ECE/CSC 506/406: Architecture of Parallel Computers

Project 1: Scheduling in OpenMP Parallel Programming Model

Version 1

Due Date: 30 September 2024, 11:59 PM

1. Objectives

- Get hands-on experience with the OpenMP library and scheduling in an HPC system.
- Understand the limits and overheads of parallelization.

In this project, you'll study Fast Fourier Transform (FFT) Implementation using *OpenMP*, a parallel programming model based on the *Shared Memory Model*. OpenMP is tailored for multi-threaded parallelism in systems with shared memory abstraction, like multi-core processors. Programming with shared memory systems is often more straightforward than with distributed memory systems, primarily because of the direct memory access it offers. This direct access enables faster communication between threads as they can access memory directly instead of transmitting messages across separate spaces. Moreover, OpenMP provides different scheduling modes for distributions of loop iterations among threads. A comparative study of these scheduling modes will provide insights into the OpenMP parallel programming model and the overheads associated with it.

2. Scheduling in OpenMP

OpenMP provides different scheduling modes for distributing loop iterations among threads, ultimately mapped to hardware threads (cores) during runtime. The choice of scheduling can impact the performance and load balancing among threads. Here are the primary scheduling modes in OpenMP.

Static Mode

In static scheduling, iterations are divided into chunks at compile time and then allocated to various threads. When the chunk size isn't specified, OpenMP evenly distributes the iterations into chunks; if the iteration count isn't a multiple of the thread count, it approximates the chunk size. This method is typically faster due to the reduced runtime overhead of determining iteration distribution.

Syntax: `#pragma omp parallel for schedule(static, chunk_size)`

Dynamic Mode

In dynamic scheduling, iterations are segmented into chunks. Threads are immediately given a new chunk once they complete their current one, continuing until all chunks are distributed. This method offers superior load balancing, mainly when workloads vary across iterations. Yet, the trade-off can be a higher overhead due to the ongoing allocation of chunks.

Syntax: `#pragma omp parallel for schedule(dynamic, chunk_size)`

Guided Mode

Guided scheduling, a form of dynamic scheduling, starts by allocating large chunks to threads. As unassigned iterations diminish, chunk size shrinks toward a set minimum or default size. The goal is to harmonize the efficiency of static scheduling with the flexibility of dynamic scheduling.

Syntax: `#pragma omp parallel for schedule(guided, chunk_size)`

3. Algorithms

The algorithm solves a 3D partial differential equation using FFTs in a parallelized manner, utilizing the FastFlow library to split the workload across multiple threads. The program performs multiple iterations, with forward and inverse FFTs, transpositions, and checksum validation at each step.

3-D Fast Fourier Transform (FFT)

1. This code is a parallel implementation of the Fast Fourier Transform (FFT)), and uses the FastFlow library for efficient parallelization. Key constants like SEED, ALPHA, and PI define mathematical parameters, while FFT block sizes control how many FFTs are processed in parallel.
2. Memory is allocated for arrays u0, u1, and twiddle, which store transformed data and coefficients.
3. The execution involves setting initial conditions, performing 1D FFTs in each dimension (x, y, z), transposing arrays, and performing time evolution over multiple iterations.
4. Parallel execution is managed using FastFlow's (Parallel For) to distribute tasks across multiple workers, optimizing performance on multicore systems.
5. The program includes timers to track the execution time of each phase (setup, FFT, evolution), and results are validated at each iteration using a checksum to ensure accuracy.

4. Datasets:

Two datasets are available: '*debug*' and '*test.'* The datasets are already configured and compiled with the source code, and you are not required to do anything. (**For grading, you need only to turn in the results of *"test"***)

3d- FFT

Dataset	**Configuration**
Debug	grid size = 64 x 64 x 64
	no of iterations = 6
Test	grid size = 512 x 256 x 256
	no of iterations = 20

5. Infrastructure

Starter code with inputs and algorithms, compilation, and run routines are provided in the project repository. By providing those, we minimize your burden and let you focus on the most important points.

You should modify the file **FT/ft.c**. The Makefile contains targets to compile these sources with the appropriate compiler and library. (**Please do not modify Makefile**)

You need to run it in the HPC clusters in a shared space. The computation ***must*** be done in an interactive cluster node, not on the login node.

