Parallel Optimisation of a Five-point Stencil

George Lancaster
User ID: qv18258 / Candidate Number: 97288

Introduction to High Performance Computing

Introduction

This report outlines and compares four approaches to parallelisation applied to a serially-optimised five point stencil algorithm. MPI optimisations include blocking send and receive, Remote Memory Access (RMA) and asynchronous message passing. An additional optimisation has been written using OpenMP.

Using asynchronous message passing on 16 processors, speeds of 0.0097 seconds, 0.521 seconds and 1.96 seconds were achieved for images of size 1024×1024 pixels (px), 4096×4096px and 8000×8000px respectively. The same optimisation when run on eight nodes with two processors per node, achieved times of 0.01 seconds, 0.136 seconds and 0.643 seconds.

Background

In many cases, parallelisation improves the performance of code through domain decomposition, which involves splitting a problem amongst multiple processors [1]. In doing this, each processor works on a smaller piece of a problem in parallel, and in theory the total amount of wall time taken to execute the code is reduced. Instead of using `MPI_scatter` to do this in this implementation, the original image is initialised on each processor and split into a number of equally sized pieces dependent on the world size. If the problem area is not divisible by the world size, then remainder rows are added to the last process.

As MPI is a distributed-memory programming model, each process running an MPI program creates its own copies of variables, which cannot be implicitly accessed by neighbouring processes. This means that in order to work together, processes must communicate with one other. In the case of the five-point stencil, the image is decomposed into row sections of size $(nx + 2) * (num\_rows + world\_size)$.

Each section contains two extra halo rows, one for the bottom row of the process above it, and one for the top row of the process below it. Neighbouring processes are required to perform a halo exchange after each call to `stencil`, in which the 'true' values for the halo rows are exchanged. This enables the top and bottom rows of each section to be calculated.

In order to build a resulting image, all processes send their section to the master process at the end of computation.

Compiler Choice & Flags

During optimisation, the program was compiled using both the GNU Compiler Collection version 7.1.0 (gcc) with OpenMPI version 2.1.1, and the Intel Compiler Collection version 16 (icc) with the Intel MPI compiler. It is important to use multiple compilers to produce optimum instructions for the target architecture.

In testing, the Intel compilers outperformed gcc, possibly due to the Intel compilers having been developed by Intel for the Intel processors used in BlueCrystal nodes.

The fastest times were measured using the compiler instructions:

```
$ mpiicc -cc=icc -std=c99 -O3 -D NOALIAS -xAVX -restrict
                          -Wall -o@
```

- `mpiicc`: Compile using the Intel MPI compiler;
- `-O3`: Aggressively maximise speed. Automatic vectorisation is enabled at this level;
- `-xAVX`: Enable processor specific optimisations for SandyBridge processors.

Optimisations

This report compares four optimisations: *a*, *b*, *c* and *d*.

Blocking Send Receive (*a*)

Optimisation *a* uses the MPI function `MPI_Sendrecv` to apply the halo exchange between calls to `stencil`. `MPI_Sendrecv` executes a blocking send and receive operation that will not return unless both operations have been successfully completed, making it implicitly thread-safe. Care must be taken when using this function, as large amounts of time can be spent waiting for the recipient of the send to post its reply, emphasising the importance of a proper communication pattern.

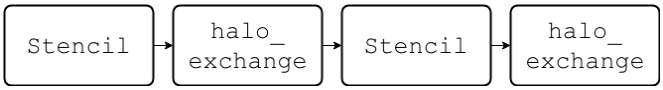

Figure 1: Program flow for optimisation *a*. A halo exchange is performed between calls to `stencil`.

The fastest times recorded for optimisation *a* when run across 16 processes are 0.0098 seconds, 0.526 seconds and 1.97 seconds for images of size 1024×1024px, 4096×4096px and 8000×8000px.

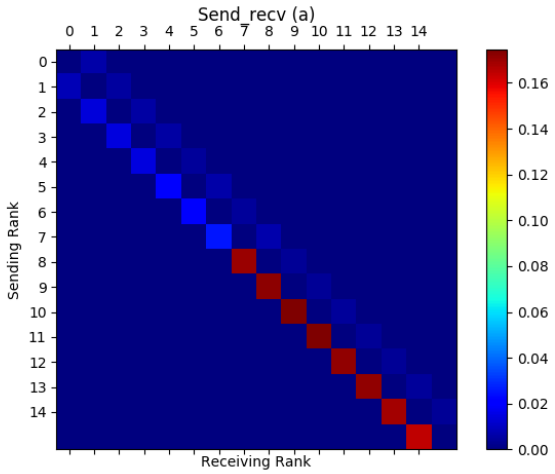

Figure 2: Communication matrix for optimisation *a* when run on problem size 4096×4096px. Communication time has been measured as time spent inside `MPI_Sendrecv`.

