łącznie zajmuje:

$$t_{c1} = r * (t_S + 4 * t_B)$$

gdzie t_S jest czasem opóźnienia, a t_B czasem transmisji jednego bajtu. Również obliczone dane każdego wiersza, muszą być przesłane z powrotem, co łącznie zajmuje:

$$t_{c2} = r * (t_S + u * t_B)$$

- Przyjmujemy że dynamiczne równoważenie obliczeń jest tak dobre, że dzieli czas obliczania danych wierszy jest dzielony przez *p-1*. (Jeden procesor nie uczestniczy w obliczeniach). Łączny czas obliczeń jest w przybliżeniu równy:

$$t_o = \frac{r * t_w}{p - 1}$$

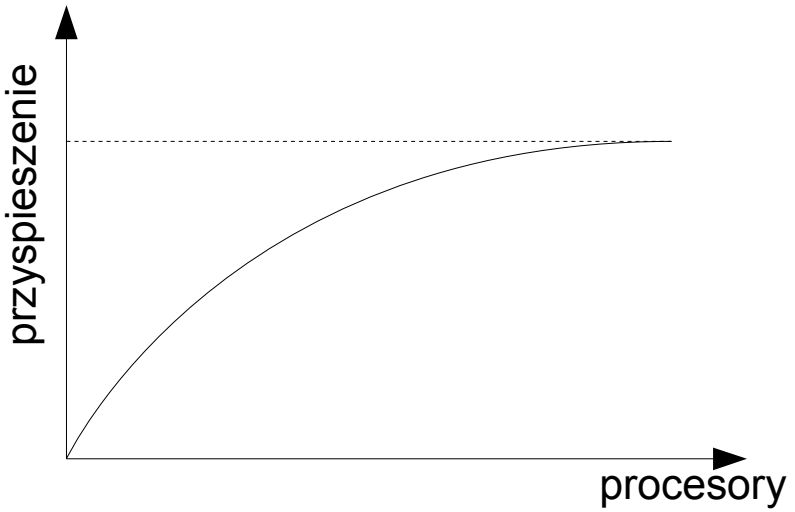
Szczegółowa analiza (2)

- Łączny czas pracy jest równy:

