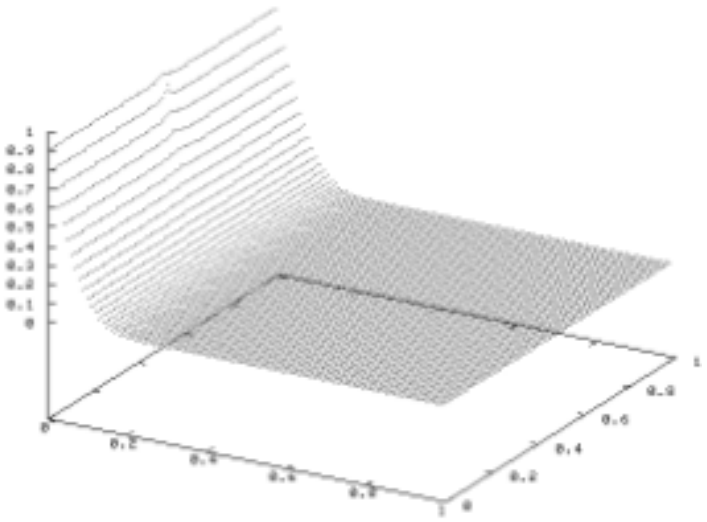
Emerging Technologies Project: Cluster Technologies

PIC codes: Eulerian data partitioning

by
Jan Christian Meyer

February 20th 2004



1 appendix: program source code

Abstract The purpose of the Emerging Technologies project to which this work belongs, is to compare the applicability of cluster computers to various problems to that of conventional supercomputers. This article details the rewriting of an existing Particle-In-Cell electrostatic simulation code targeted at SMP computers to a message-passing approach, and a comparison of the results obtained when running the resulting code both on a PC cluster and an SGI Origin supercomputer.

1 The Particle-In-Cell method at a glance

The Particle-In-Cell (hencefoth "PIC") method is a method for reducing the problem of simulating independent particles in space to solving equations on a regular and static grid partitioning of space. The method works by dividing the charge of each particle up into components for each of the grid points closest to the particle (its "cell"). The magnitude of each part is weighted by the particle's displacement from each corner of the cell, as a fraction of the cell size. The behaviour of the charge distributed on the grid is then solved as partial differential equations, and the result is interpolated at the individual particle's position by a weighted sum over its cell corners. Using a finite difference in the time domain the particle positions are then updated by approximation using the differential of the electrostatic potential at their position (i.e. the electric field strength). Although interesting, the method is already well described and not in itself of essential importance to the results presented below. A more precise description of the simulation method and its specific equations can be found in Elster's article [1] upon which this work is based, or it can (with some effort) be gleaned from the attached source code.

2 The Eulerian data partitioning

In order to create a parallel solution to any problem, the problem must be subdivided into units which are more or less independently solvable. In the case of PIC simulations, a natural approach is to divide the grid up between processes, and try to minimise the effect each subgrid has on the others. During this project a Lagrangian data partitioning has already been examined for the PIC problem; this solution terminated each time step with a global communication phase involving all subgrids. While this method easily achieves perfect computation load balance by continually redistributing the subgrid responsibilities, it was shown to scale poorly as the time involved in broadcast communication grows with the number of processes involved. This and related results are detailed in [2].

To sidestep the undesirable effect a global communication phase has on scaling, the Eulerian partitioning is a static subdivision of the grid, which sacrifices load balancing for the advantage of only requiring communication between neighboring subgrids. This partitioning is expected to scale significantly better, as the communication per process should be proportional to grid size, but inversely proportional to the number of processes involved in the computation.

The chief purpose of the following experiments is to verify the inverse proportionality by testing a static scenario on an increasing number of processors.

An effect of the Eulerian approach is that the equation solver for the simulation must be adapted to the limited communciation facilities. In this project, this meant that the PDE solver code was based on the method of successive over-relaxation (SOR). It was chosen because of the local nature of the point template it employs in each successive iteration, meaning that most computations would be process local and involve only neighbor processes along the boundaries of the process subgrid. A disadvantage of this strategy is that it requires some fixed condition in the simulation space in order to make sure that the iterations will converge upon a solution, - sadly this disallowed a few interesting tests. It is not a limitation of the chosen data partitioning, however; more a reflection of the time limitations under which the project was carried out, and that the SOR solver was prioritised as the first experiment.

