PARALLELIZATION ISSUES
AND
PARTICLE-IN-CELL CODES

A Dissertation
Presented to the Faculty of the Graduate School
of Cornell University
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

by
Anne Cathrine Elster
August 1994
PARALLELIZATION ISSUES AND PARTICLE-IN-CELL CODES

Anne Cathrine Elster, Ph.D.
Cornell University 1994

“Everything should be made as simple as possible, but not simpler.”

- Albert Einstein.

The field of parallel scientific computing has concentrated on parallelization of individual modules such as matrix solvers and factorizers. However, many applications involve several interacting modules. Our analyses of a particle-in-cell code modeling charged particles in an electric field, show that these accompanying dependencies affect data partitioning and lead to new parallelization strategies concerning processor, memory and cache utilization. Our test-bed, a KSR1, is a distributed memory machine with a globally shared addressing space. However, most of the new methods presented hold generally for hierarchical and/or distributed memory systems.

We introduce a novel approach that uses dual pointers on the local particle arrays to keep the particle locations automatically partially sorted. Complexity and performance analyses with accompanying KSR benchmarks, have been included for both this scheme and for the traditional replicated grids approach.
The latter approach maintains load-balance with respect to particles. However, our results demonstrate it fails to scale properly for problems with large grids (say, greater than 128-by-128) running on as few as 15 KSR nodes, since the extra storage and computation time associated with adding the grid copies, becomes significant.

Our grid partitioning scheme, although harder to implement, does not need to replicate the whole grid. Consequently, it scales well for large problems on highly parallel systems. It may, however, require load balancing schemes for non-uniform particle distributions. Our dual pointer approach may facilitate this through dynamically partitioned grids.

We also introduce hierarchical data structures that store neighboring grid points within the same cache-line by reordering the grid indexing. This alignment produces a 25% savings in cache-hits for a 4-by-4 cache.

A consideration of the input data’s effect on the simulation may lead to further improvements. For example, in the case of mean particle drift, it is often advantageous to partition the grid primarily along the direction of the drift.

The particle-in-cell codes for this study were tested using physical parameters, which lead to predictable phenomena including plasma oscillations and two-stream instabilities. An overview of the most central references related to parallel particle codes is also given.
Biographical Sketch

“Cognito ergo sum.” (I think, therefore I am.)

- Rene’ Decartes, Discourse on Method, 1637.

Anne Cathrine Elster was born just south of the Arctic Circle, in Mo i Rana, Norway, to Synnøve and Nils Loe Elster on October 2, 1962.

Her elementary and secondary educations where obtained at Missorado School (1968-70), Monrovia, Liberia; Brevik Barneskole and Åsen Ungdomsskole, Brevik, Norway (1970-78), followed by Porsgrunn videregående skole, Norway, were she completed her Examen Artium in 1981 with a science concentration in chemistry and physics. Awarded a scholarship through the Norway-America Association by the University of Oregon, Eugene, she spent her first year of college officially majoring in pre-business, but slanting her program towards computer science. She enrolled the following year at the University of Massachusetts at Amherst where she received her B.Sc. degree in Computer Systems Engineering cum laude in May 1985. Anne joined the School of Electrical Engineering at Cornell University in September 1985 from which received an MS degree in August 1988 with Professor Anthony P. Reeves chairing her Committee. She has accepted a position with Schlumberger Well Services at their Austin Systems Center following her Ph.D.
To the memory of

Kåre Andreas Bjørnerud
Acknowledgements

“Friends are people with whom I may be sincere. Before them, I may think aloud.” — Author’s re-write of quote by Ralph Waldo Emerson, *Friends.*

First I would like to thank my advisor Prof. Niels F. Otani and my unofficial co-advisors Dr. John G. Shaw and Dr. Palghat S. Ramesh of the Xerox Design Research Institute (DRI) for making it all possible. Their encouragement, support and advice were essential for the development and completion of this work. Thank you for believing in me and letting me get a glimpse of the world of computational physics!

Gratitude is also extended to my Special Committee member Prof. Keshav Pingali of Computer Science and his NUMA group for numerous helpful discussions, and Special Committee member Prof. Soo-Young Lee for his useful suggestions.

I also wish to thank the Cornell Theory Center, the Computer Science Department and their computer staffs, for providing invaluable help and computer resources, and Dr. Gregory W. Zack of the Xerox Design Research Institute for his support.

Special thanks are extended to all my good friends through the years. Without their encouragement and moral support, I would neither have had the confidence nor the spirit needed to get through this degree program. Most of all, I would
like to acknowledge my family including my parents, Synnøve and Nils L. Elster (Tusen takk, Mamma og Pappal) and my siblings, Johan Fredrik and Tone Bente.

Finally, I also wish to express my honor and gratitude to my sponsors including: Xerox Corporation and the National Science Foundation through my advisor’s PYI grant, the Mathematical Science Institute, and IBM through the Computer Science Department, for the support through Graduate Research Assistantships; the Norwegian Government for providing scholarships and loans; the School of Electrical Engineering and the Computer Science Department for the support through Teaching Assistantships; and the Royal Norwegian Council for Industrial and Scientific Research, Mr. and Mrs. Robinson through the Thanks to Scandinavia Inc., as well as Cornell University, for the generous fellowships.

This research was also conducted in part using the resources of the Cornell Theory Center, which receives major funding from the National Science Foundation, and New York State. Additional funding comes from the Advanced Research Projects Agency, the National Institutes of Health, IBM Corporation and other members of the center’s Corporate Research Institute.

I gratefully acknowledge all the tax payers in Norway and the United States who probably unknowingly supported my efforts through the above organizations and institutions. May their investments pay off some day!

This dissertation is dedicated to the memory of Kåre Andrew Bjørnerud, my close friend and “brother” who so unexpectedly passed away only a week after defending his Ph.D. in French Literature at Oxford in December 1992. I have not known a finer scholar.
Table of Contents

1 Introduction
   1.1 Motivation and Goals ........................................... 1
   1.1.1 Terminology ................................................. 2
   1.2 Particle Simulation Models .................................... 4
   1.3 Numerical Techniques ......................................... 5
   1.4 Contributions .................................................. 6
   1.5 Appendices ...................................................... 7

2 Previous Work on Particle Codes 8
   2.1 The Origins of Particle Codes ................................. 8
   2.2 Parallel Particle-in-Cell codes ................................ 9
      2.2.1 Vector and low-order multitasking codes ............ 10
   2.3 Other Parallel PIC References ............................... 11
      2.3.1 Fighting data locality on the bit-serial MPP .......... 11
      2.3.2 Hypercube approaches .................................... 12
      2.3.3 A MasPar approach ....................................... 15
      2.3.4 A BBN attempt ............................................. 16
      2.3.5 Other particle methods ................................... 16
   2.4 Load Balancing ................................................ 17
      2.4.1 Dynamic task scheduling ................................. 18

