**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ  
 РОССИЙСКОЙ ФЕДЕРАЦИИ**

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

**НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

**Факультет информационных технологий**

**Кафедра параллельных вычислений**

**ОТЧЕТ**

**О ВЫПОЛНЕНИИ ПРАКТИЧЕСКОЙ РАБОТЫ**

«Программирование многопоточных приложений. POSIX Threads»

студентки 2 курса, группы 22204

**Клочихиной Софьи Павловны**

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

А. Ю. Власенко

Новосибирск 2024

**СОДЕРЖАНИЕ**

ЦЕЛЬ 3

ЗАДАНИЕ 4

ОПИСАНИЕ РАБОТЫ 5

ЗАКЛЮЧЕНИЕ 6

ПРИЛОЖЕНИЕ.

Полный листинг параллельной программы на C++ 9

Полный листинг скрипта SLURM для параллельной программы 14

**ЦЕЛЬ**

1. Освоить разработку многопоточных программ с использованием POSIX Threads API. Познакомиться с задачей динамического распределения работы между процессорами.

**ЗАДАНИЕ**

Есть список неделимых заданий, каждое из которых может быть выполнено независимо от другого. Задания могут иметь различный вычислительный вес, т.е. требовать при одних и тех же вычислительных ресурсах различного времени для выполнения. Считается, что этот вес нельзя узнать, пока задание не выполнено. После того, как все задания из списка выполнены, появляется новый список заданий. Необходимо организовать параллельную обработку заданий на нескольких компьютерах. Количество заданий существенно превосходит количество процессоров. Программа не должна зависеть от числа компьютеров.

Так как задания имеют различный вычислительный вес, а список обрабатывается итеративно, и требуется синхронизация перед каждой итерацией, то могут возникать ситуации, когда некоторые процессоры выполнили свою работу, а другие - еще нет. Если ничего не предпринять, первые будут простаивать в ожидании последних. Так возникает задача динамического распределения работы. Для ее решения на каждом процессоре заведем несколько потоков. Как минимум, потоков должно быть 2:

● поток, который обрабатывает задания и, когда задания закончились,

обращается к другим компьютерам за добавкой к работе;

● поток, ожидающий запросов о работе от других компьютеров.

Можно завести третий поток, который возьмет на себя задачу подкачки работ на компьютер, при этом первый поток будет только обрабатывать задания. В таком случае третий поток, до того, как кончатся задания (соответствующий момент времени определить самостоятельно), на фоне счета будет отсылать запросы о работе и добавлять к локальному списку пришедшие задания.

Необходимо обеспечивать взаимное исключение потоков при добавлении заданий в список, удалении задач, выборке заданий для выполнения.

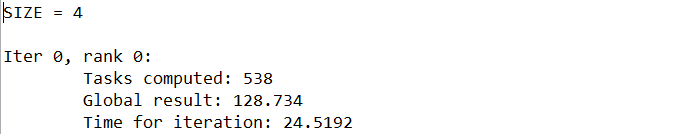
1. Программа не должна зависеть от числа процессов.
2. Использование коммуникаций MPI между процессами.
3. Использование POSIX Threads для порождения нитей в рамках процессов. Минимум в каждом процессе по 2 нити.
4. Вес заданий считать заранее неизвестным для процессов.

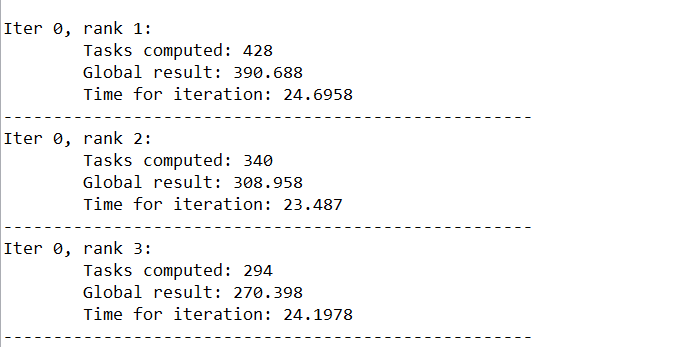
**ОПИСАНИЕ РАБОТЫ**

1. На языке программирования С++ была написана программа с использованием технологии POSIX Threads.
2. Был написан скрипт SLURM и поставлен в очередь

