An Investigation into the Speed-up of a Ray Tracer Application via the use of OpenMP and MPI

Sam Dixon
40056761@live.napier.ac.uk
SET10108 - Concurrent and Parallel Systems
School of Computing, Edinburgh Napier University, Edinburgh

The abstract goes here.

***Index Terms*—C++11, Ray Tracer, Parallel, OpenMP, MPI, Speed-up.**

I. Introduction

THE aim of this report is to document and analyse the results of an attempt to speed-up a C++11 Ray Tracer application - via the use of concurrency and parallelisation techniques. The methods being tested in this project are `OpenMP` and `MPI`.

A. Ray Tracing

Ray Tracing a rendering technique that allows an image to be generated by tracing the path of a ray as it is reflected through a virtual environment, in order to generate an accurate pixel colour on a 2D image plane. Ray Tracing aims to create photo-realistic images but the computation costs of the technique can result in significant run-times for the generation of highly detailed images. An example of how a Ray Tracer operates can be seen in Figure 1.

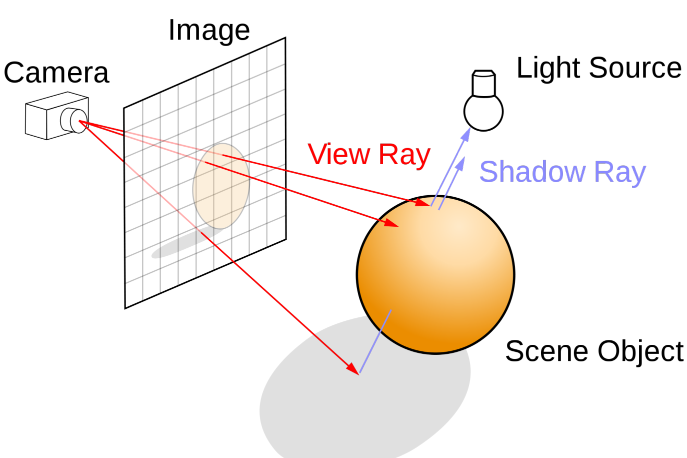


Fig. 1: A diagram of how a Ray Tracer generates an image.

The Ray Tracer being analysed in this project is based on an iterative version of the `smallpt` Ray Tracer [1]. This Ray Tracer can sample a pixel multiple times to generate a more accurate and detailed image. Two sample images are visible in Figure 2. As can be seen - the accuracy and detail of the final images depends heavily upon the number of ray samples per pixel.

Ray Tracers are ideal candidates for improvement via parallelisation as each individual ray has no dependence upon any other, thereby creating a data-parallel (or embarrassingly-parallel) problem.

Project available at: github.com/neaop/SET10108Coursework_2

B. OpenMP

`OpenMP` (Open Multi-Processing) is an open source API that allows for the implementation of shared memory multiprocessing with minimal developmental effort. `OpenMP` makes use of the C++ `#pragma` directive and the pre-processor to allow developers to flag sections of code (particularly for loops) to be parallelised. A number of different scheduling options can be implemented to alter the way in which `OpenMP` parallelises an application.

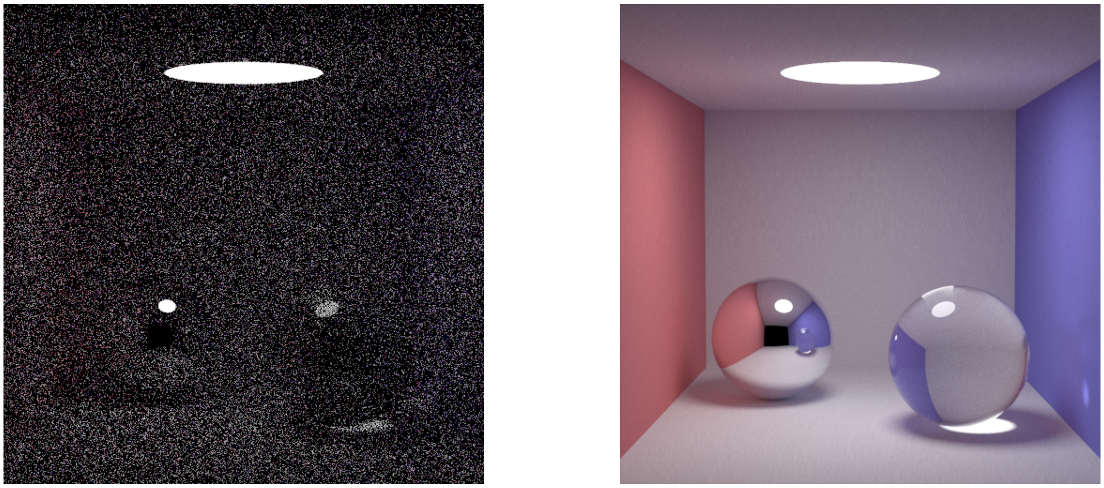
The two schedulers investigated in this project are: `Static` and inter-weaved `Dynamic`. The `Static` scheduler will break a for loop into chunks, each equal to the number of iterations divided by the number of threads. E.g. in the case of a 100 iteration loop split across 4 threads: each thread would run for 25 iterations.

