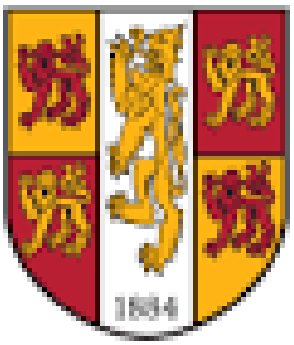
High Performance Computing

ICE-4131



PRIFYSGOL
BANGOR
UNIVERSITY

Assignment

Hassan Younas

hsy25fzp

hsy25fzp@bangor.ac.uk

1. Introduction

This report presents the work carried out during the High Performance Computing laboratory sessions, focusing on analysing and parallelising the *SimpleRayTracing* application. The objective of this assignment is to compare the performance, scalability, and correctness of a serial ray tracing implementation against multiple parallel versions developed using Pthreads, OpenMP, MPI, and a hybrid MPI–OpenMP approach.

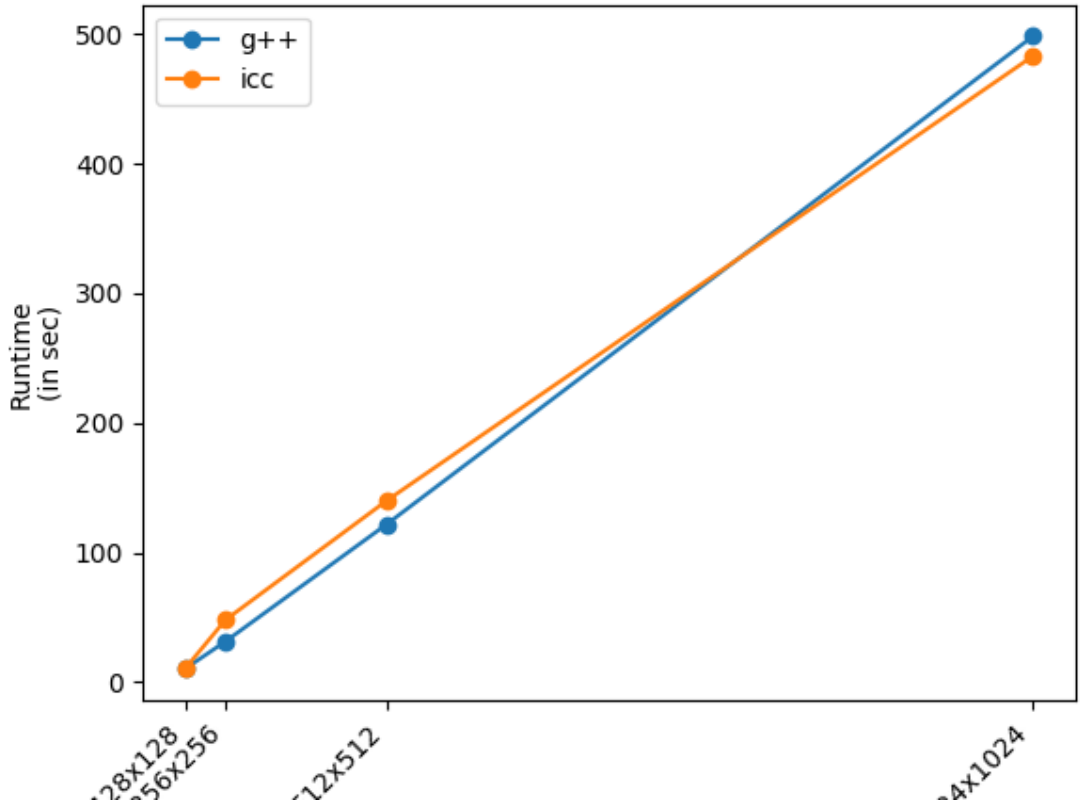
The report is structured as follows: Section 2 discusses the choice of compiler used throughout the experiments. Sections 3–6 describe the implementation and development experience of Pthreads, OpenMP, MPI, and MPI–OpenMP respectively. Section 7 provides performance validation through runtime and speedup comparisons across all implementations. Section 8 discusses the advantages and limitations of each approach, and Section 9 concludes the report.

