|  |  |  |
| --- | --- | --- |
| **Kingdom of Saudi Arabia**  **Ministry of Education**  **University of Jeddah**  **College of Computer Science and Engineering**  **Department of Computer Science and Artificial Intelligence** | Logo, company name  Description automatically generated | **المملكة العربية السعودية**  **وزارة التعليم**  **جامعة جدّة**  **كلية علوم وهندسة الحاسب**  **قسم علوم الحاسب والذكاء الاصطناعي** |

|  |
| --- |
| **Course Project** |
| **CCAI 412 Parallel Computing**  **CCCS 423 Parallel & Distributed Computing**  **2nd Semester1445 - Spring 2024**   |  |  | | --- | --- | | **name** | **UID** | | **Fares Shawli** | **2142645** | | **Saleh Almotairi** | **2041130** | | **Ibrahim Alessa** | **2040547** | | **Ahmed Al-shmrani** | **2141663** | | **Abdulrahman Maghrabi** | **2042791** | |

**Part I: (**PLO S2 / CLO2.2 / SO6**)**

Have a look at the following code snippet of a sorting algorithm. The algorithm exhibits a quadratic dependency in terms of the length of X and can be parallelized easily.

1. void sequential\_sort(std::vector<unsigned int>& X) {

2.

3. unsigned int i, j, count, N = X.size();

4. std::vector<unsigned int > tmp(N);

5.

6. for (i = 0; i < N; i++) {

7. count = 0;

8. for (j = 0; j < N; j++)

9. if (X[j] < X[i] || X[j] == X[i] && j < i)

10. count++;

11. tmp[count] = X[i];

12. }

13.

14. std::copy(tmp.begin(), tmp.end(), X.begin());

15. }

1. Explain how this sorting algorithm works.

The first loop will initialize count as 0

And the second will compare array x [j]<x[i] or if there equal and I must be bigger then j.

if so count will increase and the tmp which is a temporary copy of the array x will = X[i]

so then it will replace X with the temp array.

1. Analyze the data dependencies of each loop. Which loop is ideally suited for a parallelization with OpenMP pragmas? Consider the problem of shared variables.

Outer loop has no dependences other then count but the inner loop is dependent on reading the array X and to write to the temp array if the condition is right and also read the condition and increase the count.

1. Implement a parallel version of sequential\_sort according to your former considerations. Discuss speedup and efficiency.

|  |
| --- |
| #include <vector>  #include <algorithm>  #include <omp.h>  **void** **parallel\_sequential\_sort**(std::vector<**unsigned** **int**>& X) {  **unsigned** **int** i, j, count, N = X.size();  std::vector<**unsigned** **int**> tmp(N);  #pragma omp parallel **for** **private**(i, j, count) shared(X, tmp)  **for** (i = **0**; i < N; i++) {  count = **0**;  **for** (j = **0**; j < N; j++)  **if** (X[j] < X[i] || (X[j] == X[i] && j < i))  count++;  tmp[count] = X[i];  }  std::copy(tmp.begin(), tmp.end(), X.begin());  } |

The speedup will be drastically when the data is large and depending on the number of threads used but it there will not be much efficiency and speed up because of the dependencies .

**Part II: (**PLO S2 / CLO2.2 / SO6**)**

The Prime Factorization algorithm finds the prime factors of a given number. The Parallel Prime Factorization approach can be effective for huge numbers. This can be implemented as a shared-memory or distributed-memory versions.

Here's a high-level overview of the Parallel Prime Factorization algorithm:

* **Domain Decomposition:** In MPI, the range of numbers to be factored is split among the processes that are available; in OpenMP, it is split among the threads.
* **Local Factorization:** Each process or thread performs the prime factorization on its assigned range of numbers.
* **Communication:** In the case of MPI, the processes exchange information about the prime factors they discovered, ensuring that all processes have a complete view of the factorization findings.
* **Global Reduction:** The processes or threads combine their results to provide the final prime factorization of a given number.