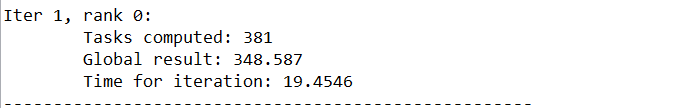


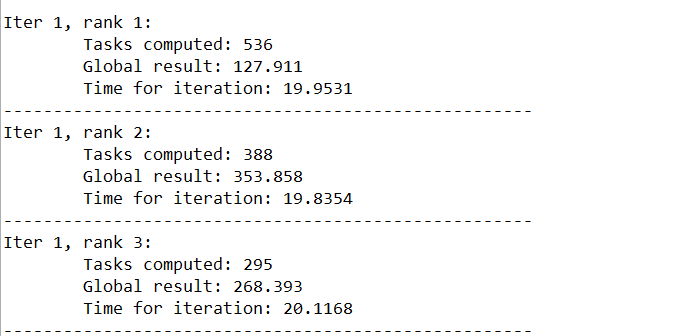
1. Запуск на 4 процессах:



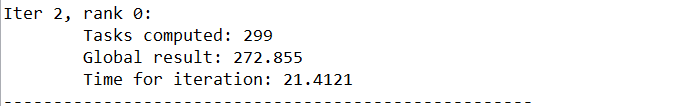


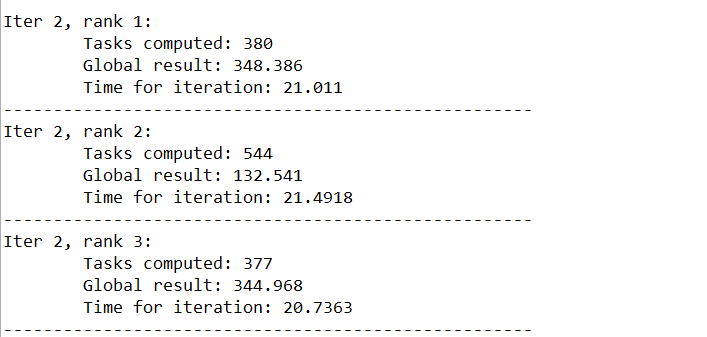


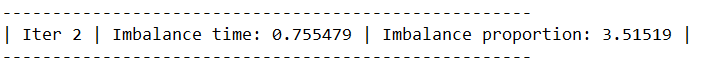


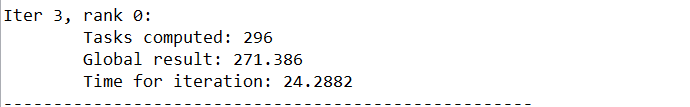


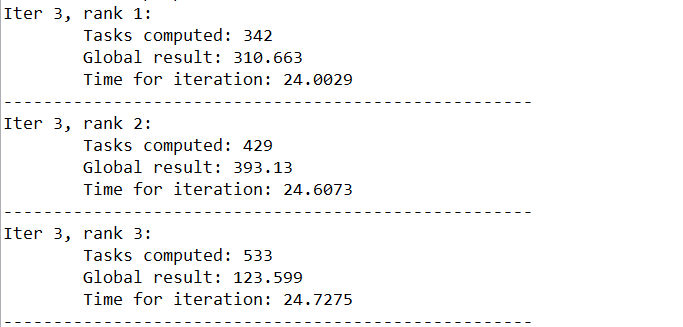


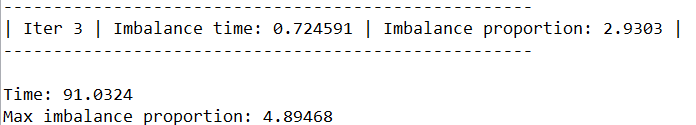












**ЗАКЛЮЧЕНИЕ**

Доля дисбаланса получилась вполне маленькой, что свидетельствует о том, что

программа работает корректно.

