Grupa 1	Kordian Kurdziel, Mateusz Maciejak	24/10/2018

Zadanie 2 - Rozmycie Gaussa w MPI

Zadaniem programu było rozmycie obrazu podanego na wejściu za pomocą algorytmu Gaussa z maską 5x5. W celu poprawy wydajności programu do zrównoleglenia jego działania należało wykorzystać standard MPI. Program przyjmuje dwa argumenty, obraz wejściowy oraz obraz wyjściowy. Liczbę procesów na jakich mają zostać wykoanane obliczenia należy podać po komendzie mpirun używając flagi -n.

Do obsługi operacji na zdjęciu została wykorzystana biblioteka OpenCV. Poniższy fragment kodu prezentuje inicjalizację programu opartego o MPI oraz pobranie identyfikatora aktualnego procesu oraz liczby procesów z jaką został uruchomiony program. Jest to potrzebne do zrównloleglenia obliczeń w dalszej części programu.

```
1     MPI_Init(&argc, &argv);
2     MPI_Comm_rank (commmunicate, &rank);
3     MPI_Comm_size(commmunicate, &numberOfProcesses);
```

Funkcja MPI_Init inicjuje mechanizm MPI. Następnie funkcja MPI_Comm_rank inicjuje do zmiennej rank aktualnie używany id procesu. Funkcja MPI_Comm_size ustawia w zmiennej numberOfProcesses ilość procesów. Obrazek jest dzielony na równe części i rozdzielany przez proces główny na do oddzielnych procesów, które w separacji rozmywają kolejne partie obrazu. Do pocięcia obrazka wykorzystywana jest funkcja biblioteki OpenCv Rect.

```
1   cv::Rect myROI(x, y, width, height);
2    cv::Mat croppedRef(picture, myROI);
3    cv::Mat cropped;
```

Do komunikacji między procesami wykorzystane zostały funkcje standardu MPI - MPI_Send i MPI_Recv. Na poniższym listingu można zobaczyć ich sygnaturę.

```
1   MPI_Send(
2        void* data,
3        int count,
4        MPI_Datatype datatype,
5        int destination,
6        int tag,
7        MPI_Comm communicator)
8   MPI_Recv(
9        void* data,
10       int count,
11       MPI_Datatype datatype,
12       int source,
13       int tag,
14       MPI_Comm communicator,
15       MPI_Status* status)
```

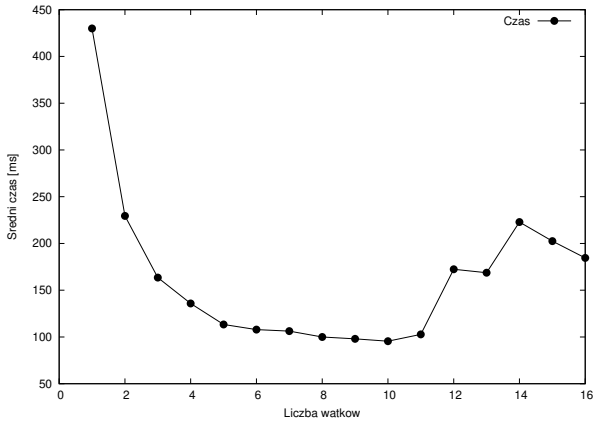
W programie główny proces służy do odczytu obrazka, podzieleniu go na równe partie, wysłaniu do procesów podrzędnych i po przetworzeniu złączenie z powrotem do spójnej całości. Główny jak i pozostałe procesy mogą być rozpoznane dzięki identyfikatorowi zwracanemu przez funckję MPI_Com_rank na początku programu. Poniższy fragment kodu przedstawia podzielenie obrazka na częśi i przesłanie ich do podrzędnych procesów.

```
1         // Split picture and send to other processes
```

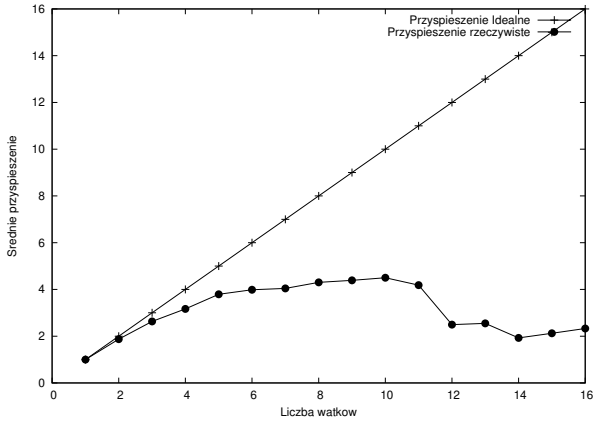
```
        for (int i = 1; i < numberOfProcesses; ++i){

            partOfImg = cutPicture(inputImg, 0, (i -1) * imgSplitSize,
                inputImg.cols, imgSplitSize);

            x = partOfImg.rows;
            y = partOfImg.cols;

            send(&x, 1, MPI_INT, i, START_VALUE_TAG);
            send(&y, 1, MPI_INT, i, END_VALUE_TAG);
            send(partOfImg.data, x * y * 3, MPI_BYTE, i, DATA_TAG);
        }
```

Samo rozmycie obrazu jest analogiczne jak w zadaniu numer 1.

Poniższe wykresy 1 i 2, przedstawiające zależność czasową oraz przyspieszenia zostały oparte na średnich wynikach programu uruchamianych lokalnie na maszynie wirtualnej. Wykorzystany został 6 rdzeniowy procesor Intel z technologią Hyperthreadingu.



Rysunek 1: Wykres zależności czasu wykonywania obliczeń od liczby wątków



Rysunek 2: Wykres przyspieszenia działania programu w zależności od liczby wątków

Można zauważyć, że po przekroczeniu fizycznej liczby rdzeni wydajność przestaje rosnąć, a więc HyperThreading nie dodaje zauważalnej różnicy w tym zadaniu. Po dalszym zwiększaniu liczby wątków wydajność programu zaczyna drastycznie maleć, co świadczy o dużych nakładach związanych z tworzeniem i komunikacją międzywątkową.