Final Project Report

High-Performance Computing, Arnaud Ruymaekers, S5298338

Intro

For the final project of High-Performance Computing, I will attempt to parallelize a program computing the Mandelbrot set using OpenMP, MPI, and CUDA.For each of those, I will try different techniques such as changing the scheduling of iterations.

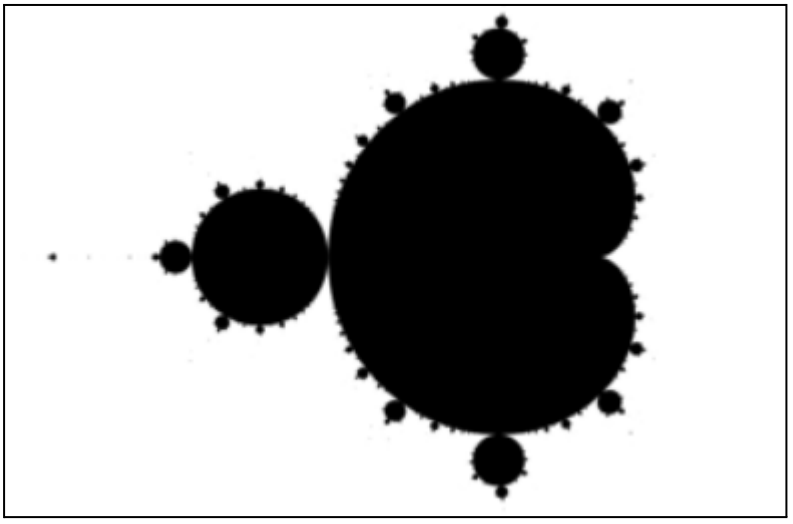
Baseline, and code analysis

Initial Run

To compute a baseline for the program, we can compile the code as follows:
> icc -std=c++11 mandelbrot.cpp -o out.o

Then we simply run the code with "> ./out.o". This runs in 243s for a resolution of 1000 and outputs an image if provided with a path:

- Output (Generated in python by converting to binary data)



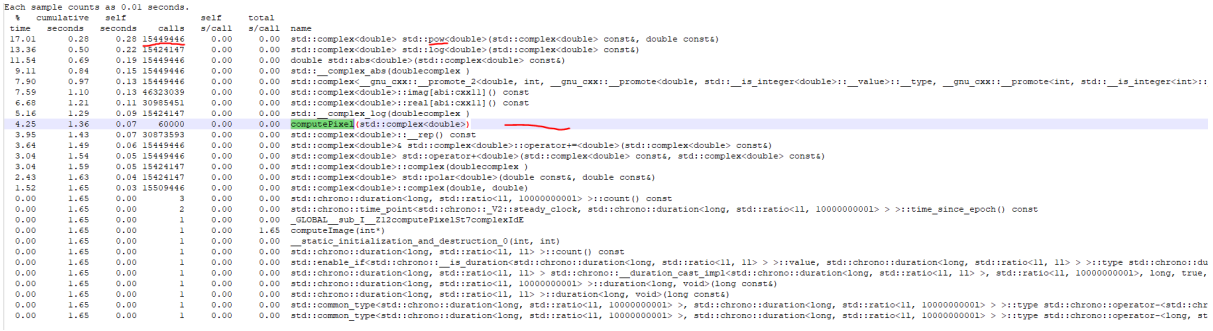
Hotspot Analysis

We can analyze the hotspots by using the gnu compiler with the -pg flag:
> g++ -pg -std=c++11 mandelbrot.cpp -o out.o
and after having executed the program, we can run the profiler (gprof) with:
> gprof -b out.o gmon.out

Note also that the actual computation for the Mandelbrot set has been moved to two functions for the sake of analysis:

- *computeImage()*: Containing the loop over all pixels and call computePixel
- *computePixel()*: That will loop over the asked amount of iterations to compute z and check for convergence

Output:



```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  s/call   s/call  name
 17.01      0.28     0.28 15449446     0.00     0.00  std::complex<double> std::pow<double>(std::complex<double> const&, double const&)
 13.36      0.50     0.22 15424147     0.00     0.00  std::complex<double> std::log<double>(std::complex<double> const&)
 11.54      0.69     0.19 15449446     0.00     0.00  double std::abs<double>(std::complex<double> const&)
  9.11      0.84     0.15 15449446     0.00     0.00  std::__complex_abs(doublecomplex )
  7.90      0.97     0.13 15449446     0.00     0.00  std::complex<__gnu_cxx::__promote_2<double, int, __gnu_cxx::__promote<double, std::__is_integer<double>::__value>::__type, __gnu_cxx::__promote<int, std::__is_integer<int>::
  7.59      1.10     0.13 46323039     0.00     0.00  std::complex<double>::imag[abi:cxx11]() const
  6.68      1.21     0.11 30985451     0.00     0.00  std::complex<double>::real[abi:cxx11]() const
  5.16      1.29     0.09 15424147     0.00     0.00  std::__complex_log(doublecomplex )
  4.25      1.36     0.07    60000     0.00     0.00  computePixel(std::complex<double>)
  3.95      1.43     0.07 30873593     0.00     0.00  std::complex<double>::__rep() const
  3.64      1.49     0.06 15449446     0.00     0.00  std::complex<double>& std::complex<double>::operator+=<double>(std::complex<double> const&)
  3.04      1.54     0.05 15449446     0.00     0.00  std::complex<double> std::operator+<double>(std::complex<double> const&, std::complex<double> const&)
  3.04      1.59     0.05 15424147     0.00     0.00  std::complex<double>::complex(doublecomplex )
  2.43      1.63     0.04 15424147     0.00     0.00  std::complex<double> std::polar<double>(double const&, double const&)
  1.52      1.65     0.03 15509446     0.00     0.00  std::complex<double>::complex(double, double)
  0.00      1.65     0.00        3     0.00     0.00  std::chrono::duration<long, std::ratio<1l, 10000000001> >::count() const
  0.00      1.65     0.00        2     0.00     0.00  std::chrono::time_point<std::chrono::_V2::steady_clock, std::chrono::duration<long, std::ratio<1l, 10000000001> > >::time_since_epoch() const
  0.00      1.65     0.00        1     0.00     0.00  _GLOBAL__sub_I__Z12computePixelSt7complexIdE
  0.00      1.65     0.00        1     0.00     1.65  computeImage(int*)
  0.00      1.65     0.00        1     0.00     0.00  __static_initialization_and_destruction_0(int, int)
  0.00      1.65     0.00        1     0.00     0.00  std::chrono::duration<long, std::ratio<1l, 1l> >::count() const
  0.00      1.65     0.00        1     0.00     0.00  std::enable_if<std::chrono::__is_duration<std::chrono::duration<long, std::ratio<1l, 1l> > >::value, std::chrono::duration<long, std::ratio<1l, 1l> > >::type std::chrono::du
  0.00      1.65     0.00        1     0.00     0.00  std::chrono::duration<long, std::ratio<1l, 1l> > std::chrono::__duration_cast_impl<std::chrono::duration<long, std::ratio<1l, 1l> >, std::ratio<1l, 10000000001>, long, true,
  0.00      1.65     0.00        1     0.00     0.00  std::chrono::duration<long, std::ratio<1l, 10000000001> >::duration<long, void>(long const&)
  0.00      1.65     0.00        1     0.00     0.00  std::chrono::duration<long, std::ratio<1l, 1l> >::duration<long, void>(long const&)
  0.00      1.65     0.00        1     0.00     0.00  std::common_type<std::chrono::duration<long, std::ratio<1l, 10000000001> >, std::chrono::duration<long, std::ratio<1l, 10000000001> > >::type std::chrono::operator-<std::chr
  0.00      1.65     0.00        1     0.00     0.00  std::common_type<std::chrono::duration<long, std::ratio<1l, 10000000001> >, std::chrono::duration<long, std::ratio<1l, 10000000001> > >::type std::chrono::operator-<long, st
```

From this profile, we can see the program spends most of its time computing each pixel. Each pixel has a varying amount of iterations depending on how early the loop exits if convergence is found, represented by how many times the “pow()” function was called. There is an average of 257.5 iterations with a minimum of 1 and a maximum of 1000 (for this run of the program). This means we will have to focus on this part to parallelize.

Vectorization

By analysis of the code we can see that there would be no benefit in trying to improve vectorization as the program doesn’t involve array operations. Each pixel has to be computed individually and is written to a single memory location.

