CSE260 Project Report

Two Approaches to Particle Simulation; OpenMPI and CUDA

Authors: Bjørn Christian Seime, Matteo Mannino, Terje Runde

Project Goal

In this project, our goal is to make particle simulation run in parallel using both CUDA and MPI implementations. The scale of parallelism between the are two vastly different, and we wish to compare implementations to gain insight on the strengths and weaknesses between the two different paradigms of parallelization.

Background

N-body simulation

Particle simulations are of huge interest in many scientific fields. Whether each particle represents an ion, a billiard ball or even entire galaxies, the structure of the problem is similar: Each particle in a system has initial position, velocity and acceleration given by the simulation characteristics, and affect each other with some gravitational or repulsive force. How to predict the motion of these particles with a discrete timestep is known as n-body simulations. [1]

It is difficult to solve and justify the correctness of such simulations through pure data. Actually, for a particle count greater than 3 it is known to be impossible [2], and that is why approximate simulations are of particular interest. Visualization of simulations helps researchers explore the behaviour and find evidence for physical models from the real world.

The simulation we wish to implement across both implementations obeys the interaction force law between particles i and j,

$$\mathbf{f}_{ij} = G\frac{m_i m_j}{\left\|\mathbf{r}_{ij}\right\|^2} \cdot \frac{\mathbf{r}_{ij}}{\left\|\mathbf{r}_{ij}\right\|},$$

Where the total force applied to particle i from all interactions 1 through N will be,

$$\mathbf{F}_i = \sum_{\substack{1 \le j \le N \\ j \neq i}} \mathbf{f}_{ij} = Gm_i \cdot \sum_{\substack{1 \le j \le N \\ j \neq i}} \frac{m_j \mathbf{r}_{ij}}{\left\|\mathbf{r}_{ij}\right\|^3}.$$

The particles will contained, and collide and bounce off boundaries defined by a rectangular grid without loss of energy.

The naïve serial implementation

A naïve serial implementation is given by UC Berkeley PhD candidate Grey Ballard [3]. For each iteration, each particle computes the force the other $n-1$ particles exerts on itself. With n particles in the system, it is easy to verify that the running time of the naïve algorithm is $O(n^2)$.

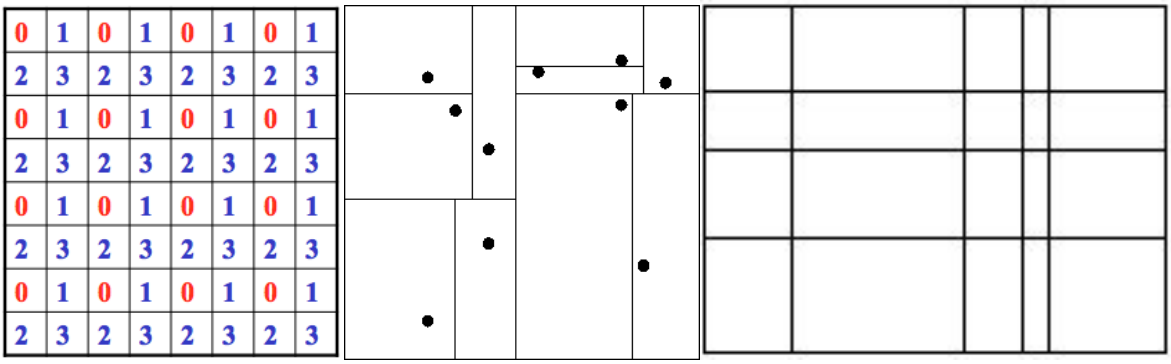
Parallelization

Researchers can get better precision per time unit by running their simulation in parallel (i.e. by reducing the time step and running on a supercomputer), and this is the prime motivation of improving the serial implementation. Best parallel speedup is achieved when the problem is divided into equally-sized chunks onto the available cores, but we also have to consider the overhead of the consequences of splitting the mesh. Each subdivision would have to pass particle information (ghosts) to each other because, indeed, each particle exerts forces on all other particles. Also, particles that moves out of a node's boundary will have to be sent to the corresponding node. A common approximation is to ignore particles that are outside some radius constant because the force will be small enough to be neglected.

Because approximations like this can be made, it becomes feasible to do particle simulation in parallel. We will now continue to dig into issues that makes particle simulation hard, and explain how we can adjust our algorithms.

Load balancing

As previously discussed, proper load balancing is vital to get the best performance out of a parallel implementation of this problem. Many algorithms exists for this purpose and we will discuss a few of them here.



Block cyclic decomposition

With block cyclic decomposition, the simulation mesh is divided into a considerable amount of sub-blocks. The sub-blocks are then distributed in modulo fashion, such that each computing unit can execute on disjoint, although scattered locations. This algorithm does a fair job approximating equal amount of particles on each computing unit in most cases. Actually, it will

converge to be optimal as the number of sub-blocks goes to infinity (i.e. each unit of work gets smaller), because the total load gets uniformly distributed on all computing units.

There is a tradeoff, however, because as the number of sub-blocks increases, the amount of data that has to be transferred between nodes also increases. This is why, for many applications, more clever algorithms is needed to make it scale.

Orthogonal Recursive Bisection

Orthogonal Recursive Bisection [4] addresses many of the problems circular mapping is suffering from. Instead of statically dividing up the mesh into a number of cells, it splits work in equal parts in a recursive manner. By doing this, it avoids the problem of dividing the mesh up in many cells unnecessary at the cost of additional overhead.

This algorithm also has some drawbacks. One of them, is that each computing unit would require extensive knowledge of how the whole mesh is splitted, in order to transfer ghost- and transfer particles to the appropriate nodes. The next algorithm presented does a tradeoff that solves this in a subtle, but simple manner.

Nicol's Rectilinear Algorithm

Nicol's Rectilinear Algorithm [5] is a simplification of the Orthogonal Recursive Bisection (ORB) algorithm, where splits have to extend the whole computation mesh. It does not do as well as ORB, but it gives the advantage that it simplifies communication: Each node will only get 4 neighbors (8 including the corners).

Data composition

How the particles are stored on each node can also be tuned for a specific application. In general, it is a good idea to choose data structures that would improve the asymptotic lookup time for nearby particles. This, of course, connected with the fact that distant particles are neglected in motion calculations.

We here discuss four different data structures that must be considered for particle storage when doing particle simulation.

Linear storage

Linear storage is very convenient for this cause, because all particles on a node would live very closely in memory. Depending on the size of each particle (i.e. how much data it holds), all particles may fit in memory - consecutively, which will be a huge advantage. Iterating through the array of particles would then require a minimum amount of memory loads, which could compromise for the $O(n)$ lookup time.

Quadtree

A quadtree is a basic tree data structure in which each internal node has exactly four children, just like a binary tree with two more childrens. Thus, it can be used to hold our two dimensional mesh of particles into a well-behaved lookup tree. It offers lookup in $O(log_4 n)$ time, significantly better than its linear counterpart.

A problem with this data structure is its complexity in our application. One would not be satisfied to just make one lookup for the list of particles for a given node, you would also have to consider all the adjacent nodes, which can be up to 8 in total. For this reason this would certainly be more scalable than the simple vector solution, if implemented correctly. But for a smaller number of particle computing nodes (or processes, rather) it would not be worth the overhead.

Chaining mesh and filling curves

Chaining the mesh in a way that bundles together nearby particles can also improve lookup time. There exists several methods for doing this, including *chaining mesh* (Hockney & Eastwood, 1981) and space-filling curves. Although these algorithms suffer from the the same problem as the quadtree, they can be just as good and work with lower overhead.

Implementation

CUDA

General Purpose GPUs, Massively Parallel Processing

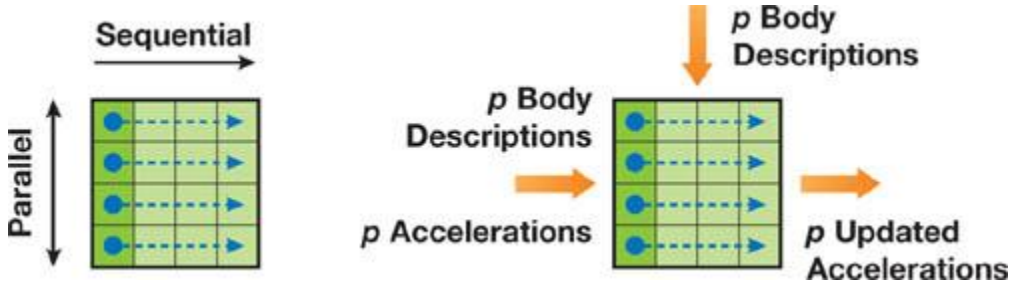
CUDA C is a heterogeneous parallel programming interface that enables exploitation of data parallelism, combining both host and device code. A CUDA program can be conceptualized by an executable function called a Kernel. A CUDA Kernel is executed in parallel by a group of threads subdivided into blocks.

Memory in CUDA is divided between global memory and shared memory. Global memory can be accessed and written to by all running threads. Shared memory, however, is only visible to all threads within a block. Shared memory is generally more efficient to read and write from, and can result in significant speed gains when data is operated over several times. Usually, a key optimization for fully utilizing the gains from shared memory involves making sure the data transfer between global and shared coalesce - all threads copy memory from a contiguous segment in global memory to shared memory.

Program execution

The initial attempt was mesh-based. We subdivided the mesh into blocks that corresponded to thread blocks, and copied particles that fell into those blocks into shared memory for updates. This approach was severely limiting as it made the scalability (number of particles) highly dependent on the available size of the shared memory. To remedy this, a subdivision method was considered, but proved difficult to implement efficiently in CUDA. Thus, this approach was abandoned in favor of a tile-based approach for shared memory usage.