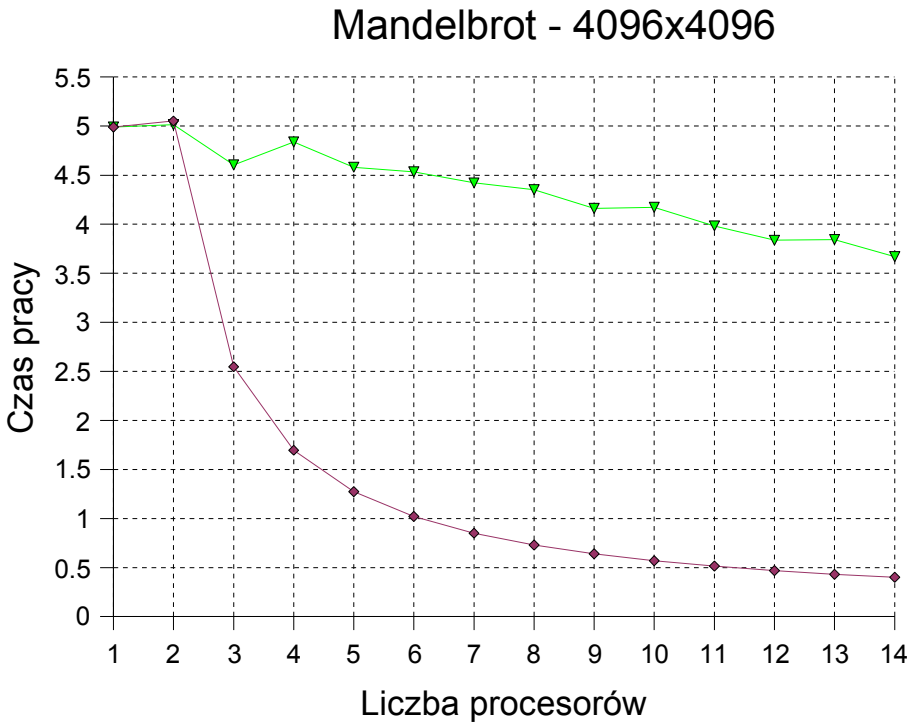
$$t_p = t_{c1} + t_{c2} + t_o$$

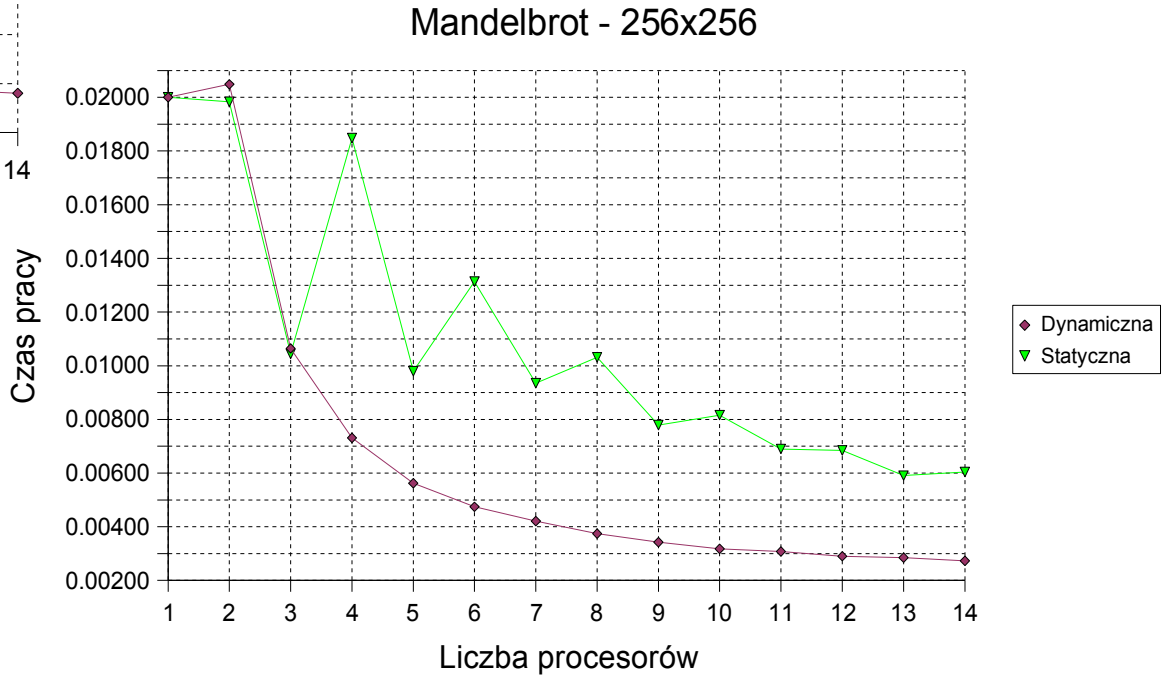
- Zauważmy, że t_{c1} i t_{c2} nie zależą od p, natomiast t_o jest dzielony w przybliżeniu przez p. Mamy więc sytuację odpowiadającą prawu Amdahla (pierwszy wykład), w którym przyspieszenie jest ograniczone z góry przez pewną wartość.

- Gdyby nie czas komunikacji, przyspieszenie byłoby równe *p-1*.
- Dla alokacji statycznej na pewno nie będzie *p-1* krotnego przyspieszenia obliczeń (wzór na t_o)