OpenMP

File: mandelbrot_omp.cpp

Parallelization

To parallelize the program using OpenMP, we can define the outer for loop as an “omp parallel for” region. Additionally, the schedule of the loop can be set up to distribute the various iterations to the threads in particular with the “schedule()” clause.

```
#pragma omp parallel for schedule(runtime)
for (int pos = 0; pos < HEIGHT * WIDTH; pos++)
```

Following this, to run experiments, we can set the number of threads using the environment variable OMP_NUM_THREADS and we can set the schedule type and block size using the OMP_SCHEDULE environment variable.

Compilation and run

To compile the code that was built for this we can use the Intel compiler as follows:
> icc -qopenmp -std=c++11 mandelbrot.cpp -o out.o

Then we can set the thread count, the schedule, and the block size and run the code using the following line:
> export OMP_NUM_THREADS=<n>;
export OMP_SCHEDULE="<schedule>,<block_size>"; ./out.o

Experiment Setup

For the experiments, I tried running from 2 to 256 on both the main server and on one of the hpcocapie nodes. Following this, with the most promising thread counts, I am trying different schedules and block sizes.

On the default schedule (so “auto”), I will attempt runs with 1, 2, 4, 8, 16, 32, 64, 128, and 256 threads.

Then, we explore different schedule configurations. I will try the static, dynamic, and guided schedules with a block size of 10, 100, and 1000 iterations.

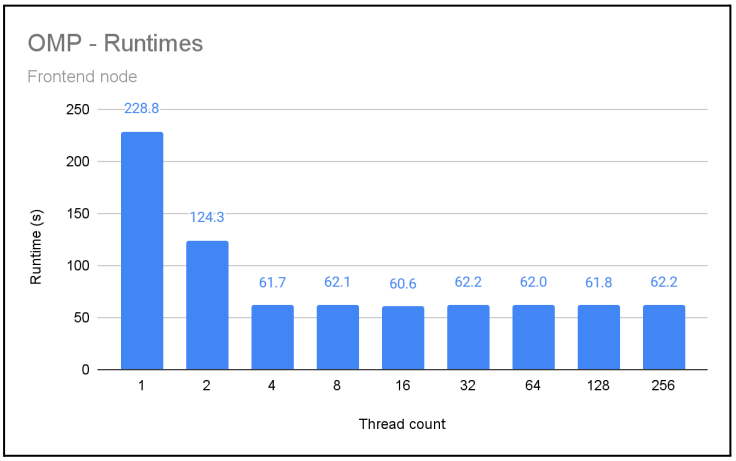
I will attempt to run these various configurations on both the frontend and on one of the hpcocapie nodes.

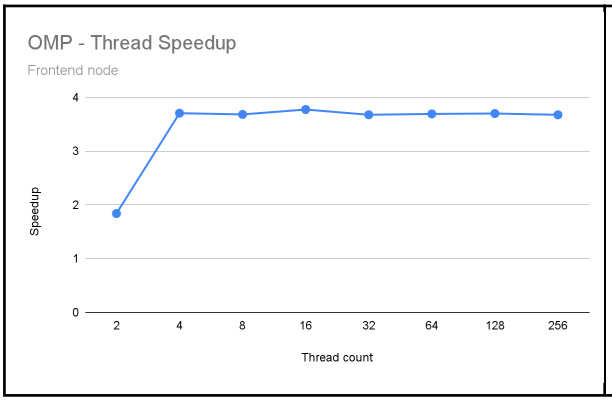
Finally, I will explore a larger problem size to see if the different scheduling options have different impacts at these scales.

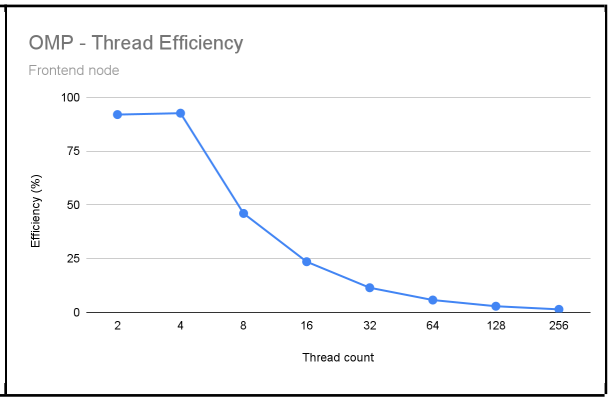
Experiment results

Front node

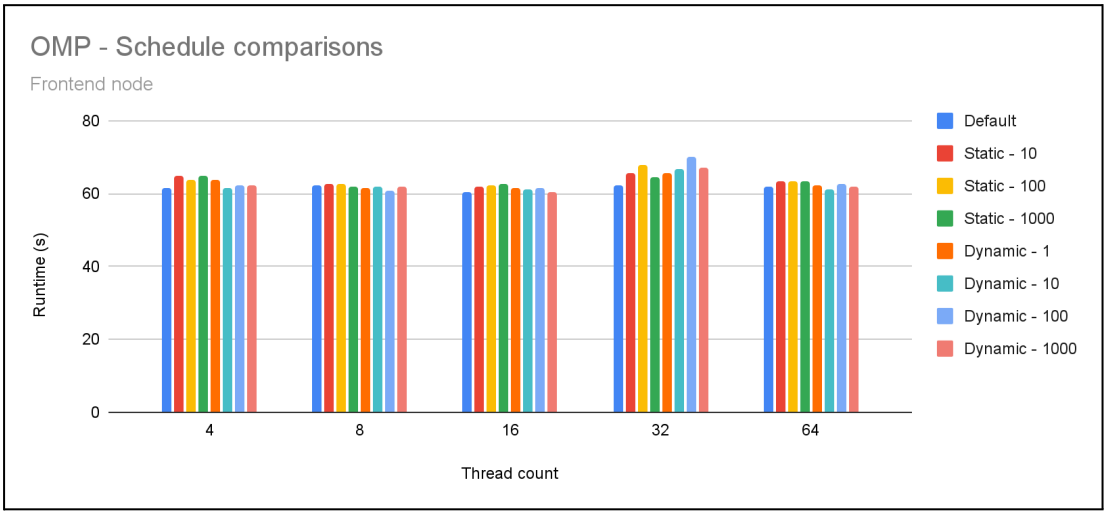
On the Frontend node, when running the program parallelized, we can see some speedup happen up to 4 threads. This makes sense as the machine has 4 cores with 1 thread each. The runtimes reach 60 seconds at 4 cores and don't improve any further beyond 4 cores.







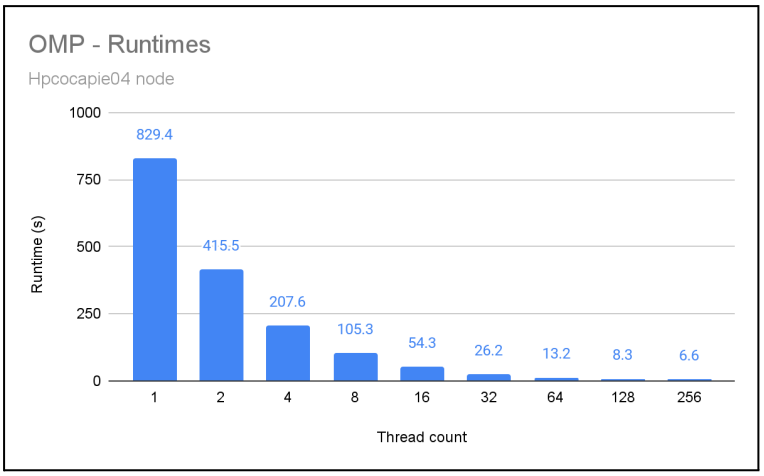
Regarding changing the schedule and block size, there are small to no changes in the runtimes. Static scheduling resulted in consistently higher runtimes compared to automatic scheduling. The dynamic schedule, however, displayed a small improvement with a block size of 10 or 100 when using 4, 8, or 16 threads.