1. **Приложение 1.** Полный листинг параллельной программы на C++.
2. #**include** <iostream>
3. #**include** <pthread.h>
4. #**include** <mpi.h>
5. #**include** <deque>
6. #**include** <valarray>
7. std::deque<int> tasks;
8. pthread\_mutex\_t mutex;
9. pthread\_cond\_t criticalCond;
10. bool isEnough = false;
11. const int L = 10000;
12. const int ITER\_COUNT = 4;
13. const int TASKS\_PER\_PROC = 400;
14. int CRITICAL = TASKS\_PER\_PROC / 10;
15. const int NO\_TASKS = -1;
16. double globalRes = 0;
17. int completedTasks = 0;
18. int size, rank;
19. void **printResult**(int, int&, double, double, double&);
20. int **countWeight**(int iterCounter, int idx) {
21. **return** abs(50 - (idx % TASKS\_PER\_PROC)) \* abs(rank - (iterCounter % size)) \* L;
22. }
23. int **getTask**() {
24. int tmp = tasks.front();
25. tasks.pop\_front();
26. **return** tmp;
27. }
28. void **doWork**(int repeatNum) {
29. double tempRes = 0;
30. **for** (int i = 0; i < repeatNum; i++)
31. tempRes += std::sin(i);
32. globalRes += tempRes;
33. completedTasks++;
34. }
35. bool **isAllWorkDone**(const int\* extraTasks) {
36. **for** (int i = 0; i < size; i++)
37. **if** (extraTasks[i] != NO\_TASKS)
38. **return** false;
39. **return** true;
40. }
41. void\* **workerThread**(void\* me) {
42. int repeatNum;
43. **while** (true) {
44. pthread\_mutex\_lock(&mutex);
45. **if** (!tasks.empty()) repeatNum = getTask();
46. **else** repeatNum = NO\_TASKS;
47. pthread\_mutex\_unlock(&mutex);
48. **if** (repeatNum != NO\_TASKS) doWork(repeatNum);
49. pthread\_mutex\_lock(&mutex);
50. **if** (isEnough) {
51. **while** (!tasks.empty()) {
52. //printf("rank %d worker left tasks = %zu\n", rank, tasks.size());
53. repeatNum = getTask();
54. doWork(repeatNum);
55. }
56. pthread\_mutex\_unlock(&mutex);
57. **break**;
58. }
59. **if** (tasks.size() <= CRITICAL) pthread\_cond\_signal(&criticalCond);
60. pthread\_mutex\_unlock(&mutex);
61. }
62. **return** nullptr;
63. }
64. void\* **managerThread**(void\* me) {
65. **while** (true) {
66. int needWorkRank;
67. MPI\_Recv(&needWorkRank, 1, MPI\_INT, MPI\_ANY\_SOURCE, rank, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);
68. **if** (needWorkRank == NO\_TASKS) **break**;
69. int extraTask;
71. pthread\_mutex\_lock(&mutex);
72. **if** (!tasks.empty() && tasks.size() > CRITICAL) {
73. extraTask = tasks.back();
74. tasks.pop\_back();
75. }
76. **else** extraTask = NO\_TASKS;
77. pthread\_mutex\_unlock(&mutex);
78. MPI\_Send(&extraTask, 1, MPI\_INT, needWorkRank, rank, MPI\_COMM\_WORLD);
79. }
80. **return** nullptr;
81. }
82. void\* **requesterThread**(void\* me) {
83. **while** (true) {
84. pthread\_mutex\_lock(&mutex);
85. pthread\_cond\_wait(&criticalCond, &mutex);
86. pthread\_mutex\_unlock(&mutex);
87. int needWorkRank = rank;
88. **for** (int i = 0; i < size; i++)
89. **if** (i != rank)
90. MPI\_Send(&needWorkRank, 1, MPI\_INT, i, i, MPI\_COMM\_WORLD);
91. int\* extraTasks = **new** int[size];
92. **for** (int i = 0; i < size; i++) {
93. **if** (i == rank) {
94. extraTasks[i] = NO\_TASKS;
95. **continue**;
96. }
97. MPI\_Recv(&extraTasks[i], 1, MPI\_INT, i, i, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);
98. }
99. **if** (isAllWorkDone(extraTasks)) {
100. **delete**[] extraTasks;
101. MPI\_Barrier(MPI\_COMM\_WORLD);
102. **break**;
103. }
104. pthread\_mutex\_lock(&mutex);
105. **for** (int i = 0; i < size; i++)
106. **if** (extraTasks[i] != NO\_TASKS)
107. tasks.push\_back(extraTasks[i]);
108. **delete**[] extraTasks;
109. pthread\_mutex\_unlock(&mutex);
110. }
111. pthread\_mutex\_lock(&mutex);
112. isEnough = true;
113. MPI\_Send(&NO\_TASKS, 1, MPI\_INT, rank, rank, MPI\_COMM\_WORLD);
114. pthread\_mutex\_unlock(&mutex);
115. **return** nullptr;
116. }
117. int **main**(int argc, char\*\* argv) {
118. int proc\_rank, procs\_num;
119. MPI\_Init\_thread(&argc, &argv, MPI\_THREAD\_MULTIPLE, nullptr);
120. MPI\_Comm\_size(MPI\_COMM\_WORLD, &procs\_num);
121. MPI\_Comm\_rank(MPI\_COMM\_WORLD, &proc\_rank);
122. rank = proc\_rank;