The `Dynamic` scheduler also breaks a single for loop into chunks, however the chunks are typically much smaller than those produced by the Static scheduler. Threads are then assigned a chunk of work, and upon completion can request a new chunk to work on.

C. MPI

`MPI` (Message Passing Interface) is a standardised method of distributed parallelism that operates by having multiple processors communicate by sending and receiving signals from one another via communication channels.

`MPI` allows a developer to build highly scalable systems by simply providing a list of IP addresses when the application is launched. A point that developers must be aware of however is the networking overhead that distributed system innately suffer from. Figure 3 depicts how the amount of time it takes to send data scales with the size of the data being sent. It should be noted that while the time taken to transfer does drop with the size of the data, eventually there is no benefit to adding more



(a) 4 Samples per Pixel (b) 16384 Samples per Pixel

Fig. 2: Two images produced by the Ray Tracer.

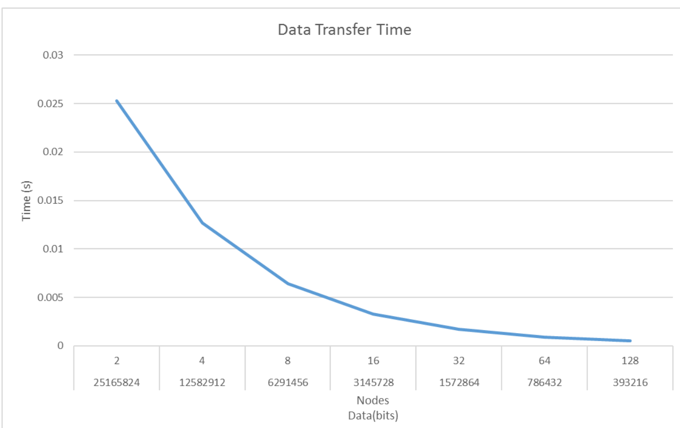


Fig. 3: A line chart depicting the the time required to send data over a network.

PCs to a distributed application. The full data from which the Chart was generated is visible in Appendix II.

II. Methodology

A. Profiling

Prior to implementing any Speed-up methods, the sequential code must first be analysed. By using the Visual Studio Performance Profiler, it is possible to evaluate the sequential code and locate the functions or methods that use the most CPU time. Once the potentially problematic areas have been identified, a suitable parallelisation method can be implemented to reduce the impact of those areas on the execution time.

B. Data Collection

To ensure fair comparison and accurate results, each implementation was tested using the same parameters. Each solution produced an image 512 pixels high by 512 pixels wide, the time taken to do so was recorded. This was repeated one-hundred times for each configuration and a mean of the total time taken was produced. All benchmarking was performed on computers with the same technical specifications, which are visible in Table I. It should be noted that all code presented in the report was run without any form of compiler optimisation.

TABLE I: PC Specifications

CPU	i7-4790k 4 Core HT @ 4.00 ghz
RAM	16gb Dual Channel DDR3
GPU	Nvidia GeForce GTX 980
OS	Windows 7 64 Bit
Bandwidth	1 Gbit/s
Latency	$\sim$ 129947 ns

C. Evaluation

As well as the average execution time, speed-up and efficiency are calculated for each technique. Speed-up is defined as:

$$S = \frac{s_t}{p_t}$$

With s_t being sequential time and p_t being parallel time. Once the speed-up of a method has been calculated, the overall efficiency of the parallelisation can be measured as follows:

$$E = \frac{S}{P}$$

S being speed-up from the previous formula and P is the number of physical cores being utilized by the application.

