Санкт-Петербургский Политехнический Университет Петра Великого

Кафедра компьютерных систем и программных технологий

Отчёт по курсовой работе

Курс: «Параллельные вычисления»

Тема: «Выполнение сортировки последовательных чисел»

Выполнил студент:

Ивашкевич О.А.
Группа: 13541/2

Проверил:

Стручков И. В.

Санкт-Петербург
2019 г.

Содержание

Курсовая работа

1.1 Цель работы

Целью данной работы является получение студентами навыков создания многопоточных программ с использованием интерфейсов POSIX Threads, OpenMP и MPI.

1.2 Программа работы

- Для алгоритма из полученного задания написать последовательную программу на языке C или C++, реализующую этот алгоритм.
- Проанализировать полученный алгоритм, выделить части, которые могут быть распараллелены, разработать структуру параллельной программы. Определить количество используемых потоков, а также правила и используемые объекты синхронизации.
- Написать код параллельной программы и проверить ее корректность на созданном наборе тестов.
- Провести эксперименты для оценки времени выполнения последовательной и параллельной программ. Проанализировать полученные результаты.
- Сделать общие выводы по результатам проделанной работы: Различия между способами проектирования последовательной и параллельной реализаций алгоритма, Возможные способы выделения параллельно выполняющихся частей, Возможные правила синхронизации потоков, Сравнение времени выполнения последовательной и параллельной программ, Принципиальные ограничения повышения эффективности параллельной реализации по сравнению с последовательной.

1.3 Индивидуальное задание

Вариант №3
Выполнить сортировку последовательных чисел.
Средство распараллеливания – **MPI**.

1.4 Компиляция и компоновка проекта

В качестве среды разработки используется MS VS 2017 с пакетом MS HPC Pack 2012 для MPI.

1.5 Ход работы

1.5.1 Формализация задачи и детали реализации

- В последовательном варианте использовалась сортировка пузырькем.
- При реализации MPI использовался алгоритм "чет-нечетных перестановок"из-за специфики распараллеливания алгоритма пузырьковой сортировки.

1.5.2 Разработка последовательной реализации

Сортировка пузырьком:

```
<...>
void swap(int* v, int i, int j)
{
  int t;
  t = v[i];
  v[i] = v[j];
  v[j] = t;
}

void sort(int* v, int n)
{
  int i, j;

  for (i = n - 2; i >= 0; i--)
    for (j = 0; j <= i; j++)
      if (v[j] > v[j + 1])
        swap(v, j, j + 1);
}
<...>
```

1.5.3 Разработка многопоточной реализации при помощи pthread

Распараллеливание подобного алгоритма подразумевает создание нескольких потоков для разделения массива чисел.

Многопоточная реализация алгоритма:

```
#include <stdio.h>
#include <pthread.h>
#include <time.h>
#include <stdlib.h>

int opt_a;
int opt_t;
int opt_r;

// number of elements in array
#define MAX 1500

// number of threads
#define THREAD_MAX 4

// array of size MAX
int *a;

// thread control parameters
struct tsk {
  int tsk_no;
  int tsk_low;
  int tsk_high;
};

void
merge(int low, int mid, int high)
{

  // n1 is size of left part and n2 is size of right part
  int n1 = mid - low + 1;
  int n2 = high - mid;

  int *left = malloc(n1 * sizeof(int));
  int *right = malloc(n2 * sizeof(int));

