Mandlebrot Benchmark

Moritz Wundke
Universitat Oberta de Catalunya

Table of Contents

MPI Mandelbrot implementation

Several strategies has been followed in the MPI implementation. The full implementation can be found in *mandle.cpp*, to use a given strategy just use the right command-line argument. See the *Compile & Run* section for more information.

Static Version

We implemented a fully static version which assigns the required task for each process on the master process. Each process will process a predefined set of columns.

Static Version – Round-Robin

A round-robin static implementation has been developed where each process starts at a predefined row but using their own rank id they exactly know which columns to work on.

Dynamic Version

The dynamic implementation will start at a predefined column and then request more work after the current column has been processed.

MPI OpenMP hybrid implementation

The MPI-OpenMP hybrid implementation can be found in *mandle.cpp*. Just build the application using the -DWITH_OMP compiler switch. See the *Compile & Run* section for more information.

In the implementation each MPI process will handle the column loop using an OpenMP directive and so parallelize that part of work. We have been analyzing what and how will affect the completion time, see section *Benchmarks* for more information.

One of the main issues that has been observed is that increasing the size of the problem (width,height) the gain in performance goes to a point where it is actually worst then using less parallelization. This has to do with the fact that the schedule policy requires us to set the maximum number of parallelization degree. The degree that fits perfectly within a given problem changes while modifying the scope of the problem itself.

We observed that the guided policy takes changes within the problem scope better and gives better average numbers then the others.

