Obliczenia równoległe

Temat : Równoległa wokselizacja modeli 3D

Data: 12.06.2013

Ocena:

.....

Prowadzący: mgr inż. Paweł Zabielski
Grupa: PS 2
Opracował: Tomasz Michaluk i Dariusz Murawski

Spis treści

1) Treść zadania

Napisać szeregową i równoległą (MPI lub OpenMP) wersję programu dokonującego wokselizacji obiektu 3D, opisanego w postaci siatki poligonów. Proponowany algorytm opisany jest tutaj. Program otrzymuje na wejściu obiekt w formacie OBJ, dokonuje jego wokselizacji i zapisuje na dysku serię obrazów – przekrojów przez uzyskany zbiór danych przestrzennych.

Obowiązkowe parametry programu to:

-i <plik> plik wejściowy

-x <szerokość> rozdzielczość generowanych danych w osi x (domyślnie: 128)

-y <wysokość> rozdzielczość danych w osi y (domyślnie: 128)

-z <głębokość> rozdzielczość danych w osi z (domyślnie: 128)

-n <liczba> liczba generowanych obrazów (domyślnie: 32)

-p <wartość>

płaszczyzna generowania przekrojów, możliwe wartości:

• c (ang. coronal) – czołowa, domyślna

• t (ang. transverse) – poprzeczna

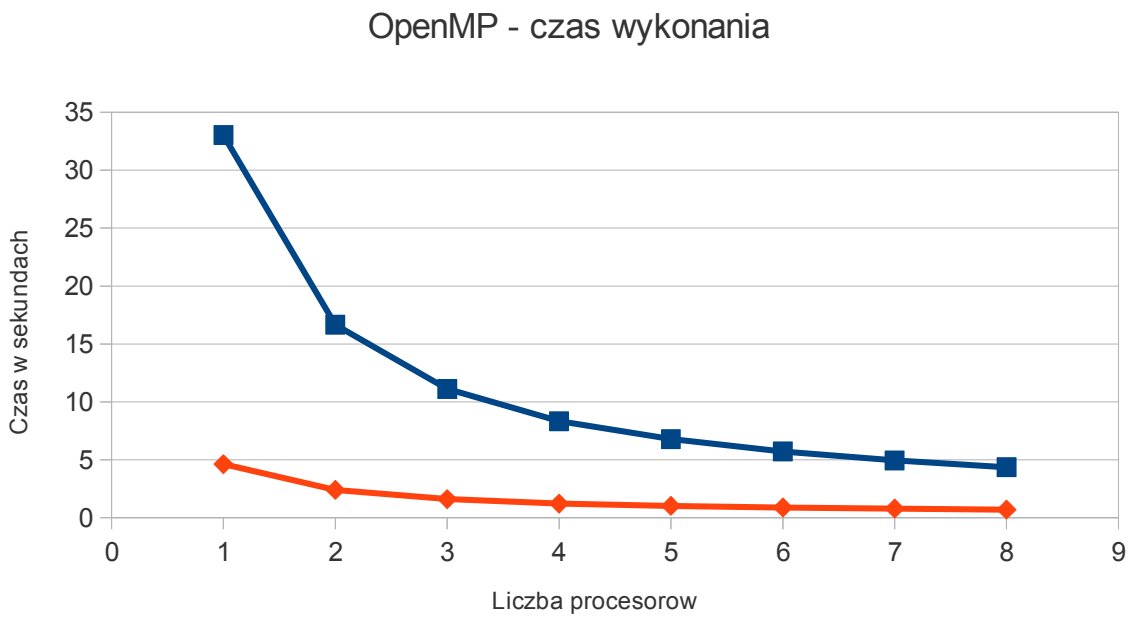
• s (ang. sagittal) - strzałkowa

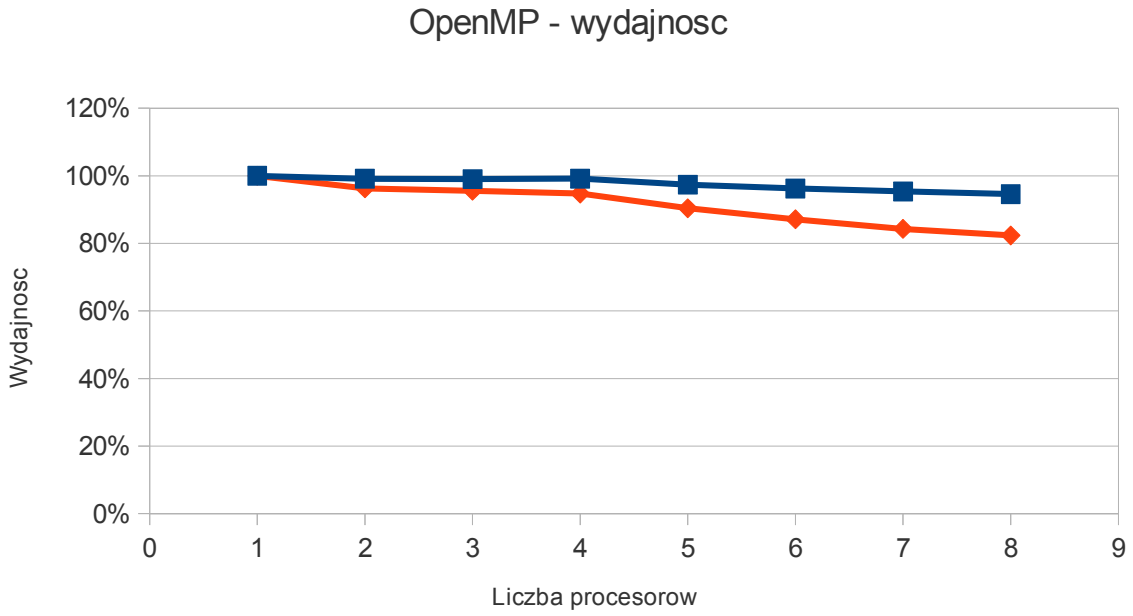
Uwaga: zakładamy, że konwertowany obiekt ma zamkniętą powierzchnię (brak dziur/nieciągłości). Po uruchomieniu program powinien zapisać do bieżącego katalogu serię obrazów wynikowych. Wszelkie komunikaty powinny być wypisywane na standardowe wyjście.