The total rank-to-rank communication time in `MPI_Sendrecv` is 1.55 seconds across all processes. This number exceeds the total execution time of the problem.

Figure 2 shows greater communication times in the second half of rank-to-rank communication across all problem sizes. This can be attributed to a communication pattern that does not favour blocking communications.

Asynchronous Message Passing (*b*)

Optimisation *b* uses `MPI_Isend` to perform the halo exchange asynchronously. Because it no longer requires the recipient to post a response, the main work of the `stencil` function is done between the calls to `MPI_Isend` and `MPI_Irecv` that are used to exchange halo rows. This allows the edge cases to be calculated in a separate loop of size $nx + 2$ after the halo rows have been received.

Figure 3: Program flow for optimisation *b*. Most of the work is carried out between `MPI_Isend` and `MPI_Recv`.

Optimisation *b* was motivated by the high communication times observed in optimisation *a*. Although the use of asynchronous message passing has reduced the total rank-to-rank communication time to 0.038 seconds for a problem size of 4096×4096px, there is only a marginal improvement in the overall execution time of the program. Figures 3 and 4 show that for some ranks, asynchronous communication time is 160× faster than when using a blocking send receive formulation. This could be because `MPI_Irecv` is non-blocking.

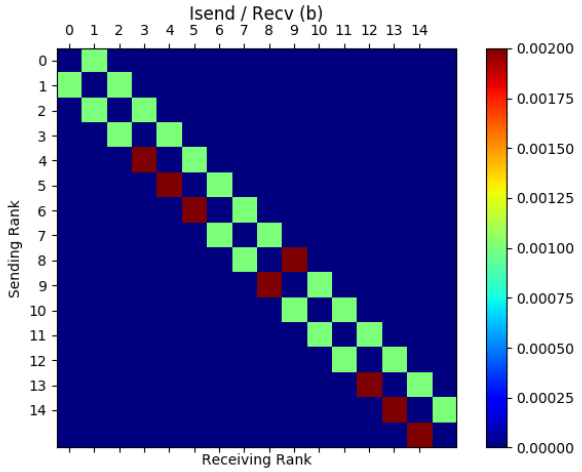

Figure 4: Communication matrix for optimisation *b* when run on problem size 4096×4096. Communication time has been measured as the time spent inside `MPI_Isend` and `MPI_Recv`.

Interestingly, as the time spent in MPI functions decreases, the time spent in `stencil` increases. A possible explanation for this is that the code is memory-bandwidth bound. As a result of this, when communication times are decreased, processes spend more time competing amongst each other for access to memory. A further analysis of this interpretation can be found in the Performance Analysis & Comparison section of the report.

Running on 16 processors, optimisation *b* achieved average execution times of 0.0097 seconds, 0.521 seconds and 1.96 seconds for images of size 1024×1024px, 4096×4096px and 8000×8000px, respectively. This is a marginal improvement over optimisation *a*.

Remote Memory Access (*c*)

Optimisation *c* uses Remote Memory Access (RMA) to open a memory window between ranks that enables one-sided communication. Data can be placed into the memory of neighbouring ranks using `MPI_Put`, without the recipient calling `MPI_Recv` or its derivatives. A 2009 study claims that RMA can reduce data movement time by 40 per cent when compared to a blocking send receive formulation [2].

In the case of the `stencil` code, this was not true. Running on 16 processors, optimisation *c* achieved average times of 0.022 seconds, 0.565 seconds and 2.30 seconds for images of size 1024×1024px, 4096×4096px and 8000×8000px respectively, making it the slowest of the three MPI optimisations. A possible reason for this is that `MPI_Win_fence` is called four times per halo exchange, which causes performance issues due to over-synchronisation. This is evidenced by a greater difference in run times for a problem size of 1024×1024px between optimisation *c* and *b*. For larger problems, the relative difference is not as significant.