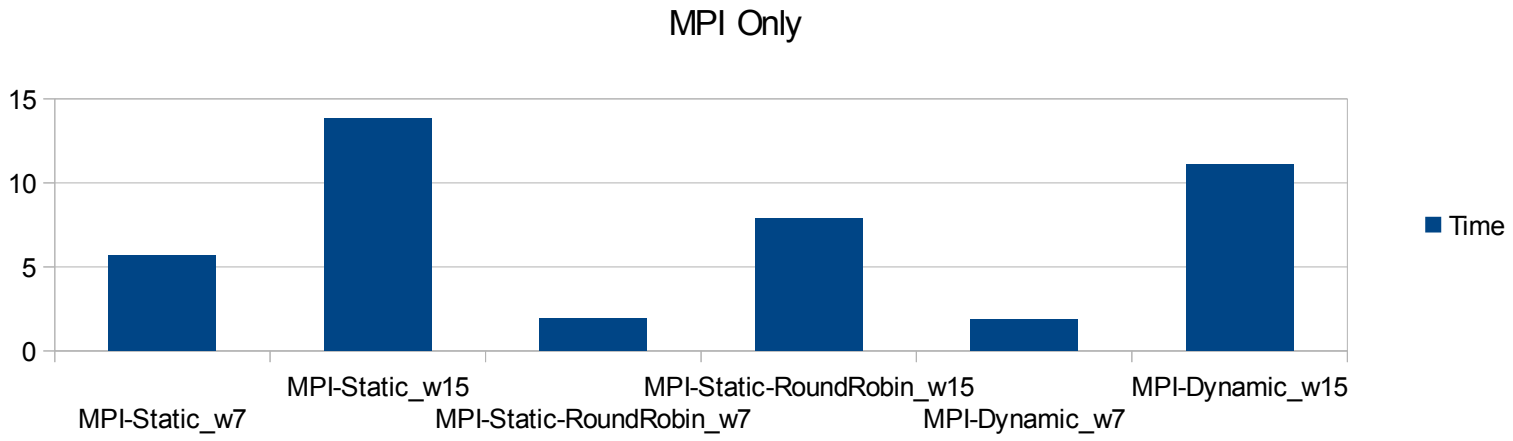
Benchmarks

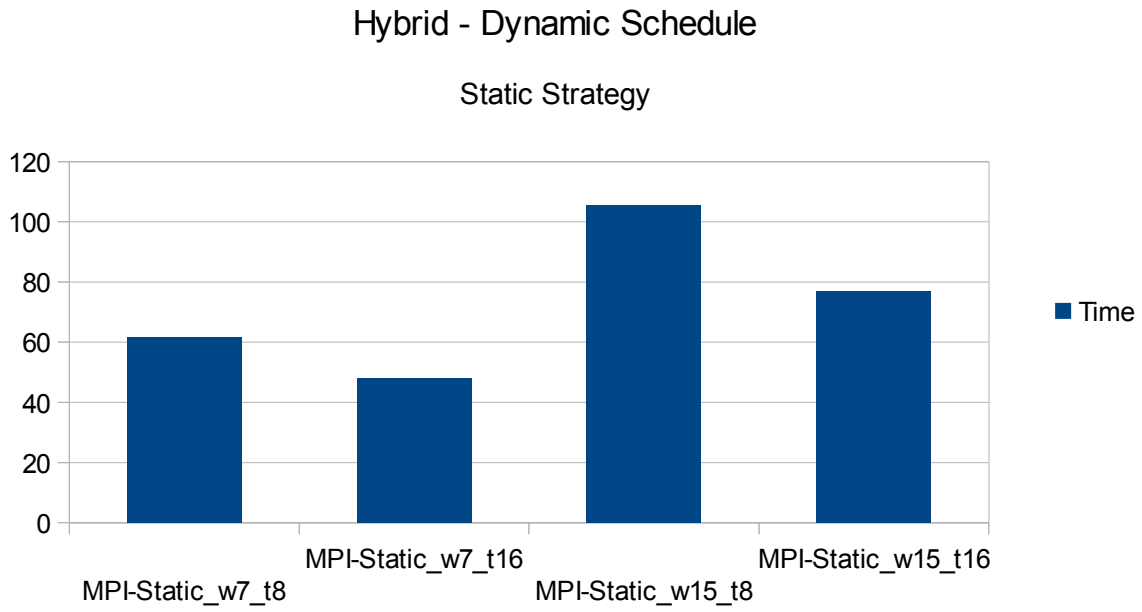
To run the benchmarks your self you just need to run the *run_mandle_test.sh* script. Each strategy will perform 4 test runs using 8 or 16 MPI with 8 or 16 OpenMP threads.

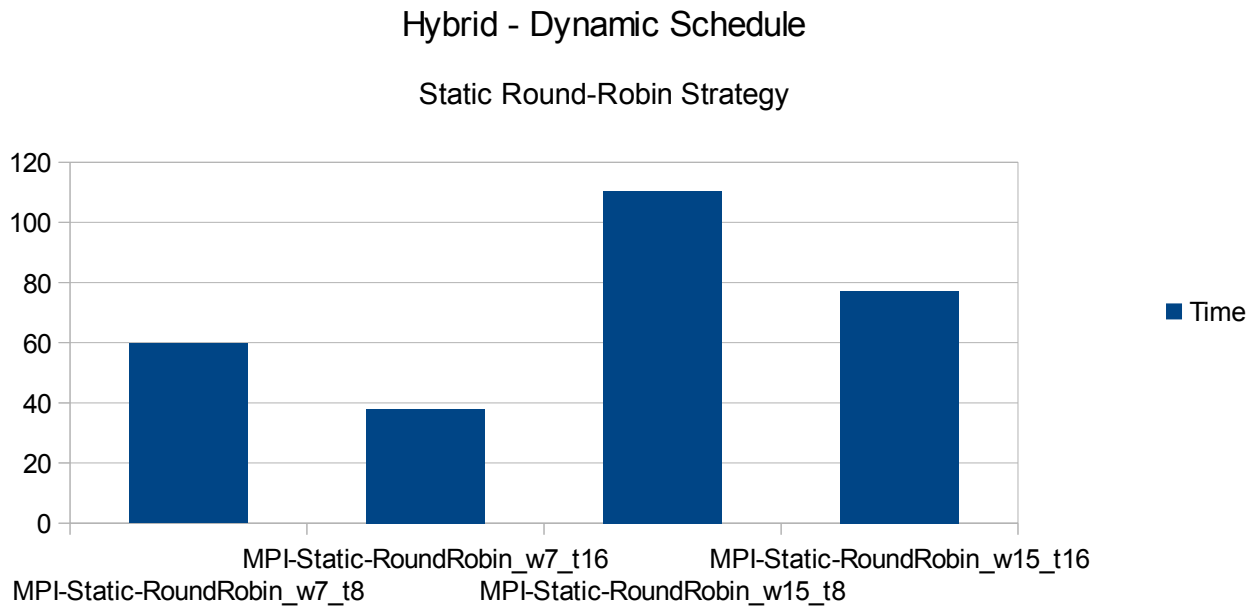
All benchmarks has been performed on a local test machine due to hang up problems in the cluster environment. The test machine is an 8 Core i7 with 8GB of ram. Performing the benchmarks on a local machine will give us also a better understanding about the comparison with the GPU implementation. The dimension of the problem has been chosen as 10000 x 10000 px. A full test run took about 32min average.