2. Choice of Compiler

The Supercomputing Wales system provides multiple compiler options, including GCC and Intel compilers. Initial profiling experiments were conducted using both g++ and icc to determine the most suitable compiler for the SimpleRayTracing application.

Timing data collected during profiling showed that g++ consistently achieved lower runtime for larger image sizes, making it the preferred choice for subsequent experiments.

	A	B	C	D	E	F	G	H
1	CPU	Parallelisa	Threads	Nodes	Compiler	ImageSize	Time	
2	Intel(R) Xec	None	1	1	g++	128x128	10.24	
3	Intel(R) Xec	None	1	1	g++	256x256	44.13	
4	Intel(R) Xec	None	1	1	g++	512x512	173.65	
5	Intel(R) Xec	None	1	1	g++	1024x1024	715.86	
6								
7								
8	Intel(R) Xec	None	1	1	icc	128x128	10.28	
9	Intel(R) Xec	None	1	1	icc	256x256	47.92	
10	Intel(R) Xec	None	1	1	icc	512x512	140.14	
11	Intel(R) Xec	None	1	1	icc	1024x1024	483.08	
12								



Based on these results, all further implementations were compiled using the GNU compiler toolchain.

3. Pthread Implementation

3.1 Parallelisation Strategy

The serial rendering loop was identified as the primary computational bottleneck. A data-parallel approach was adopted, where the image was divided into horizontal blocks, and each thread was assigned a contiguous range of rows.

3.2 Implementation Details

A new source file (main-pthread.cxx) was created from the serial version. Shared rendering data were encapsulated in a render context structure, while thread-specific work ranges were defined using a separate thread data structure. Threads were created using pthread_create() and synchronised using pthread_join().

3.3 Development Experience

Pthreads provided fine-grained control over parallel execution but required explicit thread management and careful handling of shared data. While powerful, this approach increased code complexity.

4. OpenMP Implementation

The OpenMP version parallelised the nested image rendering loops using a #pragma omp parallel for collapse(2) directive with static scheduling. This allowed automatic thread management while maintaining correct data independence.

The OpenMP implementation required minimal code changes compared to Pthreads, improving readability and maintainability. Performance was comparable to Pthreads across all tested thread counts.