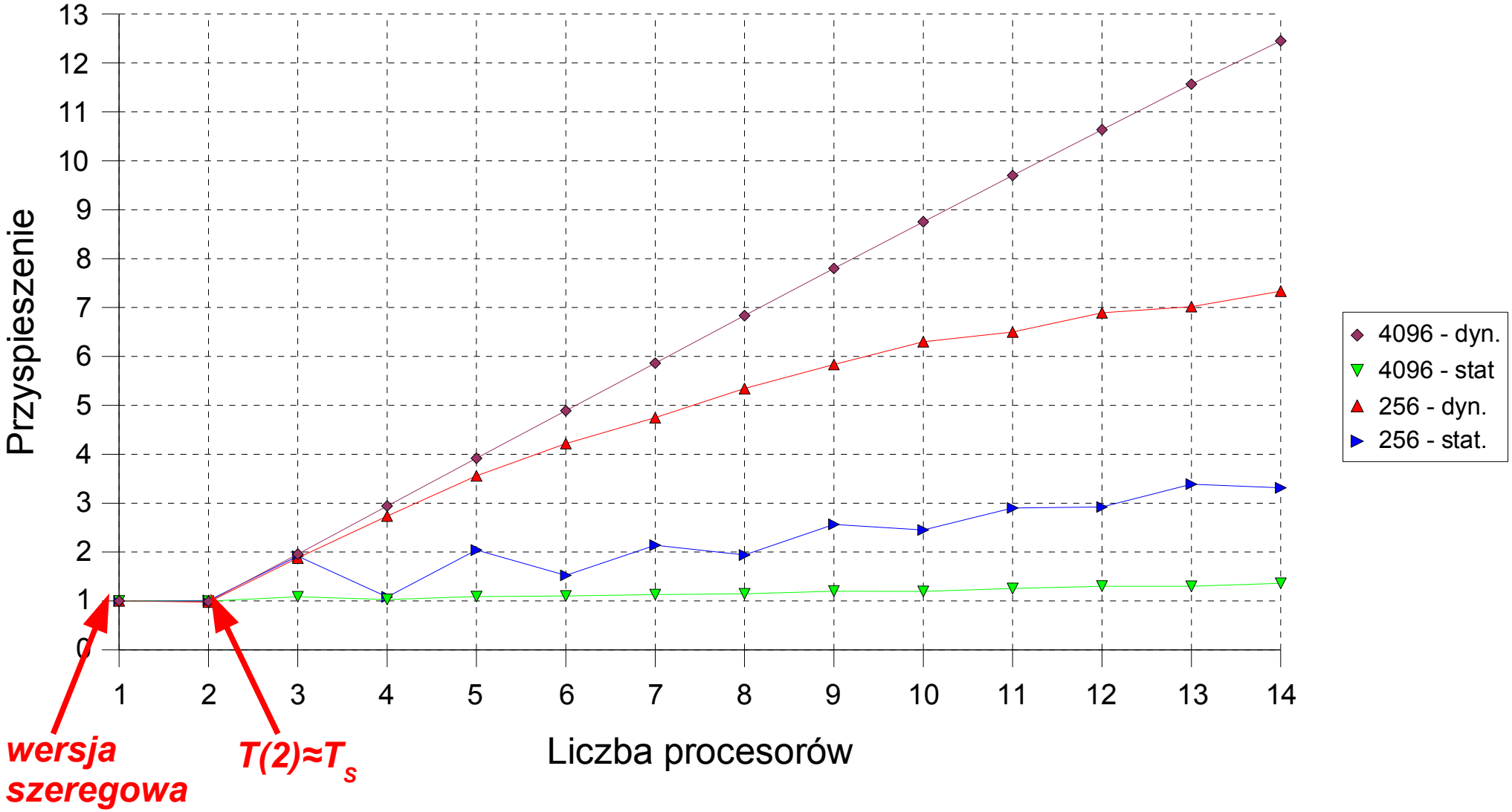
Eksperymenty - czas pracy





- Dwa wymiary obrazu 4096x4096 oraz 256x256; klaster Mordor; sieć Infiniband 10Gb/s
- Czas dla dwóch procesorów ≈ czas wersji szeregowej

Przyspieszenie



- Przypomnienie: przyspieszenie *S(p)* obliczamy dzieląc czas wersji szeregowej przez czas wersji równoległej wykorzystującej *p* procesorów.

Podsumowanie

- Wersja dynamiczna charakteryzuje się o wiele lepszą skalowalnością niż wersja statyczna.
- Dla wersji dynamicznej skalowalność jest gorsza w przypadku mniejszego zbioru danych.
  - Czas obliczeń mniej dominuje nad czasem komunikacji.
- Dla wersji statycznej jest odwrotnie.
  - Nie jest łatwo to wytłumaczyć (wpływ protokołu przesyłania długich komunikatów ?)
- Podobnego raportu oczekuję w Państwa projekcie.
- Wersje statyczną można dalej optymalizować
  - Proces master również prowadzi obliczenia.
  - Przesyłanie danych kilku wierszy jednym komunikatem.
- Programy do ściągnięcia z mojej strony **aragorn.pb.bialystok.pl/~wkwedlo**