The two equations listed above provide standardised metrics for each method or technology tested - allowing for a fair and simple comparison of the final results.

III. Implementation

As previously stated, the Ray Tracer presented here is a reimplementation of the iterative `smallpt` system [1].

Several modification had to be made to the original code to allow the implementation of parallelisation. The author of original application had written the Ray Tracer to be under 100 lines of code - which led to said code being highly obfuscated and difficult to read. Much of the development time was spent on expanding, commenting and rewriting the initial code to allow for readability and clearer functionality. Another issue with the original code base was that it was originally designed to be compiled with `GCC` meaning certain methods calls were unavailable

A. *Sequential*

Once the program was cleared up, a simple for loop was added to allow multiple iterations to be performed, and a file writer was used to allow the time taken for each iteration to be recorded. This sequential version of the code became the base for both the `OpenMP`, and `MPI` implementations.

B. *OpenMP*

Very little change had to be made to the sequential code in order to implement `OpenMP`. The most obvious location for parallelisation was the `radiance()` method, which is called from within five nested for loops. By simply inserting a `#pragma parallel for` statement above the upper-most loop - the application was made parallel.

C. *MPI*

IV. RESULTS

A. *Sequential*

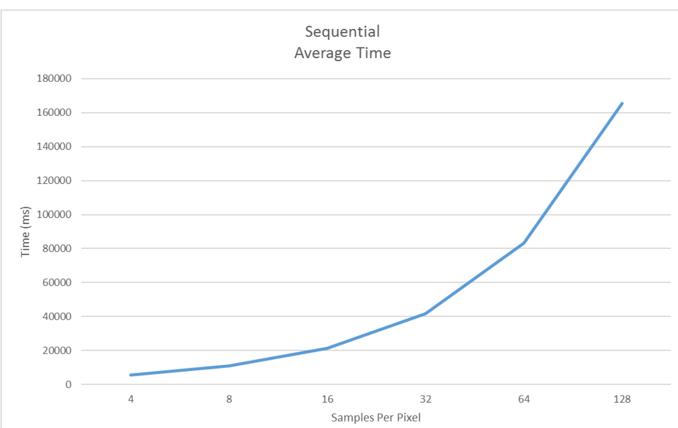


Fig. 4: A line chart indicating the increase in execution time in regard to the number of colour samples per pixel.

B. *OpenMP*

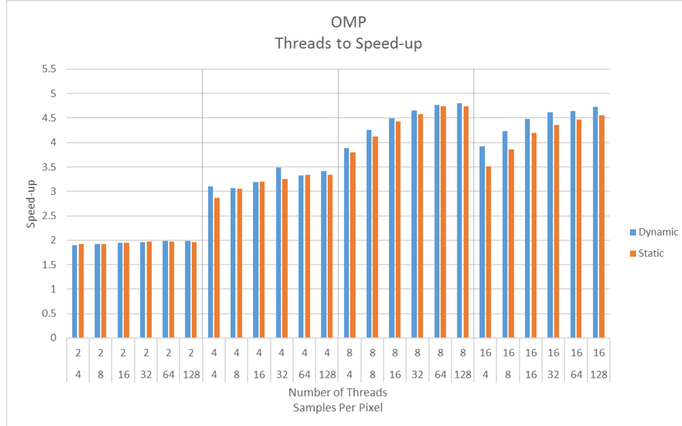


Fig. 5: A bar chart indicating the degree of Speed-up for multiple `OpenMP` schedule and thread configurations.