Several technical problems were encountered when using the Intel tools to analyse rank-to-rank communication for optimisation *c*. However, according to Application Performance Snapshot, the total rank-to-rank communication times including the time spent in `MPI_Win_create`, `MPI_Win_fence` and `MPI_Put` is 0.046 seconds for a problem size of 4096×4096px.

OpenMP (*d*)

To compare MPI to other parallel programming models, optimisation *d* has been written using OpenMP. Unlike MPI, OpenMP is a shared-memory programming model. It can spawn multiple threads of execution that have shared access to public variables. As a result of this, there is no need to perform a halo exchange as done in the previous MPI implementations.

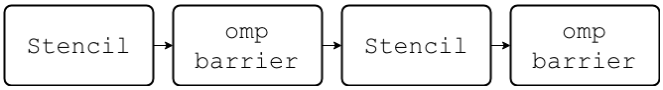

Figure 5: Program flow for optimisation *d*. A synchronisation barrier is used between calls to `Stencil`.

To work in parallel, each thread accesses its own section of the shared image variable. To ensure that all threads are working synchronously, a synchronisation barrier (`#pragma omp barrier`) is used after each call to `stencil`, which ensures that all threads are working on the same iteration. A drawback of this approach is that there are two synchronisation barriers at each iteration. The total time spent inside these barriers is 24.4 per cent of the total execution time for a problem size of 1024×1024px. This does not scale linearly with the problem size, as the wait-time per thread is only 1.8 per cent of total execution time for a problem size of 8000×8000px. Table 1 gives a full breakdown of time wasted at synchronisation barriers.

An additional drawback of OpenMP is that it can only be used on a single node. The scope of this project was

Table 1: The amount of elapsed time that is wasted at OpenMP synchronisation barriers per thread and total execution time for different problem sizes.

Problem Size	Average wait-time per thread	Total Execution Time
1024×1024	0.011	0.045
4096×4096	0.022	1.26
8000×8000	0.061	3.41

not large enough to implement a hybrid MPI/OpenMP optimisation. As a result of this, benchmarking has been limited to 16 processes.

Optimisation *d* achieved average execution times of 0.0280 seconds, 1.29 seconds and 4.67 seconds for images of size 1024×1024px, 4096×4096px and 8000×8000px, respectively. The poor performance of this optimisation can be attributed to the relative simplicity of the implementation.

Performance Analysis & Comparison

To gather accurate data, all reported execution times are given as the mean of three collected times and all outliers have been removed.

Scalability

The five-point stencil algorithm is a strong scaling problem, therefore the execution time is expected to vary with the number of processors for a fixed number of problem sizes (1024×1024px, 4096×4096px and 8000×8000px). To demonstrate the performance of each optimisation on non-square grids, another problem size of 5678×5432px has been included. This resolution was chosen as neither 5678 or 5432 are divisible by 16, the number of processors per node (ppn) in BlueCrystal.

To measure scalability, each optimisation has been run on 1 to 16 processors.

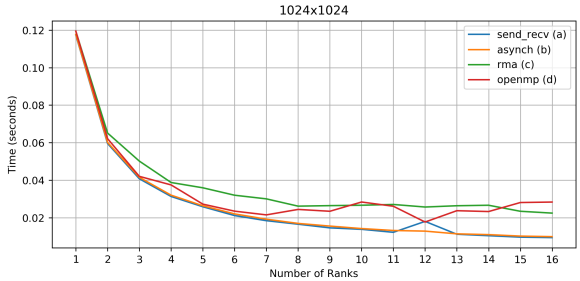

Figure 6: Execution times for all optimisations on a problem size of 1024×1024px for a varying world size.

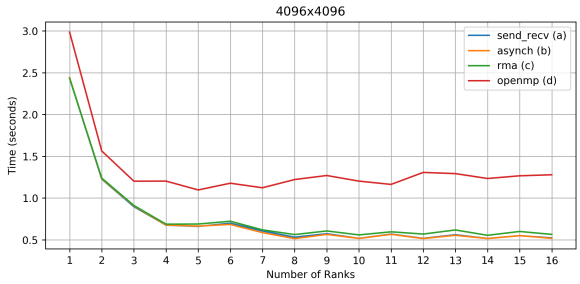

Figure 7: Execution times for all optimisations on a problem size of 4096×4096px for a varying world size.

