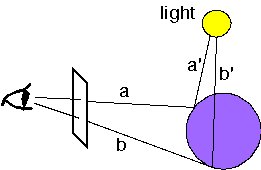
Concurrent & Parallel System CW 1

**Rasmus Munk - 20 November 2015**

# Introduction

The aim of this report is to increase the performance of a ray-tracer implementation create by Kevin Beason by using parallelism techniques such as OpenMP multithreading with different schedulers and MPI. A ray-tracer is a method to render 3d graphic environments by simulating how we see objects in the real world. This is done by using a lighting source which emits rays that bounces of the environment and its objects which in turn hits our eyes and thereby enabling us to see them. In the computer world the ray-tracer uses this approach to render images by only backtracking the rays that hits the vantage point/camera from which the environment is viewed. From each pixel we back trace it through the scene to determine its colour. The outcome colour depends on how many objects and surfaces it has hit on its way from the lightning source to the vantage point. An illustration of this can be seen below:



Source: Reference 1

This report will present itself in the following way. First of the hardware used for all experiments will be described. This is followed by the methodology used to investigate the different possibilities for a performance increase and the results these experiments. To end of a conclusion of this investigation will sum up the results and evaluate on what has been discovered and what the future steps should be in order to improve the performance even further.

# Hardware

All the experiments stated in this report were run on a PC in the games lab(B56) with the following specs:

* CPU – Intel(R) Core(TM) i7-4790K CPU @ 4.00Ghz (4 Cores)
* RAM - 16 GB
* OS - 64-bit Windows 7 Pro SP1
* Visual Studio 2013 Ultimate, 64bit, Release mode with Optimisation.

# Methodology

The forward.cpp(Reference 3) ray tracer implementation was chosen as the starting point to investigate how the performance could be increased. The reason for choosing this exact implementation was that the standard small.cpp(Reference 4) version used the deep recursion technique in the radiance function to backtrack the ray. When executed, most of the attempted sample sizes resulted in stack overflows from this recursion . This could be fixed by limiting the depth of the recursion but to avoid any of this the forward version was used because it uses a while loop instead of recursion and worked right out of the box with all the wanted sample sizes. It did though produce an image that wasn’t as good as the original version but the objective for this investigation was to improve performance of a ray-tracer not produce the best image.

To document that an increase in performance was accomplished the sequential version was first timed both in terms of the overall execution time and the individual parts of the algorithm namely Initialization, Run and Image Render. The Initialization part was where the data structures were declared and the dimensions of the image were set. The Run part was where the actual colour calculations took place in a for loop that goes through each pixel of the image. The Image Render part was where the final image file was created.

To see how the performance of the implementation would be effected by an increase in the sample size the implementation was altered to increase the sample size by the power of 2 for each iteration from 0 to 4. Which at the top would render an image with a sample size of 16.

After having a recorded the sequential performance, the next step was to introduce parallelism into the implementation. First of the timing values of the individual parts was analysed to see where most of the execution time was spent. This indicated as expected that the Run part had the greatest share in the total execution time so this would be the first area to possibly introduce parallelism.

The first parallel technique that would be used for this parallelism was OpenMP. This was done by using an increasing amount of threads that each produce a set of rays in the image that are then rendered sequentially afterwards. The OpenMP was tested with 3 different types of schedulers, dynamic, static and runtime.

The provided implementation was already developed to use OpenMP with dynamic scheduler in the Run part. This parallelism was implemented on the outer for loop that goes through each of the individual pixel to determine the colour value by using back tracing the rays. It was clear each of these rays was back traced independently without any previous execution result required between each iteration. They each operate independently as a classic data parallel problem where the amount of work that should be completed could be distributed out over multiple threads pretty easily. This parallelism point was used as the base point to investigate what type of scheduler provides the best performance with a default chunksize, would it be dynamic, static or runtime. Also how an increasing amount of executing threads would effect the execution time.

Eventually it was discovered that the runtime schedule simply uses the the scheduler type defined in the OMP\_schedule variable (Reference 2). As the results section will show, this variable was properly set to static by default because it had so similar performance measures to the explicit static scheduler.

Besides OpenMP the implementation was also changed to use MPI. This was done so that each process would produce a part of the image by assigning individual start and end points of the image height to each process. This required that when the total height was divided by the number of processes the must result had to be a real positive number. If this was not the case, the implementation wouldn’t produce the correct image because there would be rays missing from the x rows that weren’t produced because of the height being rounded of and some rays were never back tracked.