1. **Implementation:**

* Implement a program in C or C++ using OpenMP following the algorithm described above. Think about sufficient data structures for storing the elements. Run your program for different values of N, i.e. 1234567890, 23423456780, 234780234562 and 5067891002343 if your computer permit. Run your code for different amount of threads (4, 8, and 16). What do you observe? For control, always print the list of prime factors found.

Because the pc couldn’t run all the numbers because it was too long that I have to cut it down and pick the first 5 numbers.

|  |
| --- |
| #include <stdio.h>  #include <math.h>  #include <time.h>  // Function to perform prime factorization for a given number  **void** **primeFactorization**(**long** **long** n) {  // Print the prime factors  **if** (n == **1**) {  printf("1 is not a prime number.**\n**");  **return**;  }  **if** (n == **2**) {  printf("%lld**\n**", n);  **return**;  }  **while** (n % **2** == **0**) {  printf("%d ", **2**);  n = n / **2**;  }  **for** (**int** i = **3**; i <= (**int**)sqrt(n); i = i + **2**) {  **while** (n % i == **0**) {  printf("%d ", i);  n = n / i;  }  }  **if** (n > **2**)  printf("%lld ", n);  }  **int** **main**() {  **long** **long** N;  **clock\_t** start\_time, end\_time;  **double** execution\_time;  // Ask for input for N  printf("Enter the value of N: ");  scanf("%lld", &N);  // Start measuring execution time  start\_time = clock();  // Iterate from 2 to N and find prime factors  **for** (**long** **long** i = **2**; i <= N; ++i) {  printf("Prime factors of %lld are: ", i);  primeFactorization(i);  printf("**\n**");  }  // End measuring execution time  end\_time = clock();  execution\_time = ((**double**) (end\_time - start\_time)) / CLOCKS\_PER\_SEC;  // Print execution time  printf("Total execution time: %f seconds**\n**", execution\_time);  **return** **0**;  } |

OpenMP code:

|  |
| --- |
| #include <stdio.h>  #include <omp.h>  #include <math.h>  // Function to perform prime factorization for a given number  **void** **primeFactorization**(**long** **long** n) {  // Get the thread ID  **int** thread\_id = omp\_get\_thread\_num();  // Print thread ID and prime factors  **while** (n % **2** == **0**) {  printf("Thread %d: %d**\n**", thread\_id, **2**);  n = n / **2**;  }  **for** (**int** i = **3**; i <= (**int**)sqrt(n); i = i + **2**) {  **while** (n % i == **0**) {  printf("Thread %d: %d**\n**", thread\_id, i);  n = n / i;  }  }  **if** (n > **2**)  printf("Thread %d: %lld**\n**", thread\_id, n);  }  **int** **main**() {  **long** **long** N;  **int** num\_threads;  **double** start\_time, end\_time;  // Ask for input for N and number of threads  printf("Enter the value of N: ");  scanf("%lld", &N);  printf("Enter the number of threads: ");  scanf("%d", &num\_threads);  // Start measuring execution time  start\_time = omp\_get\_wtime();  // Set the number of threads  omp\_set\_num\_threads(num\_threads);  // Perform prime factorization  printf("Prime factors of numbers from 2 to %lld are: **\n**", N);  #pragma omp parallel **for**  **for** (**long** **long** i = **2**; i <= N; ++i) {  primeFactorization(i);  }  // End measuring execution time  end\_time = omp\_get\_wtime();  // Calculate and print execution time  **double** execution\_time = end\_time - start\_time;  printf("Execution time: %f seconds**\n**", execution\_time);  printf("Speedup time: %f **\n**", **115.067000** / execution\_time);  **return** **0**;  } |

* Extend your program with the necessary parallel functionality using MPI. Try to think and use the collective communication commands. Run your program for same values of *N* (from previous implementation), and different amount of processes (i.e. 4, 8, and 16). What do you observe now? For control, always print the list of prime factors found.