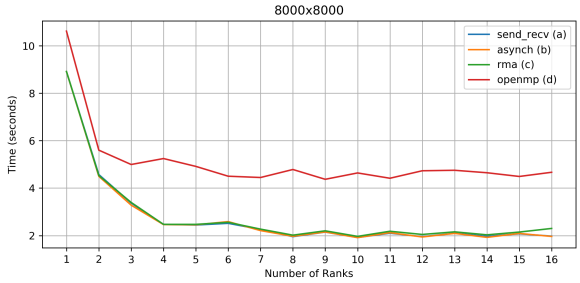

Figure 8: Execution times for all optimisations on a problem size of 8000×8000px for a varying world size.

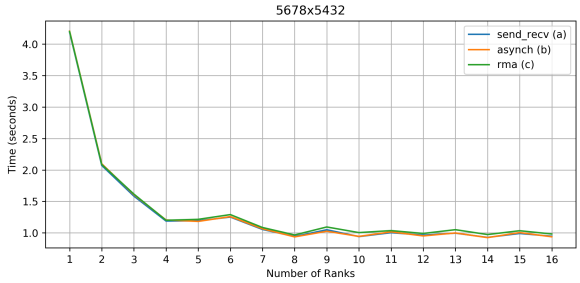

Figure 9: Execution times for all optimisations on a problem size of 5678×5432px for a varying world size.

The scaling curves for MPI optimisations, seen in figures 6-9 can be described as a sub-linear plateau. There is only a marginal improvement when running the program on more than 8 processors. For larger problem sizes, all MPI implementations scale better when run on an even number of processors. This can be observed in both figures 7 and 8.

A possible reason for this is that the Xeon E5-2670 processor shares level 3 cache and DRAM between cores, with the architecture designed to share cache between two cores [3]. An alternative explanation is that there is a greater remainder when there is an odd world-size, which impacts performance due to improper load balancing.

OpenMP doesn't follow the same pattern as the MPI optimisations. The scaling curve of OpenMP can be described as sub-linear declining, indicating that it doesn't scale as well as MPI. For the two largest problems, OpenMP scales better on odd world sizes. The reason for this is unknown.

Optimisation *b* appears to be consistently faster than the other optimisations, although only by a marginal amount. Optimisation *d*, which uses OpenMP, is the slowest of the four optimisations.

STREAM Memory Bandwidth

To compare the performance of the original serial code and the parallel code, two roofline analyses were run using Intel Advisor. As Intel Advisor only measures single core performance, the MPI result was multiplied by 16.

Intel Advisor rates the arithmetic intensity of the serial `stencil` code, compiled using icc 16 at 0.37 FLOP/Byte and a performance of 15.69 GFLOPS. In comparison, optimisation *b* compiled using mpiicc and icc 16 achieved an arithmetic intensity of 0.37 FLOP/Byte and a performance of 241.44 GFLOPS. The theoretical maximum

performance for an optimised parallel code running on one node of BlueCrystal, operating on single precision floating point numbers is 665.6 GFLOPS. Thus, the stencil code is operating at 36.27 per cent of the maximum performance available on a single BlueCrystal node.
The roofline models shown in figures 10 and 11 suggest that the code is memory-bandwidth bound for both the serial and parallel versions. This is indicated by the position of the performance point on the left hand side of the graph.

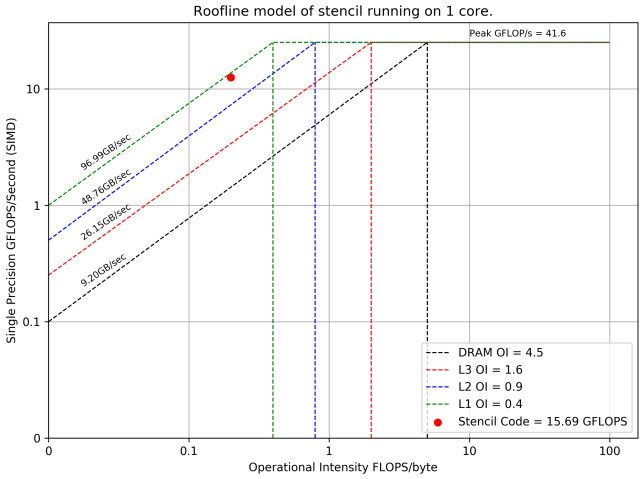

Figure 10: Roofline model for the serial stencil code when run on a single processor. The problem size used was 4096×4096px.