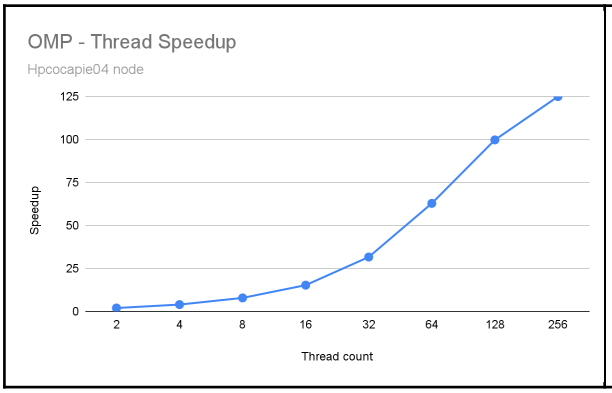


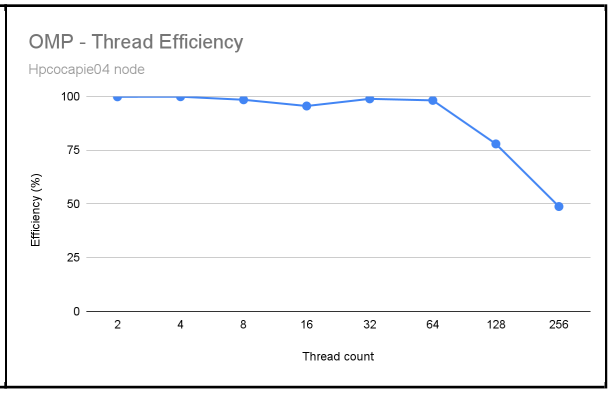
Hpcocapie03

On the hpcocapie nodes, since there are many more available threads, we can see that speedups continue to increase beyond 4 threads. We can see that the performance is lower at low counts of threads (under 16 threads) as the cores are slower on the hpcocapie nodes but there are more cores, meaning that beyond that, down from 60 seconds runtime, we can see 6 seconds runtimes with the 256 available cores.

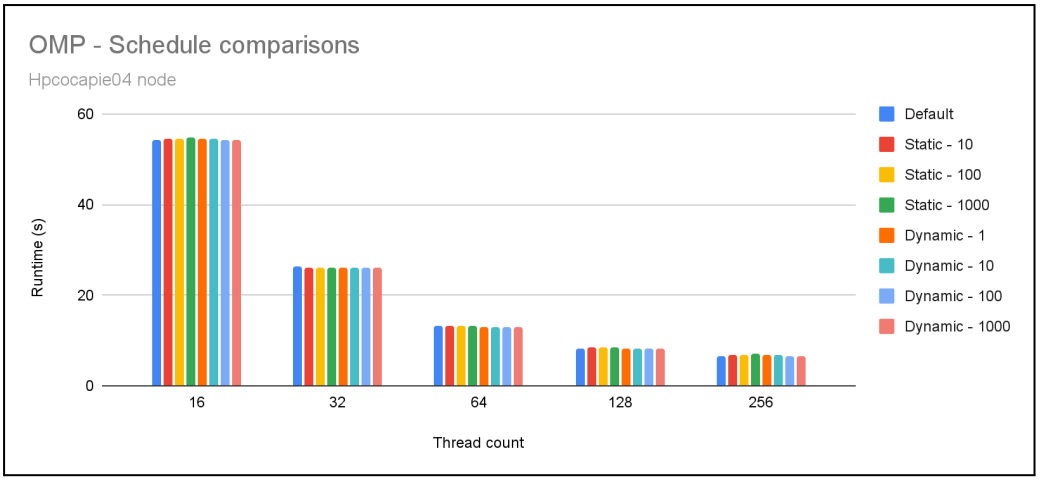
Another notable point is that the efficiency stays high overall and only starts to decrease beyond 64 threads. This is coherent with the fact that the hpcocapie nodes have 64 physical cores with each 4 threads allowing for hyperthreading but this is less efficient than the physical cores.







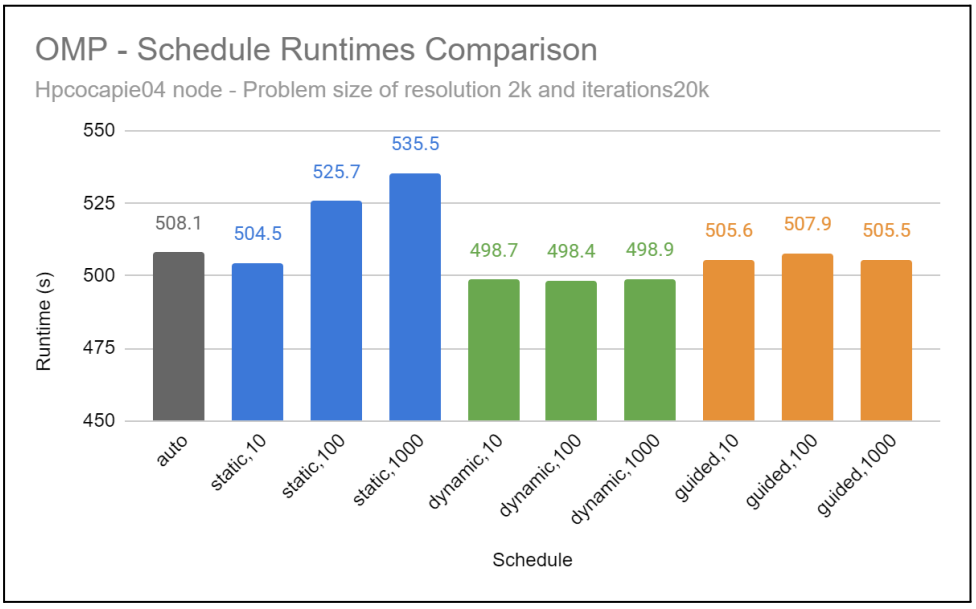
Changing the schedule at this scale, however, has no impact on the runtime.



Following this, we can attempt to increase the problem size to see if the schedule would have an impact then. To do this, I increased the size of the problem by increasing the resolution to 2k instead of 1k and increasing the number of iterations to a consequent 20k instead of 1k. The consequent increase in iterations is with the goal of attempting to create a greater imbalance in the pixel computation times.

As we can see here, there are indeed changes in the higher problem size. We have runtimes around 500s (~8.5 mins) when using 256 threads. I computed the runtime with 64 threads to approximate speedup and efficiency values. I assumed capturing a runtime with 64 threads and multiplying by 64 would be a good approximation for the runtime with 1 thread as, from what we have seen before, the efficiency of threads stays around 100% until 64 threads. With 64 threads and an auto-schedule, we get a runtime of 1020s. This leads to a runtime speedup approximation of around 128 and efficiency approximating around 50%.

We can observe that the auto schedule seems to be mostly aligned with the guided schedule, no matter the block size. Then, the static schedule is the worst performing which makes sense due to the nature of the problem and the scale of the problem. Following this, dynamic scheduling is definitely the accurate scheduling for this problem. No matter the block size, it is a net improvement compared to the other ones.



MPI

Files: mandelbrot_mpi_static.cpp, mandelbrot_mpi_dynamic_n.cpp

Parallelization

To parallelize this program we can take 2 different approaches in code. Either we split all the iterations of the loops equally between the processes, or we define a master process that will distribute the iterations or iterations blocks and receive the result after the computation that the slave processes will perform.

Static

In this approach we are splitting all the iterations of the loops equally to assign them to each process. If the amount of iterations is not perfectly divisible by the number of processes available, we distribute the remainder of the iterations between the first n processes. Computation of the number of iterations to be processed by each process:

```
// Computing the various counts of the processes
int* procIterStarts = new int[procCount];
int* procIterCounts = new int[procCount];

for (int proc = 0; proc < procCount; proc++) {
    procIterCounts[proc] = iterDiv + ((proc < remainderIter) ? 1 : 0);
    procIterStarts[proc] = (proc * (iterDiv)) + ((proc < remainderIter) ? proc : remainderIter);
}
```

Then the loop on each process can be defined as followed:

```
int* imageProcPortion = new int[iterDiv+1];

// Distributing Iterations between the processes
for (int pos = 0; pos < procIterCount; pos++)
{
    imageProcPortion[pos] = 0;

    const int row = (pos + procIterStart) / WIDTH;
    const int col = (pos + procIterStart) % WIDTH;
    const complex<double> c(col * STEP + MIN_X, row * STEP + MIN_Y);
```

Here, on each process, we define an array (imageProcPortion) that represents the portion of the image that has to be computed by it. These arrays will then need to be combined together to form the output. To do this, we define a Gather statement. It has to be a GatherV statement as we want to define the displacements (procIterStart) due to the fact we have to take into account that some processes might have one extra iteration to process:

```
// Gathering results on process 0
MPI_Gatherv(
    &imageProcPortion[0], // Send buf
    procIterCount, // Send count
    MPI_INT, // Send type
    image, // Receive buf
    procIterCounts, // Receive count
    procIterStarts, // Receive displacements
    MPI_INT, // Receive type
    0, // Root process
    MPI_COMM_WORLD // comm
);
```

Dynamic

To perform dynamic scheduling with MPI, we can build a master-slave architecture.

