

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 wykończone 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 zrównoleglenia obliczeń w dalszej części programu.

```
1  MPI_Init(&argc , &argv );
2  MPI_Comm_rank (communicate , &rank );
3  MPI_Comm_size(communicate , &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      MPIComm communicator )
8  MPI_Recv(
9      void* data ,
10     int count ,
11     MPI_Datatype datatype ,
12     int source ,
13     int tag ,
14     MPIComm 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 funkcję `MPI_Comm_rank` na początku programu. Poniższy fragment kodu przedstawia podzielenie obrazka na części i przesłanie ich do podrzędnych procesów.

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

```

2      for (int i = 1; i < numberOfProcesses; ++i){
3
4          partOfImg = cutPicture(inputImg, 0, (i - 1) * imgSplitSize,
                                inputImg.cols, imgSplitSize);
5
6          x = partOfImg.rows;
7          y = partOfImg.cols;
8
9          send(&x, 1, MPI_INT, i, START_VALUE_TAG);
10         send(&y, 1, MPI_INT, i, END_VALUE_TAG);
11         send(partOfImg.data, x * y * 3, MPI_BYTE, i, DATA_TAG);
12     }

```

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.

Jak można zauważyć dzięki użyciu standardu MPI do zrównoleglenia rozmycia obrazu metodą Gaussa, udało się znacząco przyspieszyć wykonywanie programu. Najwyższe przyspieszenie udało się uzyskać wykorzystując pełną liczbę - 6 rdzeni procesora na którym został przetestowany program. Po przekroczeniu granicy 6 procesów przyspieszenie przestało wzrastać a narzut tworzenia i przesyłania wiadomości między procesami wraz z wzrostem ich liczby powodował nawet powolne opóźnienie. Można wyciągnąć wniosek, że standard MPI dobrze nadaje się do zadań które wymagają zrównoleglenia i pozwala wykorzystać wszystkie rdzenie procesora do obliczeń. W momencie gdy liczba procesów przekracza liczbę dostępnych rdzeni przyspieszenie przestaje wzrastać, pomimo wparcia przez procesor technologii HyperThreading.