**Data Decomposition and Reduction Using OpenMP**

Please use Bridges-2 for this assignment. If you have trouble getting interactive access to a compute node using interact, you can submit batch jobs instead.

For our data decomposition example, we will parallelize Jacobi iteration using the OpenMP **parallel** and **for** directives.

**Task 1**. (10 pts) You are provided with a sequential Jacobi iteration code in **jacobi.c**. Compile the code by typing

g**cc -fopenmp -o jacobi jacobi.c -lm**Then get on a compute node using interact and run the code by typing **./jacobi**

The reason we had to compile with -fopenmp, even though jacobi.c is a sequential code, is that we are using the OpenMP timing routines. Run three times and report the average sequential execution time. Be sure to type **exit** or **Ctrl-D** to exit the compute node when you are done.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

The average sequential execution time is:

**12.433507**

**Task 2**. (10 pts) You are provided with an OpenMP version of the Jacobi iteration code in **omp\_jacobi.c**. The code has a single parallel region inside the iteration loop, with several parallel for loops inside that region. The code is correct in that all data sharing is done correctly and there are no race conditions or bad interleaving. Compile the code by typing

**gcc -fopenmp -o omp\_jacobi omp\_jacobi.c -lm**

If you are not still on a compute node, get on a compute node again by typing

**interact --ntasks-per-node=2 -t 10:00**

Set number of threads to 1 by typing

**export OMP\_NUM\_THREADS=1**  
Run the code by typing **./omp\_jacobi**  
Set number of threads to 2 by typing

**export OMP\_NUM\_THREADS=2**  
Run the code again by typing **./omp\_jacobi**

Is the parallel code with one thread about as fast as the sequential code, or is it slower? Does the code speedup with 2 threads or does it slow down? Examine the code and explain the reason for the poor performance.

1. The parallel code with one thread is slower than the sequential version, taking 18.98s compared to ~12.4s.
2. With two threads, the execution time increases significantly to 52.44s, making it even slower.
3. The poor performance is due to parallel overhead, inefficient memory access, and synchronization costs.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

**Task 3**. (20 pts) Fix the performance problem with omp\_jacobi.c that you discovered in Task 2 and recompile the code. Then get 16 cores on a compute node by typing

**interact --ntasks-per-node=16 -t 30:00**

Set OMP\_NUM\_THREADS appropriately to run with 2, 4, 8, and 16 threads. Report your times in a table. Graph the results showing the parallel speedup and comparing with linear speedup. Turn in your modified **omp\_jacobi.c** along with your writeup.

**2 threads**

A screenshot of a computer

AI-generated content may be incorrect.

**4 threads**

A screenshot of a computer

AI-generated content may be incorrect.

**8 threads**

A screenshot of a computer

AI-generated content may be incorrect.

**16 threads**

A screenshot of a computer

AI-generated content may be incorrect.

A graph with a line and a red line

AI-generated content may be incorrect.

A screen shot of a computer

AI-generated content may be incorrect.

One key optimization was replacing critical sections with OpenMP reductions. Previously, the use of #pragma omp critical introduced significant synchronization overhead, as threads had to wait for access to shared variables. By switching to reduction clauses, each thread accumulates results independently, and the final value is computed efficiently at the end, minimizing contention and improving parallel performance.

I also experimented with dynamic scheduling instead of static. This allows the workload to be distributed more flexibly among threads, preventing cases where some threads finish early while others remain overloaded. With schedule(dynamic, 500), work is assigned in smaller chunks, ensuring better load balancing, especially when dealing with irregular computation times across iterations.

Another significant enhancement involved optimizing memory access patterns. In the initial implementation, frequent accesses to neighboring elements in xnew caused cache contention, leading to performance degradation. By carefully restructuring the loop and ensuring better spatial locality, the updated version improves CPU cache utilization, reducing unnecessary memory operations and enhancing overall efficiency.

**Task 4**. (10 pts) Describe any obstacles you encountered in doing this assignment and if/how you overcame them. Summarize what you learned by doing this assignment.

One obstacle I encountered was job submission limits when trying to allocate 16 cores using interact --ntasks-per-node=16 -t 30:00. The system rejected the request due to policy constraints, so I had to adjust my resource allocation accordingly.

Another challenge was performance degradation with multiple threads. Initially, the program ran slower with more threads due to excessive synchronization from #pragma omp critical. I resolved this by using OpenMP reductions, which significantly improved efficiency.

Through this assignment, I learned how to apply data-parallel decomposition effectively and the importance of workload balancing using OpenMP’s scheduling strategies. I also gained insight into memory access optimization, which plays a crucial role in achieving better parallel speedup.