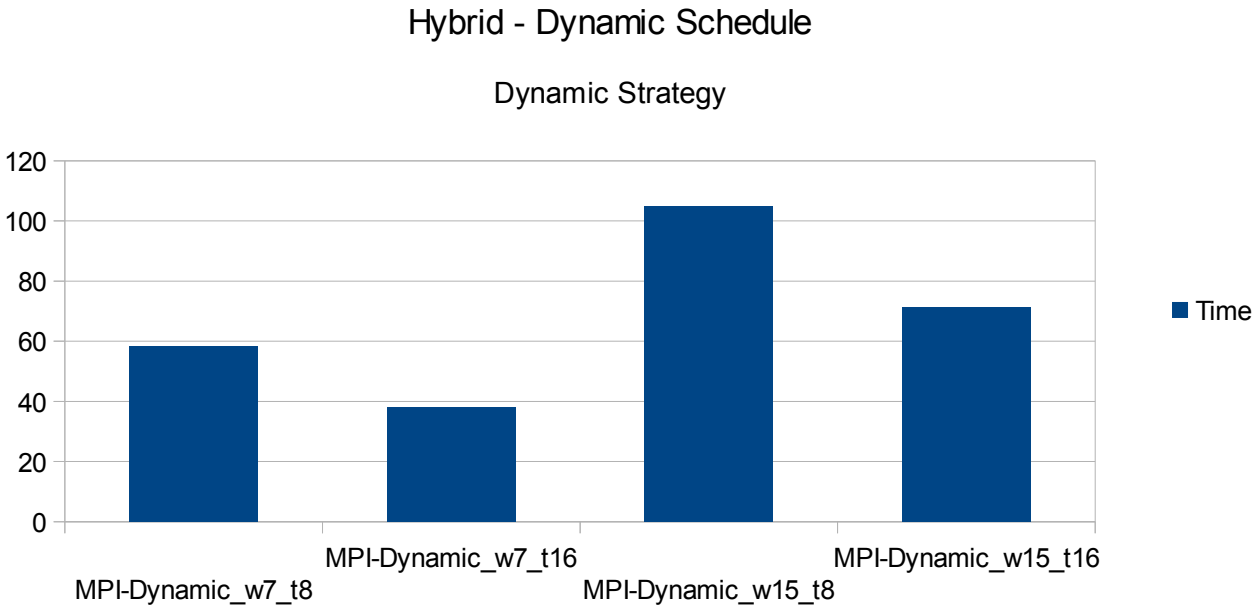
Benchmarks Comparison

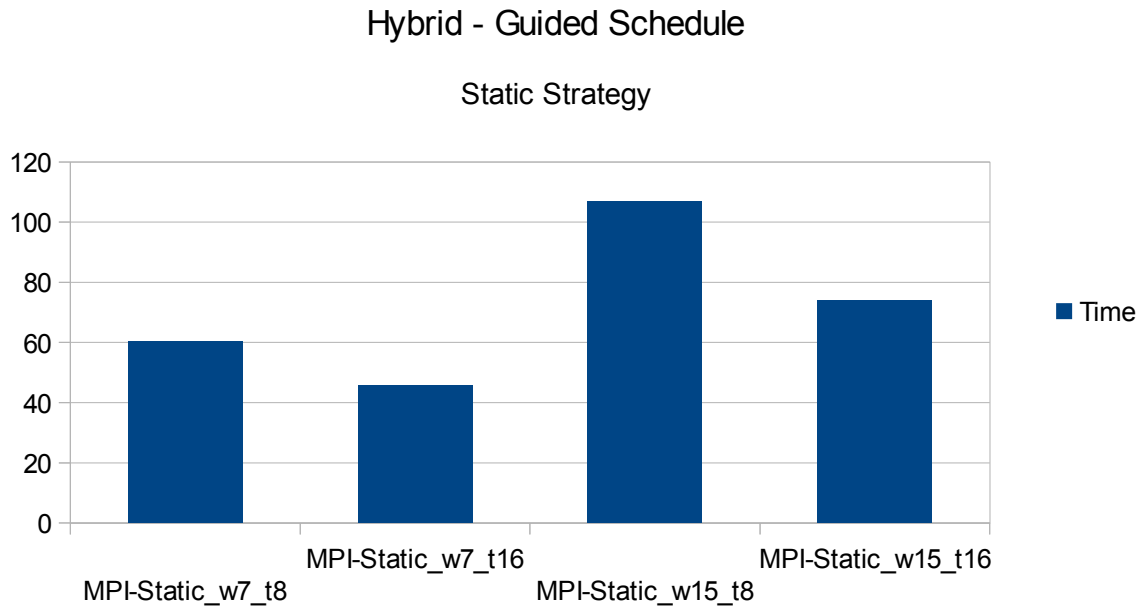
The following graphs will outline the performance analysis of all implementations, strategies and process/threads. The number of threads or processes used are indicated in the name of a test-run. wX indicates that It has been performed using x MPI worker processes and tX indicates that X OpenMP threads has been used.