123. size = procs\_num;
124. pthread\_attr\_t attrs;
125. pthread\_attr\_init(&attrs);
126. pthread\_attr\_setdetachstate(&attrs, PTHREAD\_CREATE\_JOINABLE);
127. pthread\_mutex\_init(&mutex, nullptr);
128. pthread\_cond\_init(&criticalCond, nullptr);
129. double startTime, endTime;
130. int start = TASKS\_PER\_PROC \* rank;
131. int end = start + TASKS\_PER\_PROC;
132. MULTIPLIER /= size \* 2;
133. double maxImbalanceProportion = 0;
134. **if** (!rank) startTime = MPI\_Wtime();
135. **for** (int iterCounter = 0; iterCounter < ITER\_COUNT; iterCounter++) {
136. **for** (int j = start; j < end; j++)
137. tasks.push\_back(countWeight(iterCounter, j));
138. isEnough = false;
139. pthread\_t worker;
140. pthread\_t manager;
141. pthread\_t requester;
142. double iterStartTime = MPI\_Wtime();
143. pthread\_create(&worker, &attrs, workerThread, nullptr);
144. pthread\_create(&manager, &attrs, managerThread, nullptr);
145. pthread\_create(&requester, &attrs, requesterThread, nullptr);
146. pthread\_join(worker, nullptr);
147. pthread\_join(manager, nullptr);
148. pthread\_join(requester, nullptr);
149. double iterEndTime = MPI\_Wtime();
151. printResult(iterCounter, completedTasks, iterEndTime, iterStartTime, maxImbalanceProportion);
152. globalRes = 0;
153. completedTasks = 0;
154. }
155. pthread\_attr\_destroy(&attrs);
156. pthread\_mutex\_destroy(&mutex);
157. pthread\_cond\_destroy(&criticalCond);
158. MPI\_Barrier(MPI\_COMM\_WORLD);
159. **if** (!rank) {
160. endTime = MPI\_Wtime();
161. std::cout << "Time: " << endTime - startTime << std::endl;
162. std::cout << "Max imbalance proportion: " << maxImbalanceProportion << std::endl;
163. }
164. MPI\_Finalize();
165. **return** 0;
166. }
167. void **printResult**(int iterCounter, int& completed, double iterEndTime, double iterStartTime, double& maxImbalanceProportion) {
168. double iterTime = iterEndTime - iterStartTime;
169. **for** (int j = 0; j < size; j++) {
170. MPI\_Barrier(MPI\_COMM\_WORLD);
171. **if** (j == rank) {
172. std::cout << "Iter " << iterCounter << ", rank " << rank << ":" << std::endl;
173. std::cout << "\tTasks computed: " << completed << "\n\tGlobal result: " << globalRes << "\n\tTime for iteration: " << iterTime << std::endl;
174. std::cout << "-----------------------------------------------------" << std::endl;
175. }
176. MPI\_Barrier(MPI\_COMM\_WORLD);
177. }
178. MPI\_Barrier(MPI\_COMM\_WORLD);
179. double maxIterationTime, minIterationTime;
180. MPI\_Reduce(&iterTime, &maxIterationTime, 1, MPI\_DOUBLE, MPI\_MAX, 0, MPI\_COMM\_WORLD);
181. MPI\_Reduce(&iterTime, &minIterationTime, 1, MPI\_DOUBLE, MPI\_MIN, 0, MPI\_COMM\_WORLD);
182. **if** (!rank) {
183. double imbalanceTime = maxIterationTime - minIterationTime;
184. double imbalanceProportion = (imbalanceTime / maxIterationTime) \* 100.0;
185. **if** (imbalanceProportion > maxImbalanceProportion)
186. maxImbalanceProportion = imbalanceProportion;
187. std::cout << "| Iter " << iterCounter << " | Imbalance time: " << imbalanceTime << " | Imbalance proportion: " << imbalanceProportion << " |" << std::endl;
188. std::cout << "-----------------------------------------------------" << std::endl << std::endl;
189. }
190. }
191. **Приложение 2.** Полный листинг скрипта SLURM для параллельной программы.
192. **#!/bin/bash***#SBATCH -J lab5 # Job name  
     #SBATCH -p compclass # Queue name (or "compclass\_unstable", or "gpuserv", or "a100serv")  
     #SBATCH -o lab6.%j.out # Name of stdout output file (%j expands to %jobId)  
     #SBATCH -e lab6.%j.err # Name of stderr output file (%j expands to %jobId)  
     #SBATCH -N 2 # Total number of nodes requested  
     #SBATCH -n 2 # Total number of mpi tasks requested  
     #SBATCH -t 00:10:00 # Run time (hh:mm:ss)*module load mpi/mpich-x86\_64  
       
     if [ "$#" -eq 0 ]; then  
      echo "Usage: $0 <num\_processes>"  
      exit 1  
     fi  
       
     mpicxx -pthread posix.cpp -o posix  
       
     echo -e "SIZE = $1\n"  
     mpirun -np $1 ./posix