3 The Physics of Particle Simulation Codes 25
   3.1 Particle Modeling .............................................. 25
   3.2 The Discrete Model Equations ................................ 26
      3.2.1 Poisson’s equation ....................................... 27
      3.2.2 The charge, q ............................................. 28
      3.2.3 The Plasma Frequency, $\omega_p$ .......................... 30
   3.3 Solving for The Field ......................................... 31
      3.3.1 Poisson’s Equation ....................................... 32
      3.3.2 FFT solvers .............................................. 33
      3.3.3 Finite-difference solvers ................................. 36
      3.3.4 Neumann boundaries ...................................... 37
5.2.2 Model parameters .............................................. 78
5.2.3 Result Summary ................................................. 80
5.3 Serial PIC Performance ........................................... 80
  5.3.1 Particle Updates – Velocities ................................. 82
  5.3.2 Particle Updates – Positions ................................. 82
  5.3.3 Calculating the Particles’ Contribution to the Charge Den-
sity (Scatter) ..................................................... 83
  5.3.4 FFT-solver ......................................................... 83
  5.3.5 Field-Grid Calculation ........................................ 84
5.4 Parallel PIC – Fixed Particle Partitioning, Replicated grids ........ 85
  5.4.1 Particle Updates – Velocities ................................. 85
  5.4.2 Particle Updates – Positions ................................. 86
  5.4.3 Calculating the Particles’ Contribution to the Charge Density 87
  5.4.4 Distributed memory FFT solver ............................... 88
  5.4.5 Parallel Field-Grid Calculation ............................... 91
5.5 Parallel PIC – Partitioned Charge Grid Using Temporary Borders
and Partially Sorted Local Particle Arrays ................................ 92
  5.5.1 Particle Updates – Velocities ................................. 92
  5.5.2 Particle Updates – Positions ................................. 93
  5.5.3 Calculating the Particles’ Contribution to the Charge Density 96
  5.5.4 FFT solver ......................................................... 98
  5.5.5 Parallel Field-Grid Calculation ............................... 98
5.6 Hierarchical Data structures: Cell-caching ............................ 98

6 Implementation on the KSR1 ........................................... 102
  6.1 Architecture overview ........................................... 103
  6.2 Some preliminary timing results .................................. 103
  6.3 Parallel Support on the KSR ....................................... 105
    6.3.1 C versus Fortran ........................................... 106
  6.4 KSR PIC Code ......................................................... 106
    6.4.1 Porting the serial particle code ............................. 106
  6.5 Coding the Parallelizations ....................................... 109
    6.5.1 Parallelization Using SPC ................................. 109
  6.6 Replicating Grids .................................................... 111
  6.7 Distributed Grid ..................................................... 113
  6.8 FFT Solver ......................................................... 116
  6.9 Grid Scaling ......................................................... 117
  6.10 Particle Scaling ..................................................... 117
  6.11 Testing ............................................................. 117

7 Conclusions and Future work ........................................ 121
  7.1 Future Work ......................................................... 123
A Annotated Bibliography

A.1 Introduction ......................................................... 124
A.2 Reference Books .................................................... 125
  A.2.1 Hockney and Eastwood: “Computer Simulations Using Particles” ........ 125
  A.2.2 Birdsall and Langdon: “Plasma Physics Via Computer Simulation” ......... 125
  A.2.3 Others ......................................................... 125
  A.2.4 Fox et. al: “Solving Problems on Concurrent Processors” .................. 126
A.3 General Particle Methods ........................................... 126
  A.3.1 F.H. Harlow: ”PIC and its Progeny” .................................. 126
  A.3.3 D.W. Hewett and A.B. Langdon: Recent Progress with Avanti: A 2.5D EM Direct Implicit PIC Code .................................. 127
  A.3.4 S.H. Brecht and V.A. Thomas: Multidimensional Simulations Using Hybrid Particle Codes ........................................... 127
  A.3.5 C.S. Lin, A.L. Thring, and J. Koga: Gridless Particle Simulation Using the Massively Parallel Processor (MPP) ........... 128
  A.3.6 A. Mankofsky et al.: Domain Decomposition and Particle Pushing for Multiprocessing Computers .............................. 128
A.4 Parallel PIC – Survey Articles ...................................... 129
  A.4.1 John M. Dawson ................................................. 129
  A.4.2 David Walker’s survey paper ....................................... 129
  A.4.3 Claire Max: “Computer Simulation of Astrophysical Plasmas” ........... 130
A.5 Other Parallel PIC References ...................................... 130
  A.5.1 Vector codes – Horowitz et al .................................... 130
  A.5.2 C.S. Lin (Southwest Research Institute, TX) et al ....................... 131
  A.5.3 David Walker (ORNL) ............................................. 132
  A.5.4 Lubeck and Faber (LANL) ........................................ 133
  A.5.5 Azari and Lee’s Work .............................................. 134
  A.5.6 Paulette Liewer (JPL) et. al. ...................................... 135
  A.5.7 Sturtevant and Maccabee (Univ. of New Mexico) ............... 136
  A.5.8 Peter MacNeice (Hughes/Goddard) ................................ 136

B Calculating and Verifying the Plasma Frequency ................. 137
B.1 Introduction ............................................................ 137
B.2 The potential equation ............................................... 138
  B.2.1 The FFT solver .................................................... 138
  B.2.2 The SOR solver .................................................... 138
  B.2.3 The field equations ................................................. 139
B.3 Velocity and particle positions ..................................... 140
B.4  Charge density .......................................................... 140
B.5  Plugging the equations into each other ............................ 144

Bibliography ........................................................................ 149
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Overview of Parallel PIC References – 2.5D Hybrid</td>
<td>20</td>
</tr>
<tr>
<td>2.2</td>
<td>Overview of Parallel PIC References – Liewer <em>et al.</em> 1988-89</td>
<td>21</td>
</tr>
<tr>
<td>2.3</td>
<td>Overview of Parallel PIC References – Liewer <em>et al.</em> 1990-93</td>
<td>22</td>
</tr>
<tr>
<td>2.4</td>
<td>Overview of Parallel PIC References – Walker</td>
<td>23</td>
</tr>
<tr>
<td>2.5</td>
<td>Overview of Parallel PIC References – Others</td>
<td>24</td>
</tr>
<tr>
<td>3.1</td>
<td>Plasma oscillations and time-step</td>
<td>49</td>
</tr>
<tr>
<td>4.1</td>
<td>Boundary/Area ratios for 2D partitionings with unit area</td>
<td>57</td>
</tr>
<tr>
<td>4.2</td>
<td>Surface/Volume Ratios for 3D partitionings with unit volume</td>
<td>57</td>
</tr>
<tr>
<td>5.1</td>
<td>Performance Complexity – PIC Algorithms</td>
<td>81</td>
</tr>
<tr>
<td>6.1</td>
<td>SSCAL Serial Timing Results</td>
<td>104</td>
</tr>
<tr>
<td>6.2</td>
<td>Serial Performance on the KSR1</td>
<td>107</td>
</tr>
<tr>
<td>6.3</td>
<td>Serial Performance of Particle Code Subroutines</td>
<td>108</td>
</tr>
<tr>
<td>6.4</td>
<td>Distributed Grid – Input Effects</td>
<td>116</td>
</tr>
</tbody>
</table>
List of Figures