|  |
| --- |
| #include <mpi.h>  #include <stdio.h>  #include <stdlib.h>  #include <time.h>  // Function to find prime factors of a number  **void** prime\_factors(**long** **long** **int** n, **int** rank) {  printf("Process %d: Prime factors of %lld: ", rank, n);  **for** (**long** **long** **int** i = **2**; i <= n; i++) {  **while** (n % i == **0**) {  printf("%lld ", i);  n /= i;  }  }  printf("**\n**");  }  **int** main(**int** argc, **char**\*\* argv) {  **int** rank, size;  **long** **long** **int** n = **50678**; // Number to factorize  **double** start\_time, end\_time, total\_start\_time, total\_end\_time;  **double** cpu\_time\_used;  MPI\_Init(&argc, &argv);  MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);  MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);  // Record the start time for the entire program by process 0  **if** (rank == **0**) {  total\_start\_time = MPI\_Wtime();  }  start\_time = MPI\_Wtime();  // Each process finds prime factors of a subset of numbers  **long** **long** **int** start = rank \* (n / size) + **1**;  **long** **long** **int** end = (rank + **1**) \* (n / size);  **if** (rank == size - **1**) {  // Adjust for the last process  end = n;  }    // Find prime factors for the subset of numbers  **for** (**long** **long** **int** i = start; i <= end; i++) {  prime\_factors(i, rank);  }  end\_time = MPI\_Wtime();  printf("Process %d: Execution time: %f seconds**\n**", rank, end\_time - start\_time);  // Wait for all processes to synchronize  MPI\_Barrier(MPI\_COMM\_WORLD);  // Record the end time for the entire program by process 0  **if** (rank == **0**) {  total\_end\_time = MPI\_Wtime();  printf("Total execution time across all processes: %f seconds**\n**", total\_end\_time - total\_start\_time);  }  MPI\_Finalize();  cpu\_time\_used = ((**double**) (end - start)) / CLOCKS\_PER\_SEC;  printf("CPU time used: %f seconds**\n**", cpu\_time\_used);  printf("speedup time : %f seconds**\n**", (**115.067**/cpu\_time\_used));  **return** **0**;  } |

1. **Benchmark:**

* Sketch your results (i.e. execution times) of your OpenMP/MPI program for different N and different amount of processes in two diagrams showing speed-up and parallel efficiency. You can use any plotting tool (Excel, Python, etc.) of your choice.
* In order to ‘benchmark’ your results, also make a theoretical approach to compute speed-up and efficiency according to Amdahl. Therefore, estimate the serial part of your program and compute the corresponding speed-up and efficiency values. Compare those with the ones measured and discuss your observation. Please refer to Chapter 3 in introduction to parallel programming textbook: section 3.6.3 and 3.6.4 when you are writing up the analysis part of your report.

((We sent an excel file ))

speedup=ts/tp

efficiency= speedup/num of pro \*100

1. **Report and presentation:**

* Write a final report (4-5 pages) including a brief description of your parallelization strategy and the necessary MPI communication in your code (i.e. sketch the communication idea). The report should further highlight your achieved results for both parallelization approaches (i.e. OpenMP and MPI), including your diagrams and benchmark results.
* Evaluate your results by discussing if in your opinion the results obtained are good or bad and where you might have ‘lost’ performance within your parallelization. The quality of the write-up is part of the grade. Moreover, please provide a presentation along with the report including your observations from **Part I**.

**Project Submission:**

The project is intended to be worked in groups of 4-5 students. Look for project partners and email me names not later than 30 April 2024 for group number assignment. Once groups have been assigned, they cannot be changed! Always use the assigned group numbers in any kind of correspondence.

For successfully finishing the project students have to send their final report and presentation together with all programs (i.e. source code) not later than **18 May 2024** via blackboard. One submission per group is enough. Please ensure to have your group number on the first page of the report.

Any submission after the deadline may be graded for correctness, but not credited!

In case of problems or questions students can always seek help during office hours or via email, and of course in consultations with one another.

**The evaluation will be based on:**

* First part implementation (1 Mark) X
* OpenMP correct implementation (3 Marks) X
* MPI correct implementation (3 Marks) X
* Report by providing a good benchmarking including speedup and efficiency results and analysis (2 Marks)
* Presentation + presentation showcase (1 Mark)