Master Process

The master process will first distribute iterations to all the slaves by sending a message containing the starting point of the iterations it has to process. Additionally, it will store this starting iteration for the process so that the slave process doesn't need to send it back once it is done with its task and it can just send the results.

```
// First distribute tasks to all processes
for (int i = 1; i < procCount; i++) {
    MPI_Send(
        &currentTask, // Send buf
        1, // Send count
        MPI_INT, // Send type
        i, // Destination
        0, // Tag
        MPI_COMM_WORLD // Comm
    );

    // Record what tasks have been sent and increase currentTask counter
    sentTasks[i] = currentTask;
    currentTask += blockSize;
```

Then, the master process will loop until it has received all the blocks in results back from the slave threads. The result blocks are then written to the output array starting at the starting position that was sent to the slave process.

```
// Then loop to receive back results and send new task
for (int i = 0; i < ((tasks / blockSize) + 1); i++) {
    MPI_Recv(
        payload, // Receive buf
        blockSize, // Receive count
        MPI_INT, // Receive Type
        MPI_ANY_SOURCE, // Source
        0, // Tag
        MPI_COMM_WORLD, // Comm
        &status // Status
    );

    // Process output of slave
    proc = status.MPI_SOURCE; // Process from which we receive a Message
    task = sentTasks[proc];

    // Write output to image array
    for (int i = 0; i < blockSize; i++) {
        if ((task + i) < tasks)
            image[task + i] = payload[i];
    }
```

And finally, if there are still blocks to be sent, they are sent to the slave process that we just received from. Else, if there are no more tasks to be performed, the slave process is being indicated to stop by sending it a tag of 1 instead of the usual 0.

```
// Sending a new iter or a termination signal
MPI_Send(
    &currentTask, // Send buf
    1, // Send count
    MPI_INT, // Send type
    proc, // Destination
    (currentTask < tasks)? 0: 1, // Tag (If 1, slave is done and can exit)
    MPI_COMM_WORLD // Comm
);

// In case termination signal is send, we record it and check if all have been terminated
if (currentTask >= tasks) terminatedCount++;
if (terminatedCount >= (procCount - 1)) {
    cout << "\t- All termination signals have been sent, breaking from Master." << endl;
    break;
}

// Record what tasks have been sent and increase currentTask counter
sentTasks[proc] = currentTask;
currentTask += blockSize;
```

Slave Processes

For the slave process, in an infinite loop, it will first wait to receive its first instruction from process 0, the master process.

```
// Receive Task
MPI_Recv(
    &task, // Receive buf
    1, // Receive count
    MPI_INT, // Receive Type
    0, // Source
    MPI_ANY_TAG, // Tag
    MPI_COMM_WORLD, // Comm
    &status // Status
);
```

Then, it will check whether it has to exit or not by checking the tag number. Once this is done, the different iterations in the block can start to be processed. In the loop, we also have a check to make sure we don't make the slave process compute more pixels than it has to.

```
// Dont do computation and break if tag 1 is received
if (status.MPI_TAG == 1) break;

// Distributing Iterations between the processes
for (int i = 0; i < blockSize; i++) {
    // Check if within image
    if (task + i >= tasks) break;
```

And finally, if some iterations have been performed, ie, if the slave process's main loop has not stopped yet, we send back the result of the computation to the master thread.

```
    MPI_Send(
        payload, // Send buf
        blockSize, // Send count
        MPI_INT, // Send type
        0, // Destination
        0, // Tag
        MPI_COMM_WORLD // Comm
    );
}
while (status.MPI_TAG == 0);
```

Compilation and run

To compile the code with the Intel compiler for MPI we can use the following command:
> mpiicc -std=c++11 mandelbrot.cpp -o out.o

And execute with:
> mpiexec -nolocal -host hpcocapie03.ge.infn.it -np 256 ./out.o

Then, to execute on multiple nodes, I have defined multiple files (<n>_nodes.txt) containing a list with “n” amount of nodes:
> mpiexec -nolocal -hostfile machinelists/2_nodes.txt -perhost 1 -np 2 ./out.o 10

And finally, we can also have node zero running on the frontend node with:
> mpiexec -nolocal -host localhost,hpcocapie03 -perhost 1,32 -np 33 ./out.o 1

Experiment setup

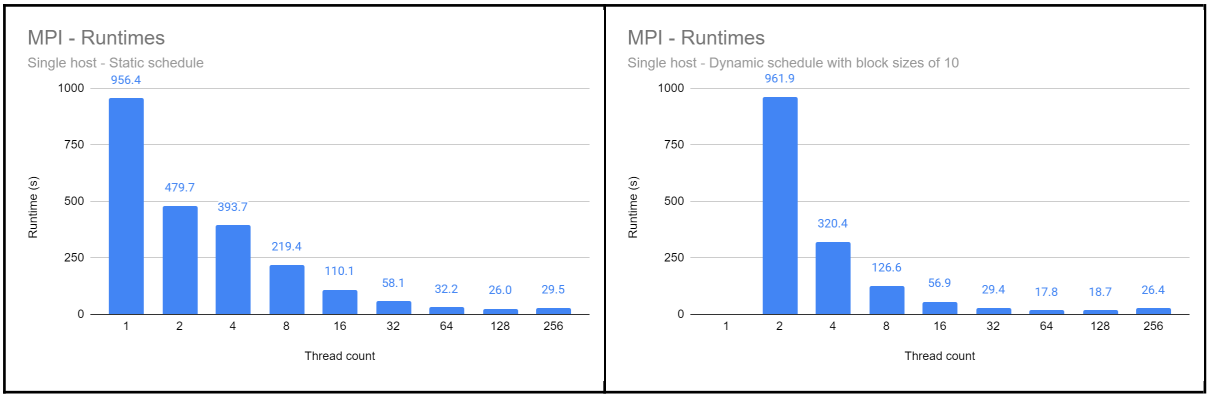
For MPI, we will compare the speedup provided on a single machine for different numbers of threads with the static implementation and with the dynamic implementation with different block sizes.

Then we will compare the same number of threads and how spreading these threads counts across various machines rather than on a single one. We will compare the efficiencies provided by each thread.

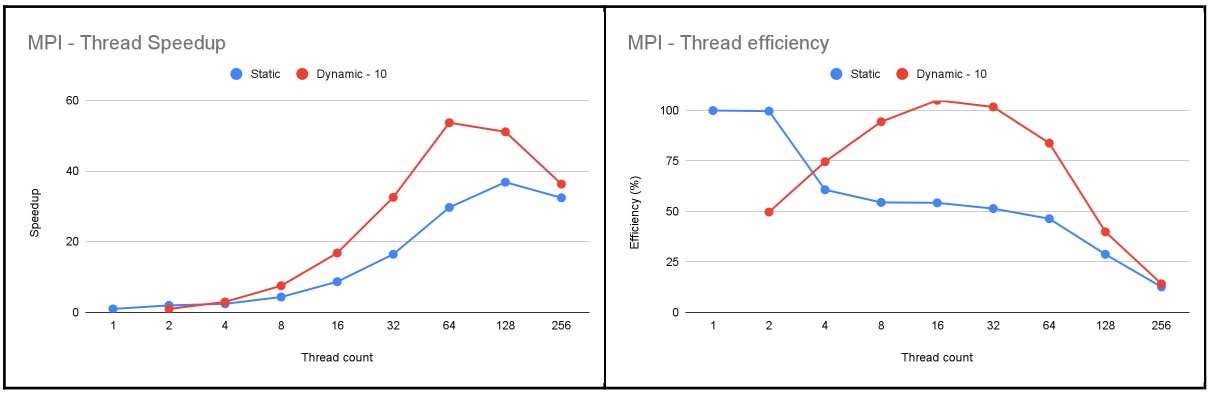
And finally, we analyze how the efficiency is impacted when each node has the same amount of threads.