3 The computers

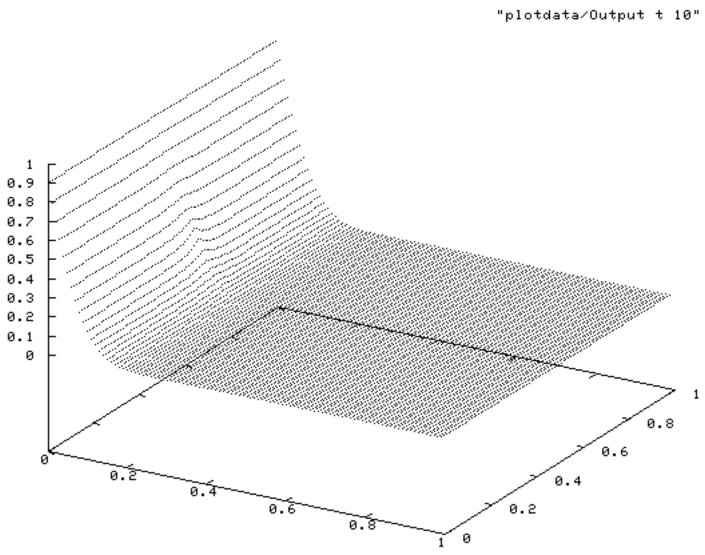
The computers which were used for testing the code were

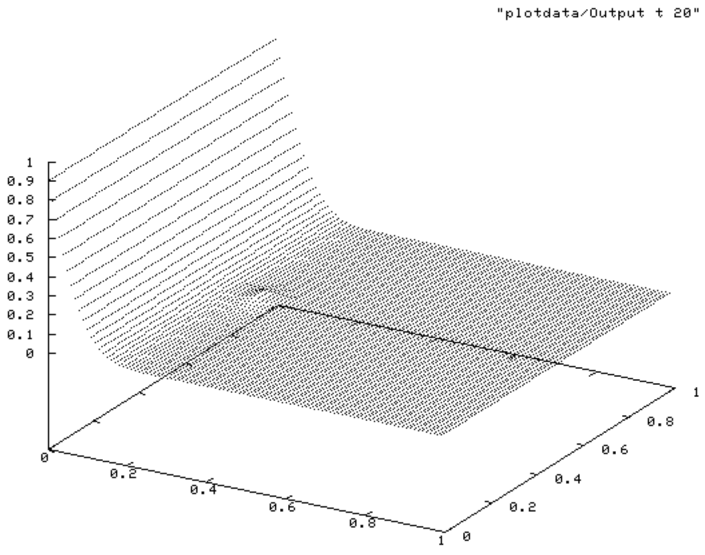
- **ClustIS**, the computing cluster of the Intelligent Systems group at the computer science dept. of Norwegian University of Science and Technology (NTNU), Trondheim, Norway
- **Gridur**, an SGI Origin belonging to the supercomputing center at said university.