The idea is to take advantage of the massive parallelization in CUDA by computing the NxN interaction matrix between each particle. The method described is outlined in GPU Gems 3 [10]. Each thread computes all N interactions with each particle.



Each particle is effectively given a thread, and each thread iterates over all existing particles and computes the accumulative interaction of each particle on the current particle. The effective use of shared memory here comes from the coalescing the writes from global memory. Contrast with the previous attempt, coalescing is not easily achieved as it required either a sort for every iteration, or a global particle array of equal size for every mesh tile combined with a process to

pass particles around bins and recognize ghost cells - similar to message passing techniques common in MPI.

The Main Kernel

```
Main Kernel: Inputs - Previous Positions, Previous Velocities

    allocate shared memory with size of the number of threads per block, to
store positions

    retrieve single particle position and velocity from previous positions
using block and thread IDs.

    set total effect onto single particle to zero.

    for each thread block tile:
        copy tile block of particle positions from global memory to shared
memory (coalesced)
        compute accumulative effect all positions in shared memory
        add accumulation to total effect
    end

    update velocity and position

    check simulation boundary collisions, and compute reflections

    copy velocity and position back into global device memory

end
```

It is evident that utilizing shared memory and memory transfers from global is crucial for optimization in CUDA. The following simple data structures were chosen for optimization.

The choice of data structure

```
Data structures to store the particles during simulation:
    // for receiving
    __device__ double2* position
    __device__ double2* velocity
```

Since we only need position to compute the interactions for our simulation, it is useful to separate it from velocity so we do not clutter the shared memory with data that will not be continuously used.

Motion Calculation Run-time Complexity

The motion calculation is described in the naive serial version section. In the serial version, for N particles, we generally have $O(N^2)$computations per iteration of the simulation. In our CUDA implement, given P threads, we achieve $O(N^2/P)$computations per iteration where is P <= N.

With this is mind, we expect the time complexity to grow by $O(N)$ until N > P, then grow by $O(N^2)$. The trick with CUDA, however, will be selecting the right block size and grid size such that we maximize occupancy on the device.

Target Hardware, C2050 Fermi

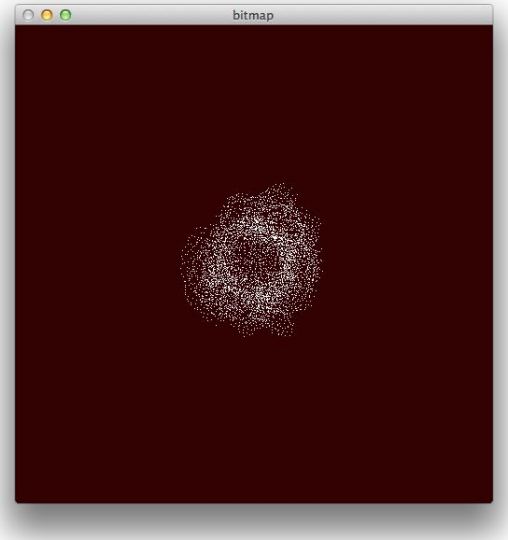
Global memory	2.62 GB
Gflops peak (DP / SP)	515 / 1030 Gflops
SM / SP per SM / total SPs	14 / 32 / 448
Shared memory per SM	48 kB / 16 kB (depending on L1 configuration)
Register file (per SM)	128 kB
L2 cache	768 kB
Memory bandwidth	144 GB/s
Performance DGEMM	31% @ 157 Gflops
Performance SGEMM	37% @ 379 Gflops

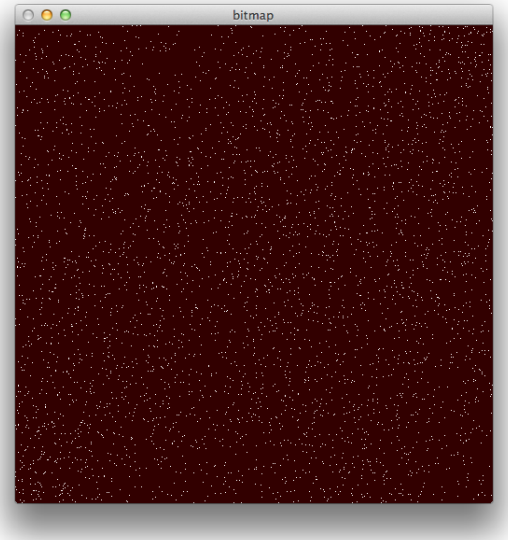
Table 1: GPU specifications for the GTX280 and the C2050.

According to the specifications, we should see the jump from $O(N)$ to $O(N^2)$ when $N > 448$ (total SPs), but definitely far more than that due to the warp scheduler per SM.

Visualization

While visualization can not be done for the target hardware, there is an OSX version that binds the texture memory on the device to a shader program wrapped around an OpenGL context. This is helpful for visual inspection of the particle demo, and has been instrumental in identifying bugs. (Note: CUDA 5.0 and Command Line Tools needs to be installed via the App Store and XCode - both are free)





OpenMPI

The MPI abstraction

Using a message passing interface to get parallel speedup on particle simulation makes it feasible to implement the concepts previously discussed. The MPI abstraction hides away the underlying network architecture through simple $Send$ and $Receive$ commands and makes it easy to develop programs. It also includes clever collective and distributive algorithms to minimize message count and sizes.

A MPI program is run as a given number of MPI processes, usually one process per physical processor core. The program is then executed on all MPI processes in parallel, exchanging data between each other if necessary. Almost all real programs do have a serial part, so a common pattern is to let one node (e.g. the one with $rank = 0$) do some extra work.

Program execution

Initialization phase

When running our program, the root node sets up a particle simulation problem with either randomly generated or user defined simulation parameters. It also generates initial partitioning, which is used to scatter particles to appropriate nodes. The choice of partitioning strategy will be discussed later.

The simulation phase

```
for each iteration:

    sort out ghost particles that will be sent to neighbors

    for each neighbor:
        send out ghost particles and particles to be moved
        receive ghost particles and particles moved to this node

    calculate the acceleration for each particle

    change the velocity and position for each particle based on acceleration
    discover and sort out particles to be moved to neighbors

    if time for plotting or rebalancing (based on iteration number):
        send all particles (including particles to be moved) to root node
end
```

All nodes, including the root node, runs the loop as described above during the simulation phase. The most expensive part is the calculation where each particle gets its acceleration, position and velocity updated. Structurally, the main loop is similar to any other iterative algorithm using message passing communication; the interesting part is where MPI nodes shares data (i.e. particles) with each other. The neighbor communication part constitutes an

implicit barrier; all nodes needs to complete the transfer part before anyone can start on the physics calculation.

It is evident that the root node acts as a master for load balancing (and plotting) and is crucial for a utilized parallel program execution.

The choice of data structure

```
Data structures used by each node the store the particles during simulation:
    // for receiving
    vector<Particle> citizens; // the nodes own particles
    vector<Particle> foreign_ghosts; // the ghost particles received from other
nodes

    // For sending. Pointers are 8-element arrays of vectors
    vector<Particle> *leavers; // particles to be transferred to neighbors
    vector<Particle> combined_leavers; // all particles leavers combined
    vector<Particle> *our_ghosts; // ghost particle to be copied to neighbors
```

Each node store it's particles in a STL vector. This data structure was chosen because of simplicity. It is easier to use than plain arrays for more complex operations like deletion and insertion. The implementation takes care of resizing when appending elements, and compacting when removing elements. Plain C++ could have been used, but it would require us code some extra logic for resizing as the number of particles per node is fluctuating during the simulation. The vector type has the property of storing all particles as a continuous memory block, which gives better iteration performance compared to tree structures or linked list. The expensive deletion cost for vectors is not that important. The chances for a particle to leave is quite small for small timesteps.

Particles that are to be sent to neighbors are sorted into one vector for each neighbor. The sorting of particles to be transferred is done at the same time as each particle get it's position updated.

Motion Calculations

The motion calculation is quite similar to how it's done in the naive serial version. Each particle get it's acceleration updated by iterating through all other particles (including ghost cells) and calculating each interaction force. The ghost particles are necessary for getting accurate calculation. Some of the nodes particles might be position right at the border to a neighbor, and may therefore interact with particles close to the other side of the border. The run-time can be described as

$$O(M * (M + G)) = O(M^2 + MG)$$

where M is the number of particle at a node, G the number of ghost particles. The MG term can be neglected for big mesh sizes and/or small interaction radiuses. If we are using only a single

node, then M = N (number of particles) and G = 0. This will give the run-time $O(M^2 + MG) = O(N^2 + N * 0) = O(N^2)$, which is the same run-time as the naive serial version.

Communication

Communication is done using MPI function calls found suitable for the amount and distribution of data we are sending. We obviously don't want to send more data than we need, and in this application communication is either 1) scattering from the root node to all nodes in the system, 2) interprocess transmission of ghost nodes or 3) gathering from all nodes to the root node. Note that the latter communication is only needed for plotting and load balancing purposes, or when the simulation ends. There are MPI functions that gets data distributed or collected with minimal message sizes.