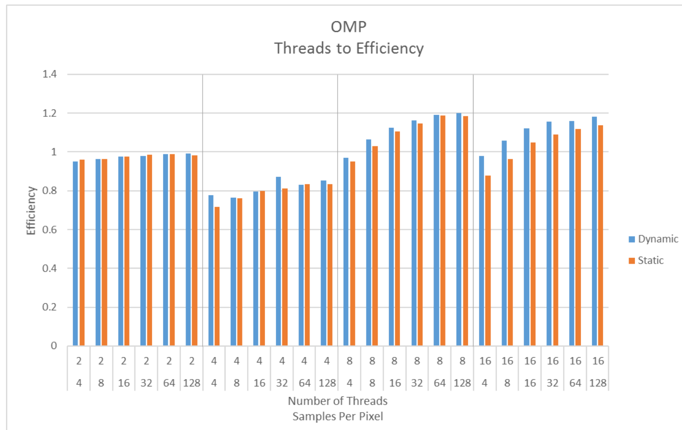


Fig. 6: A bar chart indicating the degree of Efficiency for multiple `OpenMP` schedule and thread configurations.

C. *MPI*

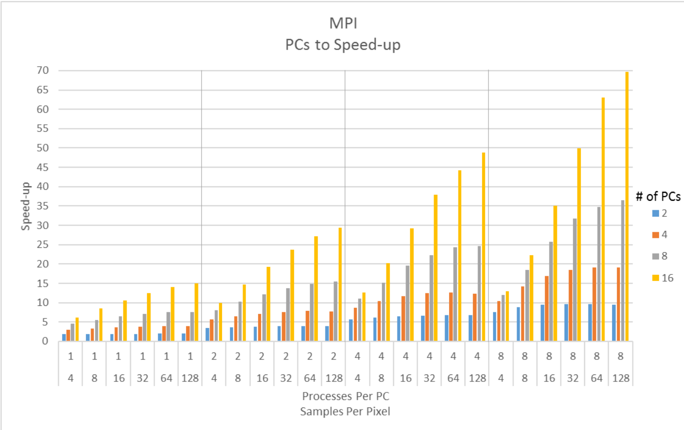


Fig. 7: A bar chart indicating the degree of Speed-up for multiple `MPI` Node and Host configurations.

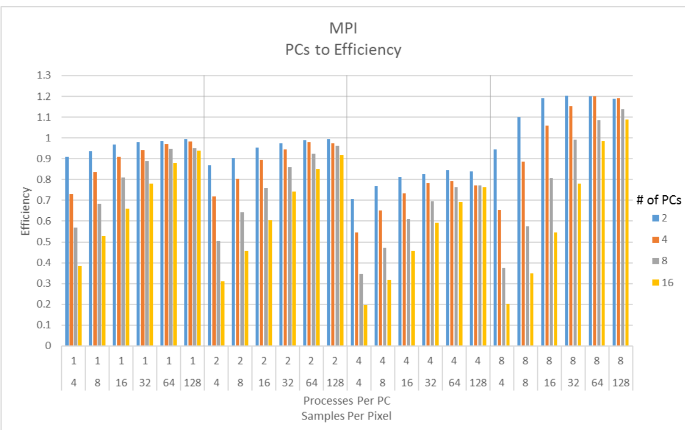


Fig. 8: A bar chart indicating the degree of Efficiency for multiple `MPI` Node and Host configurations.

D. *MPI with OpenMP*

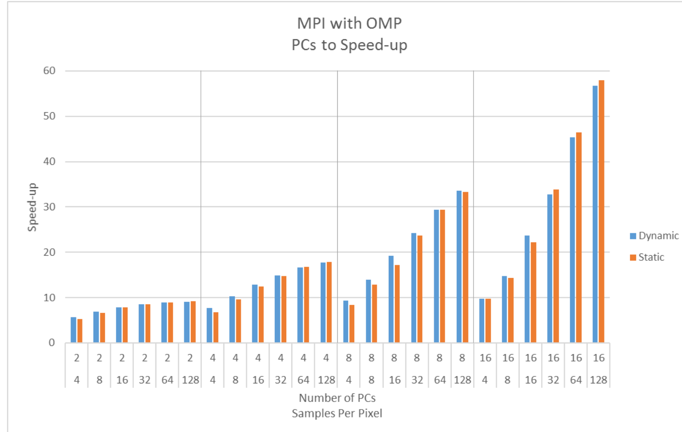


Fig. 9: A bar chart indicating the degree of Speed-up when using `MPI` and `OpenMP` with various scheduling types and number of Hosts configurations.