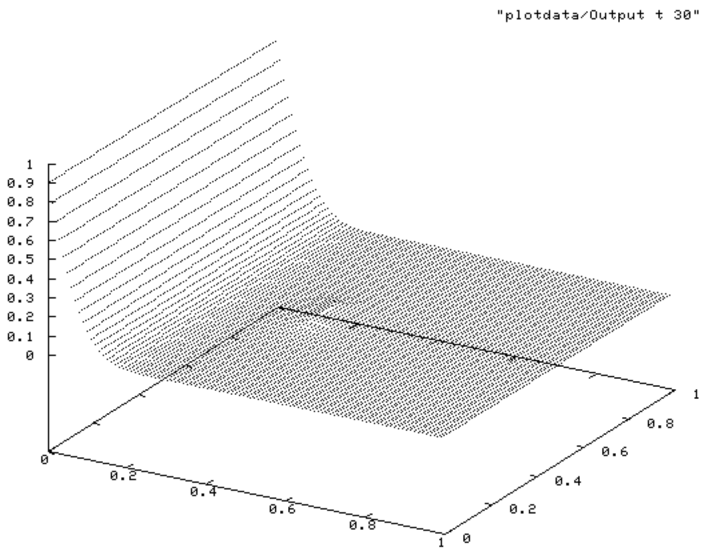
4 The test

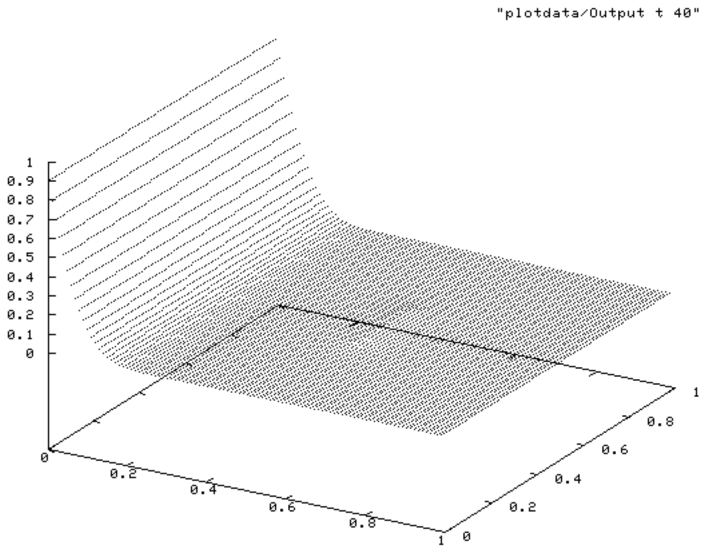
In order to test the code on both platforms to be compared, a simple system was devised in which a voltage difference is imposed on the unit square, and 9 charged particles are started close to each other at the charged end of the area. As the simulation proceeds, the charges are repelled both from the charge and one another, leading them to move across the area and scatter out towards the opposite side. This test contains the static conditions necessary to guarantee convergence (the imposed voltage difference), and also verifies that the particle physics of the code behave as expected.

After tuning the initial placements, charge quantities and time steps of the test case to create a satisfactory simulation, a 65-frame animation plotting the physical area of the simulation against the electrostatic potential at the grid points was produced. Three key frames of this animation are reproduced below; as can be seen, the particles manifest themselves as "waves" in the potential graph, rippling across the area and scattering as predicted.


"plotdata/Output t 10"
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
0.2
0.4
0.6
0.8
1
0
0.2
0.4
0.6
0.8
1


"plotdata/Output t 20"
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
0.2
0.4
0.6
0.8
1
0
0.2
0.4
0.6
0.8
1


"plotdata/Output t 30"
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
0.2
0.4
0.6
0.8
1
0
0.2
0.4
0.6
0.8
1


"plotdata/Output t 40"
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
0.2
0.4
0.6
0.8
1
0
0.2
0.4
0.6
0.8
1

5 Results

The tests performed show how computation time is affected when altering the number of nodes. Usage restrictions meant that fewer tests were performed for Gridur, but tests of 2 through 8 node computations already tend toward the desired result.