Experiment results

When running the program on a single node with varying amounts of thread, we can see that at the beginning, when using a single node as a worker, the runtime is worse on the dynamic version of the problem. This is most likely due to the cost of communicating the new task and the results back. But from 4 threads (in the dynamic version this equates to 1 master and 3 workers or slave processes, the runtime gets much better than the static version which makes sense from the nature of the problem. We can see that the best performance of the static scheduling is 26s with 128 threads whereas, for the dynamic scheduling, with 64 threads, we get down to 17.8, which is an 8s improvement.

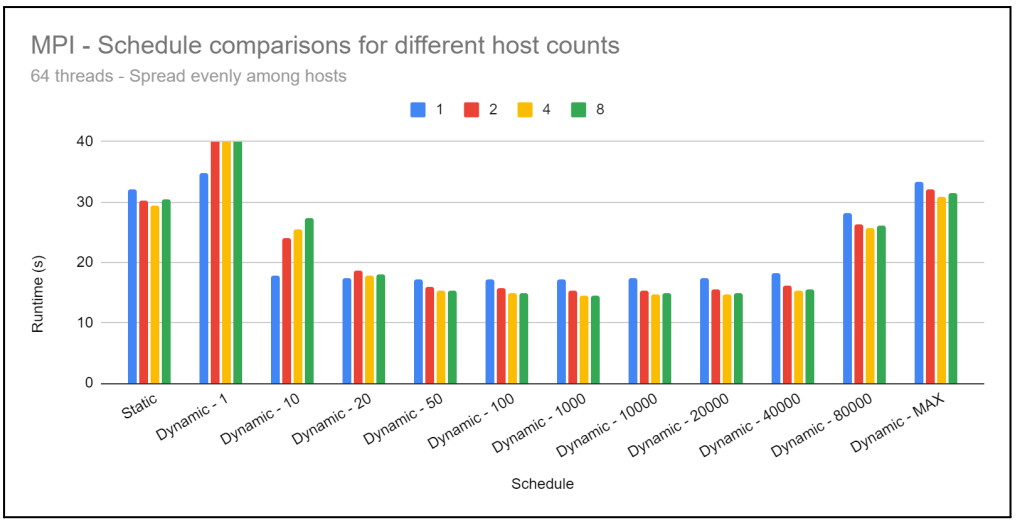


Regarding the speedups and efficiencies, with static scheduling, we can see that the efficiency per thread decreases consistently with the number of threads used. There is a hard drop after 2 threads and a second drop after 64 threads. Whereas for the dynamic scheduling, we can see that the efficiency per thread actually increases to 100% efficiency per thread for 16 and 32 threads. Beyond that, the communication costs are too high compared to the speedup advantages so the efficiency drops with a dramatic drop beyond 64 threads. The drop beyond 64 threads for both scheduling options is consistent with the fact there are 64 physical cores on the hpcocapie nodes.

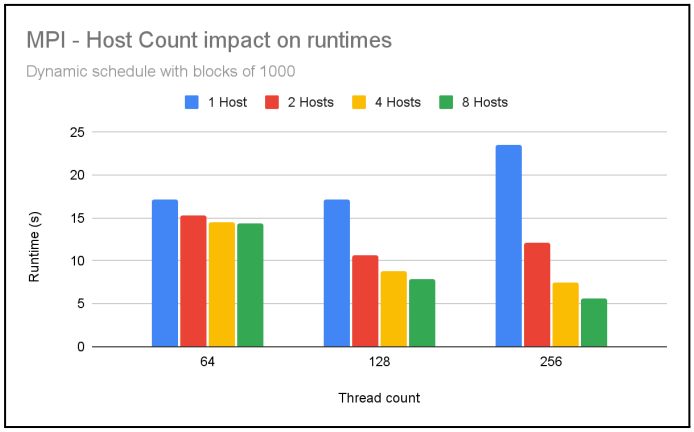


Now to compare the various scheduling options, we mostly compare the static implementation of the problem with the dynamic version with various iteration block sizes on different amounts of nodes. I chose to run all the experiments on 64 threads as this configuration gives the best times without sacrificing too much efficiency.

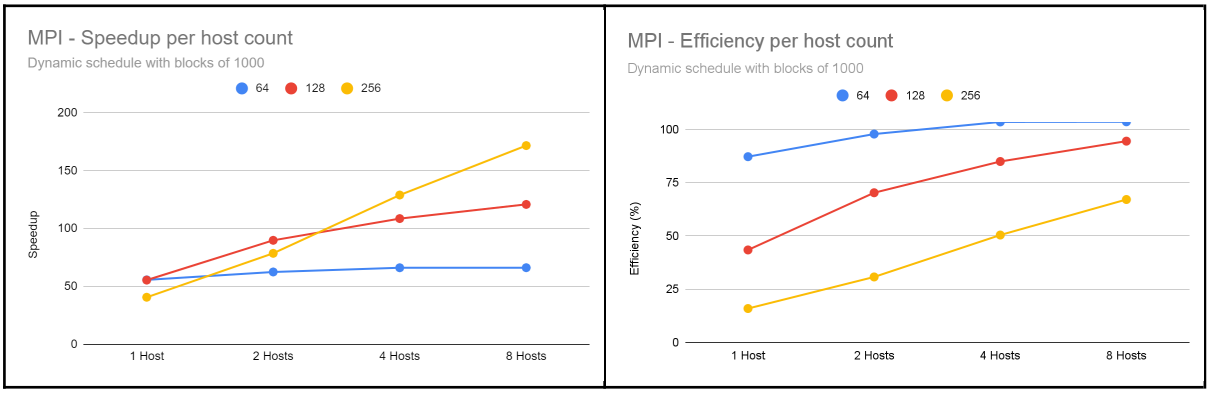
We can see from this experiment that especially at low block sizes for dynamic scheduling, the communication costs between the nodes really outweigh the advantages of dynamic scheduling. But, between block sizes of 20 and 50, the runtimes of the multi-node setups outpace the runtimes of the single-node setup. Then, we can see that the lowest runtimes reached, for all the different node counts are with dynamic scheduling with block sizes of 1000.



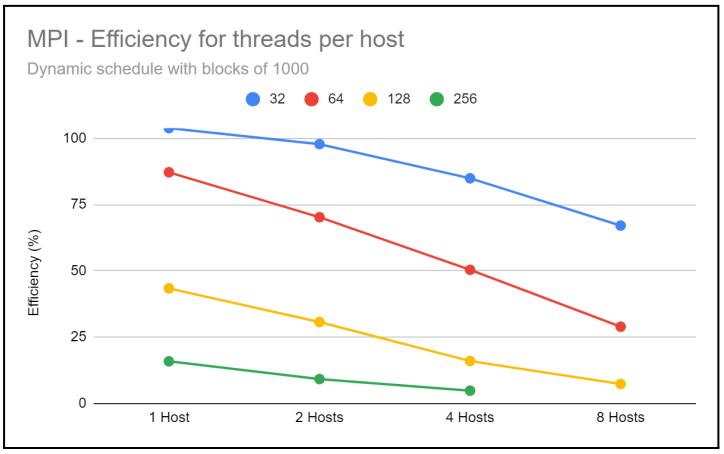
Finally, we compare the impact of the number of hosts on the runtimes with the same amount of threads spread equally over the various hosts. In this configuration we use dynamic scheduling with blocks of 1000 as this was the best configuration found so far. We can see a larger amount of threads available lead to a larger improvement in runtimes with the number of nodes available. This makes sense, as, when using 256 threads, on a single node, this is above the 64 physical cores, whereas, when using 8 hosts, this represents 32 threads per node.



Finally, we can note that the efficiency per thread constantly increases with the number of nodes the threads can be spread among.



As a last experiment, I wanted to check the efficiency with an equal amount of threads on each node, with a different amount of nodes. We can see that the efficiency per thread decreases consistently as the number of nodes increases. But, we can notice that the efficiency decreases more heavily per node for lower threads per node. For example, for 128 threads/node, the efficiency decreases at a lesser pace than for 64 threads/node.



Finally, the fastest runtime we got out of all the experiments is 5.5s with 8 hosts and 32 threads per host (this means there are a total of 256 threads). The schedule that led to this runtime was a dynamic schedule with blocks of 1000.