```

```
37    int i;
38    int j;
39
40    for (i = 0; i < n1; i++)
41      left[i] = a[i + low];
42
43    for (i = 0; i < n2; i++)
44      right[i] = a[i + mid + 1];
45
46    int k = low;
47
48    i = j = 0;
49
50    while (i < n1 && j < n2) {
51      if (left[i] <= right[j])
52        a[k++] = left[i++];
53      else
54        a[k++] = right[j++];
55    }
56
57    while (i < n1)
58      a[k++] = left[i++];
59
60    while (j < n2)
61      a[k++] = right[j++];
62
63    free(left);
64    free(right);
65  }
66
67  void
68  merge_sort(int low, int high)
69  {
70    int mid = low + (high - low) / 2;
71
72    if (low < high) {
73      merge_sort(low, mid);
74      merge_sort(mid + 1, high);
75      merge(low, mid, high);
76    }
77  }
78
79  // thread function for multi-threading
80  void *
81  merge_sort123(void *arg)
82  {
83    struct tsk *tsk = arg;
84    int low;
85    int high;
86
87    low = tsk->tsk_low;
88    high = tsk->tsk_high;
89
90    int mid = low + (high - low) / 2;
91
92    if (low < high) {
93      merge_sort(low, mid);
94      merge_sort(mid + 1, high);
95      merge(low, mid, high);
96    }
97
98    return 0;
99  }
100
101 int
102 main(int argc, char **argv)
```

```
103 {
104   char *cp;
105   struct tsk *tsk;
106
107   --argc;
108   ++argv;
109
110   opt_a = 1;
111
112   opt_r = !opt_r;
113   opt_t = !opt_t;
114
115   a = malloc(sizeof(int) * MAX);
116
117   // generating random values in array
118   printf("ORIG:");
119   for (int i = 0; i < MAX; i++) {
120     a[i] = rand() % 100;
121     printf(" %d", a[i]);
122   }
123
124   pthread_t threads[THREAD_MAX];
125   struct tsk tsklist[THREAD_MAX];
126
127   int len = MAX / THREAD_MAX;
128
129   if (opt_t)
130     printf("\nTHREADS:%d MAX:%d LEN:%d\n", THREAD_MAX, MAX, len);
131
132   int low = 0;
133
134   for (int i = 0; i < THREAD_MAX; i++, low += len) {
135     tsk = &tsklist[i];
136     tsk->tsk_no = i;
137
138     if (opt_a) {
139       tsk->tsk_low = low;
140       tsk->tsk_high = low + len - 1;
141       if (i == (THREAD_MAX - 1))
142         tsk->tsk_high = MAX - 1;
143     }
144
145     else {
146       tsk->tsk_low = i * (MAX / THREAD_MAX);
147       tsk->tsk_high = (i + 1) * (MAX / THREAD_MAX) - 1;
148     }
149
150     if (opt_t)
151       printf("RANGE %d: %d %d\n", i, tsk->tsk_low, tsk->tsk_high);
152   }
153
154   // creating 4 threads
155   for (int i = 0; i < THREAD_MAX; i++) {
156     tsk = &tsklist[i];
157     pthread_create(&threads[i], NULL, merge_sort123, tsk);
158   }
159
160   // joining all 4 threads
161   for (int i = 0; i < THREAD_MAX; i++)
162     pthread_join(threads[i], NULL);
163
164   // show the array values for each thread
165   if (opt_t) {
166     for (int i = 0; i < THREAD_MAX; i++) {
167       tsk = &tsklist[i];
168       printf("SUB %d:", tsk->tsk_no);
```

```
169         for (int j = tsk->tsk_low; j <= tsk->tsk_high; ++j)
170           printf(" %d", a[j]);
171         printf("\n");
172       }
173     }
174
175     // merging the final 4 parts
176     if (opt_a) {
177       struct tsk *tskm = &tsklist[0];
178       for (int i = 1; i < THREAD_MAX; i++) {
179         struct tsk *tsk = &tsklist[i];
180         merge(tskm->tsk_low, tsk->tsk_low - 1, tsk->tsk_high);
181       }
182     }
183     else {
184       merge(0, (MAX / 2 - 1) / 2, MAX / 2 - 1);
185       merge(MAX / 2, MAX / 2 + (MAX - 1 - MAX / 2) / 2, MAX - 1);
186       merge(0, (MAX - 1) / 2, MAX - 1);
187     }
188
189     printf("\n\nSorted array:");
190     for (int i = 0; i < MAX; i++)
191       printf(" %d", a[i]);
192     printf("\n");
193
194     return 0;
195 }
```

1.5.4 Разработка многопроцессной реализации при помощи MPI

Message Passing Interface (MPI) – программный интерфейс для передачи информации, который позволяет обмениваться сообщениями между процессами, выполняющими одну задачу. Основным средством коммуникации между процессами в MPI является передача сообщений друг другу.

В первую очередь MPI ориентирован на системы с распределенной памятью, то есть когда затраты на передачу данных велики, в то время как OpenMP ориентирован на системы с общей памятью (многоядерные с общим кешем). Обе технологии могут использоваться совместно, чтобы оптимально использовать в кластере многоядерные системы.

MPI хорошо подходит для размещения или обмена простыми общими данными: целыми числами, числами с плавающей точкой, массивами чисел и др.