3.1 3-D view of a 2-D particle simulation. Charges are thought of as “rods”. ................................................. 29
3.2 Calculation of node entry of lower corner of current cell ............. 40
3.3 Calculation of field at location of particle using bi-linear interpolation. 41
3.4 The Leap frog Method. ............................................................. 45
3.5 Two-stream instability test. a) Initial conditions, b) waves are forming, c) characteristic two-stream eye. ................. 51

4.1 Grid point distribution (rows) on each processor. ...................... 56
4.2 Inhomogeneous particle distribution ........................................ 61
4.3 X-profile for Particle Distribution Shown in Figure 4.2 ................ 61
4.4 New Grid Distribution Due to X-profile in Figure 4.3 .................. 62
4.5 Basic Topologies: (a) 2D Mesh (grid), (b) Ring. ...................... 69
4.6 Row storage versus cell caching storage ..................................... 73

5.1 Transpose of a 4-by-4 matrix on a 4-processor ring ..................... 89
5.2 Grid Partitioning: a) block-vector, b) subgrid ......................... 94
5.3 Movement of particles leaving their cells, block-vector setting ...... 95
5.4 Parallel Charge Accumulation: Particles in cell ‘A’ share grid points with the particles in the 8 neighboring cells. Two of these grid points, ‘a’ and ‘b’, are shared with particles updated by another processing node. ........................................... 97
5.5 Cache-hits for a 4x4 cell-cached subgrid .................................. 100

6.1 KSR SPC calls parallelizing particle update routines ................. 110
6.2 Parallel Scalability of Replicated Grid Approach ...................... 112
6.3 Scatter: Distributed Grid versus Replicated Grids .................. 114
6.4 Distributed Grid Benchmarks .................................................. 115
6.5 Grid-size Scaling. Replicated grids; 4 nodes; 262,144 Particles, 100 Time-steps ......................................................... 118
6.6 Particle Scaling – Replicated Grids ......................................... 119
Chapter 1

Introduction

“Once experienced, the expansion of personal intellectual power made available by the computer is not easily given up.” – Sheila Evans Widnall, Chair of the Faculty committee on undergraduate admissions at MIT, *Science*, August 1983.

1.1 Motivation and Goals

Particle simulations are fundamental in many areas of applied research, including plasma physics, xerography, astrophysics, and semiconductor device physics. So far, these simulations have, due to their high demand for computer resources (especially memory and CPU power), been limited to investigating local effects – typically using up to an order of 1 million particles.

These simulations often involve tracking of charged particles in electric and magnetic fields. The numerical techniques used usually involve assigning charges to simulated particles, solving the associated field equations with respect to simulated mesh points, applying the field solution to the grid, and solving the related equations of motion for the particles. Codes based on these numerical techniques are frequently referred to Particle-in-Cell (PIC) codes.

Highly parallel computers such as the Kendall Square Research (KSR) ma-
chine, are becoming a more and more integral part of scientific computation. By developing novel algorithmic techniques to take advantage of these modern parallel machines and employing state-of-the-art particle simulation methods, we are targeting 2-D simulations using at least 10-100 million simulation particles. This will facilitate the study of interesting global physical phenomena previously not possible.

These particle simulation parallelization methods will then be used in our continuing investigation of several problems in magnetospheric and ionospheric physics. In particular, we expect these methods to be applied to an existing simulation which models the energization and precipitation of the electrons responsible for the Aurora Borealis. Professor Otani and his students are currently examining with smaller computers the role that kinetic Alfvén waves (low frequency electromagnetic plasma waves) have in accelerating these electrons along field lines at altitudes of 1 to 2 Earth radii. Kinetic Alfvén waves have been proposed to be important in producing these “auroral” electrons, because the linear mode structure of these waves includes a component of the electric field which is oriented along the Earth’s magnetic field, ideal for accelerating electrons downward towards the Earth’s ionosphere. [Si91] [SO93].

Other codes that target parallel computers should also benefit from many of the techniques developed in this thesis.

1.1.1 Terminology

The terminology used in this thesis is based on the terms commonly associated with computer architecture and parallel processing as well as those adopted by KSR. The following “dictionary” is for the benefit of those unfamiliar with this terminology.

- **Registers**: Really fast local memory; only a few words of storage.

- **Cache**: Fast local memory.
• **Subcache**: KSR-ism for fast local memory, i.e. what *generally* would be referred to as a processor’s *cache* or *local cache*. On the KSR1 the 0.5Mb subcache is split up into a 256Kb data cache and a 256Kb instruction cache. This thesis will use the term *cache* for fast local memory. The term *subcache* may be used for emphasis when referring to the KSR.

• **Local Memory**: KSR calls the sum of their local memory *Allcache* and they hence sometimes refer to local memory as “local cache”. Note that this is not common practice. This thesis will refer to the distributed memory on each processing element as the more generally used term, *local memory*. On the KSR, the local memory consists of 128 sets of 16-way associative memory each with a page size of 16 Kb giving a total local memory of 32Mb.

• **Page**: Continuous memory block in local memory (16Kb on KSR) that may be swapped out to external memory (e.g. disk).

• **Subpage** (also called *cache line*): Minimum data-package copied into local subcache. On the KSR1 each page is divided into 128 subpages (cache lines) of 128 bytes (16 words).

• **Thrashing**: Swapping of data in and out of cache or local memory due to memory references requesting data from other processors or data spread across page or subpage boundaries.

• **OS kernel**: Operating System (OS) kernel; set of OS programs and routines.

• **Process**: Running program or program segment, typically with a lot of OS kernel support, i.e. processes typically take a while to set up (create) and release and are then sometimes referred to as *heavy weight processes* (see threads).

• **Threads**: Light-weight processes, i.e. special processes with little OS kernel over-head; On the KSR one typically spawns a parallel program into P
threads, where $P =$ no. of available processors so that there will be one thread running per processor.

- **Mach threads**: Low-level interface to the OS kernel; based on the Mach OS (as opposed to UNIX).

- **Pthreads (POSIX threads)**: Higher-level interface to Mach threads based on the IEEE POSIX (Portable Operating System Interface) standard. Most of KSR’s parallelism support is built on top of Pthreads. The KSR1 Pthreads adhere to IEEE POSIX P1003.4a standard.

Note: the terms *thread* and *processor* may be used interchangeably when talking about a parallel program that runs one thread per processor.

For further details on our test-bed, the KSR1, see Chapter 6.