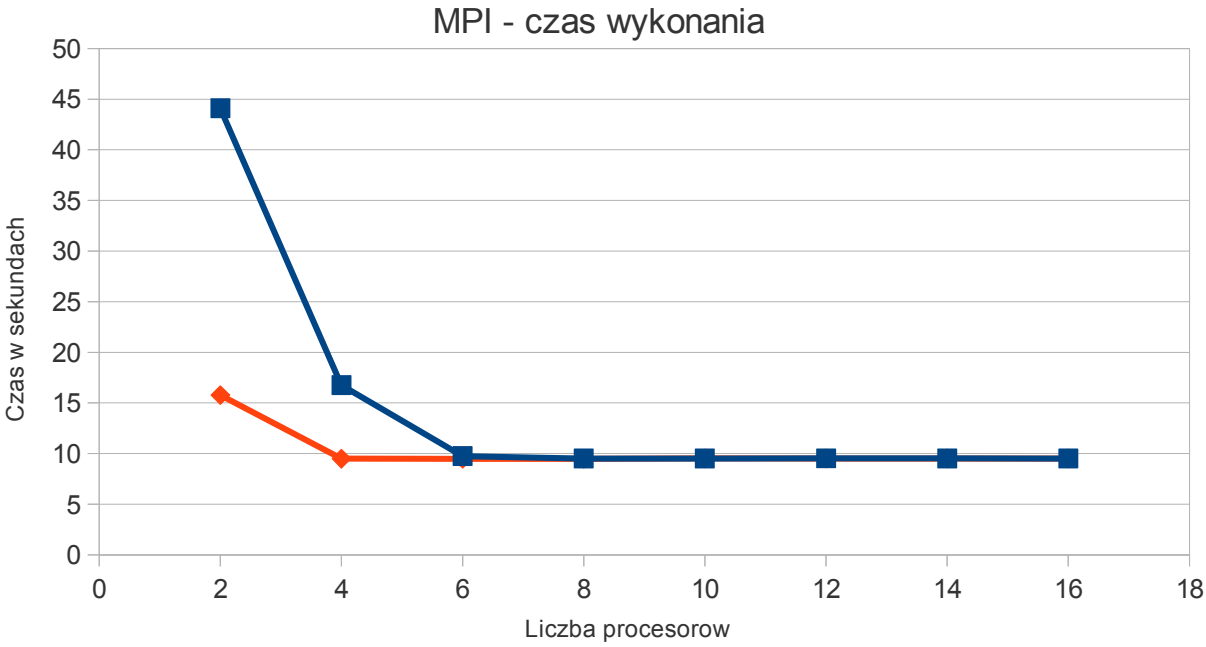
2) Wykresy i tabele

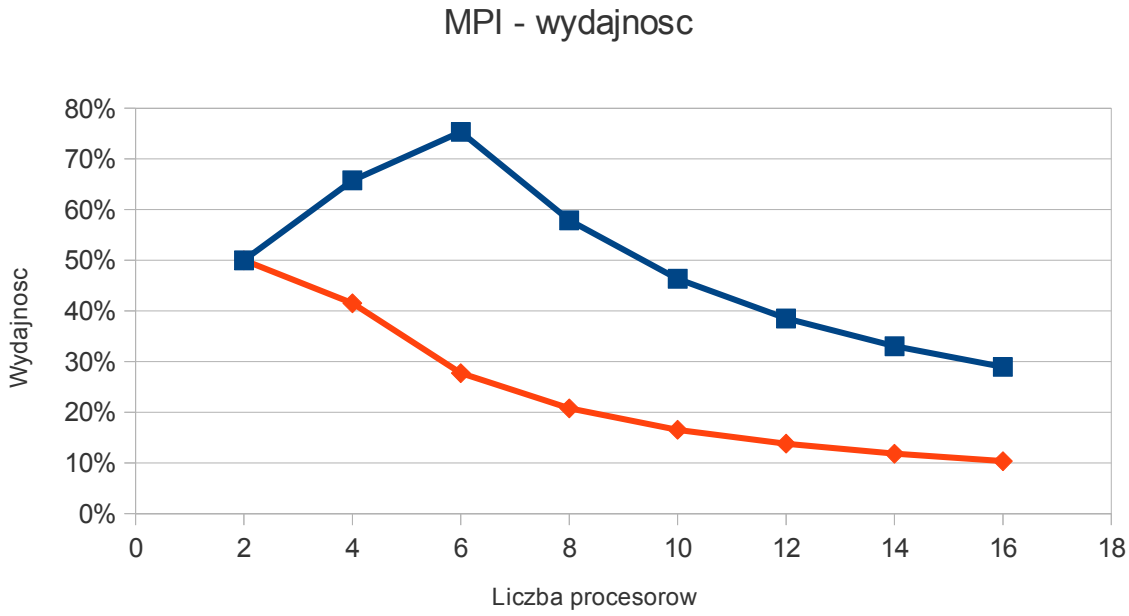
Czas wersji szeregowej głębokość drzewa 5– 33.4328 sekundy, dla głębokości 8 – 5.02999 sekund.

Niebieskie linie – głębokość drzewa QuadTree na 5, pomarańczowe – 8.






MPI - czas wykonania
Czas w sekundach
50
45
40
35
30
25
20
15
10
5
0
0
2
4
6
8
10
12
14
16
18
Liczba procesorow


MPI - wydajnosc
Wydajnosc
80%
70%
60%
50%
40%
30%
20%
10%
0%
0
2
4
6
8
10
12
14
16
18
Liczba procesorow