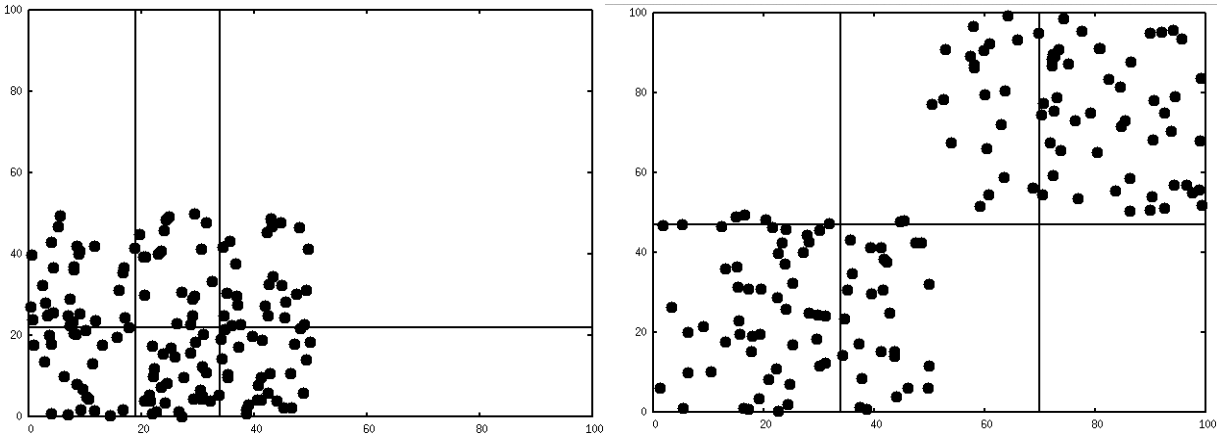
The initial particle scattering is done using the function `MPI_Scatterv`; the root node holds all the particles and knowledge about their destination on one of the MPI nodes, and distributes this to every other node. The interprocess communication of ghost nodes is equally simple; every node prepares a list of ghost particles for each adjacent cell in the partitioned mesh, and transmits them in 8 messages. This ensures that each two adjacent nodes exchange their ghosts while operating completely distributed (i.e. there is no root node interaction). There is a potential problem with this approach, however, because the startup time to send a message can be much larger than the transmission time, particularly when running on large clusters. We will discuss a few approaches to improve this later.

In our implementation, the root node as well as all the other nodes are allocating send and receive buffers with size of the largest possible scatter message, containing all particles. This also yield problems, because every non-root node potentially allocates much more space than it needs. Note that this only is a problem where nodes share memory, because the serial part of the program (i.e. the root node) would have to allocate the whole chunk anyway.

Partitioning

Orthogonal Recursive Bisection was originally meant to be our main partitioning algorithm. This decision was changed early in the implementation phase. We found the ORB algorithm itself to be feasible to implement, but the logic for doing neighbor-communication and distributing the split information would require much more work: Each process would have to know the layout on the entire splitted mesh to determine where it's neighbors are located, which can be of arbitrary count. It would prone to logic errors which would be hard to detect and debug.

We decided start off with Nicol's Rectilinear Partitioning, and maybe implement ORB later if we had enough time (which ended up not being the case). NRA and the resulting neighbor logic was easy to implement, a trade-off for a potentially worse partitioning as already discussed. However, it turns out that in many cases NRA performs very well and with less overhead than ORB. Every node got up to 8 neighbors, depending wherever it was placed at the boundary of the mesh or not. The partitioning is run on the root node in startup phase, and the partitioning is distributed as x- and y coordinates to all nodes together with the particle data.



From these figures, it is evident that Nicol's Rectilinear Algorithm performs worse in some cases (e.g. with particles in two corners).

Dynamic load balancing

As previously discussed, load balancing occurs at certain points in the program execution by the partitioning algorithm. Load balancing involves calculations on the whole mesh and the full set of particles, so this is done by 1) letting all nodes send it's particles to the root (i.e. gather), 2) do the split calculations and 3) scatter the new distribution to all nodes.

The initial particle distribution is sorted into buckets using the same algorithm, to ensure that the load is uniformly distributed from the beginning. Load balancing interval is statically determined by the user through the command line (e.g. load balance after every 100th iteration). There exists more comprehensive approaches to dynamically distribute the load, because the particles will converge to be uniformly distributed. For this purpose, however, we will adjust the time step such that the simulation terminates before particles gets uniformly distributed. We will conclude this report by suggesting improvements on this matter.

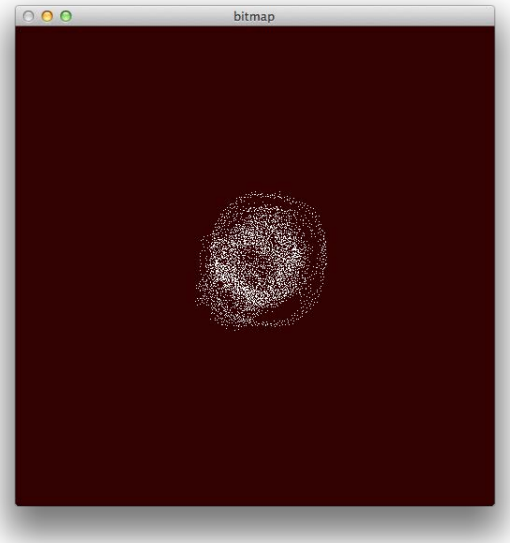
Visualization

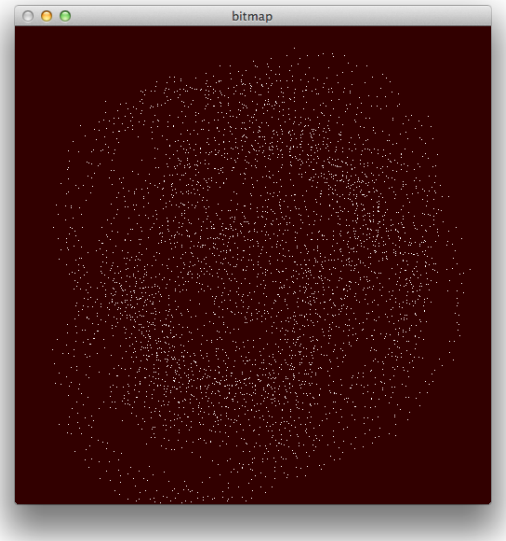
Visualization support was one of the first features we added. We used Gnuplot, as we knew this plotting framework would work on our computer cluster, Bang. The implementation was straightforward. Plotting support helped us verifying and debugging the partitioning and physics logic. The root node is the responsible for plotting, requiring all to participate by sending their particles.

Performance

CUDA

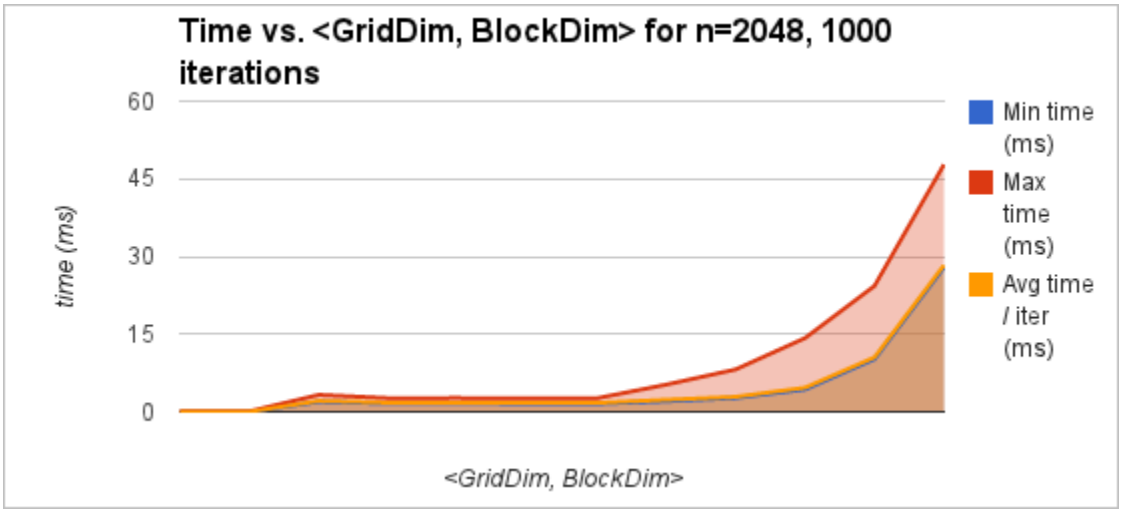
Since there is no partitioning being done, we run a simple fire works demo - clusters of particles are generated and allowed to expand as they repel each other.





In our CUDA implementation, the effective use of shared memory and the hiding of arithmetic latencies are key. Note that in order to have effective coalescing between global and shared memory, we need to align the number of threads with accessing the global memory. This means that the total number of threads will dictate the size of the shared memory used and moved from global memory by each thread block. 38 registers were used in our compilation.