### 1.2 Particle Simulation Models

Particle simulation models are often divided up into four categories [HE89]:

1. **Correlated systems** which include N-body problems and related models concerning covalent liquids (e.g. molecular dynamics), ionic liquids, stellar clusters, galaxy clusters, etc.,

2. **Collisionless systems** including collisionless plasma and galaxies with spiral structures,

3. **Collisional systems** including submicron semiconductor devices using the microscopic Monte-Carlo model, and

4. **Collision-dominated systems** including semiconductor device simulations using the diffusion model and inviscid, incompressible fluid models using vortex calculations.
In the correlated systems models there is a one-to-one mapping (correlation) between each particle simulated and physical particles modeled (atoms, molecules, ions, stars, or galaxies). Collision-dominated systems, on the other hand, take the other extreme and use a mathematical description that treats the vortex elements as a continuous, incompressible and inviscid fluid.

Our application falls in between under the second category, collisionless systems, where each simulations particle, often referred to as a “superparticle”, may represent millions of physical electrons or ions in a collisionless plasma. The numerical techniques used usually involve assigning charges to simulated particles, solving the associated field equations with respect to simulated mesh points, applying the field solution to the grid, and solving the related equations of motion for the particles. Codes based on these numerical techniques are frequently referred to Particle-in-Cell (PIC) codes.

Chapter 2 discusses the main references covering current parallel particle codes and related topics. The physics behind the collisionless plasma model is described in further detail in Chapter 3. This chapter also includes discussions of methods used in verifying the parameterizations and the physics behind the code. Tests using real physical parameters demonstrating predictable phenomena including plasma oscillation and two-stream instabilities are discussed.

1.3 Numerical Techniques

Our implementations use 2-D FFTs (assuming periodic boundaries) as the field solvers. Other techniques, including finite difference (e.g. SOR) and multigrid methods may also be used. The field is applied to each node/grid point using a 1D finite difference equation for each dimension, and then the field calculated at each particle location using bi-linear interpolation. A leap frog particle pusher is used to advance the particle positions and velocities. These numerical algorithms are explained in further detail in Chapter 3.
However, our choice of particular numerical methods is not the focus of this work; instead, we will concentrate on the general approaches of how to parallelize them as they interact with other sections of a code.

The primary goal of this work is hence to investigate how best to incorporate parallel methods within numerical algorithms with an eye towards maintaining their applicability to more sophisticated numerical methods that may be developed in the future.

1.4 Contributions

This dissertation provides an in-depth study of how to parallelize a fairly complex code with several interacting modules. Algorithmic complexity and performance analyses of the parallel algorithms developed herein, are included. These analyses highlight both the parallel performance of the individual blocks of code and also give guidelines as to how the choices of parallel methods used in each block of code may influence other parts of the code. We show how these interactions affect data partitioning and lead to new parallelization strategies concerning processor, memory and cache utilization.

Our framework is a physically distributed memory system with a globally shared memory addressing space such as the the KSR supercomputer. However, most of the new methods developed in this dissertation generally hold for high-performance systems with either a hierarchical or distributed memory system. Chapter 4 includes a discussion of distributed versus shared memory environments.

This chapter also describes a traditional parallel particle-in-cell methods using replicated and partitioned grids, as well as a novel grid partitioning approach that leads to an efficient implementation facilitating dynamically partitioned grids. This novel approach takes advantage of the shared-memory addressing system and uses a dual pointer scheme on the local particle arrays that keeps the particle locations automatically partially sorted (i.e. sorted to within the local grid partition). Load-
balancing techniques associated with this dynamic scheme are also discussed.

Chapter 4 also introduces hierarchical data structures that are tailored for both the cache size and the problem’s memory access structure, and show that further improvements can be made by considering the input data’s effect on the simulation.

Complexity and performance analyses of the methods discussed in Chapter 4 are covered in Chapter 5. Chapter 6 describes our test-bed, the KSR1 super-computer on which our test-codes was implemented in C using Pthreads on the KSR1. Optimizations were guided by the methods dictated by our analytical and experimental results.

1.5 Appendices

An annotated bibliography of the references described in Chapter 2 is provided in Appendix A. Appendix B shows how to verify that the numerical methods used produce the expected plasma oscillation by calculating and analyzing what happens during one general time-step.
Chapter 2

Previous Work on Particle Codes

“We live in reference to past experience and not to future events, however inevitable.” – Herbert G. Wells [1866-1946], Mind and the End of Its Tether

“Study the past if you divine the future.” – Confucius [551-479 B.C.], The Analects.

2.1 The Origins of Particle Codes

The origins of PIC codes date back to 1955 when Harlow and his colleagues at Los Alamos [Har88, Har64] developed them and related methods for fluid dynamics calculations. Their original work was a 1-D code that had 2 or more dimensions in mind (their 1-D code did not compete with similar codes of that era.) According to Hockney and Eastwood [HE89], this work laid the foundation for Morse and Nielson’s, and Birdsall’s Berkeley group introduction of higher-order interpolation (Cloud-in-Cell) schemes for plasmas in 1969.

The first particle models of electrostatic plasmas are, however, due to Bunneman’s group at Stanford and Birdsall and Bridges at Berkeley in the late ’50s, but their work did not use a mesh for the field calculations. Dawson [Daw83] gives a lengthy review of the field of modeling charged particles in electric and magnetic
fields and covers several physical modeling techniques typically associated with particle codes. Several good reference books on particle simulations have also been published in the last few years [HE89] [BL91] [Taj89] [BW91]. For a more detailed description of these books and some of the major papers referenced in this chapter, please see the annotated bibliography in Appendix A.

2.2 Parallel Particle-in-Cell codes

Due to their demand for computational power, particle codes are considered good candidates for parallel high performance computer systems [FJL+88] [AFKW90] [Max91] [Wal90]. However, because of their seemingly non-parallel structure, especially in the inhomogeneous cases, much work still remains – to quote Walker [Wal90]:

“... For MIMD distributed memory multiprocessors alternative decomposition schemes need to be investigated, particularly with inhomogeneous particle distributions. ”

An overview of the main references pertaining to parallel PIC codes is provided in Tables 2.1-2.5 at the end of this chapter. The central ideas provided by these references are discussed in subsequent sections.

Note that if the assignments of particles (or grids) to processors remain fixed, this is frequently called a Lagrangian decomposition. The parallel PIC codes using a pure Lagrangian decomposition will usually replicate the grid on each processor. In an Eulerian decomposition, the particles are assigned to the processors with its local grid information. As the particles move, they may migrate to another processor where their new local grid information is then stored. An Adaptive Eulerian approach will re-partition the grid in order to achieve a better load balance as particles move and “bunch” together on certain processors over time.
2.2.1 Vector and low-order multitasking codes

The first parallelizations of particle codes were done on vector machines. Nishiguchi (Osaka Univ.) et al. have a 3-page note [NOY85] that describes how they bunch particles in their 1-D code to utilize the vector processor on a VP-100 computer. Horowitz (LLNL, later Univ. of Maryland) et al. [Hor87] describes a 2D algorithm with timing analyses done on a similar 3D code for a Cray. Unlike Nishiguchi et al. who employs a fixed grid, Horowitz’s approach requires sorting in order to do the vectorization. This scheme is a bit slower for very large systems, but requires much less storage.