OpenMP

Liczba procesorów	Czas w sekundach		Przyspieszenie		Wydajność w % na procesor	
Glebokosc	5	8	5	8	5	8
1	33.0245	4.6226	1.9831675915	1	100%	100%
2	16.6524	2.40035	2.9712987539	1.9258024871	99%	96%
3	11.1145	1.61187	3.9672214254	2.8678491442	99%	96%
4	8.32434	1.21994	4.8691825105	3.7892027477	99%	95%
5	6.78235	1.02254	5.7756849165	4.5207033466	97%	90%
6	5.71785	0.88404	6.6779703516	5.2289489163	96%	87%
7	4.94529	0.783356	7.569270474	5.9010207364	95%	84%
8	4.36297	0.701699	1.9831675915	6.5877249362	95%	82%

MPI

Liczba procesorów	Czas w sekundach		Przyspieszenie		Wydajność w % na procesor	
Glebokosc	5	8	5	8	5	8
2	44.0988	15.7818	1	1	50%	50%
4	16.7642	9.50357	2.6305341144	1.6606180625	66%	42%
6	9.75788	9.48553	4.5193013236	1.6637762993	75%	28%
8	9.52285	9.49232	4.6308405572	1.6625861749	58%	21%
10	9.51725	9.53164	4.6335653681	1.6557276607	46%	17%
12	9.53628	9.52147	4.6243189168	1.6574961639	39%	14%
14	9.53005	9.52178	4.6273419342	1.6574422009	33%	12%
16	9.51841	9.51994	4.6330006797	1.6574422009	29%	10%

3) Wnioski

Wersja OpenMP i MPI są wersjami dynamicznymi.

Obserwując wykresy czasu wykonania wersji OpenMP widać, że pętla wokselizacji skaluje się bardzo dobrze. Algorytm polega na podziale pętli obliczającej kolejne „płaty” (wiersze) obiektu na poszczególne wątki dynamicznie. Prawie wszystkie zmienne można było ustawić na prywatne. Konflikt zapisu nie ma prawa wystąpić, ponieważ każdy z płatów to oddzielna tablica. Wersja dla głębokości 5 skaluje się dużo lepiej, ponieważ jeśli drzewo jest mniejsze, to więcej należy obliczeń wykonywać podczas puszczania promieni. Dzięki temu, skoro ta sekcja jest zrównoleglana, to lepiej się skaluje. Dla głębokości 8 również widać przyspieszenie, jednak mniejsze, gdyż lepsze drzewo bardzo przyspiesza algorytm.

Obserwując wykresy wersji MPI która była puszczana z flaga N (duże N), aby każdy proces był wykonywany na oddzielnym komputerze, widać, że 2 procesory działające w wersji dynamicznej obliczają obiekt o 25% dłużej. W teorii 2 procesory wersji MPI powinny wykonywać mniej więcej tyle samo czasu, co wersja szeregowa. Niestety, czas przesyłania danych oraz wstępne wczytywanie i rozsyłanie danych obiektu dodaje pewien narzut. Obserwując dalszą część wykresu dla głębokości 5 widać, że do 6 procesów wydajność zwiększa się. Spowodowane jest to tym, że obecność kolejnych procesów slave (pracujących), poprawia czas wobec wszystkich, ponieważ wątek MASTER jest tylko procesem, który rozdziela zadania i łączy wyniki. Dalszy spadek wydajności jest spowodowany coraz większym narzutem komunikacyjnym. Tak samo jak w wersji OpenMP, tutaj również widać, że głębokość drzewa która jest mniejsza, daje lepsze wyniki, ponieważ sekcja wokselizacji jest zrównoleglona.

Ogólne wnioski. Wokselizacja jest problem, w którym nie da się zrównoleglić pewnych sekcji i zostaną one zawsze takie same. Jest to ładowanie obiektu z dysku oraz zapis wyników na dysk. Musi to być robione przez jeden proces. Jednak zrównoleglanie tego problemu jest niesamowicie przydatne, jeśli obiekt ma bardzo dużo obiektów i należy wykonać dużo obliczeń. Wtedy czas sekcji który da się zrównoleglić, jest dużo większy w porównaniu do czasu wczytywania i zapisu danych. Lepsze wyniki wyszły dla OpenMP, jednak może to być spowodowane tym, że mieliśmy za mały obiekt (20 MB obj), tak więc czas pracy komputerów w wersji MPI był zbyt krótki w porównaniu do czasu transferu danych. (widać to na wykresach MPI, każdy spada do około 9.5 sekundy, nieważne jaka była głębokość drzewa)