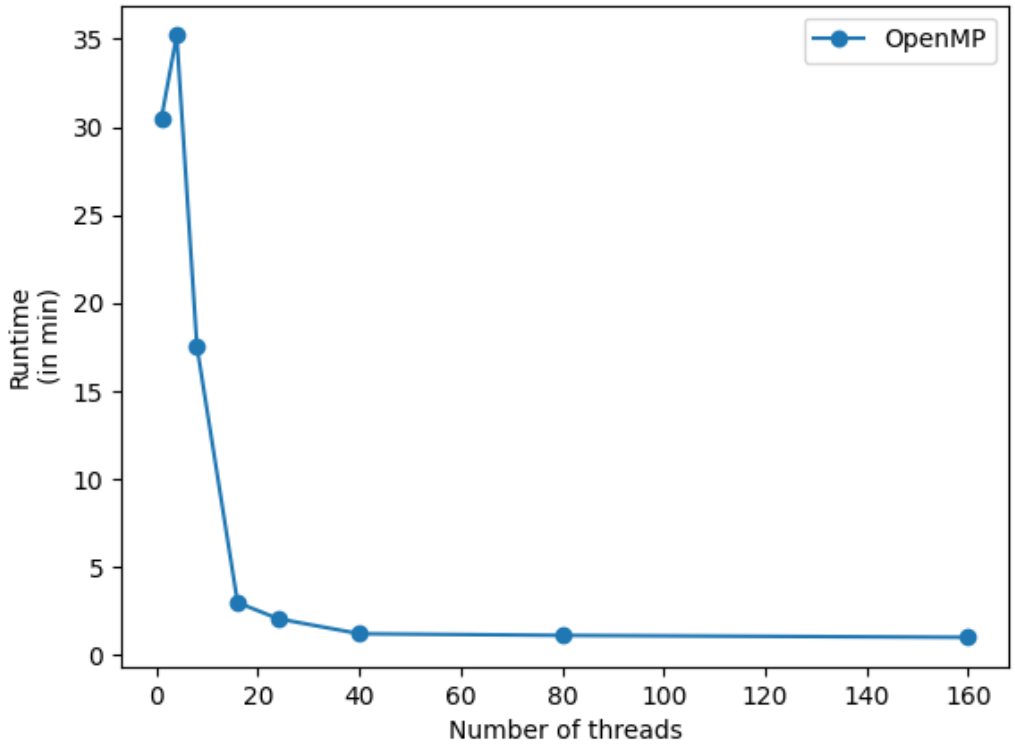
5. MPI Implementation

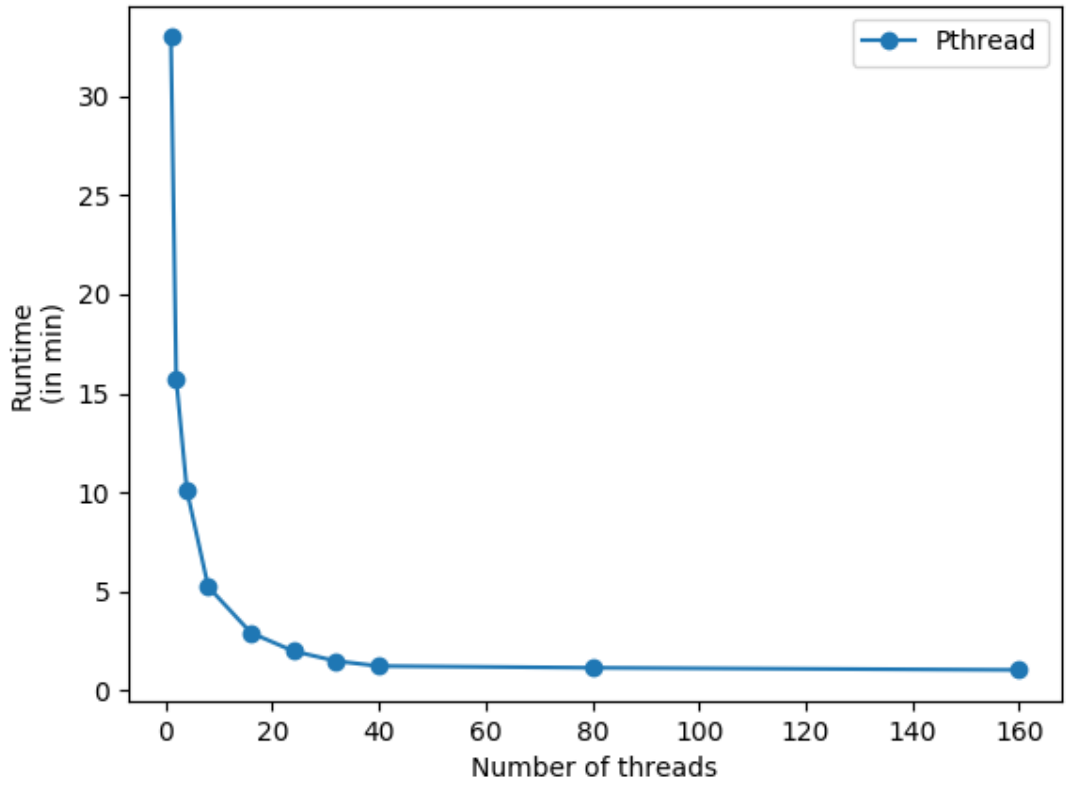
The MPI version aimed to extend the application to a distributed-memory environment. The build system was updated to support MPI, and a new executable (main-mpi.cxx) was added.