5.1 Scaling results

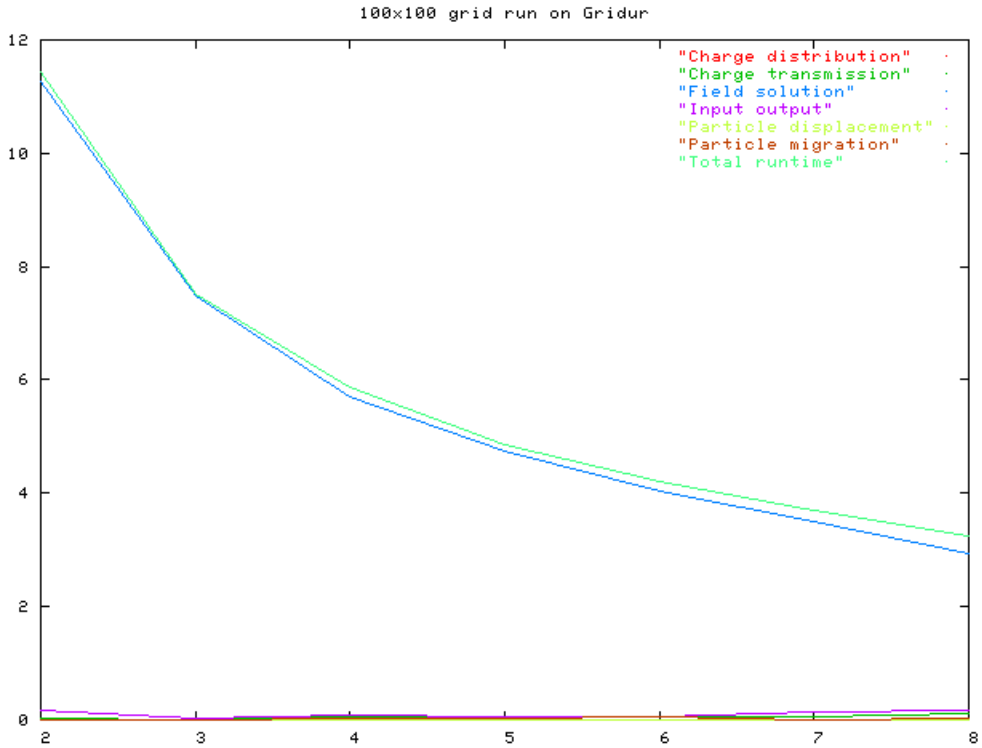


Fig.1: Timings for 100x100 point test on Gridur

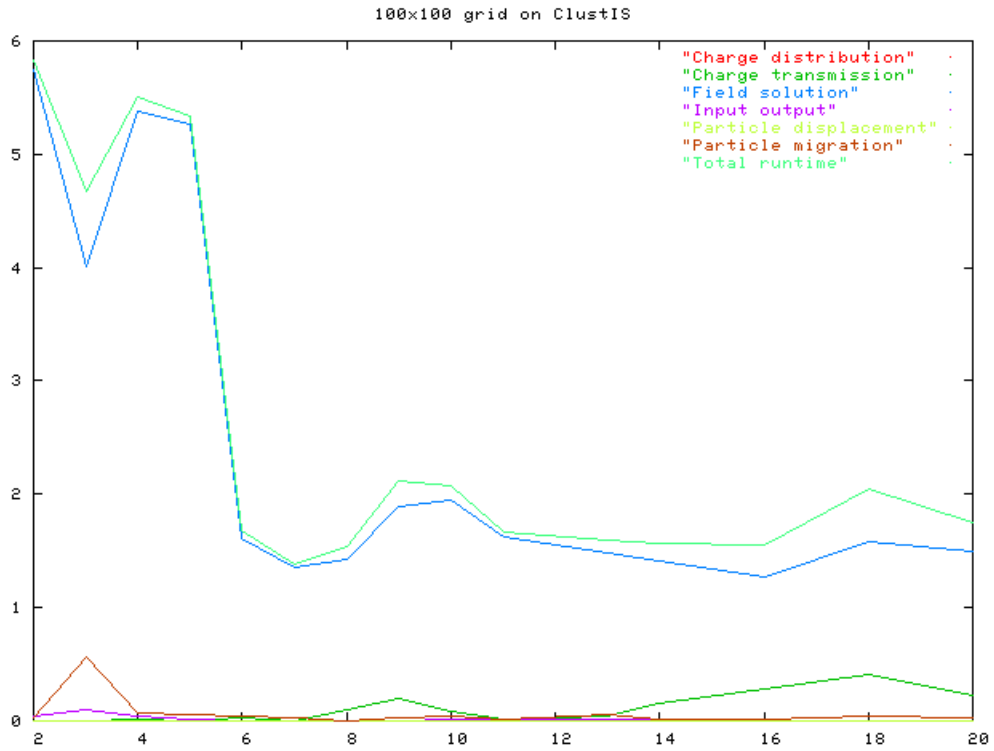


Fig.2: Timings for 100x100 point test on ClustIS

As can be seen in figures 1 and 2, the computation time falls in a roughly hyperbolic curve as the number of processors grows linearly. This is evident on both computer platforms; comparing this to the cluster timings of the Lagrangian solution in [2], there is benefit in avoiding the global summary step which would grow to dominate computation time. However, as can be seen in the measurements from ClustIS; in spite of this favorable scaling property, computation time is still bounded from below such that there is little purpose in expanding a cluster beyond a certain number of nodes for a given problem size. This makes sense - a computation will always take some minimum of time. The benefit of the Eulerian data partitioning, then, is that the machine can be scaled towards this limit as desired, without being bogged down in a communication cost which *rises* with the number of nodes after a certain point.