Алгоритм пузырьковой сортировки в прямом виде достаточно сложен для распараллеливания – сравнение пар значений упорядочиваемого набора данных происходит строго последовательно. В связи с этим для параллельного применения используется не сам этот алгоритм, а его модификация, известная в литературе как метод чет-нечетной перестановки (odd-even transposition) . Алгоритм "чет-нечетных перестановок"также сортирует n-элементов за n-итераций (n - четно), однако правила итераций различаются в зависимости от четности или нечетности номера итерации и, кроме того, каждая из итераций требует n/2 операций сравнения-обмена. Пусть (a1,a2, ...,an) - последовательность, которую нужно отсортировать. На итерациях с нечетными итерациями элементы с нечетными индексами сравниваются с их правыми соседями, и если они не находятся в нужном порядке, они меняются местами (т.е. сравниваются пары (a1, a2), (a3, a4), ..., (an-1,an) и, при необходимости, обмениваются (принимая, что n четно). Точно так же в течение четной фазы элементы с четными индексами сравниваются с их правыми соседями (т.е. сравниваются пары (a2, a3), (a4, a5), ..., (an-2,an-1)), и если они находятся не в порядке сортировки, их меняют местами. После n-фазы нечетно-четных перестановок последовательность отсортирована. Каждая фаза алгоритма (и нечетная, и четная) требует O(n) сравнений, а всего n фаз; таким образом, последовательная сложность алгоритма - O(n2).

```
1  #include "stdio.h"
2  #include "mpi.h"
3  #include "fstream"
4  double startT, stopT;
5
6
7  int* mergeArrays(int* v1, int n1, int* v2, int n2)
8  {
9    int i, j, k;
10   int* result;
```

```
11
12    result = new int[n1 + n2];
13    i = 0;
14    j = 0;
15    k = 0;
16
17    while (i < n1 && j < n2)
18      if (v1[i] < v2[j]) {
19        result[k] = v1[i];
20        i++;
21        k++;
22      }
23      else {
24        result[k] = v2[j];
25        j++;
26        k++;
27      }
28
29    if (i == n1)
30      while (j < n2) {
31        result[k] = v2[j];
32        j++;
33        k++;
34      }
35    if (j == n2)
36      while (i < n1) {
37        result[k] = v1[i];
38        i++;
39        k++;
40      }
41
42    return result;
43  }
44
45  void swap(int* v, int i, int j)
46  {
47    int t;
48    t = v[i];
49    v[i] = v[j];
50    v[j] = t;
51  }
52
53  void sort(int* v, int n)
54  {
55    int i, j;
56
57    for (i = n - 2; i >= 0; i--)
58      for (j = 0; j <= i; j++)
59        if (v[j] > v[j + 1])
60          swap(v, j, j + 1);
61  }
62
63  using namespace std;
64
65  int main(int argc, char ** argv)
66  {
67    int* data = 0;            //Our unsorted array
68    int* resultant_array; //Sorted Array
69    int* sub;
70
71    int m, n;
72    int id, p;
73    int r;
74    int s;
75    int i;
76    int move;
```

```
77    MPI_Status status;
78
79    MPI_Init(&argc, &argv);
80    MPI_Comm_rank(MPI_COMM_WORLD, &id);
81    MPI_Comm_size(MPI_COMM_WORLD, &p);
82    srand(unsigned int(MPI_Wtime()));
83
84    //Task Of The Master Processor
85    if (id == 0) {
86      n = 80000;
87      s = n / p;
88      r = n % p;
89      data = new int[n + s - r];
90
91
92      ofstream file("input");
93
94      for (i = 0; i < n; i++)
95      {
96        data[i] = rand() % 15000;
97        file << data[i] << " ";
98      }
99
100     file.close();
101
102     if (r != 0) {
103       for (i = n; i < n + s - r; i++)
104         data[i] = 0;
105
106       s = s + 1;
107     }
108
109     startT = MPI_Wtime();     //Start Time.
110     MPI_Bcast(&s, 1, MPI_INT, 0, MPI_COMM_WORLD);
111     resultant_array = new int[s];
112     MPI_Scatter(data, s, MPI_INT, resultant_array, s, MPI_INT, 0, MPI_COMM_WORLD);
113     sort(resultant_array, s);
114   }
115   else {
116     MPI_Bcast(&s, 1, MPI_INT, 0, MPI_COMM_WORLD);
117     resultant_array = new int[s];
118     MPI_Scatter(data, s, MPI_INT, resultant_array, s, MPI_INT, 0, MPI_COMM_WORLD);
119     sort(resultant_array, s);
120   }
121
122   move = 1;
123
124   //Merging the sub sorted arrays to obtain resultant sorted array
125   while (move < p) {
126     if (id % (2 * move) == 0) {
127       if (id + move < p) {
128         MPI_Recv(&m, 1, MPI_INT, id + move, 0, MPI_COMM_WORLD, &status);
129         sub = new int[m];
130         MPI_Recv(sub, m, MPI_INT, id + move, 0, MPI_COMM_WORLD, &status);
131         resultant_array = mergeArrays(resultant_array, s, sub, m);
132         s = s + m;
133       }
134     }
135     else {
136       int near = id - move;
137       MPI_Send(&s, 1, MPI_INT, near, 0, MPI_COMM_WORLD);
138       MPI_Send(resultant_array, s, MPI_INT, near, 0, MPI_COMM_WORLD);
139       break;
140     }
141
142     move = move * 2;
```

```
143   }
144
145   //Results
146   if (id == 0) {
147
148     stopT = MPI_Wtime();
149     double parallelTime = stopT - startT;
150
151     printf("\n\n\nTime: %f", parallelTime);
152
153     startT = MPI_Wtime();
154     sort(data, n);
155     stopT = MPI_Wtime();
156
157     double poslTime = stopT - startT;
158     printf("\n Time: %f \n", stopT - startT);
159     printf("\n SpeedUp: %f \n\n\n", poslTime / parallelTime);
160
161     ofstream file1("result");
162     file1 << parallelTime << " - Parallel Time \n";
163     file1 << poslTime << " - Posledov Time \n";
164     file1 << poslTime / parallelTime << " - KPD \n";
165     file1.close();
166
167     ofstream file("output");
168     for (i = 0; i < n; i++)
169     {
170       file << resultant_array[i] << " ";
171     }
172     file.close();
173
174     ofstream file2("output1");
175     for (i = 0; i < n; i++)
176     {
177       file2 << data[i] << " ";
178     }
179     for (i = 0; i < n + 1;)
180     {
181       if (i == n) {
182         file2 << ("\n Test succesfull \n");
183         break;
184       }
185       if (resultant_array[i] == data[i])
186         i++;
187       else file2 << ("\n Test not succesfull \n");
188     }
189
190     file2.close();
191   }
192
193   MPI_Finalize();
194   system("pause");
195 }
```