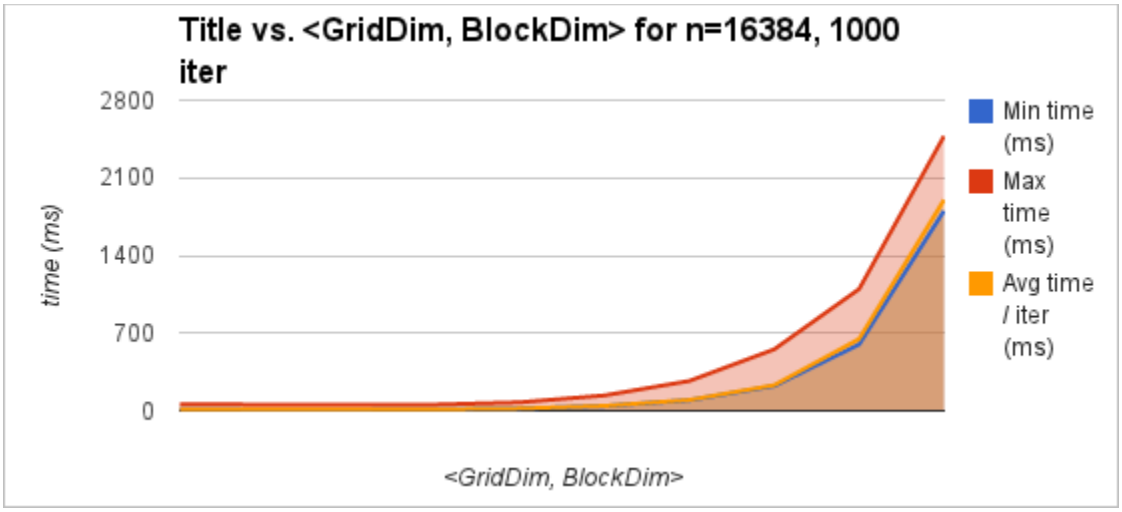
Holding the total number of particles to be constant, n = 2048, we vary Grid Blocks sizes and Thread Block sizes, starting from <1,2048>, <2,1024>, <4,512> and so on up to <2048,1>.



The curve starts to look characteristically polynomial when the Grid Size is less than the Thread Block size, as expected. This is because the gain from shared memory loses significance as fewer and fewer particles get copied into it. The latency of copying from global to shared memory starts to catch up by a rate of $O(N^2)$. Before that, we see roughly constant time, due to N staying constant and enough gains from fast shared memory access to offset the cost of moving the data. Fundamentally, there are as many processes working on the data as there is data, and thus the time cost must be constant as we hold N constant.

We see a large increase in total runtime between <2,1024> and <4,512>. Upon further testing, we find this jump happens between 768 and 769 threads per block. 768 seems to mark the maximum number of threads per block possible for CUDA to execute the kernel. If the number is greater than 768, the kernel is simply not executed, thus the low times are false.

Scaling up to N=16384, we see similar results. The x axis increments in powers of two, <32,512>, <64,256>, <128, 128>, up to, <16384,1>



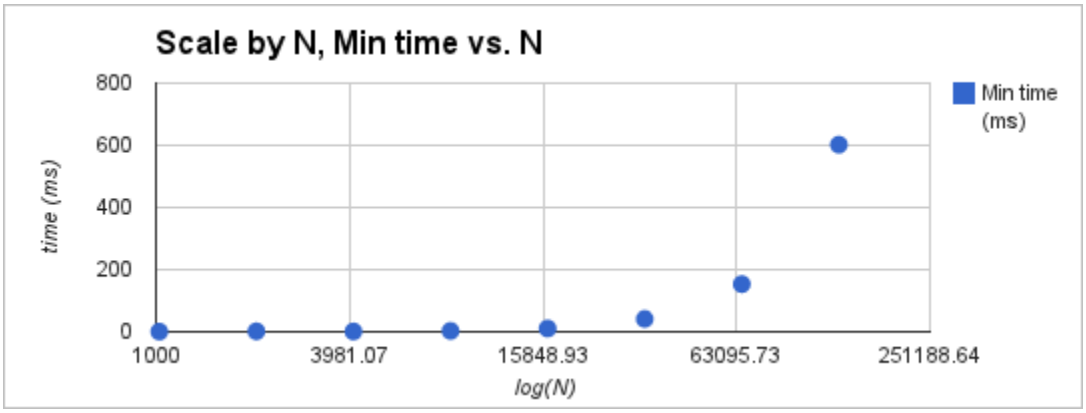
The following table shows the minimum times vs Grid and Block Dimensions. N = GridDim*BlockDim, and N is constant along the diagonals.

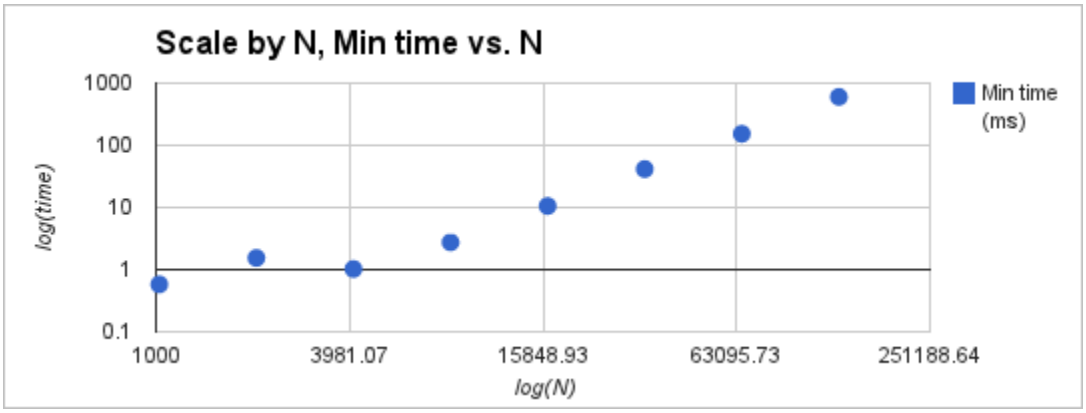
G, B	1	2	4	8	16	32	64	128	256	512
1										
2										0.95491
4									0.80208	1.91136
8								0.78572	1.60752	1.13945
16							0.78009	1.61155	1.15587	4.43232
32						0.76832	1.58537	1.0544	3.17894	13.1689
64					0.57846	1.54336	1.02454	2.75110	10.5636	43.6867
128				0.87302	2.02358	1.96496	4.01984	10.7733	42.0370	174.299
256			1.27580	2.66291	3.20156	5.88048	12.3812	41.5438	154.153	661.604
512		2.53820	4.30416	6.27046	10.7315	19.7246	43.6885	153.352	602.122	
1024	7.23148	10.1635	15.0859	24.9836	42.7931	78.6188	170.244	605.783		
2048	27.8876	38.4078	57.3100	94.8487	162.439					
4096	113.065	150.206	223.287	369.831						
8192	449.230	597.983	891.930							
16384	1802.42									
32768										

Table. Minimum times. Rows are the GridDim, and Columns are BlockDim, N = GridDim*BlockDim and is constant along the diagonals. Highlighted at the minimum times along each diagonal.

The minimum times per diagonal (N) are highlighted, and represent grid and block dimensions that give the best trade off between shared memory size and thread size where we can maximize the number of Gflops per thread. At this point, the warp scheduler can switch between warps efficiently in a way that hides the arithmetic latencies.

The following figures shows the minimum times plotted on a $\log_2$ scale on the N axis, and by a $\log_2$ scale on both time and N axes.





The linear nature of the log-log plot verifies the $O(N^2)$ complexity of the task. From this graph, we can make clear predictions on the best run time we can expect given a particle size N.

Further improvements for CUDA implementation

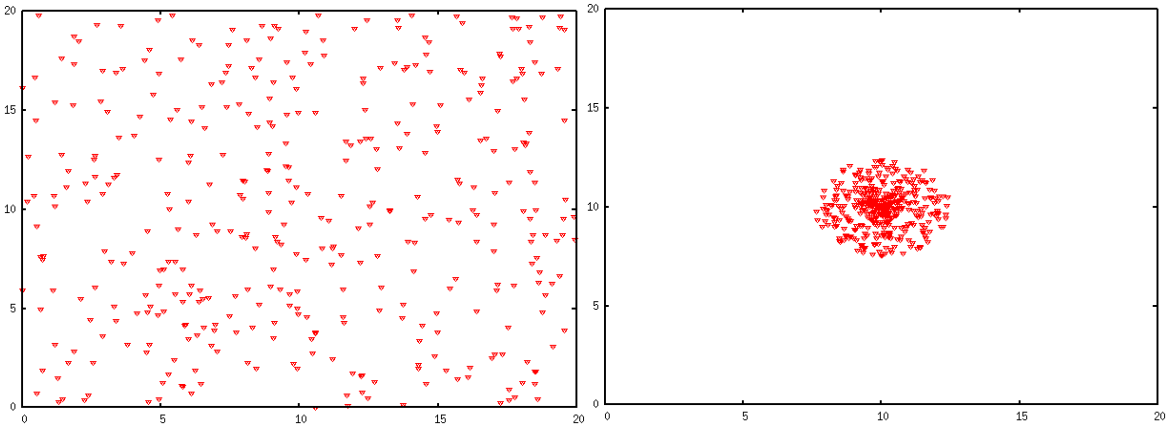
Many improvements that could be made to this, if time permitted.

- Loop unrolling the accumulative interaction calculation has been known to give high gains. [10]
- Subdivision by bin-hashing. [6]
- Do some recursive bisection prior to passing particles into the GPU.

In summary, the GPU is meant for data-parallel task with high volume data, and is generally ill-suited for methods that are sequential by nature, like sorting, unless you can get away with a quick approximation. While these methods exist, they could not be fully explored given time constraints.

MPI

We begin our discussion on performance of the MPI implementation by introducing two initial particle distributions for our simulations.



The first one is randomly distributed across the entire simulation mesh, and the second one deploys particles within a radius from the mesh center. It is obvious that the second simulation will look like an explosion due to the repulsive forces between each pair of particles, while the first will simply continue to be uniformly distributed.

A very important aspect of the parallel speedup for this simulation, is how effectively it is load balanced. If particles are uniformly distributed, load imbalance will be low through the entire simulation. As a result, dynamic load balancing will have little or no effect on performance: In fact, because load balancing introduces a huge overhead it can impact performance very badly. The second distribution is also the one of most practical interest; researchers are more curious about how particles interact a very short period of time after a specific event (e.g. a chemical reaction), rather than simulating a random phenomena.