5.2 Timing comparison

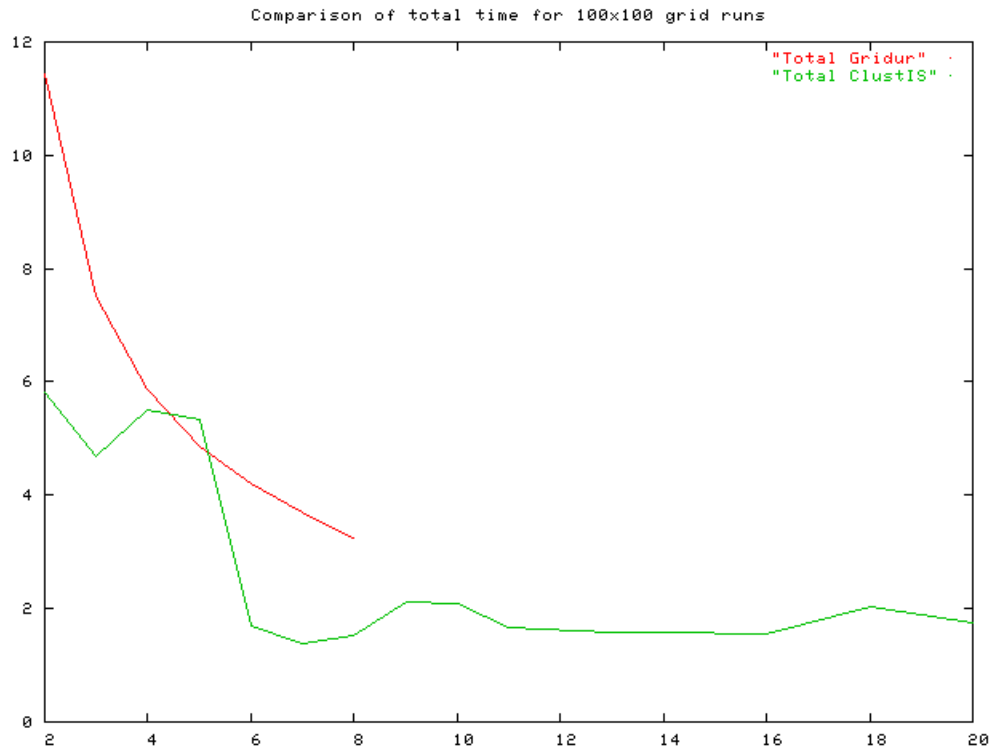


Fig.3: Comparison of total runtimes

An unforeseen result which became evident in the measurements reproduced above is that for the given code, the cluster outperforms the conventional supercomputer by approximately a factor of 2. It is not known whether this is a characteristic of the code being written from scratch with a message-passing architecture in mind, or whether it is a consequence of the low priority given to optimising absolute runtime. In any case, the plot of the total runtimes in fig.3 supports the claim that a cluster computer is not at significant disadvantage compared to a custom built supercomputer, especially if one factors in that it will often be as much as an order of magnitude cheaper to build and install.

6 A road map to the developed code

6.1 The existing code

As these efforts were based on an already finished doctoral dissertation [1] and its corresponding program code, an initial evaluation was made to determine how much of the existing material was ready to be applied within a different context given by the topic of this study. This led to the observation that while the existing code is efficient and highly optimised, this leads to a significant reduction in readability. It also makes it tricky to modify, due to tight integration between the conceptual steps of its algorithm; the equation solver part directly codepended on the data partitioning and communications, etc. An initial attempt to create code in the spirit of the original was made, in the hope of extracting sections of it into modules at a later stage. This attempt had to be abandoned, as the increase in code complexity which arises from trying to generalise the solution made for testing and correction which was far too time-consuming for the scope of the project. Only small fragments of this were salvaged (the vector structures and their functions, detailed below), and a complete rewrite was decided upon, using the original code as a "recipe book" and reference for the general algorithm and specific equations.

6.2 Use of explicit data structures

The most significant design decision of the new code was to place all significant data in explicitly declared structures, as opposed to structuring it by using arrays of language primitives.

The main disadvantage of this is that it implies a certain extra volume of dynamic memory allocation code, and the runtime overhead that comes with it. Another, more subtle disadvantage is that dynamic allocation means that memory access patterns can be difficult or impossible to predict, which can ruin a host of optimising strategies for both compilers and processors.

An advantage of structures is that they allow the program to be split into distinctly defined sections treating each datatype, and these can be tested and used in various contexts more or less independently. In addition to this modularity, it permits big programs to be created with less effort, as the details of one part can be hidden from the development of another.

This final point became the deciding factor when opting for explicit declarations of data structures. Observing the effort involved in the development of the code to analyse properties of the Lagrangian data partitioning, it appeared that a program catering to 3-dimensional systems written in the same style would be

complex to the point of becoming incomprehensible. Since execution wall-time is not relevant when examining scaleability, priority was given to lower the risk of fruitless development efforts instead of trying for a time-optimal code.

6.3 Outline of the data structures

Simple values The most fundamental structures in the code are vectors of numbers as defined by the struct declarations **int_vector** and **double_vector**, which represent vectors of integers and double floats, respectively. The value of this abstraction is that the dimensionality of the vectors is contained in the structure along with the data. Code which interfaces to these structures without assumptions about their contents will accept data of all dimensionalities uniformly, which goes a long way towards facilitating the desired 3D functionality.