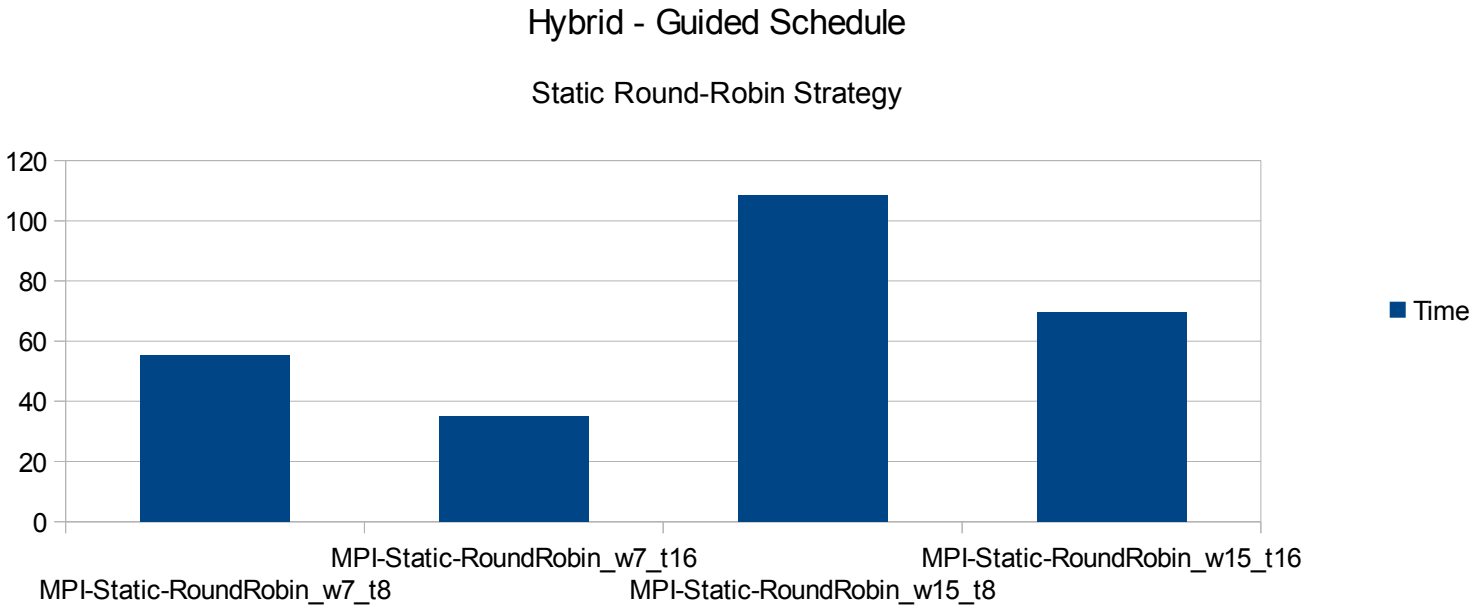


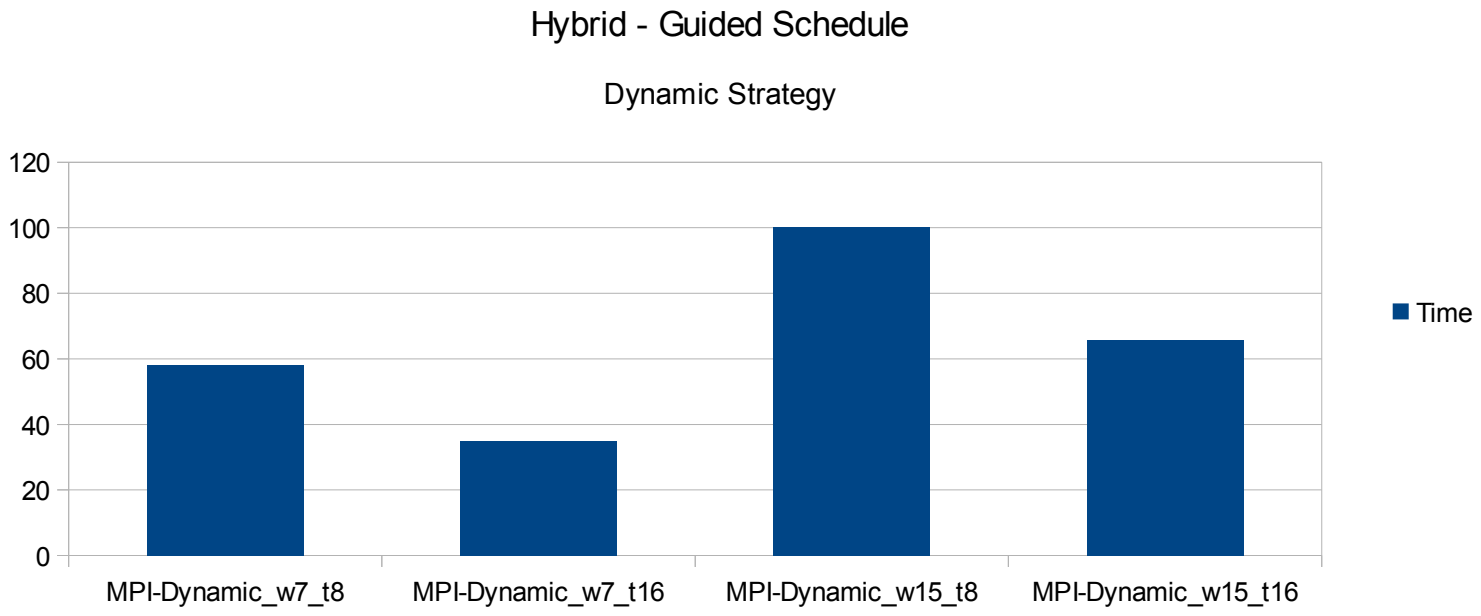


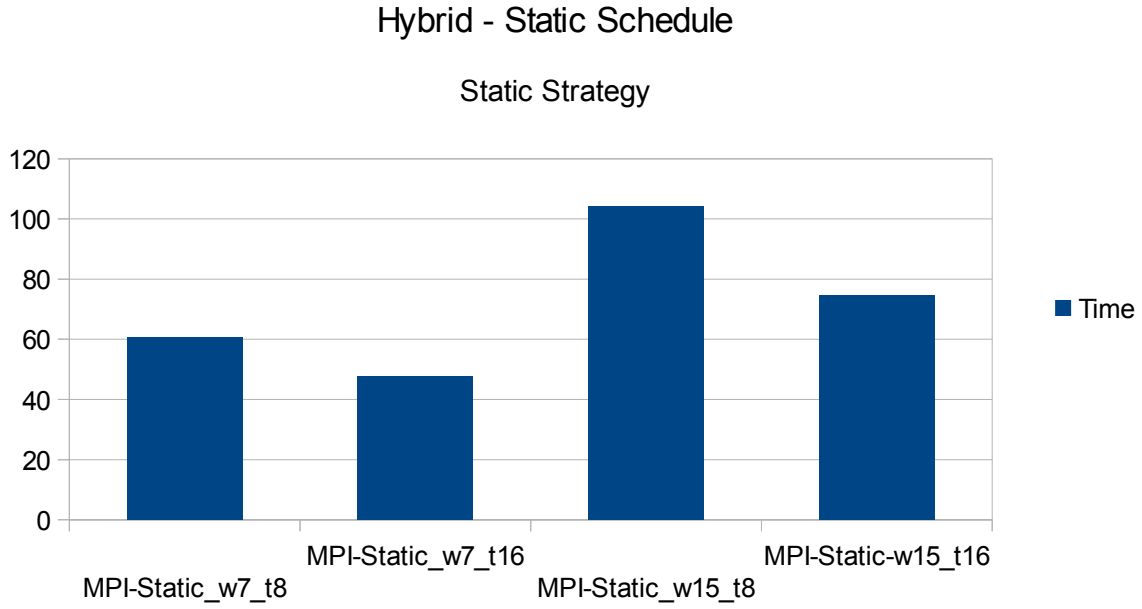

Hybrid - Dynamic Schedule
Static Round-Robin Strategy
120
100
80
60
40
20
0
MPI-Static-RoundRobin_w7_t8
MPI-Static-RoundRobin_w7_t16
MPI-Static-RoundRobin_w15_t8
MPI-Static-RoundRobin_w15_t16
Time