From here, we will continue our discussion only considering the explosion-like simulation to see how good our algorithm runs in parallel. We will do a run-time analysis, discuss the scalability of our algorithm and suggest further improvements.

Run-time analysis

Run-time analysis on the calculation part gives us the asymptotic notations of the run-time:

$$O(M * (M + G)) = O(M^2 + MG)$$

Lets assume that there are *k* MPI nodes, and that the particles are evenly distributed through the nodes, so that each node as roughly the same number of particles. We can then substitute M with $\frac{N}{k}$. The number of ghost particles per node is usually low given that the total mesh

dimension is much greater than the maximal interaction radius, so that the number of particles per node is less than the surrounding ghost particles in the neighbor nodes. G will then be less than $\frac{N}{k}$. We can now rewrite the equation as:

$$O(M^2 + MG) = O(\frac{N^2}{k^2} + \frac{N}{k} * \frac{N}{k}) = O(2 * (\frac{N^2}{k^2}) = O(\frac{N^2}{k^2})$$

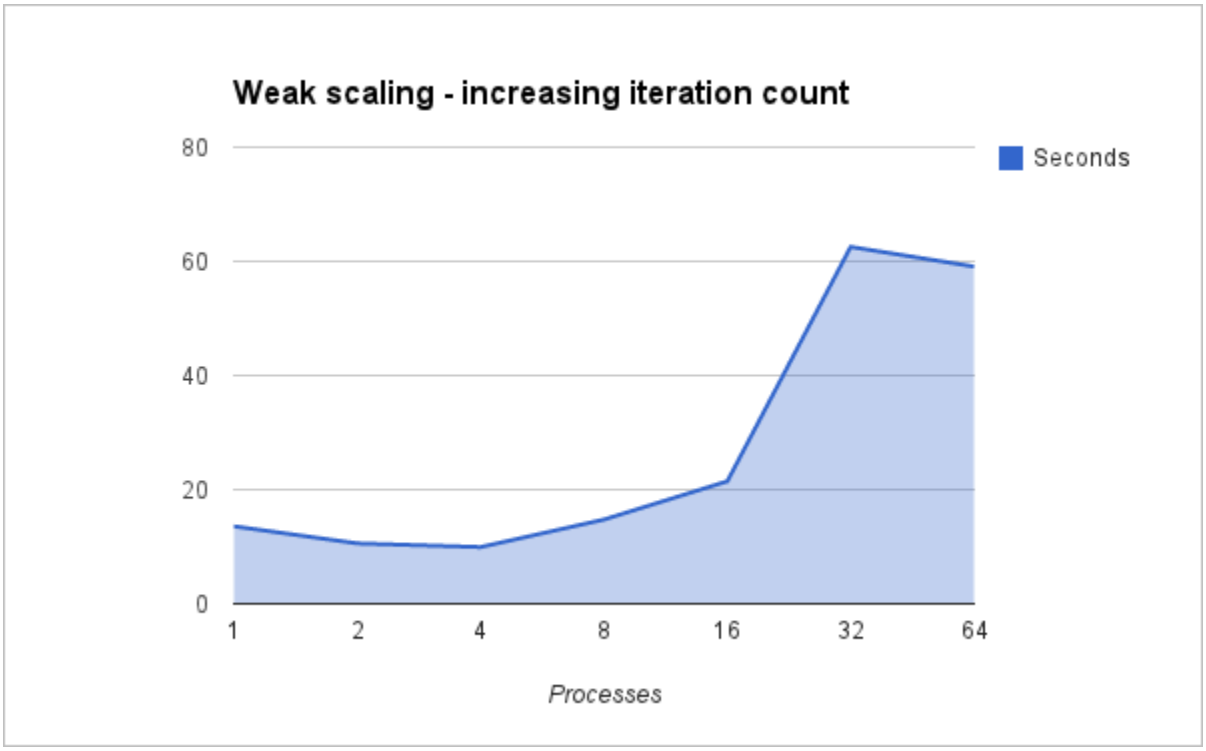
Lets assume all the nodes are just processes running on a single core processor, using an operating system with a preemptive scheduler. We'll also assume the cost of context-switching is so low that we can neglect it. The execution of the processes will then be interleaved, which gives us the total run-time of:

$$k * O(\frac{N^2}{k^2}) = O(k * \frac{N^2}{k^2}) = O(\frac{N^2}{k})$$

We now see that the total run-time is decreasing as k is increasing. This will continue as long our assumptions will hold. A superlinear speedup is therefore expected for small value of k, making it harder to measure the actual speedup. This effect would not exist if we had chosen to use a quadtree or chaining mesh for storing the particles in each node.

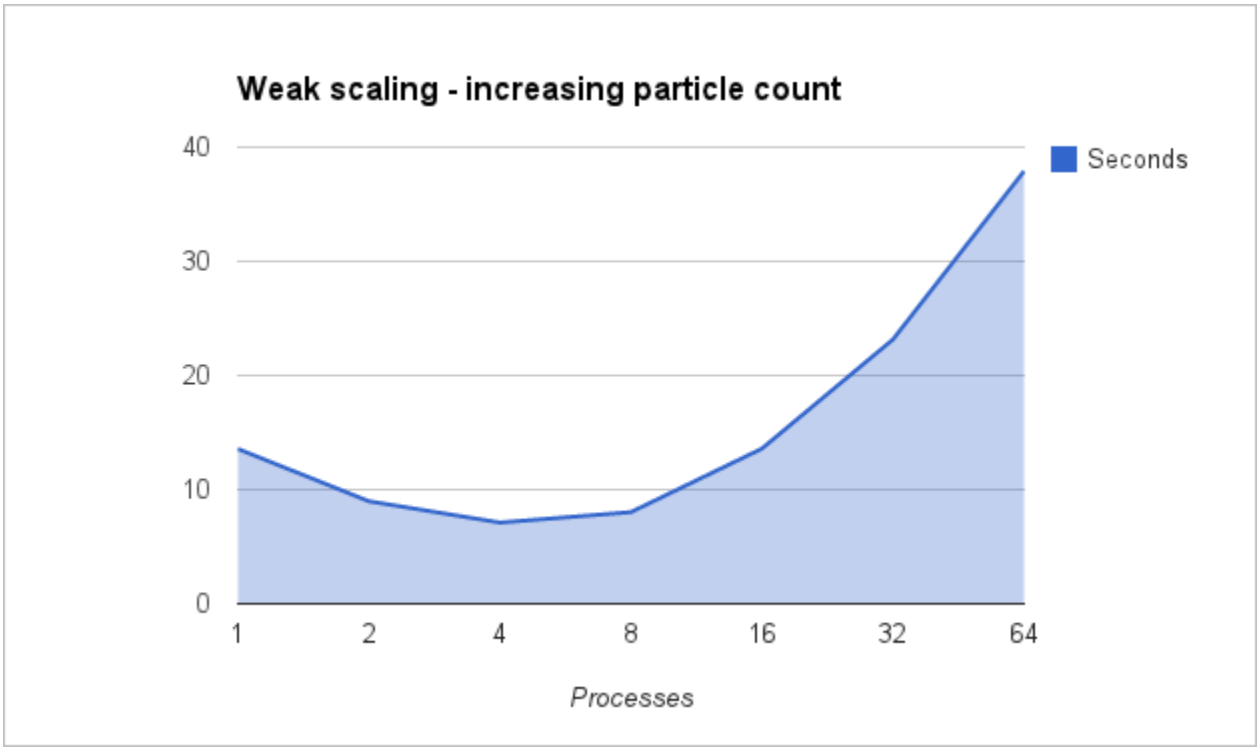
Weak scaling

We did weak scaling performance analysis in two different ways, see tabular data in Appendix A. First, we increased the number of iterations proportional with the node count. We also had to decrease the time step so that the simulation would be equivalent, only with better precision. The load balancing interval was also scaled accordingly, i.e. proportional to the iteration count. Secondly, we increased the number of particles, keeping a fixed time step and iteration count.