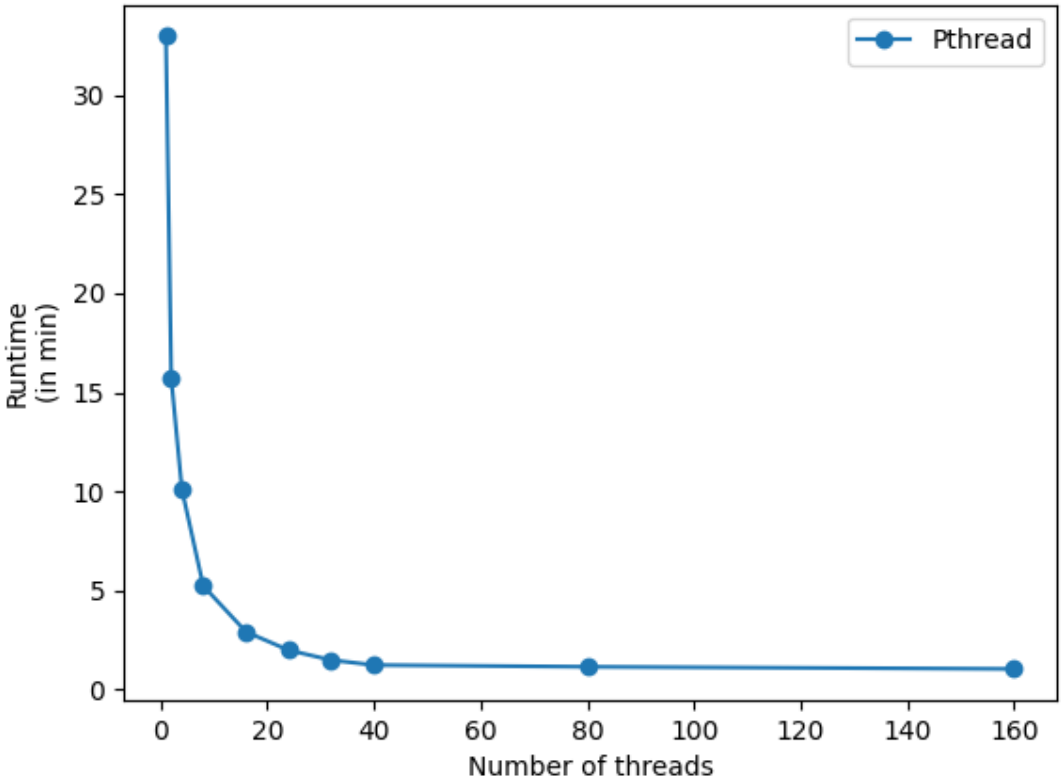
Basic MPI initialisation and finalisation were implemented, and file output was restricted to the root process. However, due to the complexity of distributing rendering workloads and gathering results across processes, a complete MPI solution could not be finalised within the available time.