To put the pieces of the image back together MPI\_Gather was used after all of the processes had completed producing their rays. By using this gathered, every executing process at the end of their execution would send their vector with the calculated pixel colours to process 0. Process 0 would then fill up a vector with all the colour values and execute the render loop with these values and output an image on the local machine. This implementation was tested in a similar way as the previous versions with the same increasing sample size increment of a power to 2 from 0 to 4. This implementation was benchmarked on both a single machine and on two machines/nodes with the same hardware configurations. To make sure that when the implementation was executed with multiple processes an MPI\_Barrier was used to make sure that no process would speed ahead.

What would happen otherwise was that while process 0 created the image file the other processes would continue their own execution eventually get a head start on the backtracking the rays for the next image. Therefore, the timing values in that instance possibly wouldn’t reflect how long it took to backtrack the rays for the entire image. For instance, if process 0 took 5 seconds to backtrack all of the rays in it’s section and process 1 took 6 seconds but because of the head start process 1 gains a second while process 0 was creating the image file it would appear that the entire backtracking in the second iteration only took 5 seconds which wouldn’t be true.

Finally, the MPI and OpenMP implementations were merged to see how the performance would be effected by using both multi threading and MPI processes. The best version from the OpenMP schedulers result was selected for this.

To calculate the speedup and efficiency values a python script was used and therefore the formula used is not available in the excel file but the calculation used can be summed up of being -> speedup = sequential time/parallel time and efficiency = speedup/hardware cores.

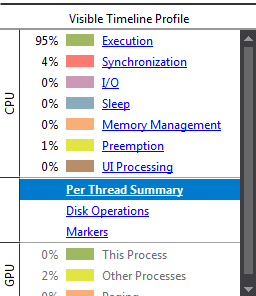
# Results

In this section the results will be presented in the following order. First of the sequential version results will be described this is followed by the parallel OpenMP and MPI versions with their individual speedup and efficiency measures.

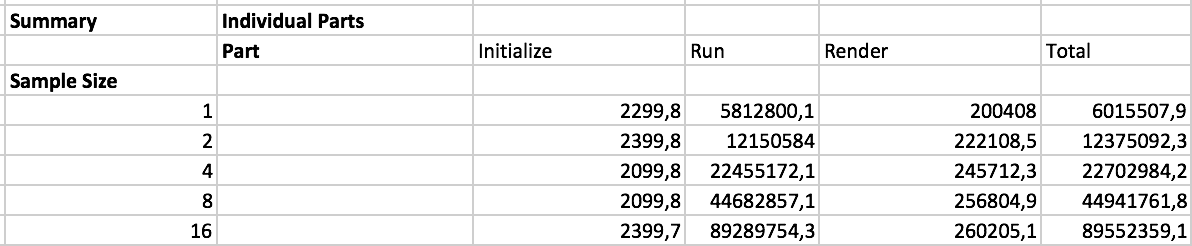
**Sequential Performance**

The sequential version was analysed and benchmarked by both timing it and running it with the concurrency visualizer. This was done to see how much time was spent on execution vs synchronization and pre-emption. As the picture shows (Figure 1.2) 95% of the time in the sequential version was spent on execution with the rest being shared by synchronization and pre-emption. Also in the graph below it shows (Figure 1.1) the results from the sequential benchmarking that displays how the increase in sample size effected the execution time.

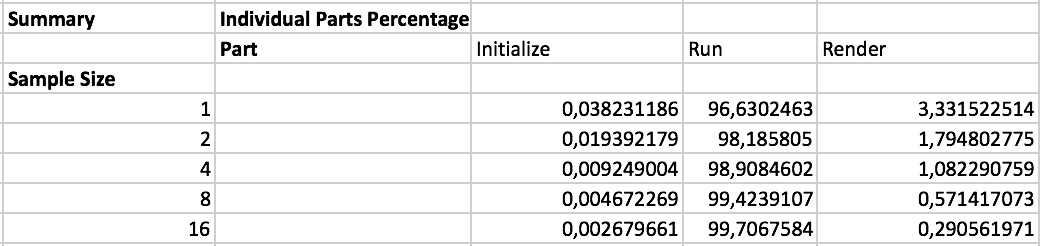
**Figure 1.1**

 **Figure 1.2**

The sequential implementation was also analysed to how the individual parts were effected by this increase in sample size. The results from the individual parts analysis can be in the table below:

**Figure 1.3**

As the tables depicts, for all of the sample sizes most of the execution time was spent in the Run part. Also when the sample size was increased the Run part was the only one that gained a significantly increase in execution time from 5,8 seconds with a sample size of 1 to 89,2 seconds with a sample size of 16. This was not so surprising though because the Run part was responsible for backtracking all the rays that have hit the camera lens. Also as the table beneath shows (Figure 1.4) the Run part was responsible for 96,6 percent of the total execution time at sample size 1 to 99,7 percent at sample size 16. Therefore, it was concluded that the best chance to gain performance would be to investigate whether the Run part could be parallelised.