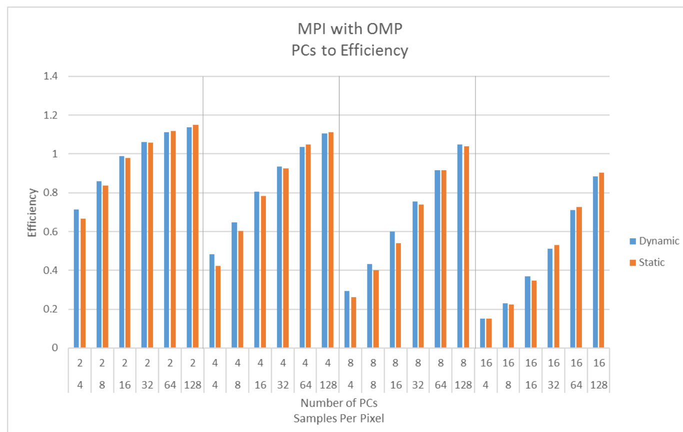


Fig. 10: A bar chart indicating the degree of Efficiency when using `MPI` and `OpenMP` with various scheduling types and number of Hosts configurations.

V. Conclusion

References

[1] K. Beason. (2014) *smallpt: Global Illumination in 99 lines of C++*. (Accessed on 12/14/2016). [Online]. Available: http://www.kevinbeason.com/smallpt/

Appendix A

TABLE II: Data transfer times over a network.

Nodes	Width	Height	Chunk Size	Pixels Per Node	Vec size (bits)	Total Data (bits)	Bandwidth bits/s	Latency(s)	Time (s)
2	512	512	256	131072	192	25165824	1000000000	0.00013	0.02530
4	512	512	128	65536	192	12582912	1000000000	0.00013	0.01271
8	512	512	64	32768	192	6291456	1000000000	0.00013	0.00642
16	512	512	32	16384	192	3145728	1000000000	0.00013	0.00328
32	512	512	16	8192	192	1572864	1000000000	0.00013	0.00170
64	512	512	8	4096	192	786432	1000000000	0.00013	0.00092
128	512	512	4	2048	192	393216	1000000000	0.00013	0.00052

Appendix B

TABLE III: Sequential Benchmarks

Samples	Mean Time
4	5735
8	10794
16	21282
32	41829
64	83256
128	165345

Appendix C

TABLE IV: OMP Speed-up and Efficiency

Schedule	Samples	Threads	Mean Time	Speed-up	Efficiency
Dynamic	4	2	3014	1.90279	0.95139
Dynamic	8	2	5600	1.92750	0.96375
Dynamic	16	2	10895	1.95337	0.97669
Dynamic	32	2	21389	1.95563	0.97782
Dynamic	64	2	42033	1.98073	0.99036
Dynamic	128	2	83384	1.98293	0.99147
Dynamic	4	4	1848	3.10335	0.77584
Dynamic	8	4	3525	3.06213	0.76553
Dynamic	16	4	6684	3.18402	0.79601
Dynamic	32	4	11987	3.48953	0.87238
Dynamic	64	4	25062	3.32200	0.83050
Dynamic	128	4	48495	3.40953	0.85238
Dynamic	4	8	1477	3.88287	0.97072
Dynamic	8	8	2538	4.25296	1.06324
Dynamic	16	8	4731	4.49841	1.12460
Dynamic	32	8	8998	4.64870	1.16217
Dynamic	64	8	17454	4.77002	1.19251
Dynamic	128	8	34447	4.79998	1.20000
Dynamic	4	16	1463	3.92003	0.98001
Dynamic	8	16	2551	4.23128	1.05782
Dynamic	16	16	4749	4.48136	1.12034
Dynamic	32	16	9054	4.61995	1.15499
Dynamic	64	16	17938	4.64132	1.16033
Dynamic	128	16	34953	4.73050	1.18262
Static	4	2	2987	1.91999	0.95999
Static	8	2	5604	1.92612	0.96306
Static	16	2	10903	1.95194	0.97597
Static	32	2	21237	1.96963	0.98481
Static	64	2	42101	1.97753	0.98877
Static	128	2	84127	1.96542	0.98271
Static	4	4	1999	2.86893	0.71723
Static	8	4	3538	3.05088	0.76272
Static	16	4	6654	3.19838	0.79959
Static	32	4	12881	3.24734	0.81184
Static	64	4	24933	3.33919	0.83480
Static	128	4	49619	3.33229	0.83307
Static	4	8	1508	3.80305	0.95076
Static	8	8	2621	4.11828	1.02957
Static	16	8	4806	4.42821	1.10705
Static	32	8	9130	4.58149	1.14537
Static	64	8	17541	4.74637	1.18659
Static	128	8	34886	4.73958	1.18490
Static	4	16	1631	3.51625	0.87906
Static	8	16	2799	3.85638	0.96409
Static	16	16	5073	4.19515	1.04879
Static	32	16	9606	4.35447	1.08862
Static	64	16	18609	4.47396	1.11849
Static	128	16	36336	4.55045	1.13761