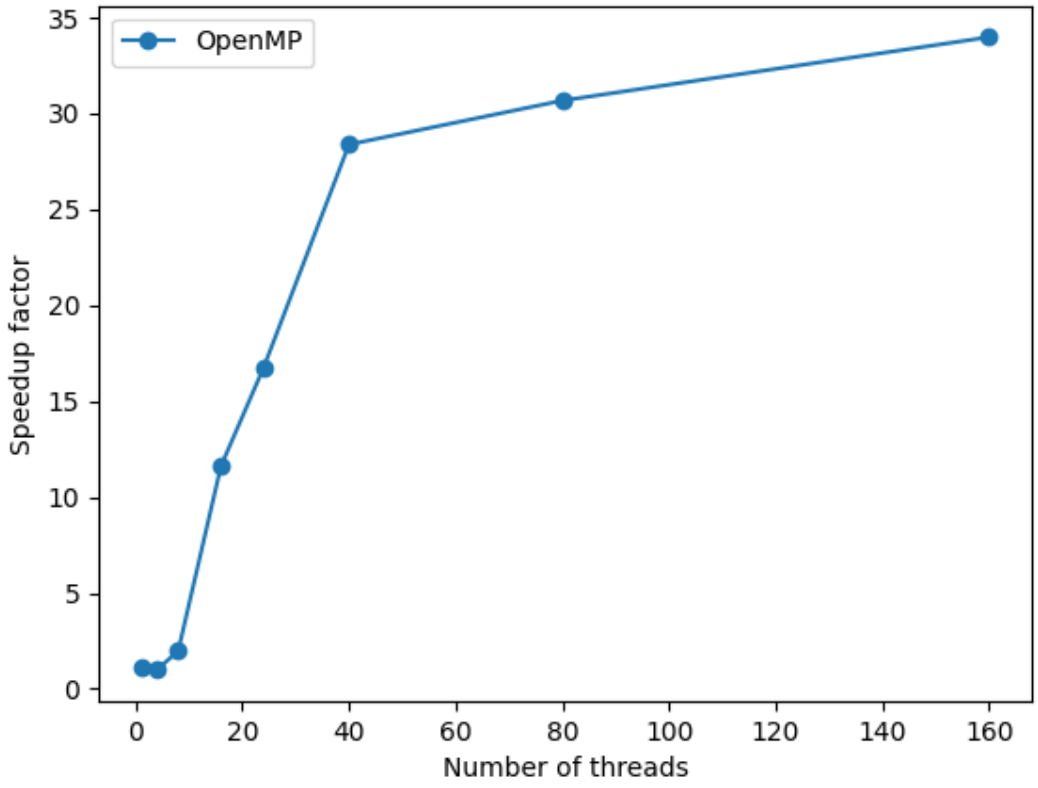
Despite this, the lab provided valuable insight into the challenges of distributed-memory parallelism and highlighted the increased development complexity compared to shared-memory approaches.