**Figure 1.4**

**Theoretical speedup**

With these collected sequential times, it was possible to calculate what execution time could be expected if the work got distributed equally amongst the 4 available cores with a single node/machine compared to the 8 cores when executed on two nodes.

For instance, with a sample size of 16 the sequential execution took 89,5 seconds. When this is divided by the number of hardware cores (4) linear speed would be acquired with an execution time of 22,3 seconds. Running on two machines this would be 89,5/8 = 11,1875 seconds.

**Parallel Performance**

The results from the OpenMP scheduler experiments can be seen in the graph below (Figure 2.1). From this graph it can be extracted that with default chunk sizes the static and runtime schedulers performed with similar speedup and efficiency values. Overall though the dynamic scheduler did prove itself faster when there was more than 1 thread executing the code. With a single thread the 1 thread execution the static and runtime scheduler did prove themselves were slightly better speedup. This can’t really be seen in the graph but by looking at the test data, for instance with a sample size of 16 the dynamic had a speedup of 0,966 while the static had a speedup of 0,980. This pattern continues for all of the other different sample sizes with 1 thread execution. This indicates that the dynamic scheduler has a slightly bigger overhead than the static scheduler. The reason for this could be that with the dynamic scheduler the thread requests new iterations every time it has completed some work. Whereas the static scheduler simply gets a set of iterations assigned to it. This requesting from the work queue could be the cause of the bigger overhead. It should be said as the graph depicts that this minimal overhead disappears with the gains of having extra threads executing and requesting work simultaneously.

**Figure (2.1)**

As displayed in the graph (Figure 2.1) the OpenMP Dynamic Default(chunksize) with 8 threads on the 4 cores gained the most speedup from around 3,8 times with a sample size of 1 to just under 4,5 with a sample size of 16. The reason for this speedup actual exceeding the theoretical “linear” speedup value could be attributed to the way the dynamic scheduler works. That is, since the dynamic scheduler doesn’t have an evenly distributed set of iterations for each thread but merely each thread requests more work once its has done an iteration it can perform better than the static scheduler in this instance because of the variance between how many ray bounces each iteration has to calculate. This variance is the result of the variable set of bounces that each ray can have i.e. if a ray hits a ball the thread has to calculate the reflected ray as well, whereas simply hitting a wall straight way ends the back tracking.

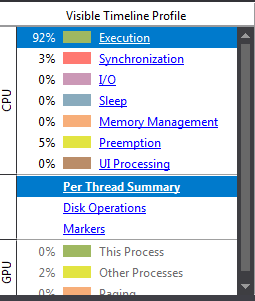
What’s a bit odd though is that the dynamic version with 8 threads at a sample size of 4 goes down in speedup to match the runtime and static 8 threads versions at around 3,8 where it would be expected to be around 4,2 to follow the curve. The reason for this is unknown but it could be that the CPU was simply busy with other things as well during the execution of the test since in this instance all of the threads are in use which doesn’t leave anyone left for other applications to execute instructions on. The overall things to take away is that all of the different configurations provided speedup to the execution time (except for the single threaded versions) with a clear pattern that when the number of threads increases the speedup does as well. This does not mean though that this trend would continue beyond 8 threads, if the number of threads are increased beyond the number of available hardware threads it would undouble result in a loss of speedup because an increased interleaving between the threads. Besides calculating speedup, the efficiency of these configurations was also calculated and can be seen in the graph on the next page (Figure 2.2)

**Figure (2.2)**

The most efficient configuration from the experiments was the same one that had the greatest speedup. This was not so surprising since the efficiency values are based on the speedup values. Generally, for the four best speedup and efficiency versions it showed that an increase in the problem size improved the efficiency and speedup until they hit a sample size of 8 where the improvement flattened out to a slight increase after that.

Besides speedup and efficiency values the best performing “8 threads dynamic scheduler” was also analysed with the concurrency visualizer (Figure 2.3) to see what the increase amount of threads had done to the distribution between executing, synchronization and pre-emptive.

**Figure (2.3)**

**** Figure(2.3) Shows that the execution had suffered by 3% from the sequential version(Figure 1.2) and that the synchronization had gone down by 1%. These decreases had been claimed by the Pre-emption that had gone up by 4%. An increased “blocking time/Pre-Emption”(Reference 5) was not so surprising when the CPU is executing with all of its available hardware threads at the same time.