6. Initial Setup

- Log in to the HPC cluster using unityid@login.hpc.ncsu.edu and provide the required credentials. **This is the login node; you must not execute your code here.**
- You should move to the shared storage **cd /share/$GROUP/$USER.** Type "**whoami**" in the terminal you should see your unityid.
- Move the project repo into this folder using *scp* or *sftp* command. e.g. (**scp Project_1.zip falam3@login.hpc.ncsu.edu:/share/ece506f24/falam3** (**Remember your shared folder would be different**)
- Request an interactive node using **bsub -Is -n 32 -W 10 tcsh. (More details on this in HPC training videos)**
- Now, you are ready to modify and execute your code.

7. Running the Code

- *Get into the main folder, and you will find the bin FT folder* that contains the compiled binaries and FFT source code, respectively. It also contains a Makefile that you would use to compile and run the code.
- ***make clean*** to remove prebuilt binaries.
- **make debug** to run the Debug benchmarks.
- **make test** to run the Test benchmarks.
- You can **change the number of threads** by changing the variable **THREADS** in Makefile.

Required Software

Cluster nodes will have all required packages.

8. General Guidelines

- **Start as early as possible**. It will help you understand the infrastructure early and use the rest of the time for coding.
- Run the serial version of the algorithms. (Set THREADS=1)
- Completely understand the code before modifying it.

9. Problems

- Identify the number of loops that are present in the code. Use timing primitives to record the execution time of each loop.
- Change the number of threads to 4 and 8 and record and report the timing. Report the Default Scheduling policy in OpenMP.
- Change the scheduling policy to Dynamic and Static with a Chunk Size of 10 for threads=4 and report the overall execution time.
- Optimize the code to minimize the overall execution time. You can use whatever optimization policy you can come up with. (**You can choose to use additional cores**) (only for **CSC/ECE-506**)

10. Report

Your report should be at most three pages. (Font type - Arial, Font size - 11)

Things will be reported in the report in the strict order provided here.

1. Report the number of loops and what each of the loops is doing.

2. Bar graph of speedup of each loop (with THREADS=4 and 8) normalized to serial code (THREAD=1).
3. Explanation of default scheduling policy of OpenMP. Provide a reference of where you found it and briefly explain it.
4. Bar graph of overall speedup of Static and Dynamic Scheduling policy with THREAD=4 relative to THREAD=1.
5. Bar graph of speedup with the proposed optimization relative to THREAD=1. Provide a brief explanation. (only for CSC/ECE 506).

Run the required workload three times for each graph and use the average. Comparing only the time spent on calculation would eliminate fluctuation in reading/printing. It will give you a better idea of where time is spent (the program has I/O, memory allocation, and deallocation, which are all significant for large inputs).

Please submit the numerical data in an Excel sheet, along with the report in PDF format.

11. Submission Guidelines

Submit the source files **ft.c** in Gradecope. More details will be posted on GradeScope.

12. Grading

ECE/CSC-406

- Successful code compilation in Gradescope (**20%**). **(It is your responsibility to ensure that the code compiles on Gradecope)**
- Identify the number of loops that are present in the code. Use timing primitives to record the execution time of each loop. (**20%**)
- Change the number of threads to 4 and 8 and record and report the timing. Report the Default Scheduling policy in OpenMP. (**20%**)
- Change the scheduling policy to Dynamic and Static with a Chunk Size of 10 for threads=4 and report the overall execution time. (**30%**)

ECE/CSC-506

- Successful code compilation in Gradescope (**10%**). **(It is your responsibility to ensure that the code compiles on Gradecope)**

- Identify the number of loops that are present in the code. Use timing primitives to record the execution time of each loop. (**20%**)
- Change the number of threads to 4 and 8 and record and report the timing. Report the Default Scheduling policy in OpenMP. (**20%**)
- Change the scheduling policy to Dynamic and Static with a Chunk Size of 10 for threads=4 and report the overall execution time. (**20%**)
- Optimize the code to minimize the overall execution time. You can use whatever optimization policy you can develop (**30%**). (**CSC/ECE-506**)

13. Self-grading

- For your convenience, Gradescope Self-grader is provided. It tests only the functional correctness of your code and not parallelization efforts. The grader will compile your code, run it, and compare the output with numdiff tool.
- ***The Project folder contains a validation folder to match the outputs.***
- Currently, custom Makefile is not supported. Your code will be compiled using the given Makefile.