APPENDIX D

TABLE V: MPI Speed-up and Efficiency (Part 1)

Samples	Hosts	Nodes	Mean Time	Speed-up	Efficiency
4	2	2	3150	1.82063	0.91032
8	2	2	5762	1.87331	0.93665
16	2	2	11001	1.93455	0.96728
32	2	2	21369	1.95746	0.97873
64	2	2	42198	1.97298	0.98649
128	2	2	83011	1.99184	0.99592
4	2	4	1651	3.47365	0.86841
8	2	4	2982	3.61972	0.90493
16	2	4	5578	3.81535	0.95384
32	2	4	10724	3.90050	0.97513
64	2	4	21024	3.96005	0.99001
128	2	4	41535	3.98086	0.99521
4	2	8	1012	5.66700	0.70837
8	2	8	1758	6.13993	0.76749
16	2	8	3277	6.49435	0.81179
32	2	8	6312	6.62690	0.82836
64	2	8	12331	6.75176	0.84397
128	2	8	24648	6.70825	0.83853
4	2	16	759	7.55599	0.94450
8	2	16	1225	8.81143	1.10143
16	2	16	2232	9.53495	1.19187
32	2	16	4343	9.63136	1.20392
64	2	16	8684	9.58729	1.19841
128	2	16	17395	9.50532	1.18816
4	4	4	1959	2.92751	0.73188
8	4	4	3224	3.34801	0.83700
16	4	4	5856	3.63422	0.90856
32	4	4	11096	3.76974	0.94243
64	4	4	21441	3.88303	0.97076
128	4	4	42075	3.92977	0.98244
4	4	8	999	5.74074	0.71759
8	4	8	1681	6.42118	0.80265
16	4	8	2971	7.16324	0.89541
32	4	8	5533	7.55991	0.94499
64	4	8	10622	7.83807	0.97976
128	4	8	21225	7.79011	0.97376
4	4	16	656	8.74238	0.54640
8	4	16	1038	10.39884	0.64993
16	4	16	1812	11.74503	0.73406
32	4	16	3344	12.50867	0.78179
64	4	16	6569	12.67408	0.79213
128	4	16	13404	12.33550	0.77097
4	4	32	549	10.44627	0.65289
8	4	32	762	14.16535	0.88533
16	4	32	1257	16.93079	1.05817
32	4	32	2265	18.46755	1.15422
64	4	32	4340	19.18341	1.19896
128	4	32	8681	19.04677	1.19042

Appendix E

TABLE VI: MPI Speed-up and Efficiency (Part 2)