OpenMP Increased Chunk size (Dynamic, Static)

**MPI Single and Two Nodes**

The speedup and efficiency results of the MPI implementation can be seen in graphs (Figure 2.1 and Figure 2.2). From these it can be extracted that the MPI Single node versions had a slightly better efficiency than their MPI Two nodes counterpart even though the Two Node versions executed the program in a shorter time frame with a greater speedup. This was not a surprise since the Single Node versions speedup values were closer to or above their linear speedup value of 4 compared to their MPI Two Node version counterpart of 8. The cause of this properly lies with the overhead it took for the MPI to gather the data back to the root process over the communication channel (Ethernet in this case) from the other processes that were not running on the local hardware.

This slowed down the execution time compared to doing a local gather from the local memory between the processes where it was directly available in main memory. The same pattern showed in the efficiency graph(Figure 3.2) where both “Single Node with 8 processes and 12 processes” were both more efficient at every sample size than the most efficienct Two Nodes experiement which was the “Two Nodes 16 processes”. Take for instance the sample size of 16, here two best (8 and 12) had an effeciency of 1,0752 and 1,0329 whereas the “Two Nodes 16 processes” version had an efficiency of 0,92466. It should be pointed out though that the gap between them were diminised from an initial gap at sample size 1 of 0,20048 to 0,10829 at sample size 16 and this was proberly due to the networking overhead became a smaller part of the overall execution time as the problem size increased.

Eventhough the Single Node was more efficient with its processes, this dosen’t mean that the trend would continue with an ever increasing amount of processes running on a Single Node. For instance the best speedup and effeciency on the Single Node was with 8 running processes. Here it clearly shows that when going beyond 8 processes on this Single Node as was done with the “Single Node with 12 processes” it showed that simply starting more processes won’t give a greater performance but can indeed hurt both the speedup and effeciency gains. As depicted in the graph(Figure 3.1 & 3.2) where the 8 processes version at sample size 16 had a speedup of 4,30101598 whereas the 12 processes had a speedup of 4,131813 and in terms of efficiency the 12 processes with the same sample size was 0,0423 less efficient.

The reason for this was that the Single Node only had 8 hardware threads available and at 12 processes executing simultainously they had to continuously wait for access to an available thread which slowed down the execution and hurt both the speedup and efficiency.

**Figure 3.1**

**Figure 3.2**

**MPI with dynamic OpenMP**

The speedup and efficiency results of the combined version with MPI running on two nodes and OpenMP with default dynamic scheduler can be seen in the graphs (Figure 4.3 & 4.4). The takeaway from these was that the merger gave an overall gain in speedup compared to their counterpart implementations with the same amount of thread or processes. For instance, the “MPI Two Nodes 2 Processes OpenMP Dynamic 2 Threads” configuration had a speedup of 3,29 at sample size 8 whereas the “MPI Two Nodes 2 Processes” on its own had a speedup of 1,73 and the “OpenMP Dynamic Default 2 Threads” had a speedup of 1,90. So combining them clearly gave a significant speedup but as the data shows, combining these two parallel implementations didn’t give a 1 to 1 speedup in the sense that it isn’t simply a case of adding the two individual speedup to determine the combined speedup as 1,90+1,73 would give a speedup of 3,63 but as the data shows the outcome was 0,34 beneath this. In terms of how the efficiency was effected by this merger the data shows that the OpenMP with dynamic scheduler improved/”pulledup” the efficiency of the MPI with two nodes versions in general and in some instances even succeeded the base OpenMP efficiency. An example of this can be seen in (Figure 4.2) with the merged version “MPI Two Nodes 12 Processes OpenMP Dynamic 8 Threads” at sample size 16 had an efficiency of 1,1113 whereas the MPI version on it’s own had an efficiency of 0,820 and the OpenMP had 1,1074.

The overall threshold at which the combined versions gained more speedup than the best performing ”MPI Two Nodes” version (16 processes) was with the “MPI Two Nodes 2 Processes OpenMP Dynamic 8 Threads” at sample size 2 to 16. Here it shows that the 16 processes MPI version on it’s own had a better speedup at sample size 1 with a speedup of 5,78 whereas the merged version had 5,62. But as figure 4.1 shows that the merged version gained a bigger speedup from the increased sample size to having a 0,4657 better speedup at sample size 16 compared to the “MPI Two Nodes 16 processes” version.