Something that would be interesting to explore is to see if we could combine the thread efficiency we got with OpenMP with the inter-node communication abilities provided by MPI. In this regard, I built a Hybrid implementation that will be described in the following section…

OpenMP-MPI Hybrid

Files: mandelbrot_hybrid_static.cpp,
mandelbrot_hybrid_dynamic_n.cpp

Setup

To setup the code for a hybrid version between OpenMP and MPI, we can start from the code built for the MPI section and define the loop over the pixels to be an OpenMP for loop as follows:

- Static:

```
// Distributing Iterations between the processes
#pragma omp parallel for schedule(runtime)
for (int pos = 0; pos < procIterCount; pos++)
```

- Dynamic:

```
// Distributing Iterations between the processes
#pragma omp parallel for schedule(runtime)
for (int i = 0; i < blockSize; i++) {
```

Compilation and run

To compile, we can use the MPI Intel compiler along with the OpenMP tag “-qopenmp”:
> mpiicc -qopenmp -std=c++11 mandelbrot.cpp -o out.o

Following this, we can execute by setting the environment variables for the number of threads and the schedule followed by the MPI execution instruction. The difference from previously is that we have to send the environment variables to the hosts on which we will run the program with the –env tag:
> mpiexec -host hpcocapie04 -np 8
--env OMP_NUM_THREADS 4
--env OMP_SCHEDULE "auto" ./out.o

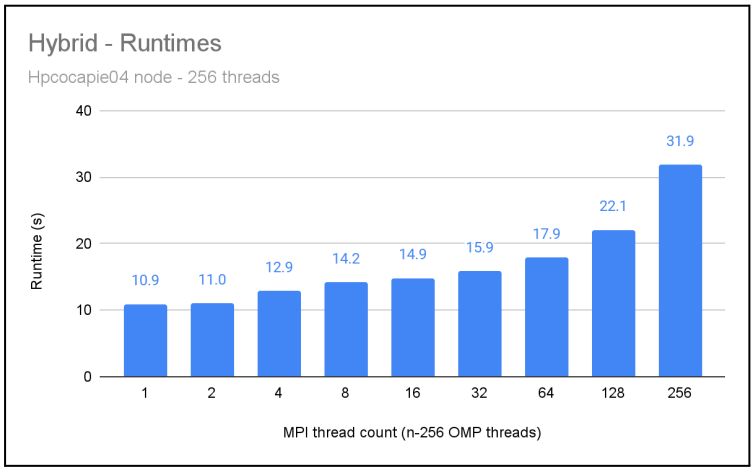
Experiment setup

To analyze how the combination of MPI and OpenMP fairs in combination, I wanted to first try out on a single host how, on a single host, with a total of 256 threads and how the proportion of threads assigned to OpenMP and assigned to MPI would affect the runtimes. And how the efficiencies compare to one another.

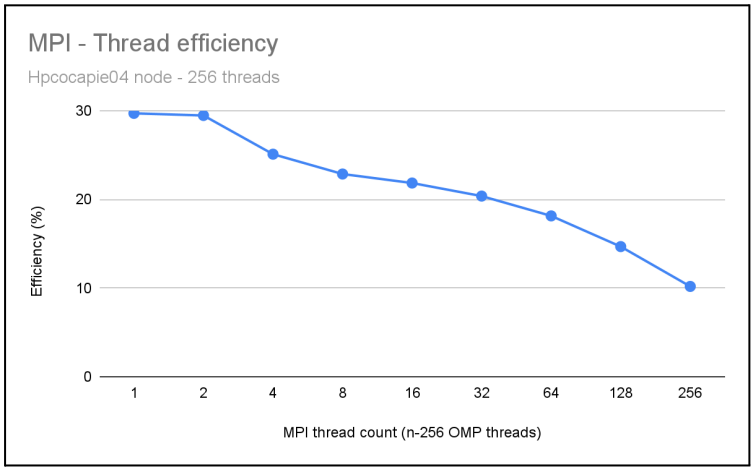
Then, to have a more accurate picture of the thread efficiency at the highest power available, I wanted to increase the problem size to a 2k resolution and 2k iterations. I will compare various proportions like previously but on this larger problem and using 1, 2, 4, and 8 hosts.

Experiment results

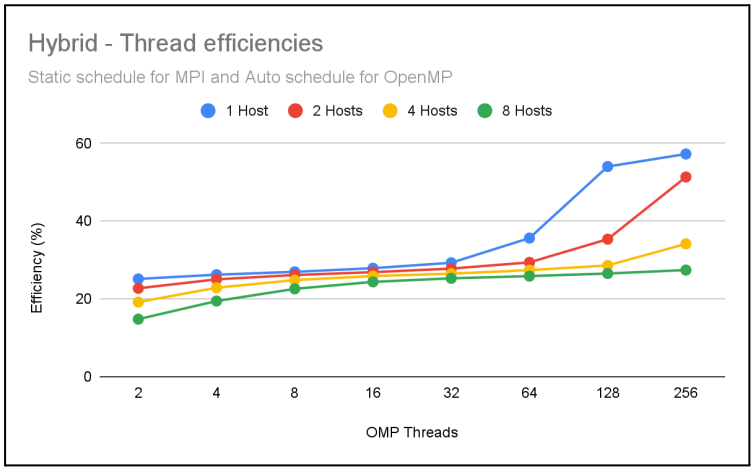
When running 256 threads on the hpcocapie04 node, we can see that running the program with a smaller proportion of MPI processes the better the performance gets. It is most likely due to the fact MPI relies on sending data as messages between processes which involves a lot of overhead compared to the shared memory accesses of the OpenMP parallelization.



Regarding the efficiency of the threads, we can note that for 1 and 2 MPI threads, this represents 256 and 128 OpenMP threads respectively, we can see that efficiency is the highest.



Now, to compare on multiple hosts, with a larger problem scale, we can still see the same efficiency pattern exhibited on 1 host. But then, on multiple hosts, we can note that with a larger number of hosts, the efficiency gained from larger proportions gets less and less. This is most likely due to the communication involved with a larger amount of hosts.



I wanted to try out different scheduling for MPI but I figured that the optimal setup for a Hybrid between OpenMP and MPI would be to spread a static amount of iterations to each host and let OpenMP handle the irregular nature of the problem. But regarding the scheduling of OpenMP, we noted before that the scheduling had a minimal impact but the “auto” scheduling was a decent choice.

Finally, as the result of this larger problem, the baseline value (so on one host and using one thread) was 7558 seconds, so above two hours. Now when running the problem on a single host with just OpenMP, we get it down to 51 seconds, while with MPI at the maximum configuration, we get around 30 seconds. Now combining the two, with 256 OpenMP threads on 8 hosts, we get a runtime of 13.5 seconds, which represents an efficiency of 20% per thread.

CUDA

Files: mandelbrot_cuda.cpp

Parallelization

First to do the parallelization, we want to define the kernel function. Inside this function, we must catch the block and thread id to know what pixel it is currently responsible for.

```
__global__ void computePixels(int* image, int width, int height, double step, double min_x, double min_y, int iterations) {
    // Computing i and j from block and thread number
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;

    if ((col >= width) || (row >= height)) return;
```

Something to notice was that the complex variable had to be converted to something CUDA-compatible. In this case, the analog for this in CUDA is “cuDoubleComplex” and operations for those CUDA complexes are to be used:

```
cuDoubleComplex c = make_cuDoubleComplex(col * step + min_x, row * step + min_y);

// z = z^2 + c
cuDoubleComplex z = make_cuDoubleComplex(0, 0);
for (int i = 1; i <= iterations; i++)
{
    z = cuCadd(cuCmul(z, z), c);

    // If it is convergent
    if (cuCabs(z) >= 2)
    {
        image[(width * row) + col] = i;
        break;
    }
}
```

Finally, we need to call this kernel from the main function by specifying the blocks per grid and the threads per block. Here, we will keep the blocks per grid static, we make them directly proportional to the size of the problem and the size of the threads per block.

```
int* image;
cudaMallocManaged(&image, size);

// Defining threads and blocks of threads
dim3 threadsPerBlock(16, 16);
dim3 blocksPerGrid((WIDTH / threadsPerBlock.x) + 1, (HEIGHT / threadsPerBlock.y) + 1);

// Starting kernel
computePixels<<<blocksPerGrid, threadsPerBlock>>>(image, WIDTH, HEIGHT, STEP, MIN_X, MIN_Y, ITERATIONS);
cudaDeviceSynchronize();
```

Compilation and run

To compile the code, we can use CUDA compiler nvcc and directly run it using:
> nvcc -std=c++11 mandelbrot_cuda.cu -o out.o -run