4) Kod źródłowy

Wersja szeregowa:

```
void Grid::voxelization(){
	std::vector<realtype>* zList = NULL;
	for(unsigned i = 0 ; i < xRes ; ++i){
		for(unsigned j = 0 ; j < yRes ; ++j){
			delete zList;
			zList = NULL;
			this->setTmpPoint2D(i, j);
			zList = quadTree->calcRayIntersects(tmpPoint);
			if(zList == NULL){
				for(unsigned k = 0 ; k < zRes ; ++k){
					voxels[i][j][k] = false;
				}
			}else{
				for(unsigned k = 0 ; k < zRes ; ++k){
					realtype zc = this->calculateCellZCenter(k);
					//std::cout << "zc: " << zc << ", zList.size =
" << zList->size() << std::endl ;
					int isectsCounter = 0;
					// liczenie wartosci z przeciec mniejszych od
wartosci Z centrum komorki
					for(unsigned m = 0 ; m < zList->size() ; ++m){
						if( zc > zList->at(m) ){
							isectsCounter++;
						}else{
							break;
						}
					}
					if( (isectsCounter % 2) == 0 ){
```

```
                                    voxels[i][j][k] = false; // wewnatrz
                              }else{
                                    voxels[i][j][k] = true; //zewnatrz
                              }
                        }
                  }
            }
      }
      delete zList;
}
```

Wersja OpenMP

```
void Grid::voxelization(unsigned procNum){
      omp_set_num_threads(procNum);
      unsigned i,j;

      bool*** voxelsPointer = voxels;
      QuadTree* quadTreePointer = quadTree;
      BoundingBox aabb = gridAABB;

      unsigned width = xRes;
      unsigned height = yRes;
      unsigned depth = zRes;

#pragma omp parallel for private(i,j)  firstprivate(height,depth,aabb)
shared(voxelsPointer, quadTreePointer) schedule(dynamic)
      for(i=0; i < width ; ++i){

            //#pragma omp parallel for private(j)
firstprivate(height,depth,aabb) shared(voxelsPointer, quadTreePointer)
schedule(dynamic)
            for(j=0; j < height ; ++j){
                  std::vector<realtype>* zList = NULL;

                  Point2D tmpPoint = this->setTmpPoint2D(i, j);
                  zList = quadTreePointer->calcRayIntersects(tmpPoint);
                  if(zList == NULL){
```

```
                        for(unsigned k = 0 ; k < depth ; ++k){
                              voxelsPointer[i][j][k] = false;
                        }
                  }else{
                        for(unsigned k = 0 ; k < depth ; ++k){
                              realtype zc = this-
>calculateCellZCenter(k,aabb,depth);
                              //std::cout << "zc: " << zc << ", zList.size =
" << zList->size() << std::endl ;
                              int isectsCounter = 0;
                              // liczenie wartosci z przeciec mniejszych od
wartosci Z centrum komorki
                              for(unsigned m = 0 ; m < zList->size() ; ++m){
                                    if( zc > zList->at(m) ){
                                          isectsCounter++;
                                    }else{
                                          break;
                                    }
                              }
                              if( (isectsCounter % 2) == 0 ){
                                    voxelsPointer[i][j][k] = false; //
wewnatrz
                              }else{
                                    voxelsPointer[i][j][k] = true;
//zewnatrz
                              }
                        }
                  }
                  delete zList;
            }
      }
}
```