**Figure 4.1**

In terms of efficiency the most efficient MPI version was the ”Single Node 8 Processes” with an efficiency of 1,075 at sample size 16. The breaking point in which the merged version became more efficient than the this was with the “MPI Two Nodes 4 Processes with OpenMP default dynamic Threads 4” version with an efficiency of 1,0823. But as (Figure 4.2) shows in terms of sample size 1 and 2 no one has a better efficiency than the “MPI Single Node 8 Processes”(light blue line). Beyond this sample size the merged versions starts to become more efficient than it. This clearly shows that to make up for the efficiency loss of transferring the data over the network at least 4 OpenMP threads have to be running and that the sample size of the ray tracer should be over 2 before a distribution over 2 nodes becomes more efficient than running 8 processes locally.

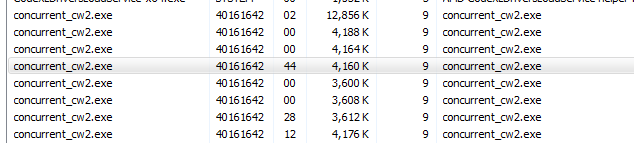
**Figure 4.2**

**Figure 4.3**

**Figure 4.4**

Overall it is a bit strange that the speedup and efficiency continues to grow when the maximum amount of hardware threads were specified to run at the same time across multiple processes as in the case with the “MPI Two Nodes 16 Processes OpenMP Dynamic 8 Threads” performs the best both in terms of speedup and efficiency. It would be expected that with executing 8 processes on each machine where all of which have 8 threads running as showed in (Figure 4.5) that the threads/processes would be “fighting” over the same resources and in result would hurt the speedup as in the case with the “Single Node 12 Processes” had a worse speedup than the “Single Node 8 Processes” version. But according to the data this didn’t happen. It would be interesting to run the “Concurrency Visualizer” while the MPI are executed but how to do this is not known at this point. One reason could be that the OS switches between the processes in such a way that the synchronization is minimal but this is just a hypothesis.

**Figure** **4.5 -** 16 total MPI processes running (8 on each machine) with 8 OpenMP threads



# Conclusion

What can be concluded from this investigation is that to increase the performance of the forward.cpp implementation in terms of both efficiency and speedup the “MPI Two Nodes 16 Processes OpenMP Dynamic 8 Threads” version performed the best at sample size 16 with 9,231 in speedup and 1,1539 in efficiency. The combination of MPI and OpenMP proved to be a good choice in that the OpenMP improved both the efficiency and speedup of the MPI versions and both OpenMP and MPI contributed to a combined increase in speedup.

Overall from the combined versions it showed that increasing the amount of OpenMP threads gave a greater amount speedup and efficiency compared to just increasing the amount of MPI processes. An example of this can be seen in (Figure 4.3 & 4.4) where the data shows that it gave a speedup of 5,51 and efficiency of 0,689 to run 1 process on each of the two nodes with 4 OpenMP threads each with the dynamic scheduler than running 8 processes on two nodes with 1 OpenMP thread that gave a speedup of 5,21 and an efficiency of 0,651.

On it’s own MPI in terms of efficiency the Single Node versions performed better than their Two Node counterparts. This was properly due to the overhead that comes from transferring the pixel data across the network. This does not mean though that it would be better to run the application on a single node because as the data shows(Figure3.1) the two node configurations gave a greater speed because of having 8 CPU cores available to execute the algorithm on instead of 4. Best of both worlds would be to have a single node with 8 cores to avoid the network traffic but in the long run this would be a never ending spiral where an even increasing amount of cores within a single node would be desirable. This is not practical and properly not cost beneficial so multiple nodes should be used to get the improve speedup but with a great attention to minimizing the amount of data that is sent across the network which slows down the execution the most.

In terms of OpenMP the dynamic scheduler proved to be the best choice for this implementation. This was properly due to the fact that the amount of work for each iteration varies in terms of the amount of rays that it has to back trace depends on the objects they have hit on their way from the lighting source to the vantage point/camera. This could be either a reflective surface in which the iteration requires more work to calculate the colour of the pixel or non reflective surface where the iteration simply stops. An interesting thing to further investigate would be to see how the chunk size of the different schedulers effects the performance of the implementation. Also for further work the guided scheduler should be used instead of runtime since runtime just uses either dynamic, static or guided all dependent on what the OMP\_schedule variable has been set to.

# References

1. <https://www.cs.unc.edu/~rademach/xroads-RT/RTarticle.html>
2. <https://software.intel.com/en-us/articles/openmp-loop-scheduling>
3. <http://www.kevinbeason.com/smallpt/forward.cpp>
4. <http://www.kevinbeason.com/smallpt/smallpt.txt>
5. https://msdn.microsoft.com/en-us/library/ee329541.aspx