Samples	Hosts	Nodes	Mean Time	Speed-up	Efficiency
4	8	8	1259	4.55520	0.56940
8	8	8	1971	5.47641	0.68455
16	8	8	3286	6.47657	0.80957
32	8	8	5881	7.11257	0.88907
64	8	8	10988	7.57699	0.94712
128	8	8	21727	7.61012	0.95126
4	8	16	710	8.07746	0.50484
8	8	16	1049	10.28980	0.64311
16	8	16	1753	12.14033	0.75877
32	8	16	3037	13.77313	0.86082
64	8	16	5633	14.78005	0.92375
128	8	16	10734	15.40386	0.96274
4	8	32	518	11.07143	0.34598
8	8	32	714	15.11765	0.47243
16	8	32	1091	19.50687	0.60959
32	8	32	1882	22.22582	0.69456
64	8	32	3417	24.36523	0.76141
128	8	32	6691	24.71155	0.77224
4	8	64	477	12.02306	0.37572
8	8	64	586	18.41980	0.57562
16	8	64	825	25.79636	0.80614
32	8	64	1320	31.68864	0.99027
64	8	64	2397	34.73342	1.08542
128	8	64	4535	36.45976	1.13937
4	16	16	934	6.14026	0.38377
8	16	16	1278	8.44601	0.52788
16	16	16	2012	10.57753	0.66110
32	16	16	3352	12.47882	0.77993
64	16	16	5921	14.06114	0.87882
128	16	16	10997	15.03546	0.93972
4	16	32	579	9.90501	0.30953
8	16	32	735	14.68571	0.45893
16	16	32	1103	19.29465	0.60296
32	16	32	1762	23.73950	0.74186
64	16	32	3063	27.18119	0.84941
128	16	32	5619	29.42605	0.91956
4	16	64	453	12.66004	0.19781
8	16	64	535	20.17570	0.31525
16	16	64	728	29.23352	0.45677
32	16	64	1104	37.88859	0.59201
64	16	64	1883	44.21455	0.69085
128	16	64	3393	48.73121	0.76143
4	16	128	441	13.00454	0.20320
8	16	128	484	22.30165	0.34846
16	16	128	608	35.00329	0.54693
32	16	128	837	49.97491	0.78086
64	16	128	1320	63.07273	0.98551
128	16	128	2376	69.58965	1.08734

Appendix F

TABLE VII: MPI with OMP Speed-up and Efficiency

Samples	Schedule	Hosts	Nodes	Mean Time	Speed-up	Efficiency
4	Dynamic	2	2	1002	5.72355	0.71544
8	Dynamic	2	2	1572	6.86641	0.85830
16	Dynamic	2	2	2694	7.89978	0.98747
32	Dynamic	2	2	4927	8.48975	1.06122
64	Dynamic	2	2	9367	8.88822	1.11103
128	Dynamic	2	2	18150	9.10992	1.13874
4	Dynamic	4	4	744	7.70833	0.48177
8	Dynamic	4	4	1044	10.33908	0.64619
16	Dynamic	4	4	1652	12.88257	0.80516
32	Dynamic	4	4	2797	14.95495	0.93468
64	Dynamic	4	4	5014	16.60471	1.03779
128	Dynamic	4	4	9351	17.68207	1.10513
4	Dynamic	8	8	610	9.40164	0.29380
8	Dynamic	8	8	778	13.87404	0.43356
16	Dynamic	8	8	1111	19.15572	0.59862
32	Dynamic	8	8	1729	24.19260	0.75602
64	Dynamic	8	8	2840	29.31549	0.91611
128	Dynamic	8	8	4925	33.57259	1.04914
4	Dynamic	16	16	591	9.70389	0.15162
8	Dynamic	16	16	734	14.70572	0.22978
16	Dynamic	16	16	898	23.69933	0.37030
32	Dynamic	16	16	1278	32.73005	0.51141
64	Dynamic	16	16	1834	45.39586	0.70931
128	Dynamic	16	16	2917	56.68324	0.88568
4	Static	2	2	1077	5.32498	0.66562
8	Static	2	2	1613	6.69188	0.83648
16	Static	2	2	2716	7.83579	0.97947
32	Static	2	2	4937	8.47255	1.05907
64	Static	2	2	9297	8.95515	1.11939
128	Static	2	2	17986	9.19298	1.14912
4	Static	4	4	845	6.78698	0.42419
8	Static	4	4	1116	9.67204	0.60450
16	Static	4	4	1701	12.51146	0.78197
32	Static	4	4	2825	14.80673	0.92542
64	Static	4	4	4963	16.77534	1.04846
128	Static	4	4	9288	17.80200	1.11263
4	Static	8	8	682	8.40909	0.26278
8	Static	8	8	844	12.78910	0.39966
16	Static	8	8	1234	17.24635	0.53895
32	Static	8	8	1770	23.63220	0.73851
64	Static	8	8	2837	29.34649	0.91708
128	Static	8	8	4973	33.24854	1.03902
4	Static	16	16	587	9.77002	0.15266
8	Static	16	16	751	14.37284	0.22458
16	Static	16	16	956	22.26151	0.34784
32	Static	16	16	1234	33.89708	0.52964
64	Static	16	16	1793	46.43391	0.72553
128	Static	16	16	2855	57.91419	0.90491