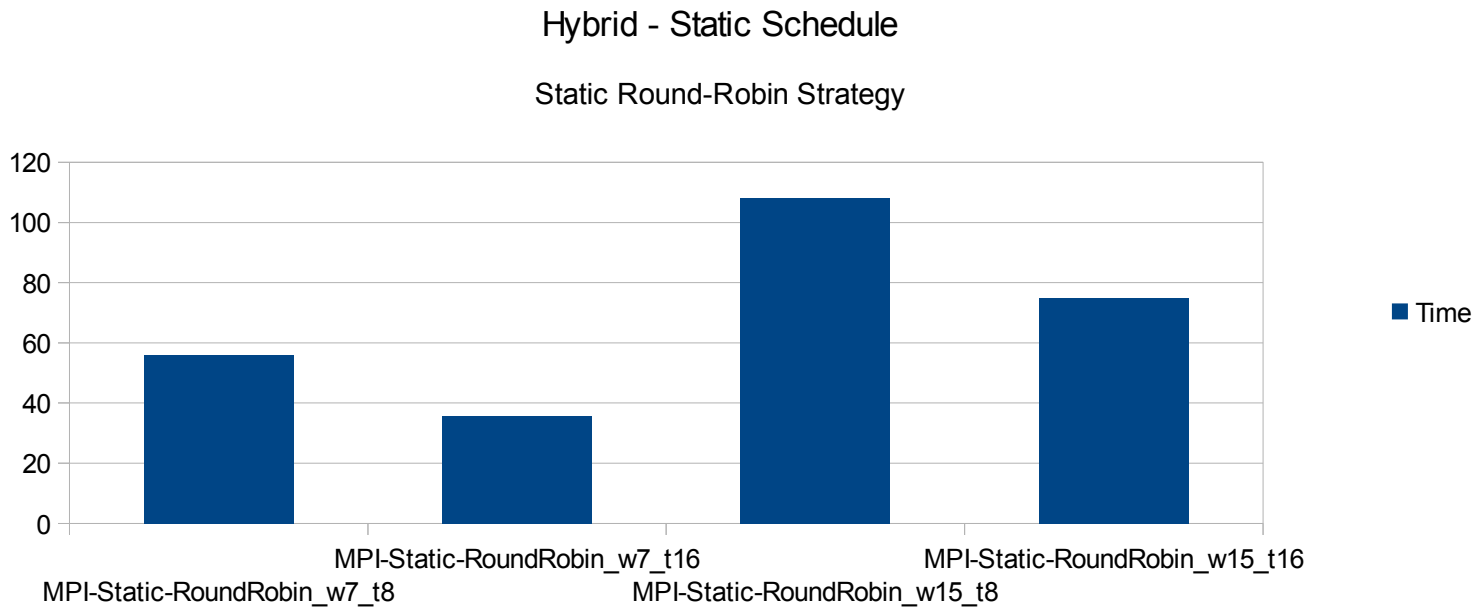

Hybrid - Dynamic Schedule
Dynamic Strategy
120
100
80
60
40
20
0
MPI-Dynamic_w7_t8
MPI-Dynamic_w7_t16
MPI-Dynamic_w15_t8
MPI-Dynamic_w15_t16
Time