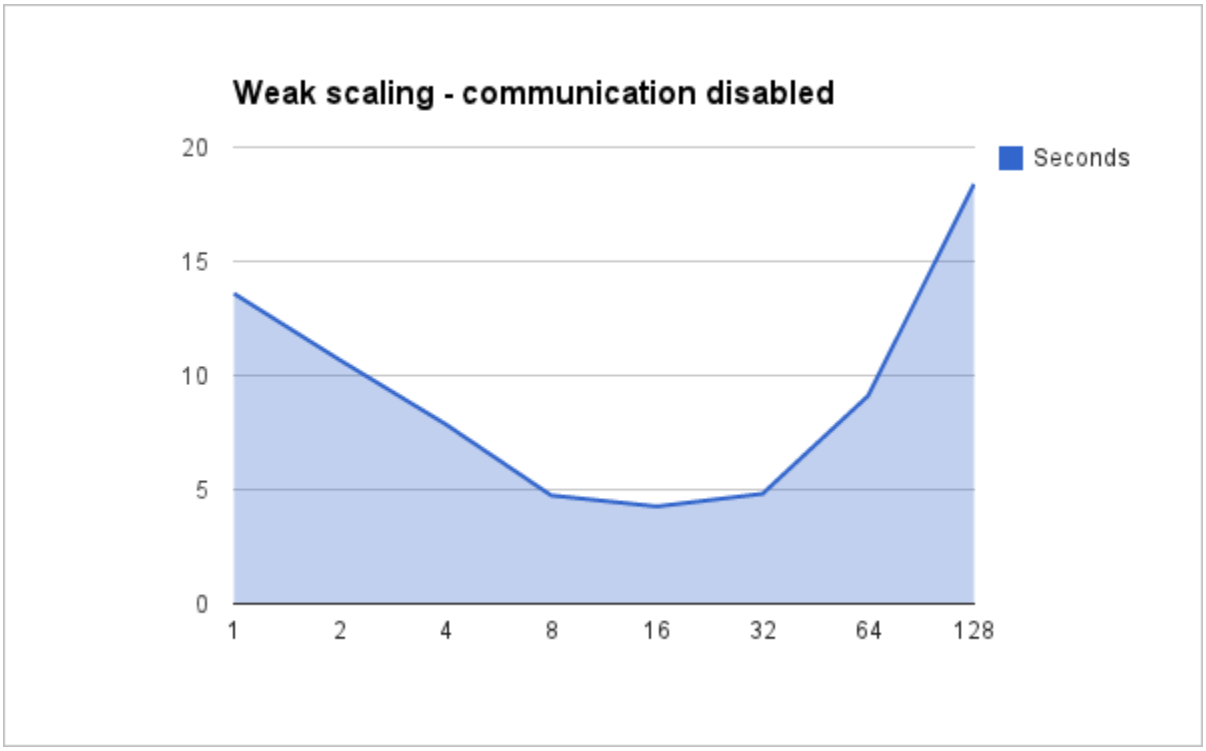
If perfect weak scaling were achieved, the run time in seconds would remain constant as iteration count and number of processes increases. We are experiencing superlinear scaling as we increase the number of processes from 1 through 4 (for reasons previously discussed), but the effect disappears when the process count increases further. However, it can be seen from the figure that our implementation scales reasonably well for up to 16 processes.

In our computing environment, it does not scale for larger numbers of processes. This is expected as the message passing overhead increases dramatically. It can be seen that a 64 process simulation scales better than a 32 process; this is likely because the mesh is allowed to have an equal number of splits in both dimensions: The number of ghosts that has to be sent in each direction is the same, figuratively.



The same results is evident in our second weak scaling measurement results, where we increased the particle count.

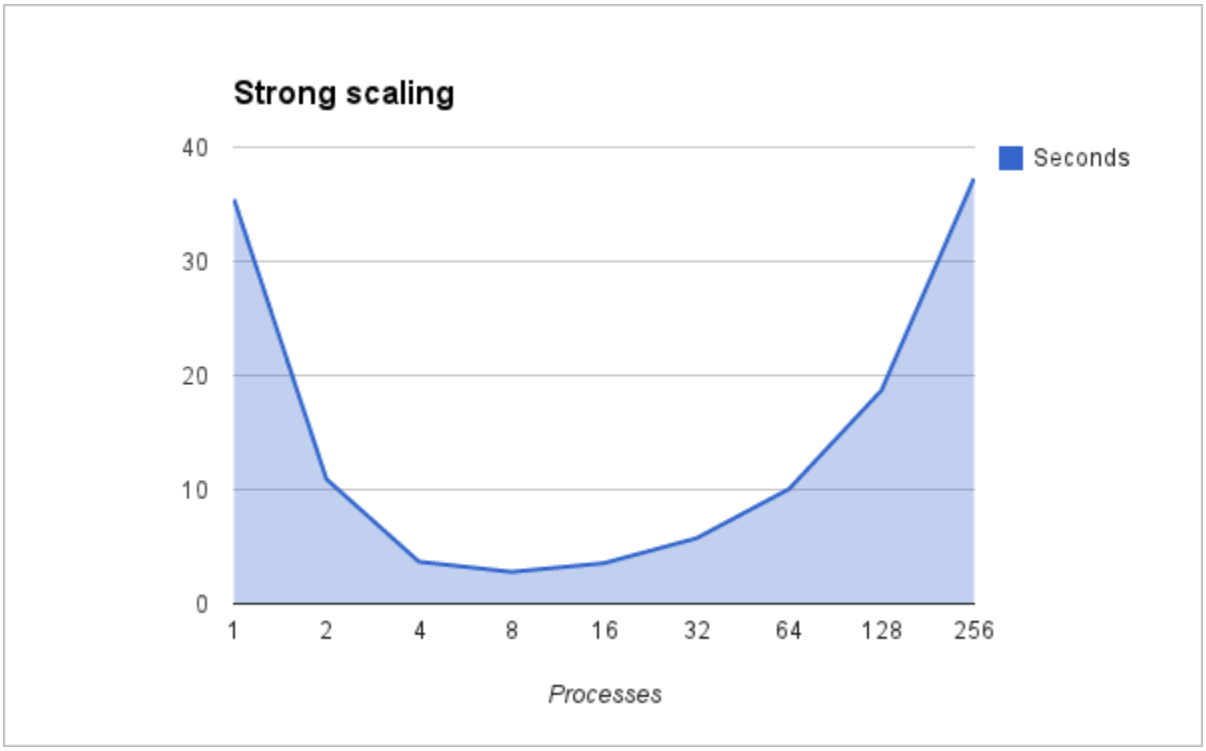
The reason for this lack of scaling is partly because of the overhead of transferring ghost particles. We did runs with disabled communication between nodes (with increasing particle count), which gave superlinear scalability for each step up to 16 processes. It indeed outputs a wrong simulation, but we are easily able to identify a bottleneck and make the overall performance peak with more processes. The following figure clearly shows a performance boost with disabled communication.



Strong scaling

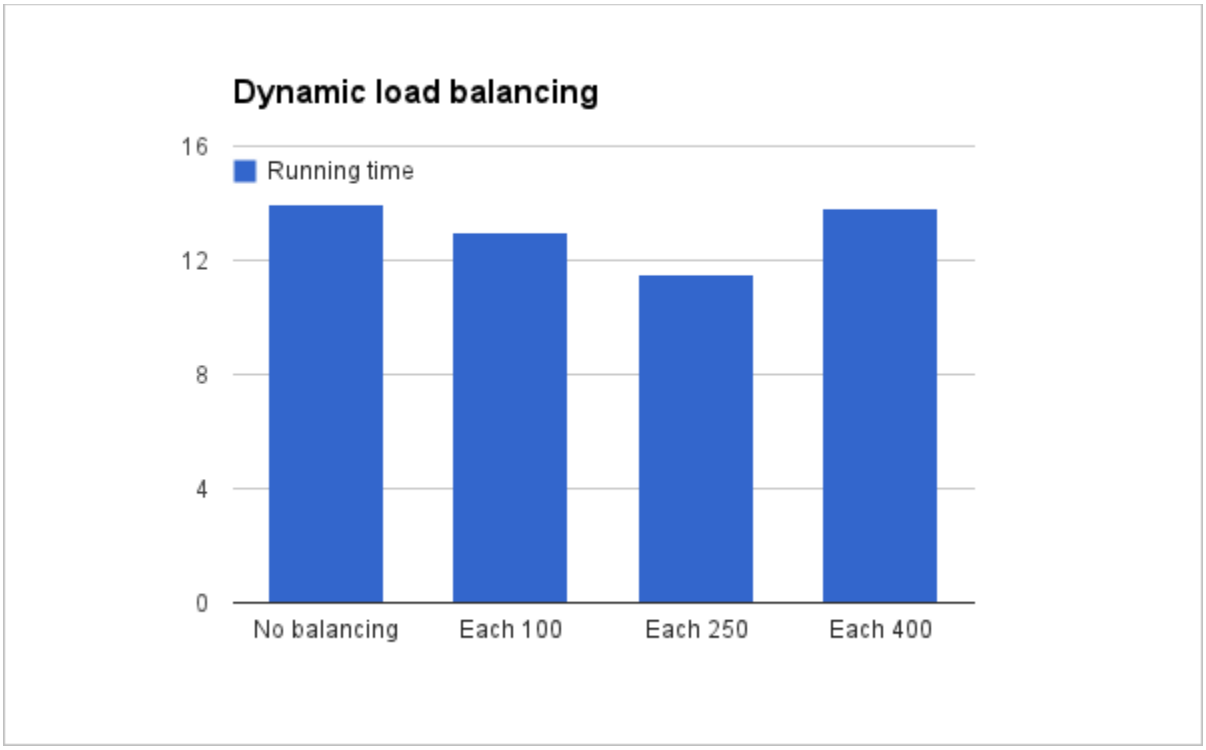
We measured strong scaling by increasing the number of processes on the same problem. The 1-process simulation took around 35 seconds and it scales rather good using up to 4 processes. With more cores, the message passing overhead impairs the running time; the 8 process version runs just slightly faster than with 4 processes. Also, the frequency of load balances will affect performance and must be tuned for the specific simulation.

As already discovered, a lot of messages constricts the running time of our program. To get even more performance, hybrid implementations with e.g. OpenMP should be considered. It introduces parallelism without the overhead of passing messages around all the time. Each MPI process could then distribute its load on multiple cores to achieve better performance.



Our results here strongly connects with the observed results from the previous step, when discussing weak scaling. Our algorithm does not perform very well when message passing is used on many nodes because the overhead of passing messages becomes too big.