Experiment setup

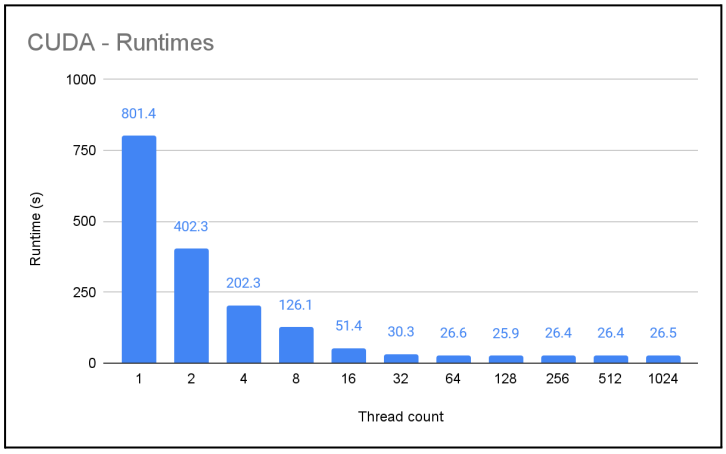
To try different configurations, we will vary the number of threads per block and also the topological setup (for example 32 threads either as 8 x 4 or 32 x 1). We will test using a single thread per block to the full capacity of 1024 threads per block.

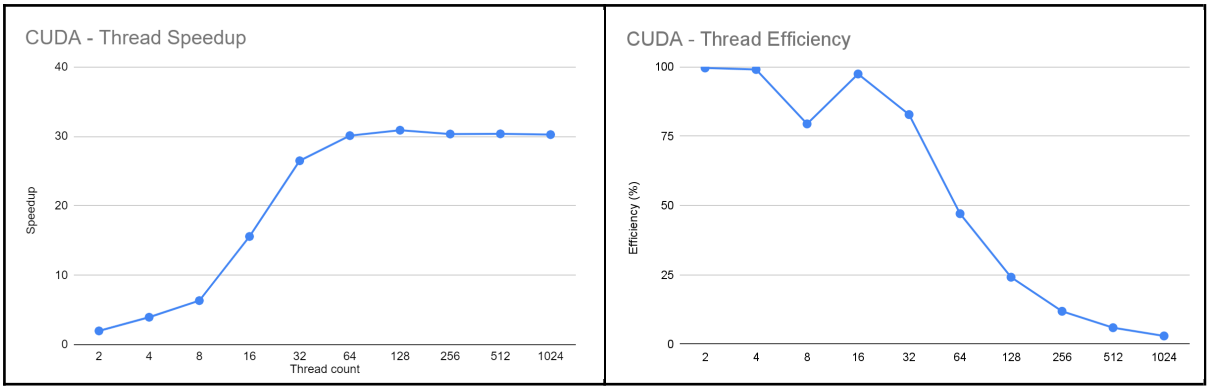
Then per thread count, we will vary the configuration of the block to see if the topology matters in this problem.

And finally, we will attempt to change the problem size by modifying the resolution and the number of iterations to see if that has an impact on the speedup and efficiency values.

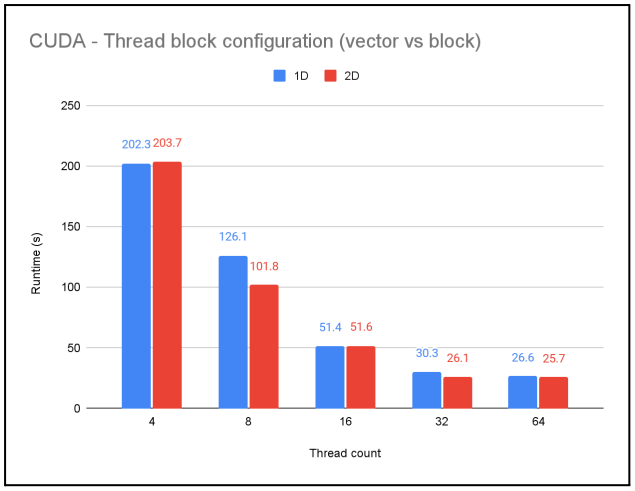
Experiment results

To start with, I decided to run the problem with a resolution of 5000 to show meaningful speedup and scaling (the amount of convergence iterations has, however, not been changed). We can see from these graphs that there is good speedup happening up to 32 threads but beyond that, the efficiency rapidly declines.

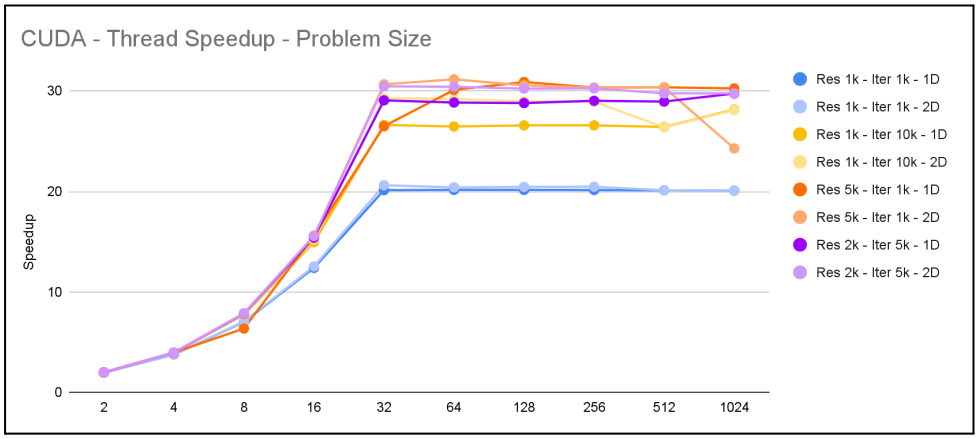




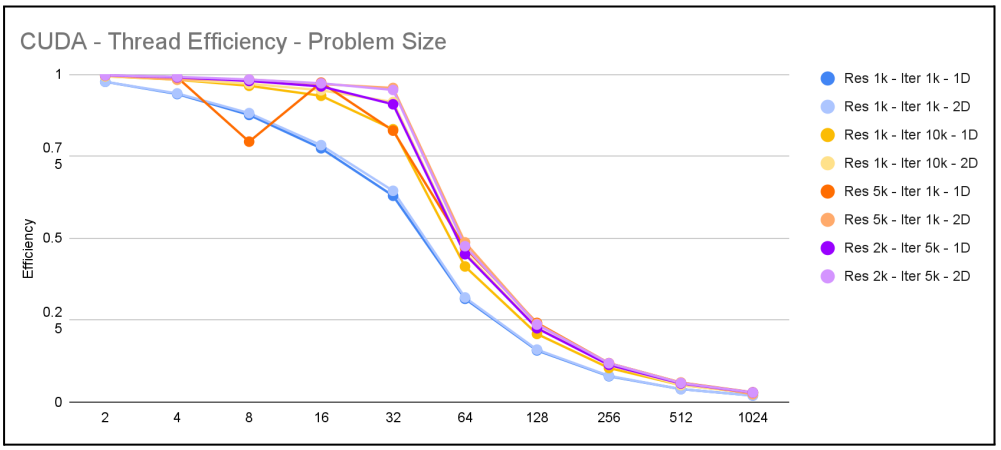
Then, with the same setup (5000 resolution, 1000 iterations), we can see that there is a small or no improvement with the 2D thread. What is interesting is that where the improvement is the most significant, is the non-square thread blocks, 8 threads are made with 4 x 2, and 32 are made with 8 x 4.



And, finally, we can see that with the default problem size, the speedup seems capped at 20x. Then, when increasing the problem size, the speedup per thread and therefore the efficiencies improve. We can see that, overall, the 2D blocks provide a more significant speedup than their vector counterpart for the same number of threads.



Regarding the efficiency of the threads, we can see that the larger the problem, the more efficient the threads. We can see that a more balanced version of a larger problem (2000 resolution and 5000 iterations) leads to more efficient threads compared to versions where the resolution is much higher but iteration remains at 1000 iterations (leading to a less uneven problem) or where the iterations are much larger and the resolution remaining at 1000 (leading to more imbalance in pixel processing times).