Global values The term "global" can be slightly ambiguous in the context of parallel programming - there are values which are global across all nodes during execution, and there is the more conventional understanding of a global value as being a variable which is accessible from all subroutines of a program executing on one node. In this program, the global-everywhere sense is referred to as "simulation parameters" and the global-on-a-node sense as "process parameters", and their respective structures are named with this in mind.

The simulation parameters are broadcast once at the beginning of execution, and are constant for the duration of the program. They include such data as the size of the physical system to be simulated, number of particles, number of computational nodes, etc.

The process parameters contain some fixed node-specific data, but also include mutable buffers and tables for housekeeping on the individual node. They include data about the node's portion of the simulated system, particle positions, communication buffers, etc.

Particles The variable properties of a particle are its position and velocity; these are represented by vectors of double values. The constant properties of particles (mass, charge, drag, etc.) are stored with the simulation parameters, to avoid redundancy. Since particles come and go across processes, the set of particles needs to be stored in a fashion which permits adding and removing elements - this is accomplished by maintaining a linked list of each process' local particle set, using the structure **particle_set**.

The grid The simulation grid is a discretisation of the physical system into an integer coordinate system. It is represented by arrays of **grid_point** structures by each computational node, corresponding to that node's part of the simulated system. The points are stored with their location in terms of the entire simulation grid, the node's own portion of it (henceforth referred to as "subgrid"), and

the point they correspond to in the physical system being simulated. Also stored are charge, resulting electrostatic potential and electric field strength which are to be computed.

There are also a number of structures related to communication of data; these are specific enough to their application that they are discussed along with the descriptions of the functions which create, transmit and read them.

6.4 Code construct for n-dimensional loops

A construct which is used extensively in the source code is one which creates a loop iterating on all permutations of a vector. While this is straightforward for a single case with a fixed number of dimensions (N nested for-loops where N is the dimensionality of the vector), it becomes a non-trivial undertaking when the vector may be of any size. The manner in which this is accomplished is that a lower and an upper boundary vector is established, containing the minimal and maximal allowed values along each axis respectively. An iteration vector is then initialised with the minimal values, and incremented along the first dimension where the maximum is greater than the minimum. When the end is reached, the next allowable dimension is incremented, and the original reset. This proceeds until the iterating vector equals the maximum vector. In the code this appears as nested while loops, as examplified below:

```
while( ! compare_int_vectors( maximum, iterator ) ) {
while( 1 ) {
Try incrementing each dimension, break at first permissible
}
Do something with the updated iteration vector
}
```

Any reader intending to read the provided source code is advised to get aquainted with this method, as it often appears. Note that the increment occurs at the beginning of the loop, so the initialisation data for the iterating vector is actually $minimum - 1$ for the first permissible dimension.

6.5 Overview of program execution

A brief outline of how the program proceeds is given below; each step is examined in greater detail later.

1. Process 0 reads simulation global data from file, and broadcasts it to allprocesses.

2. Each process independently computes the relation between the grid, all subgrids and the physical domain, and initialises storage space for its own subgrid.

3. Process 0 reads initial particle data from file, and sends each particle to itsrespective governing process based on its position.

4. Simulation loop begins: Each process opens files for particle data and timings for this loop iteration.

5. Each process distributes the charge of each of its particles onto the grid points forming the enclosing cell.

6. Each process communicates the charge distributed along its subgrid boundaries to the neighboring processes.

7. Each process runs the loop of the iterative solver; this involves repeatedly computing a successively improving approximation of the electrostatic potential generated by the distributed charge, communicating the approximation along subgrid boundaries to the neighboring processes, and starting over with the received values *from* the neighboring processes.

8. Each process differentiates the resulting electrostatic potential values, obtaining electric field strengths at each grid point.

9. All processes managing particles locate the cells which these are presently in, interpolates the electric field strength at the particles' present position, and updates their position accordingly.

10. Any process noticing a particle which has moved beyond a process boundary sends this particle to the appropriate neighbor process.

11. If simulation is not yet finished, the loop started in point 4 iterates.

12. All allocated resources are freed.

6.6 Step 1: Simulation global data

This step has process 0 reading and broadcasting data which is common to all processes, and the remaining processes receiving and storing them.

Data are read from a filename supplied in the first argument to the invoked program. This is done using the function **parse_parameters**, which begins on page 47 of the appendix. It is simply a number of calls to the library function **scanf**, to fill a copy of the structure **simulation_parameters** with values.