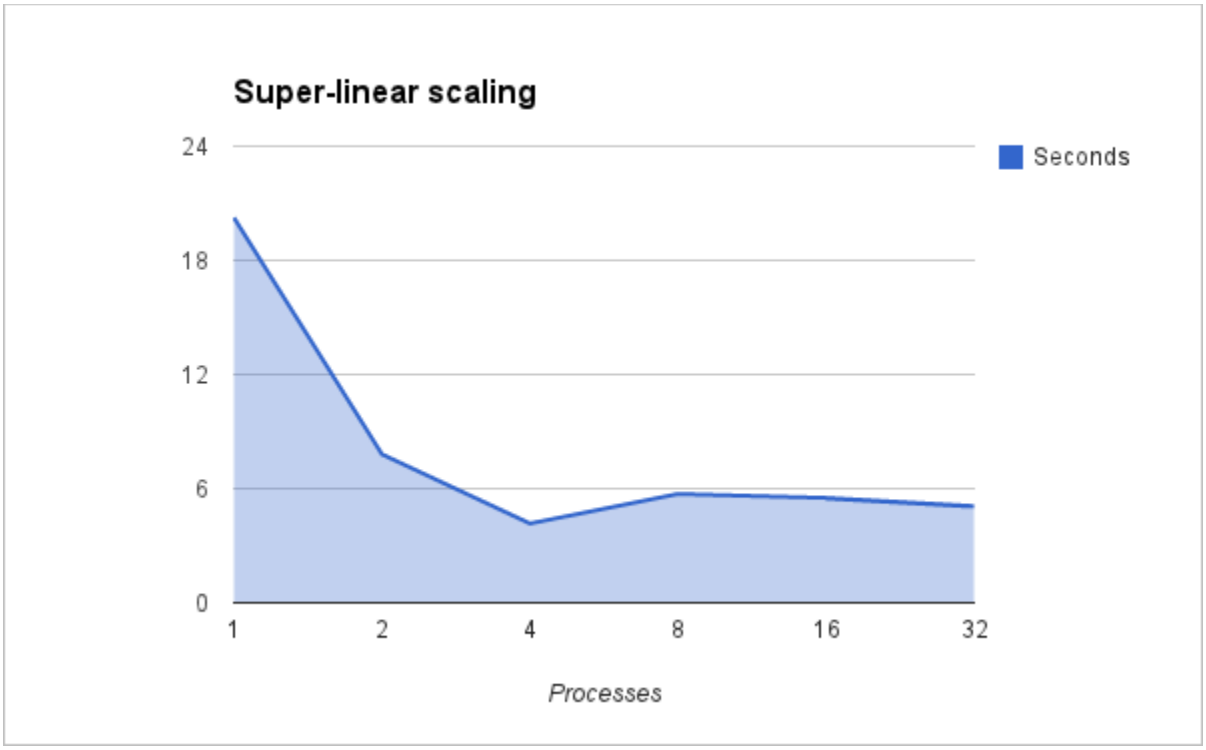
Dynamic load balancing



We constructed an example using 5000 particles and 1000 iterations, and ran it using 4 different configuration for dynamic load balancing. Load balancing every 250 iteration (load balancing on iteration number 250, 500 and 750) was the best configuration for this example. This example shows how hard it is to get good performance improvements using dynamic load balancing. Doing load balancing is a costly operation because every node has to send it's particle to the root node, and constitutes a second bottleneck of our implementation. The root node needs to do the load balance algorithm, which in our case, is an iterative algorithm where average and worst run-time differs a lot depending on the particle positions.

A natural step further would be to make the load balancing algorithm distributed so reduce the extra load for the root node. The load balancing should also be initialized only when needed, by for instance using a clever heuristic to make the decision.

Super-linear speedup



We did several performance testing on Bjorn's laptop using a constant problem size, but with increasingly number of processes, from 1 to 32. Going from 1 to 2 processes reduced the running time by roughly ⅔, which can be credited to the superlinear scaling effect we discussed. Increasing the number of processes to 4 also yielded a smaller running time, also due to the superlinear scaling. The processor is a dual core, but it has Hyper Threading support, which might give a solid gain (somewhere between 1-40% in some benchmarks [8]). The reduction in running time is 47% from 2 to 4, so Hyper Threading can't explain the speed-up alone.

The performance stabilizes when increasing the number of processes even more, probably due to the overhead with increasing neighbor communications.

Further improvements for MPI implementation

Our main focus for the MPI implementation was to get the code correct, easy to debug and fast up and running. Applying optimizations had lower priority, and we planned to implements some obvious optimization if we had time before the performance benchmarking. This was not the case, and the result was a code that did not scale as well as we had hoped on. Here is a list of optimizations we could implement (some more realistic than other):

- Use a better data structure for storing the particles (reduce the cost of searching for nearby particles, insertion and removal). Relevant data structures are chaining mesh and quad tree.
- Change the code to do all the ghost particle transfers in parallel. The current code is communicating with the neighbors in a blocking fashion. The current implementation

forces every node to send and receive ghost particles from the node above before communicating with the node below (except for the topmost node of course). This forced serial way of communication is a serious bottleneck for simulations with high number of nodes.

- Use arrays where number of elements are bounded above or constant. This will increase chances for loops being vectorized.
- Use OpenMP for running the physic calculation in parallel inside a process.
- Reduce the branching when checking which neighbor to move particle / send ghost particle to. This could have been done using some kind of a smart constant lookup structure like a hashmap.
- Implement a distributed partition algorithm for load balancing, and for deciding when it's wise to do load balancing.
- Implement the ORB partitioning algorithm as first planned.
- Implement a heuristic to decide which timestep to use for each iteration.

CUDA vs. MPI

Our results shows that for some simulation meshes, the CUDA implementation scales amazingly well as the particle count increases. The MPI implementation will also give a performance boost with increasing number of cores, but it drops as the number of messages increases.

In CUDA, the optimization centers around maximizing grid occupancy, effectively using the small amount of shared memory by coalescing the writes from global memory to shared memory, while finding the optimal thread size such that the warp scheduler can switch contexts effectively to hide the arithmetic work being done. Any communication between thread blocks is done through shared memory between threads, and global memory between blocks. There is no way to directly communicate between threads. This makes the main bottlenecks in CUDA generally related to memory usage efficiency.

In MPI, message passing and thread communication are the main bottlenecks. This overhead is greater than of CUDA because communication is done further away from the hardware. CUDA outperforms pure MPI in this particular problem simply due to it's massive number of parallel threads, but hybrid solutions with e.g. OpenMP could justify the difference.

Correctness

We tested correctness of the implementation by checking of both of them gave the same results for a given simulation. Our test simulation was done using 10 particles with hard coded initial positions. All physics constants were of course equal. The particle positions were compared for iteration 1 and 10, using average and maximum difference for x and y values as measurement. The average position difference was rather large, 1.94% for first iteration and 3.81% the tenth iteration. Maximum difference was 4.91% for iteration 1 and 22.11% for iteration 10.

We had code reviews for both simulations, and we found no difference in physics calculations that could introduce such a large error. We are also quite confident that every particle is interacting with each other, as after hours of scanning each other's code, and carefully constructing simulations that might expose problems, we simply could not figure it out in time. We believe our simulations do N by N interactions fairly, and are currently uncertain as to why these interactions could differ. The only thing we can think of is hardware differences in the ALU, but it seems like a longshot.

Conclusion

During this project, we made a number of observations and gained much experience with both MPI and CUDA. Working with the two, and attempting algorithms that worked well for one but not the other, taught us a great deal about the limitations of both.

Our performance analysis shows that the MPI implementation suffers from two distinct problems, which both is caused by the overhead of passing messages around. CUDA does much better, but has some constraints on automatically scaling to size. CUDA code has to be designed for a specific range of particles in mind, as the shared memory and block/thread sizes need to correspond and be set at compile time. The main overhead in CUDA is writes from global to shared memory, and poor selection of thread blocks for warp scheduling.

With respect to particle simulations, it has come clear to us that particle simulations are parallelized most efficiently when approximate particle distributions throughout the simulation time is known. With this prior knowledge, a number of implementation choices hereby discussed can be exploited for best parallel performance.

A number of improvements can be done with our implementations to speed it up further. For the OpenMPI implementation, adequate data structures for particle organization can improve the asymptotic run-time of the algorithm, which obviously can help a lot. Also, less overhead data types could be used to maximize loop vectorization. OpenMP is a different approach to speed up our loops which avoids explicit message passing, so a hybrid solution can be much more scalable.

Also, making the particle simulation algorithm more dynamic by computing time step and load balancing intervals on-the-fly can improve performance for large simulations. For many applications, frequent load balancing is only important in early stages of the simulation (i.e. before they all spread out). Doing load balancing with exponential backoff intervals (e.g. 2nd, 4th, 8th, 16th iterations and so on) could improve running time in many cases. This also requires prior knowledge on the behaviour of the simulation to be work well; there is still no free lunch.