Appendix

The raw data, runtimes and charts can be found on the following google sheet:
HPC - Final Project - Runtimes

OMP Runtimes

Frontend runtimes

Threads	Default	Static - 10	Static - 100	Static - 1000	Dynamic - 1	Dynamic - 10	Dynamic - 100	Dynamic - 1000
1	228.810							
2	124.329							
4	61.731	64.846	63.724	64.829	63.770	61.548	62.155	62.326
8	62.117	62.841	62.690	61.779	61.595	61.995	60.760	61.857
16	60.618	61.829	62.300	62.650	61.508	61.358	61.498	60.471
32	62.232	65.775	67.729	64.587	65.694	66.839	70.102	67.247
64	61.958	63.274	63.397	63.448	62.394	61.112	62.505	61.950
128	61.834							
256	62.242							

Hpcocapie03 runtimes

Threads	Default	Static - 10	Static - 100	Static - 1000	Dynamic - 1	Dynamic - 10	Dynamic - 100	Dynamic - 1000
1	829.392							
2	415.453							
4	207.644							
8	105.318							
16	54.257	54.490	54.513	54.755	54.452	54.663	54.155	54.393
32	26.216	26.113	26.084	26.018	26.037	25.959	25.960	25.960
64	13.202	13.188	13.162	13.170	13.096	13.071	13.032	13.036
128	8.317	8.515	8.480	8.468	8.342	8.173	8.094	8.110
256	6.640	6.837	6.754	6.992	6.855	6.894	6.583	6.618

Increased problem size (2k resolution and 20k iterations)

Schedule	Schedule Type	Runtime (s)	Speedup	Efficiency
auto	Auto	508.060	128.557	0.502
static,10	Static	504.530	129.457	0.506
static,100	Static	525.653	124.255	0.485
static,1000	Static	535.481	121.974	0.476
dynamic,10	Dynamic	498.681	130.975	0.512
dynamic,100	Dynamic	498.361	131.059	0.512
dynamic,1000	Dynamic	498.870	130.925	0.511
guided,10	Guided	505.623	129.177	0.505
guided,100	Guided	507.860	128.608	0.502
guided,1000	Guided	505.505	129.207	0.505

MPI Runtimes

Hosts	Proc (/ host)	TOT	Static	Dynamic - 1	Dynamic - 10	Dynamic - 20	Dynamic - 50	Dynamic - 100	Dynamic - 1000	Dynamic - 10000	Dynamic - 20000	Dynamic - 40000	Dynamic - 80000	Dynamic - MAX	Dynamic - 1000 - With Master On localhost
1	1	1	956.447												956.647
1	2	2	479.680	992.565	961.865									955.232	474.753
1	4	4	393.660	340.620	320.431									588.917	237.602
1	8	8	219.380	140.150	126.614									261.798	106.175
1	16	16	110.124	69.253	56.896									120.526	53.554
1	32	32	58.116	46.299	29.370	29.220	28.896	28.889	28.800	28.850	29.178	29.307	30.136	60.535	29.597
1	64	64	32.203	34.902	17.825	17.479	17.189	17.209	17.147	17.376	17.345	18.204	28.108	33.468	22.710
1	128	128	25.963	52.713	18.723	17.940	16.630	16.367	17.219	17.825	18.154	23.250		27.291	16.205
1	256	256	29.506	90.913	26.355	23.578	22.410	22.698	23.476	23.259	27.670			29.973	23.431
2	1	2	479.170		1036.436										
2	2	4	394.754		317.826										
2	4	8	238.870		137.210										
2	8	16	115.339		65.103										
2	16	32	57.279	148.917	35.796	31.710	28.865	28.520	28.500	28.500	28.308	28.978	31.379	59.252	
2	32	64	30.282	155.146	24.122	18.750	15.982	15.665	15.285	15.390	15.546	16.207	26.259	32.159	
2	64	128	18.697		21.574	14.510	11.392	11.750	10.639	10.849	11.194	16.352		18.669	
2	128	256	18.198		32.538	20.139	13.589	13.619	12.159	13.188	16.997			18.330	
2	256	512	24.115		55.671	34.328	25.206	22.295	20.248	22.969				24.160	
4	1	4	394.850												
4	2	8	240.421												
4	4	16	125.761												
4	8	32	59.531												
4	16	64	29.382	157.151	25.438	17.865	15.330	14.981	14.445	14.688	14.739	15.364	25.676	30.856	
4	32	128	16.475		21.852	12.636	9.758	9.406	8.802	8.963	9.266	14.383		16.772	
4	64	256	11.313		21.985	14.242	8.970	8.347	7.414	7.864	10.383			11.635	
4	128	512	13.801		32.156	21.294	14.822	13.551	11.660	13.312				14.330	
4	256	1024	20.740		78.779	37.264	25.607	23.947	19.236					20.480	
7	1	7	286.263												
7	2	14	144.100												
7	4	28	73.421												
7	8	56	34.986												
7	16	112	17.778	156.458	22.753	13.520	10.710	11.860	8.860	9.397	9.691	13.501		17.789	
7	32	224	10.642	159.204	21.496	12.985	7.633	6.957	6.105	6.552	8.507			10.507	
7	64	448	9.263		23.910	14.345	9.530	8.330	6.346	7.684				8.883	
7	128	896	12.230		38.676	23.688	15.676	14.388	10.710					11.914	

7	256	1792												
8	1	8	238.924											
8	2	16	126.130											
8	4	32	65.497											
8	8	64	30.447	175.110	27.452	18.120	15.439	14.925	14.435	15.004	14.955	15.628	26.136	31.502
8	16	128	15.648		22.274	12.884	9.216	8.691	7.912	8.258	8.446	13.524		15.950
8	32	256	9.536		21.723	13.150	7.403	6.586	5.571	5.934	8.470			9.591
8	64	512	8.720		23.475	14.180	9.619	8.236	6.450	7.647				7.909
8	128	1024	11.504		56.609	24.390	16.451	14.507	12.680					11.703
8	256	2048												

Hybrid Runtimes

MPI \ TOTAL	1	2	4	8	16	32	64	128	256
1	830.912	416.105	341.523	207.150	109.269	56.422	28.648	15.432	10.929
2		416.110	342.160	208.990	109.434	56.449	28.629	16.767	11.018
4			341.564	207.260	109.468	56.625	28.762	17.444	12.935
8				207.484	109.607	56.748	28.908	18.369	14.200
16					110.760	57.196	29.371	18.837	14.861
32						58.138	30.287	19.888	15.935
64							32.156	21.758	17.898
128								26.714	22.120
256									31.879

Threads per host

OMP Threads	1 Host	1 Hosts	4 Hosts	8 Hosts
2	117.477	64.966	38.486	24.950
4	112.586	58.974	32.265	18.972
8	109.514	56.461	29.715	16.339
16	105.842	54.929	28.528	15.139
32	100.810	53.122	27.865	14.594
64	82.855	50.231	26.943	14.269
128	54.655	41.763	25.807	13.913
256	51.576	28.756	21.614	13.461

CUDA Runtimes

Thread Block size	**Thread count**	**Res 1k - Iter 1k - 1D**	**Res 1k - Iter 1k - 2D**	**Res 1k - Iter 10k - 1D**	**Res 1k - Iter 10k - 2D**	**Res 5k - Iter 1k - 1D**	**Res 5k - Iter 1k - 2D**	**Res 2k - Iter 5k - 1D**	**Res 2k - Iter 5k - 2D**
1 x 1	1	32.595	32.595	313.367	313.367	801.427	801.427	628.430	628.430
2x1	2	16.682	16.682	157.660	157.660	402.310	402.310	315.255	315.255
2 x 2	4	8.672	8.655	79.672	79.587	202.283	203.684	158.600	158.377
4 x 2	8	4.652	4.623	40.605	40.342	126.128	101.817	80.201	79.833
4 x 4	16	2.632	2.600	20.964	20.596	51.409	51.640	40.798	40.394
8 x 4	32	1.619	1.581	11.770	10.700	30.250	26.138	21.620	20.628
8 x 8	64	1.617	1.597	11.840	10.726	26.615	25.713	21.780	20.657
16 x 8	128	1.617	1.593	11.790	10.815	25.941	26.225	21.820	20.783
16 x 16	256	1.618	1.592	11.790	10.828	26.413	26.425	21.650	20.766
32 x 16	512	1.620	1.620	11.860	11.880	26.392	26.395	21.710	21.110
32 x 32	1024	1.623	1.623	11.127	11.152	26.486	32.997	21.123	21.108