Санкт-Петербургский государственный политехнический университет

Кафедра компьютерных систем и программных технологий

**Отчёт по курсовой работе**

**Дисциплина**: Параллельные вычисления

#### Тема: Создание многопоточных программ на языке C с использованием Pthreads. Создание многопроцессного приложения средствами MPI.

Выполнил студент гр. 13541/2 Муравьев Ф.Э.

(подпись)

Руководитель Стручков И. В.

(подпись)

“ ” 2017 г.

Санкт-Петербург

2017

#### Работа №1. Создание многопоточных программ на языке C с использованием Pthreads

Целью данной работы является получение навыков создания многопоточных программ с использованием интерфейсов POSIX Threads.

Ключ компилятора -pthread.

##### **Программа работы**

1. Для алгоритма из полученного задания написать последовательную программу на языке C или С++, реализующую этот алгоритм.
2. Для созданной последовательной программы необходимо написать 3-5 тестов, которые покрывают основные варианты функционирования программы. Для создания тестов можно воспользоваться механизмом Unit-тестов среды NetBeans, или описать входные тестовые данные в файлах. При использовании NetBeans необходимо в свойствах проекта установить ключ компилятора -pthread.
3. Написать код параллельной программы и проверить ее корректность на созданном ранее наборе тестов. При необходимости найти и исправить ошибки.
4. Провести эксперименты для оценки времени выполнения последовательной и параллельной программ. Проанализировать полученные результаты.

**Вариант индивидуального задания:**

Вариант №3: Выполнить сортировку последовательности чисел.

**Выполнение задания:**

1. **Для алгоритма из полученного задания написать последовательную программу на языке C или С++, реализующую этот алгоритм.**

Вариант №3: Выполнить сортировку последовательности чисел.

В качестве алгоритма сортировки был выбран алгоритм быстрой сортировки (Quick sort). Быстрая сортировка – рекурсивный алгоритм сортировки. Выбрав элемент в списке, быстрая сортировка делит с его помощью список на две части. В первую часть попадают все элементы, меньшие выбранного, а во вторую – большие элементы. Далее алгоритм применяется реккурентно к обеим частям списка.

Была написана последовательная программа на языке С++, реализующая данный алгоритм.

Листинг 1.1. Последовательная программа алгоритма быстрой сортировки

|  |
| --- |
| //QuickSort - реализация алгоритма быстрой сортировки  //@param a sort array  //@param n index the last of array elements  #include <iostream>  #include <fstream>  #include <ctime>  using namespace std;  template <typename T>  void quickSort(T\* a, const long n) {  long i = 0, j = n;  float pivot = a[n / 2]; // pivot element  do {  while (a[i] < pivot) i++;  while (a[j] > pivot) j--;  if (i <= j) {  std::swap(a[i], a[j]);  i++; j--;  }  } while (i <= j);  if (j > 0) quickSort(a, j);  if (n > i) quickSort(a + i, n - i);  }  int main(){  cout<<"Enter size of array: ";  int size;  cin>>size;  int \* a=new int[size];  int first\_value=0;  int last\_value=10000;  int format\_out=10;  int elements\_in\_row=10;  //Text file for results  ofstream fout("results.txt");  fout<<"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\tArray:"<<endl<<endl;  for(int i=0; i<size; i++)  {  a[i]=first\_value+rand()%last\_value;  fout.width(format\_out);  fout.setf(ios::left);  if(i%elements\_in\_row==0 && i!=0) fout<<endl;  fout<<a[i]<<"\t\t";  }  //Start quick sorting array  cout<<endl<<"Start sorting array (Quick sort)"<<endl;  unsigned int start\_time = clock(); // start time  quickSort(a, size-1);  unsigned int end\_time = clock(); // finish time  unsigned int search\_time = end\_time - start\_time; // execution time  cout<<"Finish sorting array. Execution time = "<<search\_time<<" milliseconds"<<endl;  fout<<endl<<endl<<endl<<"Start sorting array (Quick sort)"<<endl;  fout<<"Finish sorting array. Execution time = "<<search\_time<<" milliseconds"<<endl<<endl;  //Print results (sort array)  fout<<endl<<endl<<"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\tSort array:"<<endl;  for(int i=0; i<size; i++)  {  fout.width(format\_out);  fout.setf(ios::left);  if(i%elements\_in\_row==0 && i!=0) fout<<endl<<endl;  fout<<a[i]<<"\t\t";  }  fout.close();  delete [] a;  return 0;  } |

Сам алгоритм представлен функцией quickSort в листинге 2.