Таким образом, задача была распараллелена на кол-во процессов задаваемое пользователем в качестве аргумента.

1.5.5 Тестирование

Тестирование производительности MPI

Для тестирования производительности MPI на примере функции вычисления экспоненты разработаем тест, который замеряет время выполнения с высокой точностью через функции MPI.

```
1 < ... >
2
3     stopT = MPI_Wtime();
```

```
    double parallelTime = stopT - startT;

    startT = MPI_Wtime();
    sort(data, n);
    stopT = MPI_Wtime();

    double poslTime = stopT - startT;

    startT = MPI_Wtime();
    pthsort(data, n);
    stopT = MPI_Wtime();

    double pthTime = stopT - startT;

    ofstream file1("result");
    file1 << parallelTime << " - Parallel Time \n";
    file1 << poslTime << " - Posledov Time \n";
    file1 << pthTime << " - Pthread Time \n";
    file1.close();
< ... >
```

Результаты тестирования производительности(4 потока):

№	numbers	Simple, s	MPI, s
1	10000	0.924464	0.0662966
2	20000	3.49665	0.268839
3	40000	13.3649	1.03102
4	80000	54.0817	3.98443

Как и ожидалось, с увеличением кол-ва чисел в массиве распараллеливание MPI дает все большее преимущество по сравнению с простым алгоритмом.

Вычислим зависимость от количества процессов MPI и pthread для массива размером 10000 чисел (вычисление производится на процессоре с 4 ядрами):

process count	Simple, s	MPI	Pthread, s
1	0.909266	0.901637	0.008503
2	0.916537	0.261306	0.007064
4	0.905833	0.0696099	0.006640

Сверхлинейный характер ускорения для ряда экспериментов обусловливается квадратичной сложностью алгоритма сортировки в зависимости от объема упорядочиваемых данных.

Тестирование алгоритмов

Тест основывается на том, что результат всех трех алгоритмов с одним и тем же исходным массивом должен быть одинаковым:

```
< ... >

{for (i = 0; i < n + 1;)
    {
      if (i == n) {
        file2 << ("\n Test succesfull \n");
        break;
      }
      if (resultant_array[i] == data[i])
        i++;
      else file2 << ("\n Test not succesfull \n");
    }

< ... >
```

Подобный тест повторяется для всех трех тестовых деревьев.

1.6 Вывод

В ходе работы было разработано несколько алгоритмов для сортировки последовательности чисел.

Многопоточная реализация не является общим решением: приходится жертвовать производительностью на небольших данных, кроме того потоки занимают большое количество памяти. Многопоточную реализацию имеет смысл применять, когда для каждого потока есть трудоемкая подзадача, которую можно выполнять параллельно, чтобы контекстные переключения не так значительно влияли на общую производительность.

Наиболее значимый выигрыш в производительности сортировки у MPI наблюдается при переходе от одного процесса к двум. Также неплохой выигрыш наблюдается при переходе от двух процессов к четырем. Последующее наращивание количества процессов не сильно улучшает производительность.