Data are communicated using the function **send_simulation_parameters**, which begins on page 80 of the appendix. This function serialises a **simulation_parameters** structure into an array of bytes using **MPI_Pack**, and broadcasts it using **MPI_BCast**.

The serialised object is received using the function **receive_simulation_parameters**, which begins on page 81 of the appendix. This function simply unpacks the data packed previously, and stores it in a **simulation_parameters** structure.

6.7 Step 2: Determine position on the grid

How the individual process relates to the grid is determined in its entirety by one call to the function **eulerian_section**, which begins on page 26 of the appendix. This function allocates a special array of subgrid structures for convenient lookup of data regarding its own neighbor processes. It then factorises the number of processes to create a global map of subgrids, and assigns each process to its place on this map. This procedure is somewhat elaborate, because it must account for cases where the number of processes along an axis does not evenly divide the number of grid points to be partitioned.

As there is quite a bit of computation involved in determining the subgrid positiions, origins and sizes, this data is all stored away for more convenient lookup when the simulation begins. This is done with the subgrid map lookup table, which is a simple table indexed on subgrid position (among subgrids) and yielding the process rank as a value. Since the subgrid position is a vector, it is linearised into a number to make this lookup more efficient. The linearisation is by the function **linearise_int_vector** given on page 30 of the appendix, and is a straight numeric conversion of the index vector read as a number written in base M, where M is the greatest value to appear along any dimension of the table. E.g., in a 12-process run factorised into $(3, 4)$ subgrids, the process at $(2, 3)$ will get the key $11 = 2 * 4^1 + 3 * 4^0$. Once the lookup table is generated, it is employed in determining the ranks of the processes in the aforementioned neighborhood.

6.8 Step 3: Transmission of initial particle distribution

This step is largely similar to step 1, but is accomplished using the functions **parse_particles**, **group_transmissions**, **transmit_initial_particles** and **receive_initial_particles**, found on pages 49, 77, 78 and 79, respectively. The only significant difference is in the use of **group_transmissions** which uses the subgrid map lookup table to determine target processes for each particle not initially local to process 0. There is also an extra communication phase in which each process determines how many (if any) particles it will receive in the subsequent transmission.

6.9 Step 4: Opening files for output of data and timings

This phase employs the string-generating utility functions found on pages 44 through 46 of the appendix to create file names for ouput particle data and timing data uniquely idetifying process rank and time step for output as the time step progresses.

6.10 Step 5: Distribution of charge

In this step each particle is examined locally, and the corner points which form its enclosing cell are determined. The particle's contributed charge to each corner point charge value is then computed by a weighted division of the particle charge. The charge contribution per cell unit length is precomputed in the global parameters, and is now multiplied by the particle's displacement from the individual corner point along each axis. The result is a division in which the sum of all corner contributions becomes the particle charge. If a contribution belongs to a remotely hosted corner point, it is linked to a transmission buffer in a **charge_contribution** structure to be sent in the following step.

6.11 Step 6: Transmission of charge contributions

In a step conceptually similar to the initial transmission of particles, the functions **transmit_charge_contributions** and **receive_charge_contributions** (pages 74 and 72 of the appendix) are used to pack, send, unpack and store any charge contributions made by particles to grid points which are hosted outside their local grid.

6.12 Step 7: The iterative solver

The solver is at the heart of the program, providing the solution to the Poisson differential equation which governs the forces that displace the particles. Its task is to compute and electric field vector for each grid point, based on the already computed charge level at each point and its neighbors. A few strategies are available for this purpose, most notably the FFT-method and the method of Successive Over-Relaxation (SOR). The original intention during this project was to implement several solvers to see how their performance compares, but sadly time was only sufficient to create an SOR solver which works on 2-dimensional systems.

Do note that the solver is the only part of this program which is not created to be fully n-dimensional. Insofar as an eulerian data partitioning is assumed (such that each node primarily communicates with its neighbors), it can be substituted for more or less any approach which uses the charge and outputs the field in the given format. Particle acceleration, migration and displacement are, to the

best of the author's knowledge, completely independent of the internals of this module.

The solver initially computes an approximation to each grid point's electrostatic potential based on its own and the surrounding potential values, expressed in a 5-point SOR template. The mathematics of this computation are documented in the original paper by Elster. Surrounding values are fetched locally or remotely by the same function; **neighbor_potential** on page 84 of the appendix. These values are all buffered, and set at the end of the iteration.