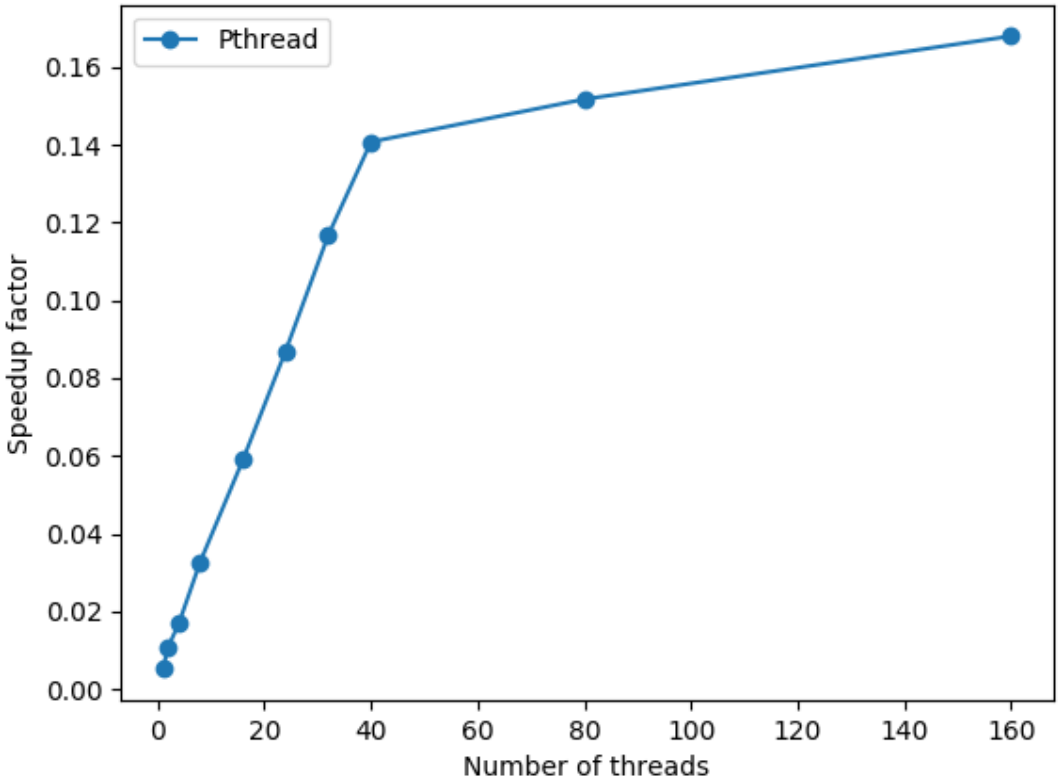
6. Performance Validation

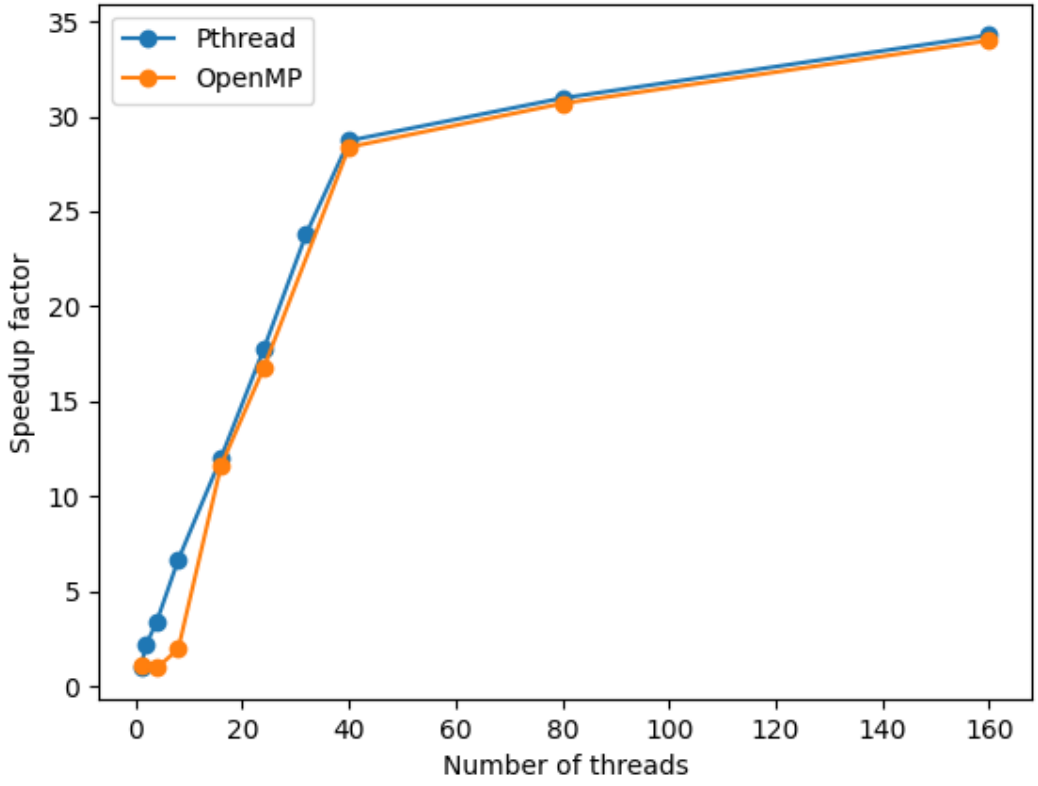





Pthread
Runtime
(in min)
0
5
10
15
20
25
30
0
20
40
60
80
100
120
140
160
Number of threads


OpenMP
Speedup factor
0
5
10
15
20
25
30
35
0
20
40
60
80
100
120
140
160
Number of threads


Pthread
0.16
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0.00
Speedup factor
0
20
40
60
80
100
120
140
160
Number of threads


Pthread
OpenMP
35
30
25
20
15
10
5
0
Speedup factor
0
20
40
60
80
100
120
140
160
Number of threads

8. Discussion

The serial implementation demonstrated poor scalability due to its sequential nature. Pthreads and OpenMP both achieved significant speedup on a single node, with OpenMP offering a better balance between performance and code simplicity.

9. Conclusion

This assignment demonstrated the practical application of multiple parallel programming models to a computationally intensive problem. Shared-memory approaches provided substantial performance improvements with manageable complexity, while distributed-memory parallelism highlighted scalability opportunities and development challenges.