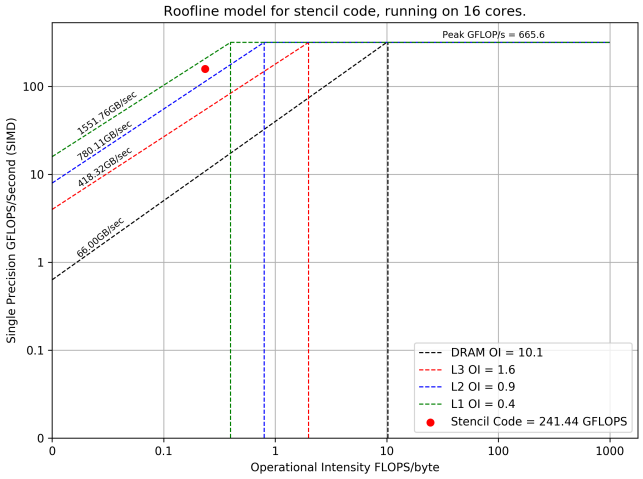

Figure 11: Roofline model for optimisation *b* when run on 16 processors. The problem size used was 4096×4096px.

MPI Time & Mulit-node Performance

The amount of time spent in MPI functions is low compared to the amount of time spent in `stencil` across all MPI optimisations. This leaves very little room for improvement by reducing MPI time. Additionally, figure 11 suggests that the parallel code is memory-bandwidth bound. A possible way to improve performance is to increase the memory-bandwidth by running the code on multiple nodes. Figure 12 shows the result of doing so.
For a problem size of 8000×8000px the execution time is reduced to 0.64 seconds when run on 8 nodes with 2 processors per node, compared to 1.96 seconds when run on 16 processors on a single node. This significant speedup of approximately 3× can be explained by a reduction in cache contention between processors.

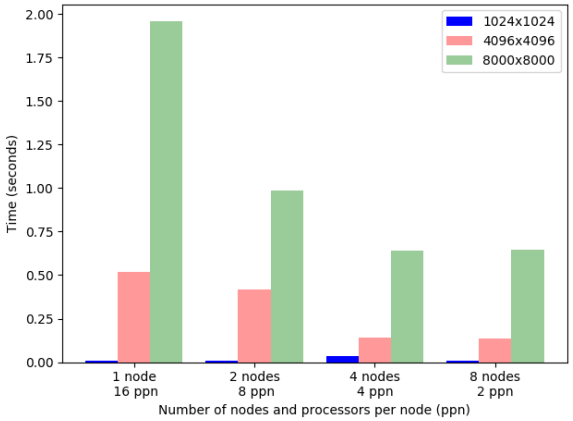

Figure 12: Run times of optimisation *b* , when run on 16 processors across multiple nodes.

Further Work

Further gains to performance could be seen by the implementation of proper load-balancing, which involves allocating remainder rows evenly between processes. An initial implementation was tested, however it failed when run on a single processor.
Additionally, a hybrid MPI/OpenMP implementation could be used to run OpenMP across multiple nodes. Other studies have had success using this approach in the past [4].

Conclusion

Optimisation *b* was found to the the fastest optimisation by a small margin and is the optimisation submitted for marking.
Due to the code being memory-bandwidth-bound, significant gains to performance can be seen by running the code on multiple nodes, which increases the memory-bandwidth and therefore reduces the cache-contention between processors.

References

[1] Mauro Bianco. An interface for halo exchange pattern. www.prace-ri.eu/IMG/pdf/wp86.pdf. Accessed: 10-12-2018.

[2] Torsten Hoefler, James Dinan, Darius Buntinas, Pavan Balaji, Brian Barrett, Ron Brightwell, William Gropp, Vivek Kale, and Rajeev Thakur. Mpi+ mpi: a new hybrid approach to parallel programming with mpi plus shared memory. *Computing*, pages 1121–1136, 2013.

[3] Intel. Xeon Processor E5-2670 product specification. https://ark.intel.com/products/64595/Intel-Xeon-Processor-E5-2670-20M-Cache-2-60-GHz-8-00-GT-s-Intel-QPI-. Accessed: 10-12-2018.

[4] Rolf Rabenseifner, Georg Hager, and Gabriele Jost. Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. *2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pages 427–436, 2009.