Parallelization efforts on two production codes, ARGUS and CANDOR, is described by Mankofsky et al. [M+88]. They include low-order multiprocessing on systems such as the Cray X-MP and Cray 2. ARGUS is a 3-D system of simulation codes including modules for PIC codes. These modules include several field solvers (SOR, Chebyshev and FFT) and electromagnetic solvers (leap frog, generalized implicit and frequency domain), where they claim their 3D FFT solver to be exceptionally fast. One of the most interesting ideas of the paper, however, is how they used the cache as storage for particles that have left their local domain whereas the local particles got written to disk. A cached particles was then tagged onto a local particle in its new cell when it got swapped in. Their experience with CANDOR, a 2.5D electromagnetic PIC code, showed that it proved efficient to multi-task over particles (or a group of particles) within a field block. They note the trade-off due to the fact that parallelization efficiency increases with the number of particle groups, whereas the vectorization efficiency increases with the number of particles per group. The speed-up for their implementation on the Cray X-MP/48 showed to reach close to the theoretical maximum of 4.

Horowitz et al. [HSA89] later describe a 3D hybrid PIC code which is used to model tilt modes in field-reversed configurations (FRCs). Here, the ions are modeled as collisionless particles whereas the electrons are treated as an inertia-
less fluid. A multigrid algorithm is used for the field solve, whereas the leap frog method is used to push the particles. Horowitz multi-tasks over 3 of the 4 Cray 2 processors in the multigrid phase by computing one dimension on each processor. The interpolation of the fields to the particles was found computationally intensive and hence multi-tasked achieving an average overlap of about 3 due to the relationship between task length and the time-slice provided for each task by the Cray. (For the Cray they used, the time-slice depended on the size of the code and the priority at which it ran.) The particle push phase similarly got an overlap of about 2 (many, but simpler calculations). These results are hence clearly dependent on the scheduling algorithms of the Cray operating system.

### 2.3 Other Parallel PIC References

With the introduction of distributed memory systems, several new issues are arising in the parallelization of particle codes. Data locality is still the central issue, but rather than trying to fill up a set of vector registers, one now tries to minimize communication costs (global data interaction). In either case, where available, one would like to fill up local cache lines.

#### 2.3.1 Fighting data locality on the bit-serial MPP

C.S. Lin et al. [LTK88,Lin89b,Lin89a] describe several implementations of a 1-D electrostatic PIC code implemented on the MPP (Massively Parallel Processor), a 128-by-128 toroidal grid of bit-serial processors located at Goddard.

They first describe a gridless model [LTK88] where particles are mapped to processors and the grid computations are avoided by using the inverse FFT to compute the particle positions and the electric field. However, since the MPP is a bit-serial SIMD (single instruction, multiple data) architecture with a grid topology and not much local memory, they found that the overhead in communication when computing the reduction sums needed for this technique when computing
the charge density was so high that 60% of the CPU time was used in this effort.

In an earlier study, they mapped the simulation domain directly to the processor array and sorted the particles according to their cell every time step. This was found to be highly inefficient on the MPP due to the excessive I/O required between the array processors and the staging memory. They also point out that the scheme would not remain load-balanced over time since the fluctuations in electrical forces would cause the particles to distribute non-uniformly over the processors.

Lin later [Lin89b,Lin89a], uses a particle sorting scheme based on rotations (scatter particles that are clustered through rotations).

The author simulates up to 524,000 particles on this machine using an FFT solver. The implementation uses 64 planes to store the particles. This approach fills only half the particle planes (processor grid) with particles to make the sorting simpler by being able to shuffle (“rotate”) the data easily on this bit-serial SIMD machine. The spare room was used in the “shuffling” process. Here congested cells had part of their contents rotated to their northern neighbor, and then to their western neighbor, if necessary, during the “sorting” process.

This implementation is clearly tied to the MPP architecture. For nodes with more computational power and a different interconnection network, other techniques will probably prove more useful.

2.3.2 Hypercube approaches

Unlike the MPP, hypercubes such as the Intel iPSCs, the JPL Mark II, and the NCUBE, have floating point processors (some even with vector boards) and a lot more local memory. Their advantage is in their relatively high degree of inter-node connectivity (the number of interconnections between nodes grows logarithmically in the number of nodes) that provides perfect near-neighbor connections for FFT algorithms as well as $O(\log N)$ data gathers and scatters. The nodes of these systems are still fairly homogeneous (with the exception of some I/O processors).
Lubeck and Faber (LANL) [LF88] cover a 2-D electrostatic code benchmarked on the Intel iPSC hypercube. A multigrid algorithm based on Fredrickson and McBryan efforts are used for the field calculations, whereas they considered 3 different approaches for the particle push phase.

Their first approach was to assigning particles to whichever processor has the cell information (observing strict locality). The authors rejected this approach based on the fact that their particles tended to congregate in 10% of the cells, hence causing serious load-imbalance. The second alternative they considered was to relax the locality constraint by allowing particles to be assigned to processors not necessarily containing their spatial region. The authors argue that the performance of this alternative (move either the grid and/or particles to achieve a more load-balanced solution) would be a strong function of the particle input distribution.

The alternative they decided to implement replicated the spatial grid among the processors so that an equal number of particles could be processed at each time-step. This achieves a perfect load balance (for homogeneous systems such as the iPSC). To us, however, this seems to require a lot of extra overhead in communicating the whole grid to each processor at each time-step, not to mention having to store the entire grid at each processor. They do, however, describe a nice performance model for their approach. The authors comment that they found the partitioning of their PIC algorithm for the hypercube “an order of magnitude greater” compared with a shared memory implementation.

Azari and Lee (Cornell) have published several papers related to Azari’s work on hybrid partitioning for PIC codes on hypercubes [ALO89, AL91, AL90, AL92, Aza92]. Their underlying motivation is to parallelize Particle-In-Cell (PIC) codes through a hybrid scheme of Grid and Particle Partitions.

Partitioning grid space involves distributing particles evenly among processor and partitioning the grid into equal-sized sub-grids, one per available processor element (PE). The need to sort particles from time to time is referred to as an
undesirable **load balancing** problem (dynamic load balancing).

A particle partitioning implies, according to their papers, that all the particles are evenly distributed among processor elements (PEs) no matter where they are located on the grid. Each PE keeps track of the same particle throughout the entire simulation. The entire grid is assumed to have to be stored on each PE in order to keep the communication overhead low. The storage requirements for this scheme are larger, and a global sum of the local grid entries is needed after each iteration.

Their hybrid partitioning approach combines these two schemes with the argument that by partitioning the space one can save memory space on each PE, and by partitioning the particles one may attempt to obtain a well-balanced load distribution which would lead to a high efficiency. Their hybrid partitioning scheme can be outlined as follows:

1. the grid is partitioned into equal subgrids;
2. a group of PEs are assigned to each block;
3. each grid-block is stored in the local memory of each of the PEs assigned to that block;
4. the particles in each block are initially partitioned evenly among PEs in that block.