Appendix A: System information

Dirac

GPU: C2050 Fermi, Mem Bandwidth 144 GB/s, 768kB L2 Cache , 14 SM, 32 SP per SM, Gflops peak (DP/SP) 515/103, 262 GB global memory, 48kB/16kB L1/Shared memory config
Compiler: NVCC, CUDA 4.2

Bang

System: Bang HPC Cluster (35x Dell PowerEdge 1950 compute nodes)
CPU: 2x Intel Xeon E5345 @ 2.33Ghz, 8MB L2 Cache [7]
Compiler: GCC 4.4.6, OpenMPI 1.4.3

Bjorn's laptop

System: Apple Macbook Pro 13" Early 2011
CPU: Intel Core i7 2620M @ 2.7Ghz, 4MB L3 Cache [8]
Compiler: Apple clang version 4.1, OpenMPI 1.6.3

Appendix B: Benchmarks

CUDA

All the times were done with 1000 time step iterations. All tests were run on carver under interactive queues.

n = 1024, 1000 iter				
GridDim, BlockDim	**Min time (ms)**	**max time (ms)**	**Avg time / iter (ms)**	**Total Runtime (ms)**
1, 1024	0.008352	0.338048	0.0100029	10.0029
2, 512	0.954912	1.657696	1.082108	1082.108
4, 256	0.80208	1.33648	0.905808	905.808
8, 128	0.785728	1.274144	0.8787	878.7
16, 64	0.780096	1.312992	0.881956	881.956
32, 32	0.76832	1.284224	0.869429	869.429
64, 16	0.578464	1.335136	0.669535	669.535
128, 8	0.873024	2.72832	0.98361	983.61
256, 4	1.275808	4.251744	1.417772	1417.772
512, 2	2.538208	6.337728	2.659323	2659.323
1024, 1	7.231488	12.291808	7.372661	7372.661

n = 2048, 1000 iter				
GridDim, BlockDim	**Min time (ms)**	**Max time (ms)**	**Avg time / iter (ms)**	**Total Runtime (ms)**
1, 2048	0.012448	0.094112	0.013327	13.327
2, 1024	0.00816	0.089376	0.009585	9.585
4, 512	1.91136	3.276608	2.095825	2095.825

8, 256	1.60752	2.639744	1.754477	1754.477
16, 128	1.611552	2.661536	1.760015	1760.015
32, 64	1.585376	2.5928	1.728	1728
64, 32	1.54336	2.604704	1.702105	1702.105
128, 16	2.023584	5.239648	2.324239	2324.239
256, 8	2.662912	8.176064	2.884088	2884.088
512, 4	4.30416	14.242432	4.650045	4650.045
1024, 2	10.163584	24.330751	10.51463	10514.63
2048, 1	27.887615	47.761986	28.295919	28295.919

n = 4096, 1000 iter				
GridDim, BlockDim	**Min time (ms)**	**Max time (ms)**	**Avg time / iter (ms)**	**Total Runtime (ms)**
2, 2048	0.012768	0.074464	0.013608	13.608
4, 1024	0.008704	0.075968	0.00943	9.43
8, 512	1.139456	6.577888	1.253302	1253.302
16, 256	1.155872	6.44192	1.264427	1264.427
32, 128	1.0544	5.749696	1.155263	1155.263
64, 64	1.024544	5.531232	1.120392	1120.392
128, 32	1.96496	9.20544	2.086978	2086.978
256, 16	3.201568	13.641184	3.389544	3389.544
512, 8	6.270464	22.511871	6.532528	6532.528
1024, 4	15.08592	45.038017	15.529591	15529.591
2048, 2	38.407806	77.938431	39.107784	39107.784
4096, 1	113.06582	160.891205	113.79528	113795.28

n = 8192, 1000 iter				
GridDim, BlockDim	**Min time (ms)**	**Max time (ms)**	**Avg time / iter (ms)**	**Total Runtime (ms)**
16, 512	4.43232	22.266624	4.733953	4733.953
32, 256	3.178944	17.543873	3.440734	3440.734
64, 128	2.751104	15.773184	3.007915	3007.915
128, 64	4.01984	19.201536	4.28529	4285.29
256, 32	5.88048	24.336224	6.192309	6192.309
512, 16	10.731584	38.685696	11.206341	11206.341
1024, 8	24.983648	80.653023	25.9042	25904.2
2048, 4	57.310047	158.260391	58.781544	58781.544
4096, 2	150.20694	278.533783	153.332352	153332.352
8192, 1	449.230438	613.529053	456.02182	456021.82

n = 16384, 1000 iter				
GridDim, BlockDim	**Min time (ms)**	**Max time (ms)**	**Avg time / iter (ms)**	**Total Runtime (ms)**
32, 512	13.16896	60.450527	13.921448	13921.448
64, 256	10.563616	55.001408	11.317458	11317.458
128, 128	10.773312	53.855839	11.520149	11520.149
256, 64	12.381216	55.558018	13.086437	13086.437
512, 32	19.724607	76.316605	20.670088	20670.088
1024, 16	42.793121	138.362183	44.38744	44387.44
2048, 8	94.84877	268.829376	97.787277	97787.277
4096, 4	223.287903	553.816284	230.708984	230708.984
8192, 8	597.983215	1097.388916	647.318359	647318.359
16384, 1	1802.422607	2475.834717	1901.21814	1901218.14

n = 32768, 1000 iter				
GridDim, BlockDim	**Min time (ms)**	**Max time (ms)**	**Avg time / iter (ms)**	**Total Runtime (ms)**
64, 512	43.686783	186.677597	45.783257	45783.257
128, 256	42.037025	187.646942	44.115234	44115.234
256, 128	41.543839	187.346588	43.638573	43638.573
512, 64	43.688545	180.465561	45.798763	45798.763
1024, 32	78.618813	268.92099	81.697433	81697.433
2048, 16	162.439301	509.660675	168.146957	168146.957
4096, 8	369.83194	1114.084229	388.291656	388291.656
8192, 4	891.930115	2165.014404	1053.129272	1053129.272

MPI

All configurations were run at least 3 times, with minimum timings listed below. Some of test run shad large fluctuation, and were sometimes ran up to 10 times. All test were run on Bang, except the superlinear scaling tests, which were run on Bjorn's laptop.

Strong scaling

5000 particles, 300 iterations, load balancing each 50 iteration.

Number of processes	Running time
1	35.4372 sec
2	10.9213 sec
4	3.6837 sec
8	2.7934 sec
16	3.5576 sec
32	5.7401 sec
64	10.0490 sec

128	18.6982 sec
256	37.2463 sec

Weak scaling, increasing number of particles

100 iterations, load balancing each 50 iteration

Number of processes	Number of particles	Running time
1	5000	13.5650 sec
2	7071	8.9801 sec
4	10000	7.1136 sec
8	14142	8.0278 sec
16	20000	13.5945 sec
32	28284	23.1582 sec
64	40000	37.8930 sec
128	56569	72.8621 sec
256	80000	118.7794 sec

Weak scaling, increasing number of particles, no neighbor communication

100 iterations, load balancing each 50 iteration

Number of processes	Number of particles	Running time
1	5000	13.5883 sec
2	7071	7.8716 sec
4	10000	7.8716 sec
8	14142	4.7477 sec
16	20000	4.2629 sec
32	28284	4.8181 sec
64	40000	9.1164 sec
128	56569	18.3848 sec

Weak scaling, increasing number of iterations

5000 particles, timestep inverse proportional to number of iterations

Number of processes	Number of iteration	Load balancing interval	Running time
1	100	50	13.5761 sec
2	200	100	10.5832 sec
4	400	200	9.9383 sec
8	800	400	14.7556 sec
16	1600	800	21.4666 sec
32	3200	1600	64.4109 sec
64	6400	3200	59.0942 sec
128	12800	6400	148.3382 sec
256	25600	12800	130.9703 sec

Super-linear scaling

5000 particles, 300 iterations (ran on Bjorn's laptop)

Number of processes	Running time
1	20.2400 sec
2	7.8051 sec
4	4.1665 sec
8	5.7198 sec
16	5.5130 sec
32	5.0795 sec

Dynamic load balancing

5000 particles, 1000 iteration

Balancing	**Running time**
No balancing	13.9525 sec
Each 100	12.9762 sec
Each 250	11.4840 sec
Each 400	13.8590 sec

Appendix C: Correctness test

10 iterations, 10 particles. Values calculated on x and y coordinates between each particle for CUDA vs MPI.

	Iteration 1	Iteration 10
Position average difference	1.94%	3.81%
Position max difference	4.91%	22.11%

References

[1]: http://en.wikipedia.org/wiki/N-body_simulation
[2]: http://en.wikipedia.org/wiki/N-body_problem#General_considerations:_solving_the_n-body_problem
[3]: http://www.cs.berkeley.edu/~ballard/cs267.sp11/hw2/
[4]: http://ww2.cs.mu.oz.au/498/notes/node52.html
[5]: http://bmi.osu.edu/hpc/slides/Saule10-NCST.pdf
[6]: http://www.dps.uibk.ac.at/~cosenza/teaching/gpu/nv_particles.pdf
[7]: http://ark.intel.com/products/28032
[8]: http://ark.intel.com/products/52231
[9]: http://www.anandtech.com/show/3871/the-sandy-bridge-preview-three-wins-in-a-row/9
[10]: http://http.developer.nvidia.com/GPUGems3/gpugems3_ch31.html

Team Evaluation

(1) Team members

A: Matteo Mannino

B: Terje Runde

C: Bjørn Christian Seime

(2) Time estimation

	A	B	C
meetings	1 + 3	3 + 3	3 + 3
coding	10+30	12 + 20	13 + 30
writeup	2+15	0 + 28	0 + 24
planning (alone)	4+4	2 + 0	1 + 1
total	69	68	75

(3)

(a) As described in the progress report, we divided the work in two parts: Bjørn and Terje worked with the OpenMPI implementation while Matteo did the CUDA implementation. This was the easiest solution for us because of colliding schedules.

(b) We completed two working implementations of a particle simulation between two radically different paradigms. We are quite happy with what we've done together as a team, and this project taught us a lot about what we got out of the course. We feel our knowledge in parallel processing, and computer science in general, has improved, and furthermore we know our shortcomings and areas where our knowledge could improve.

c) We are all about equal in programming skill. As stated before, Bjørn and Terje are classic CS students (took architecture in a course at some point), and were able to help Matteo understand those aspects much better than before. Matteo did the CUDA part alone, but also generally skilled in discussing algorithms.

(d) The lessons learned are fairly outlined in the report. One thing: when the professor says something will be difficult, we should listen.

(e) One milestone we failed was attending the potluck party the Friday prior to the project delivery. We were simply unable to find time. In the future, no turkey will be uneaten, no pastry unravaged.