Листинг 2. Функция quickSort

|  |
| --- |
| template <typename T>  void quickSort(T\* a, const long n) {  long i = 0, j = n;  float pivot = a[n / 2]; // pivot element  do {  while (a[i] < pivot) i++;  while (a[j] > pivot) j--;  if (i <= j) {  std::swap(a[i], a[j]);  i++; j--;  }  } while (i <= j);  if (j > 0) quickSort(a, j);  if (n > i) quickSort(a + i, n - i);  } |

1. **Для созданной последовательной программы необходимо написать 3-5 тестов, которые покрывают основные варианты функционирования программы.**

Главным критерием правильности работы программы является отсортированный массив. Были произведены запуски программы с разным размером исходного массива, данные в массиве – целые числа в диапазоне от 0 до 32767. Полученные результаты:

**Запуск программы производится на оборудовании:**

Intel Core i3-5005U CPU 2.00GHz

2 ядра

4 логических процессора

Для начала произведем запуск программы с использованием **1-ого потока** для выборки чисел в 1000000:

Массив из 1000000 элементов:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| № | Время выполнения  t (мс) | Среднее время выполнения  (мс) | Погрешность отдельного измерения | Среднеквадратичная ошибка среднего арифметического | Доверительный интервал |
| 1 | 381,848 | 391,4801 | -9,6321 | 2,511238 | 5,680419 |
| 2 | 380,354 | -11,1261 |
| 3 | 394,458 | 2,9779 |
| 4 | 395,53 | 4,0499 |
| 5 | 390,326 | -1,1541 |
| 6 | 409,037 | 17,5569 |
| 7 | 390,478 | -1,0021 |
| 8 | 390,735 | -0,7451 |
| 9 | 393,679 | 2,1989 |
| 10 | 388,356 | -3,1241 |

1. **Написать код параллельной программы и проверить ее корректность на созданном ранее наборе тестов. При необходимости найти и исправить ошибки.**

Листинг 3.1. Параллельная программа алгоритма быстрой сортировки

|  |
| --- |
| #include <pthread.h>  //QuickSort - реализация алгоритма быстрой сортировки  //@param a sort array  //@param n index the last of array elements  #include <iostream>  #include <fstream>  #include <ctime>  #include <pthread.h>  #define ERROR\_CREATE\_THREAD -11  using namespace std;  struct pthrData  {  long size;  int \*a;  unsigned int rec\_level;  };  void\* quickSort(void\* thread\_data) {  unsigned int recursion\_level=2;  pthrData \*data=(pthrData\*) thread\_data;  int \*a=data->a;  long n=data->size;  long i = 0, j = n;  int pivot = a[n / 2]; // pivot element  do {  while (a[i] < pivot) i++;  while (a[j] > pivot) j--;  if (i <= j) {  std::swap(a[i], a[j]);  i++; j--;  }  } while (i <= j);  if (data->rec\_level<recursion\_level) {  pthread\_t thread1;  bool flag1=true;  pthread\_t thread2;  bool flag2=true;  if (j > 0)  {  pthrData new\_data;  new\_data.a=a;  new\_data.size=j;  new\_data.rec\_level=++data->rec\_level;  flag1=pthread\_create(&thread1, NULL, quickSort, &new\_data);  if (flag1) {  printf("main error: can't create thread\n");  exit(ERROR\_CREATE\_THREAD);  }  }  if (n > i)  {  pthrData new\_data;  new\_data.a=a+i;  new\_data.size=n-i;  new\_data.rec\_level=++data->rec\_level;  flag2=pthread\_create(&thread2, NULL, quickSort, &new\_data);  if (flag2) {  printf("main error: can't create thread\n");  exit(ERROR\_CREATE\_THREAD);  }  }  if(!flag1) pthread\_join(thread1, NULL);  if(!flag2) pthread\_join(thread2, NULL);  }  else  {  if(j > 0)  {  pthrData new\_data;  new\_data.a=a;  new\_data.size=j;  quickSort(&new\_data);  }  if (n > i)  {  pthrData new\_data;  new\_data.a=a+i;  new\_data.size=n-i;  quickSort(&new\_data);  }  }  pthread\_exit(0);  }  int main(){  cout<<"Enter size of array: ";  int size;  cin>>size;  int \* a=new int[size];  int first\_value=0;  int last\_value=32767;  int format\_out=10;  int elements\_in\_row=10;  //Text file for results  ofstream fout;  fout.open("results.txt");  fout<<"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\tArray:"<<endl<<endl;  for(int i=0; i<size; i++)  {  a[i]=first\_value+rand()%last\_value;  fout.width(format\_out);  fout.setf(ios::left);  if(i%elements\_in\_row==0 && i!=0) fout<<endl;  fout<<a[i]<<"\t\t";  }  pthrData data;  data.a=a;  data.size=size;  data.rec\_level=0;  //Start quick sorting array  cout<<endl<<"Start sorting array (Quick sort)"<<endl;  unsigned int start\_time = clock(); // start time  //create id thread  pthread\_t thread;  pthread\_create(&thread, NULL, quickSort, &data);  //wait finish pthread  pthread\_join(thread, NULL);  unsigned int end\_time = clock(); // finish time  unsigned int search\_time = end\_time - start\_time; // execution time  cout<<"Finish sorting array. Execution time = "<<search\_time<<" milliseconds"<<endl;  fout<<endl<<endl<<endl<<"Start sorting array (Quick sort)"<<endl;  fout<<"Finish sorting array. Execution time = "<<search\_time<<" milliseconds"<<endl<<endl;  //Print results (sort array)  fout<<endl<<endl<<"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\tSort array:"<<endl;  for(int i=0; i<size; i++)  {  fout.width(format\_out);  fout.setf(ios::left);  if(i%elements\_in\_row==0 && i!=0) fout<<endl<<endl;  fout<<a[i]<<"\t\t";  }  fout.close();  delete [] a;  return 0;  } |