Walker (ORNL) [Wal89] describes a 3-D PIC code for the NCUBE, but does not have a full implementation of the code. He uses the “quasi-static crystal accumulator”, some kind of gather-scatter sorter proposed by G. Fox. *et al.* (See also Walker’s general reference in Appendix A.)

Liewer (JPL) has also co-authored several papers on hypercube implementations. Her 1988 paper with Decyk, Dawson (UCLA) and G. Fox (Syracuse) [LDDF88], describes a 1-D electrostatic code named 1-D UCLA, decomposing the physical domain into sub-domains equal in number to the number of processors
available such that initially each sub-domain has an equal number of processors. Their test-bed was the Mark III 32-node hypercube.

The code uses the 1-D concurrent FFT described in Fox et al. For the particle pushing phase, they divide their grid up into \((N - p)\) equal-sized sub-domains. However, the authors point out how they need to use a different partitioning for the FFT solver in order to take advantage of the hypercube connection for this phase. (They need to partition the domain according to the Gray Code numbering of the processors.) The code hence passes the grid array among the processors twice at each time step. In the conclusions, they point out that this may not be the case if a finite difference field solution is used in place of the FFT.

In the paper published a year later [LZDD89] they describe a similar code named GCPIC (General Concurrent PIC) implemented on the Mark IIIib (64-processors) that beats the Cray X-MP. Liewer also co-authored a paper [FLD90] describing a 2-D electrostatic code that was periodic in one dimension, and with the option of bounded or periodic in the other dimension. The code used 2 1-D FFTs in the solver. Liewer et al. has recently developed a 3D code on the Delta Touchstone (512 node grid) where the grids are replicated on each processor [DDSL93,LLFD93]. Liewer et al. also have a paper on load balancing described later in this thesis.

### 2.3.3 A MasPar approach

MacNeice’s paper [Mac93] describes a 3D electromagnetic PIC code re-written for a MasPar with a 128-by-128 grid. The code is based on Oscar Buneman’s TRISTAN code. They store the third dimension in virtual memory so that each processor has a grid vector. They use a Eulerian decomposition, and hence need to sort after each time-step. A finite-difference scheme is used for the field solve, whereas the particle push phase is accomplished via the leap frog method. Since they assume systems with relatively mild inhomogeneities, no load balancing
considerations were taken. The fact they only simulate 400,000 particles in a 105-by-44-by-55 system, i.e. only about one particle per cell and hence under-utilizing the 128-by-128 processor grid, we assume was due to the memory limitations of the MasPar used (64Kb/processor). The MasPar is a Single Instruction Multiple Data (SIMD) machine, i.e. the processors share the instruction stream, but operate on individual (local) data.

2.3.4 A BBN attempt

Sturtevant and Maccabee [SM90] describe a plasma code implemented on the BBN TC2000 whose performance was disappointing. They used a shared-memory PIC algorithm that did not map well to the architecture of the BBN and hence got hit by the high costs of copying very large blocks of read-only data. Like the BBN, our test-bed, the KSR1, is also a shared-address space system with distributed memory. By focusing more on data locality issues, we will later show how we overcame some of the obstacles that face Sturtevant and Maccabee.

2.3.5 Other particle methods

D.W. Hewett and A.B. Langdon [HL88] describe the direct implicit PIC method and some relativistic extensions. The code uses an iterative solution based on ADI (alternating direction implicit) as a 2-D direct field solver. The paper does point out that only minimal consideration was given to algorithms that may be used to implement the relativistic extensions. Some concepts were tested on a 1-D code.

S.H. Brecht and V.A. Thomas [BT88] describe the advantages and disadvantages in using a hybrid particle code to simulate plasmas on very large scale lengths (several $\lambda_D$). By treating the electrons as a massless fluid and the ions as particles, some physics that magnetohydrodynamics (MHD) codes do not provide (MHD assumes charge neutrality, i.e. $\rho = 0$), can be included without the costs of a full particle code. They avoid solving the potential equations by assuming that the
plasma is quasi-neutral \( n_e \approx n_i \), using the Darwin approximation where light waves can be ignored, and assuming the electron mass to be zero. They hence use a predictor-corrector method to solve the simplified equations.

J. Ambrosiano, L. Greengard, and V. Rokhlin [AGR88] advocate the use of modern hierarchical solvers, of which the most general technique is the fast multipole method (FMM), to avoid some of the local smoothing, boundary problems, and aliasing problems associated with PIC methods when used to simulate cold plasmas and beams, and plasmas in complicated regions. The paper describes the FMM method for gridless particle simulations and how it fares with respect to the aforementioned problems associated with PIC methods.

2.4 Load Balancing

Aside from the scrolling methods previously mentioned that Lin developed for the MPP, Liewer et al. [LLDD90] have implemented a dynamic load balancing scheme for a 1D electromagnetic PIC code on the MarkIII Hypercube at Caltech/JPL. The code is based on the electrostatic code GCPIC (see 2.5.1). Load balancing was achieved by their Adaptive Eulerian approach that has each processor calculate an approximation of the plasma density profile and using it to compute the grid partitioning. This calculation requires all processors to broadcast the location of their sub-grid boundaries and their current number of particles. They point out that the actual plasma density profile could be used directly (computed for the field solution stage), but that it would require more communication to make the density profile global. Other methods, such as particle sorting, were assumed to require a much larger amount of communication and computation overhead. Results from test cases with 5120 particles run on 8 processors were provided. In the load-balancing case, the particle distribution was approximated every 5 time-steps.
2.4.1 Dynamic task scheduling

Hennessy et al. [SHG92,SHT+92] have also done some interesting load-balancing studies for hierarchical N-body methods that are worth investigating.

The first paper, entitled “Implications of hierarchical N-body Methods” concentrates on analyzing two N-body methods: the Barnes-Hut Method and the Fast Multipole Method. The paper stresses caching of communicated data and claims that for most realistic scaling cases, both the communication to computation ratio, as well as the optimal cache size, grow slowly as larger problems are run on larger machines. They also show that the programming complexity and performance overheads substantially increases when going from shared-memory to distributed memory implementations.

The second paper, entitled “Load Balancing and Data Locality in Hierarchical N-body methods” focuses on how to achieve effective parallelizations through simultaneously considering load balancing and optimizing for data locality for three main hierarchical N-body methods. In addition to the Barnes-Hut and FMM methods considered in the first paper, they also consider a recent method for radiosity calculations in computer graphics.

The latter turns out to require a very different approach to load-balancing since in this case, the work associated with a patch in one iteration is not a good indicator of the work associated with it in the next. Unable to find an effective predictive mechanism that could provide load balancing, the best approached the authors ended up with was something they call cost-estimates+stealing. This uses profiling or cost-estimates to initialize the task queues at each processor, and then uses on-the-fly task stealing to provide load balancing.