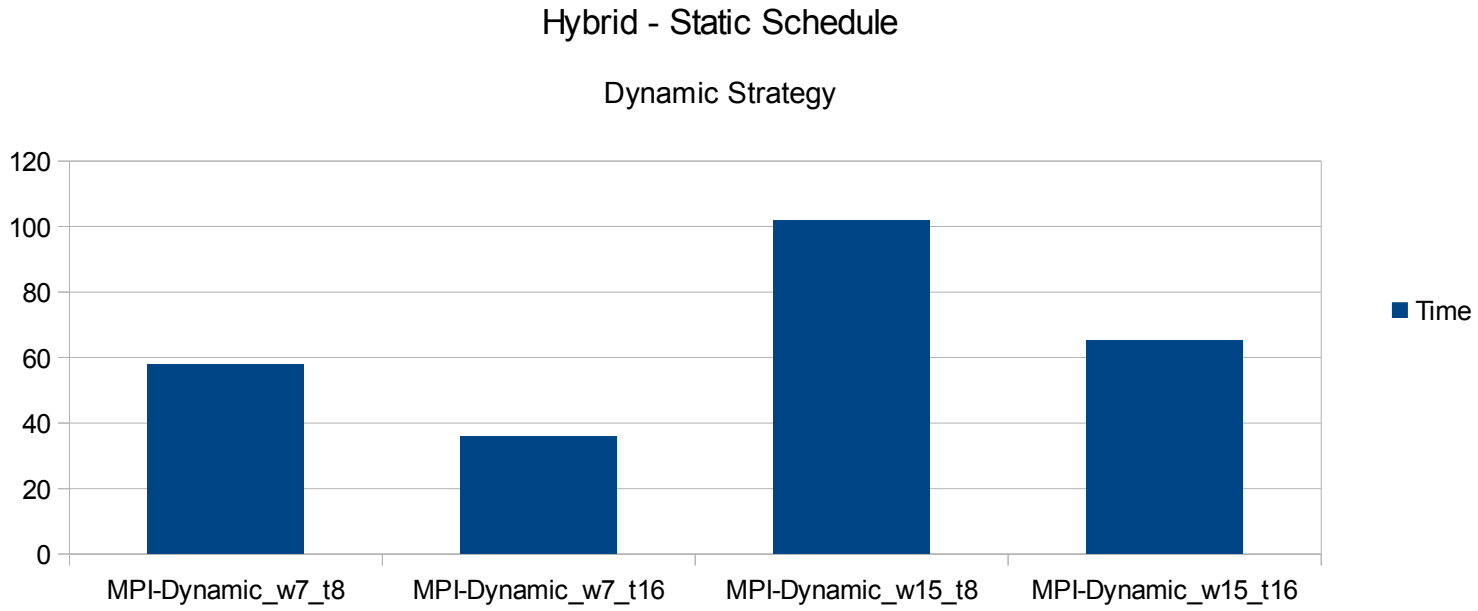

Hybrid - Guided Schedule
Static Strategy
120
100
80
60
40
20
0
MPI-Static_w7_t8
MPI-Static_w7_t16
MPI-Static_w15_t8
MPI-Static_w15_t16
Time


Hybrid - Guided Schedule
Static Round-Robin Strategy
120
100
80
60
40
20
0
MPI-Static-RoundRobin_w7_t8
MPI-Static-RoundRobin_w7_t16
MPI-Static-RoundRobin_w15_t8
MPI-Static-RoundRobin_w15_t16
Time


Hybrid - Guided Schedule
Dynamic Strategy
120
100
80
60
40
20
0
MPI-Dynamic_w7_t8
MPI-Dynamic_w7_t16
MPI-Dynamic_w15_t8
MPI-Dynamic_w15_t16
Time


Hybrid - Static Schedule
Static Strategy
120
100
80
60
40
20
0
MPI-Static_w7_t8
MPI-Static_w7_t16
MPI-Static_w15_t8
MPI-Static-w15_t16
Time


Hybrid - Static Schedule
Static Round-Robin Strategy
120
100
80
60
40
20
0
MPI-Static-RoundRobin_w7_t8
MPI-Static-RoundRobin_w7_t16
MPI-Static-RoundRobin_w15_t8
MPI-Static-RoundRobin_w15_t16
Time


Hybrid - Static Schedule
Dynamic Strategy
120
100
80
60
40
20
0
MPI-Dynamic_w7_t8
MPI-Dynamic_w7_t16
MPI-Dynamic_w15_t8
MPI-Dynamic_w15_t16
Time

De-balancing

The problem could actually get de-balanced if the number of threads used by the OpenMP schedule goes to low or even to high increasing the thread creation overhead. In our test cases we have 8 physical cores available and so the number of threads/processes is likely to affect the outcome. Take a look at the first chart (MPI only), thus some strategies perform better than others if we do not balance the number of processes with the number of threads we are getting a high negative impact. 7 processes perform better then 15.

The same can be observed with the hybrid approaches, apart from the fact that it is better having more OpenMP threads then MPI processes because OS threads are easier t manage then actual bound processes.

Overhead

The overhead in terms of parallelization is obvious. First, a high rate of MPI processes will result in message overhead where having more OpenMP and less MPI processes the compared overhead is less. Also the scheduling method and the OpenMP chunk size will impact the overhead, less thread creations will result in less overhead.

The implementations

The first issue we see is that any of our hybrid approaches perform worse than the MPI only approach. This has to do with the overhead added using OpenMP and how we have setup our threads, lowering the number of threads/processes to be always less the total number of physical cores would have lower that overhead (thread stalling, resource trashing, ...).

Comparing the schedule options the guided schedule performed better in general and the dynamic scheduling very close behind while the static scheduling seams to be the worst.

Analyzing the 3 implementation strategies we see that both the dynamic and the round-robin strategies seams to be very close to each other. The dynamic strategy adds more overhead thus it is not really noticeable on a single-machine environment, in a real scenario the round-robin approach would theoretically perform better then the dynamic one.

Pros/Contras

It is obvious that a mixed implementation will be useful, but this always comes down to the problem that has to be solved. Using a hybrid approach introduces complexity, not only the complexity of the implementation, which of course has to be considered, but the complexity when designing the solution. A hybrid approach in turn can take advantage over others due to it's flexibility, we are using the benefits of both world: the communication skills of MPI, which can run tasks on different nodes located on different physical machines, and the power of OpenMP to create lightweight threads to use the power of the local node.

Where to use one or the other

A hybrid approach can be considered if we require memory coherency at a node, high data movement within the node, we want to synchronize memory instead of adding blocking barriers or we have node decomposition problems.

In turn if we have distributed memory or our task does not have a heavy data exchange within the

same node we are better served with a non hybrid approach using MPI for example. The same applies to an only OpenMP approach in case we do not require any MPI benefit.