1. **Провести эксперименты для оценки времени выполнения последовательной и параллельной программ. Проанализировать полученные результаты.**

Главным критерием правильности работы программы является отсортированный массив. Были произведены запуски программы с разным размером исходного массива, данные в массиве – целые числа в диапазоне от 0 до 32767. Полученные результаты:

Теперь произведем запуск программы с использованием **2-ух потоков** для выборки чисел в 1000000:

Массив из 1000000 элементов:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| № | Время выполнения  t (мс) | Среднее время выполнения  (мс) | Погрешность отдельного измерения | Среднеквадратичная ошибка среднего арифметического | Доверительный интервал |
| 1 | 72,631 | 72,7171 | -0,0861 | 1,242068 | 2,809557 |
| 2 | 69,447 | -3,2701 |
| 3 | 72,494 | -0,2231 |
| 4 | 67,593 | -5,1241 |
| 5 | 69,844 | -2,8731 |
| 6 | 73,553 | 0,8359 |
| 7 | 79,177 | 6,4599 |
| 8 | 69,534 | -3,1831 |
| 9 | 73,883 | 1,1659 |
| 10 | 79,015 | 6,2979 |

Теперь произведем запуск программы с использованием **4-ух потоков** для выборки чисел в 1000000:

Массив из 1000000 элементов:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| № | Время выполнения  t (мс) | Среднее время выполнения  (мс) | Погрешность отдельного измерения | Среднеквадратичная ошибка среднего арифметического | Доверительный интервал |
| 1 | 110,52 | 123,6166 | -13,0966 | 3,305834 | 7,477795 |
| 2 | 128,96 | 5,3434 |
| 3 | 133,58 | 9,9634 |
| 4 | 113,854 | -9,7626 |
| 5 | 139,773 | 16,1564 |
| 6 | 123,753 | 0,1364 |
| 7 | 114,314 | -9,3026 |
| 8 | 111,317 | -12,2996 |
| 9 | 131,963 | 8,3464 |
| 10 | 128,132 | 4,5154 |

#### Работа №2. Создание многопроцессного приложения средствами MPI

**Вариант индивидуального задания:**

Вариант №3: Выполнить сортировку последовательности чисел.

**Выполнение задания:**

1. **Для алгоритма из полученного задания с использованием средств MPI была разработана программа сортировки последовательности чисел.**