The test bed for both papers is the experimental DASH multiprocessor located at Stanford. It has 16 processors organized in 4 clusters where each cluster has 4 MIPS R3000 processors connected by a shared bus. The clusters are connected together in a mesh network. Each processor has two level of caches that are kept
coherent in hardware, and share an equal fraction of the physical memory.

Both the applications (N-body simulations) and test-bed (DASH) are fairly different from our setting, but their use of dynamic task scheduling to achieve load balancing, is worth noting. Chapter 4 discusses our idea of using processor system information such as “load” and “distance” that relates to this approach.
<table>
<thead>
<tr>
<th>Author(s)</th>
<th>Type</th>
<th>Architecture</th>
<th>Parallel methods</th>
<th>Field solver</th>
<th>Particle pusher</th>
<th>Max. pt cls simulated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mankofsky et al.</td>
<td>2.5D hybrid</td>
<td>Cray X-MP</td>
<td>multi-tasking</td>
<td>3D-FFT</td>
<td>leap frog</td>
<td></td>
</tr>
<tr>
<td>[M⁺88]</td>
<td>(ARGUS) 3D</td>
<td>Cray 2 &amp; X-MP/48</td>
<td>multi-tasking</td>
<td></td>
<td>leap frog &amp; others</td>
<td></td>
</tr>
<tr>
<td>Horowiz et al.</td>
<td>2.5D hybrid</td>
<td>Cray 2</td>
<td>multi-tasking</td>
<td>Multigrid</td>
<td>leap-frog</td>
<td>up to $10^6$</td>
</tr>
<tr>
<td>[HSA89]</td>
<td>(4 proc.)</td>
<td>(4 proc.)</td>
<td></td>
<td></td>
<td></td>
<td>43x43x43</td>
</tr>
<tr>
<td>Azari &amp; Lee</td>
<td>2.5D hybrid</td>
<td>Intel iPSC/2</td>
<td>hybrid partition</td>
<td>predictor -</td>
<td>sort pt cls</td>
<td>16,384</td>
</tr>
<tr>
<td>(1989-92)</td>
<td>quasi-neutral</td>
<td>hypercube</td>
<td>corrector</td>
<td>each $\Delta t$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[ALO89]</td>
<td>Darwin</td>
<td>(32 proc)</td>
<td>(subgrids replicated)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[AL90,AL91]</td>
<td>(n_e ≈ n_i,</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[AL92]</td>
<td>ignore light,</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[Aza92]</td>
<td>$m_e = 0$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Author(s)</td>
<td>Type</td>
<td>Architecture</td>
<td>Parallel methods</td>
<td>Field Solver</td>
<td>Particle pusher</td>
<td>Max. ptcls simulated</td>
</tr>
<tr>
<td>-------------------</td>
<td>----------</td>
<td>------------------</td>
<td>------------------</td>
<td>--------------</td>
<td>--------------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>Liewer, Decyk,</td>
<td>1-D</td>
<td>JPL</td>
<td>Eulerian</td>
<td>FFT</td>
<td>Gaussian</td>
<td>720,896 ptcls</td>
</tr>
<tr>
<td>Dawson &amp; Fox,</td>
<td>electro-</td>
<td>MarkIII</td>
<td>decomp.</td>
<td>rand. gen</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[LDDF88]</td>
<td>static</td>
<td>hypercube</td>
<td></td>
<td>for velocity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Liewer, Decyk,</td>
<td>1-D</td>
<td>JPL</td>
<td>Eulerian</td>
<td>FFT</td>
<td>buffer</td>
<td>8128 grid pts</td>
</tr>
<tr>
<td>[LD89]</td>
<td>electro-</td>
<td>MarkIII</td>
<td>(load bal.</td>
<td></td>
<td>ptcls</td>
<td></td>
</tr>
<tr>
<td></td>
<td>static</td>
<td>hypercube</td>
<td>tried in</td>
<td></td>
<td>leaving</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GCPIC</td>
<td></td>
<td>[LLDD90])</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Author(s)</td>
<td>Type</td>
<td>Architecture</td>
<td>Parallel methods</td>
<td>Field Solver</td>
<td>Particle pusher</td>
<td>Max. ptcls simulated</td>
</tr>
<tr>
<td>------------------</td>
<td>----------------</td>
<td>------------------</td>
<td>------------------</td>
<td>--------------</td>
<td>-----------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>Ferraro, Decyk,</td>
<td>2-D electro-</td>
<td>JPL</td>
<td>replicate</td>
<td>two</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Liewer</td>
<td>static</td>
<td>MarkIII hypercube</td>
<td>grid on each</td>
<td>1-D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[FLD90]</td>
<td></td>
<td></td>
<td>processor</td>
<td>FFTs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Liewer, Krucken,</td>
<td>3-D electro-</td>
<td>Delta</td>
<td>replicate</td>
<td>??</td>
<td>??</td>
<td>$1.47 \times 10^8$</td>
</tr>
<tr>
<td>Ferraro, Decyk</td>
<td>static</td>
<td>Touchstone</td>
<td>grid on each</td>
<td></td>
<td></td>
<td>(3000 grid pts)</td>
</tr>
<tr>
<td>[DDSL93] preprint</td>
<td></td>
<td>512-node</td>
<td>processor</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 2.4: Overview of Parallel PIC References – Walker

<table>
<thead>
<tr>
<th>Author(s)</th>
<th>Type</th>
<th>Architecture</th>
<th>Parallel methods</th>
<th>Field Solver</th>
<th>Particle pusher</th>
<th>Max. pt cls simulated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Walker</td>
<td>3D, but no full-scale PIC implemented</td>
<td>hypercube (NCUBE considered)</td>
<td>dynamic routing of grid pts</td>
<td>FFT assumed</td>
<td>only</td>
<td>estimates only</td>
</tr>
<tr>
<td>[Wal89]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Walker</td>
<td>3-D SOS simulated</td>
<td>hypercube</td>
<td>Adaptive Euler</td>
<td>commercial SOS code</td>
<td>only</td>
<td>estimates only</td>
</tr>
<tr>
<td>[Wal90]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Author(s)</td>
<td>Type</td>
<td>Architecture</td>
<td>Parallel methods</td>
<td>Field Solver</td>
<td>Particle pusher</td>
<td>Max. ptcls simulated</td>
</tr>
<tr>
<td>----------------</td>
<td>------------</td>
<td>--------------</td>
<td>------------------</td>
<td>--------------</td>
<td>-----------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>Lin et al.</td>
<td>1-D</td>
<td>MPP</td>
<td>reduction sums</td>
<td>“gridless”</td>
<td>inverse FFT</td>
<td>FFT</td>
</tr>
<tr>
<td>[LTK88]</td>
<td>electrostatic</td>
<td>(16,384 proc.)</td>
<td></td>
<td>1-D FFT</td>
<td>load bal. via shifts</td>
<td>524,000</td>
</tr>
<tr>
<td>Lin</td>
<td>1-D</td>
<td>MPP</td>
<td>sorting etc.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[Lin89a]</td>
<td>electrostatic</td>
<td>(bit-serial)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[Lin89b]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lubeck &amp; Faber</td>
<td>2-D</td>
<td>Intel iPSC</td>
<td>replicate grid on each processor</td>
<td>multigrid</td>
<td>sort ptcls each Δt</td>
<td>Gridpts: &lt; 14884 = (122)^2</td>
</tr>
<tr>
<td>[LF88]</td>
<td>electrostatic</td>
<td>(64 proc)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MacNeice (1993)</td>
<td>3-D</td>
<td>MasPar</td>
<td>store 3rd D in virtual memory</td>
<td>finite difference</td>
<td>leap frog</td>
<td>200 000 e^- ion pairs</td>
</tr>
<tr>
<td>[Mac93]</td>
<td>electromagn.</td>
<td>128x128</td>
<td></td>
<td></td>
<td></td>
<td>105x55x55 Δt's</td>
</tr>
</tbody>
</table>
Chapter 3