GPU implementation – OpenCL

We have chosen to implement our GPU mandelbrot set using OpenCL and the AMD OpenCL SDK [1].

Hello World

The sample CUDA program, like the sample OpenCL programs are based on the same idea. The actual code that will be executed on the GPU gets uploaded to the GPU and the GPU then executes it. This is the same idea as creating a vertex/pixel shader in traditional GPU rendering.

In the case of the example the part that gets run on the GPU is the hello kernel:

```
__global__
void hello(char *a, int *b)
{
        a[threadIdx.x] += b[threadIdx.x];
}
...
hello<<<dimGrid, dimBlock>>>(ad, bd);
```

As show in the following snipped *a* and *b* are transferred to the device to be used as arguments for the kernerl (the *__global__ void hello* function):

```
cudaMemcpy( ad, a, csize, cudaMemcpyHostToDevice );
cudaMemcpy( bd, b, isize, cudaMemcpyHostToDevice );
```

After the function has been executed on the GPU we will transfer *a* back to the host:

```
cudaMemcpy( a, ad, csize, cudaMemcpyDeviceToHost );
```

Used organization

The GPU thread organization I finally adapted is to use as much jobs as possible, one for each pixel. It was the simplest way to setup the scenario and with a modern card I achieved the best results with it. I also tried different approaches as having a job for each 4x4 block, this this can get achieved easily by modifying the job assignment and adding a small loop in the kernel program.

Multi-GPU in CUDA and OpenCL

In OpenCL Multi-GPU can be added very quickly. Just create a new CommandQueue with clu_create_command_queue using a different device id and enqueue tasks on each queue. Once a queue for each device has been created the job assignment could be like the following snippet:

```
...
// Create a queue per device
for(cl_int i = 0; i < num_devices; i++) {
        cmd_queue[i] = clu_create_command_queue( ... , i, ...);
}
...
// Create a block of work for each device
size_t work_block[2] = {width, height/num_devices};
for(cl_int i = 0; i < num_devices; i++) {
        size_t work_offset[2] = {0, work_block[1]*i};
        // We are using monocrom images, of you are using
        // 3 chars per pixel you have to multiply per 3!
        size_t offset = device_work_offset[1]*width;
        clEnqueueNDRangeKernel(cmd_queue[i], ... , work_offset, device_work_size, ... );
        // Read in a non-blocking fashion because we know no one
        // will write where we will read
        clEnqueueReadBuffer(cmd_queue[i], ... , CL_FALSE, ... );
 }
// Make sure all is done
for(i = 0; i < num_devices; i++) {
        clFinish(cmd_queue[i]);
}
...
```

The idea for CUDA is similar, the idea is to send the GPU with a given device id the data and the work to be processed and to read back the portion that has been processed by the given device.

GPU vs CPU

Having build both a CPU and a GPU version we directly see that the GPU version is about 70 faster then the CPU implementation.

We setup two test with the same dimension of 10000 x 10000 px. The CPU version took 10.69633032 seconds to complete while the GPU version only took 0.149057 seconds.

Scaling the GPU implementation

The scaling of a GPU implementation seams to be obvious at the first glance but we get fast to the border of being able to scale more. The idea to massively scale a OpenCL/CUDA implementation is to create a hybrid version using MPI to scale over physically separated GPU nodes.

Compile & Run

Compiling and Building

First of all make sure you have an OpenCL implementation installed and set the right path to your installation in the building script (*build.sh*). Otherwise you will not be able to build the OpenCL sample. Some OpenCL implementations use different headers which could lead to compile errors.

Go into the source folder and execute the following script:

```
$ ./build.sh
```

You can customize what to build and what to enable to create different binaries with different features. The following compiler options are available for MPI and the hybrid implementation:

Compiler Option	Description
-DDEBUG	Enable debug output
-DWITH_OMP	Enable OpenMP hybrid support
-DWITH_X11 -lX11	Result will be drawn using a X11 window
-DWITH_PBM	Enable PBM creation after the Mandelbrot set has been created.
-DWITH_BENCHMARK	If set no PBM files or X11 output will be generated. Use this for benchmarking.
-DSET_OMP_MODE=[0..2]	Set the OpenMP schedule mode. 0 for static, 1 for dynamic and 2 guided
-DOMP_CHUNK=[1..]	Set the OpenMP chunk size. By default 1.

Table 1 – Available compiler options

Run

The easiest way to run the samples is to use the provide *run_** scripts. Customize the scripts to adapt them to your own needs.

References

- [1] AMD OpenCL SDK: http://developer.amd.com/tools/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk/
- [2] MPI & CUDA: http://*eprints.dcs.warwick.ac.uk/270/1/sc-lu.pdf*