Wersja MPI

```
// to co wyzej ale wierszami
void Grid::voxelizationMaster_MPI_DynamicRows(int masterID, int
processQuantity){
      std::cout << "# Dynamiczne rozdzielanie - proces master " << masterID
<< " voxelizuje" << processQuantity << std::endl;
```

```
	int activeSlaves;
	unsigned nextRow, nextColumn;
	char dataBuffer[zRes*yRes];
	MPI_Status status;

	activeSlaves = processQuantity - 1;
	nextRow = 0;
	nextColumn = 0;

	// Wyslanie kazdemu  poczatkowego numeru wierszanad ktorym pracuje
	for(int i = 0 ; i < activeSlaves ; i++){

		MPI_Send(&nextRow, 1, MPI_UNSIGNED, i+1, 0, MPI_COMM_WORLD);
		nextRow++;
		if( nextRow == xRes ){
			nextRow = 0;
			nextColumn++;
		}
	}

	do{
		MPI_Recv(dataBuffer, zRes*yRes, MPI_CHAR, MPI_ANY_SOURCE,
MPI_ANY_TAG, MPI_COMM_WORLD, &status);
		activeSlaves--;
		// Numer odebranego wiersza to etykieta komunikatu
		unsigned recRow = status.MPI_TAG;
		unsigned slave = status.MPI_SOURCE;
		// Skopiowanie wyniku
		//std::cout << nextRow << " " << nextColumn << std::endl;
		for(unsigned i = 0 ; i < yRes ; i++){
			memcpy(voxels[recRow][i], &dataBuffer[i*yRes],
sizeof(char)*zRes);
		}
		//std::cout << "robie wiersz " << nextRow << std::endl;
		if (nextRow < xRes) {
			MPI_Send(&nextRow,1,MPI_UNSIGNED,slave,0,MPI_COMM_WORLD);
			activeSlaves++;
```

```
                  nextRow++;
//                if( nextRow == xRes ){
//                      nextRow = 0;
//                      nextColumn++;
//                }
            } else {
                  //std::cout << "MASTER WYLACZA SLAVE" << std::endl;
                  // Nie, wyślij sygnał zakończenia procesu podrzędnego
                  unsigned end= 0xFFFFFFFF;
                  MPI_Send(&end,1,MPI_UNSIGNED,slave,0,MPI_COMM_WORLD);
            }



      }while(activeSlaves > 0);
}
// to co wyzej ale calymi wierszami
void Grid::voxelizationSlaves_MPI_DynamicRows(int slavesID, int
processQuantity){
      std::cout << "# Dynamiczne rozdzielanie - proces slave " << slavesID
<< " voxelizuje" << processQuantity << std::endl;


      char dataBuffer[zRes*yRes];
      unsigned row;
      std::vector<realtype>* zList = NULL;
      MPI_Status status;


      while(1){
            MPI_Recv(&row, 1, MPI_UNSIGNED, MASTER_IDD ,0,MPI_COMM_WORLD,
&status);
            if( row == 0xFFFFFFFF ){
                  //std::cout << "# SLAVE" << slavesID << ", KONCZE
PRACE !!!" << std::endl;
                  return;
            }
            //std::cout << "# slave" << slavesID << ", wiersz: " << row <<
", kolumna: " << column << std::endl;
            for(unsigned j = 0 ; j < yRes ; ++j){
                  // Obliczenie "wiersza" o podanym x i y (czyli serii
kwadracikow po wybranym X
```

```
                  delete zList;
                  zList = NULL;
                  this->setTmpPoint2D(row, j);
                  zList = quadTree->calcRayIntersects(tmpPoint);
                  if(zList == NULL){
                        for(unsigned k = 0 ; k < zRes ; ++k){
                              dataBuffer[k+j*yRes] = 0;
                        }
                  }else{
                        for(unsigned k = 0 ; k < zRes ; ++k){
                              realtype zc = this->calculateCellZCenter(k);
                              int isectsCounter = 0;
                              // liczenie wartosci z przeciec mniejszych od
wartosci Z centrum komorki
                              for(unsigned m = 0 ; m < zList->size() ; ++m){
                                    if( zc > zList->at(m) ){
                                          isectsCounter++;
                                    }else{
                                          break;
                                    }
                              }
                              if( (isectsCounter % 2) == 0 ){
                                    dataBuffer[k+j*yRes] = 0; // wewnatrz
                              }else{
                                    dataBuffer[k+j*yRes] = 1; //zewnatrz
                              }
                        }
                  }
            }
            // Przeslanie obliczonego elementu
            MPI_Send(dataBuffer,zRes*yRes,MPI_CHAR, MASTER_IDD
,row,MPI_COMM_WORLD);
      }
}
```