6.13 Step 8: Computing the field

At the end of the final solver iteration, the field vectors are computed in a similar fashion to the potentials; the equation is different, but the code and communication pattern are largely the same. This step happens within the solver, as the potential values are discarded when the solver returns.

6.14 Step 9: Accelerating the particles

In this step (which also takes place in the final iteration of the solver), the field vectors of enclosing cell corners are interpolated to a weighted sum at each particle's position. This operation is the inverse of that which takes place when charge is split; the method of weighting each corner's partition by the in-cell particle displacement is identical.

6.15 Step 10: Migrating particles

The final step of the algorithm proper entails looping over all particles to see if any have migrated beyond the boundaries of the local grid. If any such particles are found, they are linked to a transmission buffer to be sent to their new host process at the end of the step. This communication is handled by the function **migrate_particles**, found on page 72 of the appendix.

7 Caveats and shortcomings

7.1 Particles at subgrid corners

The code behaves incorrectly in a case covered on page 87 of the appendix, in the function **neighbor_field**. This is the function which calculates the electric field for a particle position, based on the neighboring grid points. The case is that when a particle is in a cell for which each corner belongs to a different subgrid, its governing process will not have received all the relevant field strengths

(particularly, the value from the process diagonally opposite across the cell will be missing). This case was overlooked during development, and discovered too late to be handled correctly. Present behaviour is to print a warning and not accelerate the particle at all - as the rare particles passing through such cells during the tests usually had a velocity already, this minor glitch was not noticeable in the results. However, this case should be considered in particular if a particle is configured with initial position inside such a cell.

7.2 Frequent passing of small messages

The iterative solver uses frequent communication of short messages to approximate the field along subgrid boundaries. While it is not a logical fault of this code, for runs with a great number of nodes this was found to occasionally saturate and crash the communications subsystem of ClustIS.

7.3 Accuracy of double precision floating-point representation

This issue really concerns the gcc c-compiler, but deserves mention here since that compiler is popular for linux-based clusters, and its behaviour gave rise to the longest debugging session of the development. The issue was with representing the number 0.1 in a test, but similar peculiarities may occur for any number which does not have an exact binary floating point representation.

When such a number is used as the denominator in a division, the result may be slightly smaller than the exact solution, e.g. 1/0.1 becomes 0.99999... instead of 10. In the test (which was derived for development purposes only), this fraction was to be chopped to the floor-function integer; as the C standard specifies that this is the default behaviour when casting double to int, this mechanism was trusted to provide an imperfect but consistent result.

The CPUs on the computational nodes of ClustIS work with 80-bit fpu registers; the IEEE double floating point standard specifies only 64 bits of precision. What happened with the fraction 1/0.1 was that when the 80 fpu register bits were chopped to 64 bits in memory, the outcome was the IEEE double float of value 10 which cast to integer without problems. If the cast was performed with the value stored in a register, the 80-bit representation gave the value 9.

This oddity led to errors which were extremely hard to pin down until it was noticed that their occurrence varied with the optimization level of the build; with aggressive optimization, the fraction was precomputed and stored in a 64-bit representation in the object code, without optimization the 80-bit (undesired) chop occurred. On the bug tracking mailing lists of gcc, this behaviour has been

commented to be compliant with the ANSI standard for C, and is not subject to change. What remains from the outlined ordeal are the calls to the library function **floor** on pages 38 and 92 of the appendix (which enforce the use 64-bit double value consistently), and a clear reminder to keep close track of rounding errors when dealing with floating point values.

8 Extensions and further work

A great deal of work can be put into optimising the attached code. This ranges from pure technicalities which can improve speed, e.g. substituting some of the constant variables with preprocessor macros, etc. to altering fundamental behaviour such as how to treat particles in boundary areas.

Beyond sheer speed optimisation, the most natural extension on this work is to expand the SOR solver to be parameterised with respect to the dimensionality of the grid. Also, it would be interesting to perform experiments with a greater range of different solution strategies.

Finally, an expansion which would extend the work into uncharted territories would be to develop a decent load-balancing algorithm for the Eulerian data partitioning; i.e. develop some mechanism by which the subgrids would vary in size attuned to the number of particles to be handled by each processor. A literature search in the initial phase of this work did not yield any approach which would be appliccable in the absence of a process with a "global overview".

9 Bibliography

1. Anne C. Elster, "Parallelization issues and particle-in-cell codes", 1994
2. Snorre Boasson, "Porting PIC Code From Pthreads To MPI", 2003