Листинг 1.1. Программа алгоритма быстрой сортировки с использованием средств MPI

|  |
| --- |
| /\* quicksort \*/  #include <stdio.h>  #include <stdlib.h>  #include <mpi.h>  #include <time.h>  #define N 10000000  void showElapsed(int id, char \*m);  void showVector(int \*v, int n, int id);  int \* merge(int \*v1, int n1, int \*v2, int n2);  void swap(int \*v, int i, int j);  void qsort(int \*v, int left, int right);  double startTime, stopTime;  void showElapsed(int id, char \*m)  {  printf("%d: %s %f secs\n",id,m,(clock()-startTime)/CLOCKS\_PER\_SEC);  }  void showVector(int \*v, int n, int id)  {  int i;  printf("%d: ",id);  for(i=0;i<n;i++)  printf("%d ",v[i]);  putchar('\n');  }  int \* merge(int \*v1, int n1, int \*v2, int n2)  {  int i,j,k;  int \* result;  result = (int \*)malloc((n1+n2)\*sizeof(int));  i=0; j=0; k=0;  while(i<n1 && j<n2)  if(v1[i]<v2[j])  {  result[k] = v1[i];  i++; k++;  }  else  {  result[k] = v2[j];  j++; k++;  }  if(i==n1)  while(j<n2)  {  result[k] = v2[j];  j++; k++;  }  else  while(i<n1)  {  result[k] = v1[i];  i++; k++;  }  return result;  }  void swap(int \*v, int i, int j)  {  int t;  t = v[i];  v[i] = v[j];  v[j] = t;  }  void qsort(int \*v, int left, int right)  {  int i,last;  if(left>=right)  return;  swap(v,left,(left+right)/2);  last = left;  for(i=left+1;i<=right;i++)  if(v[i]<v[left])  swap(v,++last,i);  swap(v,left,last);  qsort(v,left,last-1);  qsort(v,last+1,right);  }  int main(int argc, char \*\*argv)  {  int \* data;  int \* chunk;  int \* other;  int m,n=N;  int id,p;  int s;  int i;  int step;  MPI\_Status status;  startTime = clock();  MPI\_Init(&argc,&argv);  MPI\_Comm\_rank(MPI\_COMM\_WORLD,&id);  MPI\_Comm\_size(MPI\_COMM\_WORLD,&p);  showElapsed(id,"MPI setup complete");  if(id==0)  {  int r;  int first\_value=0;  int last\_value=32767;  s = n/p;  r = n%p;  data = (int \*)malloc((n+s-r)\*sizeof(int));  for(i=0;i<n;i++)  data[i] = first\_value+rand()%last\_value;  if(r!=0)  {  for(i=n;i<n+s-r;i++)  data[i]=0;  s=s+1;  }  showElapsed(id,"generated the random numbers");  MPI\_Bcast(&s,1,MPI\_INT,0,MPI\_COMM\_WORLD);  chunk = (int \*)malloc(s\*sizeof(int));  MPI\_Scatter(data,s,MPI\_INT,chunk,s,MPI\_INT,0,MPI\_COMM\_WORLD);  showElapsed(id,"scattered data");  qsort(chunk,0,s-1);  showElapsed(id,"sorted");  }  else  {  MPI\_Bcast(&s,1,MPI\_INT,0,MPI\_COMM\_WORLD);  chunk = (int \*)malloc(s\*sizeof(int));  int r;  r = n%p;  data = (int \*)malloc((n+s-r)\*sizeof(int));  MPI\_Scatter(data,s,MPI\_INT,chunk,s,MPI\_INT,0,MPI\_COMM\_WORLD);  showElapsed(id,"got data");  qsort(chunk,0,s-1);  showElapsed(id,"sorted");  }  step = 1;  while(step<p)  {  if(id%(2\*step)==0)  {  if(id+step<p)  {  MPI\_Recv(&m,1,MPI\_INT,id+step,0,MPI\_COMM\_WORLD,&status);  other = (int \*)malloc(m\*sizeof(int));  MPI\_Recv(other,m,MPI\_INT,id+step,0,MPI\_COMM\_WORLD,&status);  showElapsed(id,"got merge data");  chunk = merge(chunk,s,other,m);  showElapsed(id,"merged data");  s = s+m;  }  }  else  {  int near = id-step;  MPI\_Send(&s,1,MPI\_INT,near,0,MPI\_COMM\_WORLD);  MPI\_Send(chunk,s,MPI\_INT,near,0,MPI\_COMM\_WORLD);  showElapsed(id,"sent merge data");  break;  }  step = step\*2;  }  if(id==0)  {  FILE \* fout;  stopTime = clock();  printf("%d; %d processors; %f secs\n", s,p,(stopTime-startTime)/CLOCKS\_PER\_SEC);  showElapsed(id,"opening out file");  fout = fopen("result","w");  for(i=0;i<s;i++)  fprintf(fout,"%d\n",chunk[i]);  fclose(fout);  showElapsed(id,"closed out file");  }  MPI\_Finalize();  } |

1. **Протестируем эффективность полученной программы.**

**Запуск программы производится на оборудовании:**

Intel Core i3-5005U CPU 2.00GHz

2 ядра

4 логических процессора

Для начала произведем запуск программы с использованием **1-ого потока** для выборки чисел в 1000000, 10000000:

Массив из 1000000 элементов:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| № | Время выполнения  t (мс) | Среднее время выполнения  (мс) | Погрешность отдельного измерения | Среднеквадратичная ошибка среднего арифметического | Доверительный интервал |
| 1 | 958 | 863.4 | 94,6 | 10.86 | 24.58 |
| 2 | 865 | 1,6 |
| 3 | 841 | -22,4 |
| 4 | 849 | -14,4 |
| 5 | 864 | 0,6 |
| 6 | 845 | -18,4 |
| 7 | 864 | 0,6 |
| 8 | 854 | -9,4 |
| 9 | 848 | -15,4 |
| 10 | 846 | -17,4 |

Массив из 10000000 элементов:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| № | Время выполнения  t (с) | Среднее время выполнения  (с) | Погрешность отдельного измерения | Среднеквадратичная ошибка среднего арифметического | Доверительный интервал |
| 1 | 13,726 | 13,7707 | -0,0447 | 0,015023 | 0,033982 |
| 2 | 13,775 | 0,0043 |
| 3 | 13,764 | -0,0067 |
| 4 | 13,750 | -0,0207 |
| 5 | 13,736 | -0,0347 |
| 6 | 13,843 | 0,0723 |
| 7 | 13,827 | 0,0563 |
| 8 | 13,726 | -0,0447 |
| 9 | 13,725 | -0,0457 |
| 10 | 13,835 | 0,0643 |

Теперь произведем запуск программы с использованием **2-ух потоков** для выборки чисел в 1000000, 10000000:

Массив из 1000000 элементов:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| № | Время выполнения  t (мс) | Среднее время выполнения  (мс) | Погрешность отдельного измерения | Среднеквадратичная ошибка среднего арифметического | Доверительный интервал |
| 1 | 567 | 468,6 | 98,4 | 12,94965 | 29,2921 |
| 2 | 453 | -15,6 |
| 3 | 496 | 27,4 |
| 4 | 484 | 15,4 |
| 5 | 443 | -25,6 |
| 6 | 437 | -31,6 |
| 7 | 453 | -15,6 |
| 8 | 481 | 12,4 |
| 9 | 435 | -33,6 |
| 10 | 437 | -31,6 |

Массив из 10000000 элементов:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| № | Время выполнения  t (с) | Среднее время выполнения  (с) | Погрешность отдельного измерения | Среднеквадратичная ошибка среднего арифметического | Доверительный интервал |
| 1 | 6,265 | 6,4761 | -0,2111 | 0,026689 | 0,060371 |
| 2 | 6,560 | 0,0839 |
| 3 | 6,420 | -0,0561 |
| 4 | 6,512 | 0,0359 |
| 5 | 6,521 | 0,0449 |
| 6 | 6,513 | 0,0369 |
| 7 | 6,521 | 0,0449 |
| 8 | 6,480 | 0,0039 |
| 9 | 6,520 | 0,0439 |
| 10 | 6,449 | -0,0271 |

Теперь произведем запуск программы с использованием **4-ех потоков** для выборки чисел в 1000000, 10000000:

Массив из 1000000 элементов:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| № | Время выполнения  t (мс) | Среднее время выполнения  (мс) | Погрешность отдельного измерения | Среднеквадратичная ошибка среднего арифметического | Доверительный интервал |
| 1 | 359 | 356,9 | 2,1 | 2,208318 | 4,995214 |
| 2 | 343 | -13,9 |
| 3 | 349 | -7,9 |
| 4 | 359 | 2,1 |
| 5 | 364 | 7,1 |
| 6 | 351 | -5,9 |
| 7 | 360 | 3,1 |
| 8 | 365 | 8,1 |
| 9 | 359 | 2,1 |
| 10 | 360 | 3,1 |

Массив из 10000000 элементов:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| № | Время выполнения  t (с) | Среднее время выполнения  (с) | Погрешность отдельного измерения | Среднеквадратичная ошибка среднего арифметического | Доверительный интервал |
| 1 | 4,248 | 4,2764 | -0,0284 | 0,014967 | 0,033855 |
| 2 | 4,305 | 0,0286 |
| 3 | 4,323 | 0,0466 |
| 4 | 4,257 | -0,0194 |
| 5 | 4,300 | 0,0236 |
| 6 | 4,323 | 0,0466 |
| 7 | 4,280 | 0,0036 |
| 8 | 4,289 | 0,0126 |
| 9 | 4,277 | 0,0006 |
| 10 | 4,162 | -0,1144 |

**Выводы:**

Эффективность выполнения параллельной программы алгоритма быстрой сортировки по сравнению с выполнением последовательной программы была доказана.