The Physics of Particle Simulation Codes

“Even if the open windows of science at first makes us shiver ... in the end, the fresh air brings vigour, and all the great spaces have a splendour of their own.” – Bertrand Russell, *What I Believe*, 1925.

3.1 Particle Modeling

The goal of this chapter is to show how to implement some of the typical numerical algorithms associated with particle simulations. As a benefit to those who are experienced in computing, but not necessarily with physics, we also review the basic physics behind some of the equations modeled, i.e. how one arrives at the equations and what they mean with respect to the physical world they are describing. Based on our understanding of the physical system we are trying to model, we then show how this knowledge can be used to verify and test the codes. Finally, we describe examples of interesting physical phenomena that can be investigated through the use of particle codes.

The numerical techniques described in this chapter represent only a few of the many numerical methods available. Several other techniques, some more complex,
but possibly more accurate, exist. However, we feel the techniques chosen give a representative flavor of the most common techniques, and may also prove to be more suitable for our future goal, i.e. parallelization than other more complex algorithms.

The discussions following concentrate on 2D simulations, the model we chose for our simulation. Although the physical world is clearly 3D and the methods used extend to multiple dimensions, we chose to model only 2D since the numerical computations otherwise would take a very long time for the larger interesting problems (which capture possible global effects) – even on present supercomputers. This is the reason many current serial particle codes model only 1D. As more computational power becomes available, one might either decide to expand to a 3D model, do an even larger-scale 2D (or 1D) simulation, or obtain more detailed solutions (using finer grids). Until the whole universe can be accurately modeled in a short amount of time, physical models will always be subject to trade-offs!

### 3.2 The Discrete Model Equations

This section looks at where the equations that were modeled stem from in the physics. In our simulations using the Acceleration model, we assumed the the equations being modeled were:

\[
\nabla^2 \Phi = -\frac{\rho}{\epsilon_0}, \quad (3.1)
\]

\[
\mathbf{E} = -\nabla \Phi, \quad (3.2)
\]

\[
d\mathbf{x}/dt = \mathbf{v}, \quad (3.3)
\]

\[
d\mathbf{v}/dt = \frac{q\mathbf{E}}{m} \quad (3.4)
\]

The following subsections will describe these equation in more detail.
3.2.1 Poisson’s equation

The field equation $\nabla^2 \Phi = -\rho/\epsilon_0$ stems from the differential form of Maxwell’s equations which say electric fields with space charges obey:

$$\nabla \cdot \mathbf{D} = \rho_f,$$

(3.5)

where $\mathbf{D}$ is the electric “displacement” and $\rho_f$ is the source density or density of free charges. $\nabla \cdot \mathbf{D}$ is also called the divergence of $\mathbf{D}$.

Combined with the equation for matter (with no polarizations or bound charges)

$$\mathbf{D} = \epsilon_0 \mathbf{E},$$

(3.6)

and with $\mathbf{E} = \text{grad } \Phi \triangleq \nabla \Phi$, we have:

$$\text{div } \epsilon_0 \mathbf{E} = \rho$$

(3.7)

$$\text{div } \nabla \Phi = -\frac{\rho}{\epsilon_0} \text{ or } \nabla^2 \Phi = -\frac{\rho}{\epsilon_0}.$$  

(3.8)

For a 2-D scalar field $\Phi(x, y)$, the sequential application of the differential operators grad and div also yields the above Poisson’s equation:

$$\nabla^2 \Phi = \frac{\partial^2 \Phi}{\partial x^2} + \frac{\partial^2 \Phi}{\partial y^2}$$

(3.9)

or

$$\nabla^2 \Phi = -\frac{\rho}{\epsilon_0}.$$  

(3.10)

$\mathbf{v} = d\mathbf{x}/dt$, and $d\mathbf{v}/dt = q\mathbf{E}/m$ follow from Newton’s law, where the force $\mathbf{F}$ is the sum of the electric force $q\mathbf{E}$.

Let us now take a closer look at the charge $q$ and the charge density $\rho$ and what they mean in our simulation.
3.2.2 The charge, \( q \)

The charge density, \( \rho \), is usually thought of as units of \( \text{charge}/(\text{length})^3 \) (charge per volume). In SI-units, the length is measured in meters. Now, volume indicates a 3-D system, whereas our simulation is a 2-D one. To “convert” our 2-D simulation to a 3-D view, one can think of each simulation particle representing a rod-like entity of infinite length that in the 3-D plane intersects the x-y plane perpendicularly at the simulations particle’s location \((x_0, y_0)\) as shown in Figure 3.1.

Consider a volume \( h_z h_x h_y \), where \( h_z \) is the height in units of the unit length, with \( h_x \) and \( h_y \) being the grid spacing in \( x \) and \( y \), respectively, and \( N_p \) being the number of simulation particles. Considering \( q_L \) as the charge per unit length, each rod would hence represent the charge \( q_L h_z \). Consequently, the total charge \( Q \) in the volume is:

\[
Q = N_p q_L h_z \tag{3.11}
\]

As mentioned, the charge density \( \rho \) is \( Q/\text{volume} \):

\[
\frac{Q}{\text{volume}} = \frac{(N_p q_L h_z)}{(h_z h_x h_y)} \tag{3.12}
\]

\[
= N_p q_L / (h_x h_y) \tag{3.13}
\]

notice that \( h_z \) always cancels out and that \( \rho = N_p q_L / (h_x h_y) \) has the right units:

\[
\frac{\text{(no-dim)}(\text{charge/length})}{(\text{length})^2} = \frac{\text{charge}}{(\text{length})^3}. \tag{3.14}
\]

In other words, in our 2-D simulation, the charge density, \( \rho \), is defined so that:

\[
h_x h_y \sum_{\text{gridpoints}(i,j)} \rho_{i,j} = q_L N_p. \tag{3.15}
\]

In our simulations, we have set the electron charge \((q)\), the number of simulation particles \((N_p)\), number of grid points \((N_g)\), grid spacings \((h_x, h_y)\), a well as the
Figure 3.1: 3-D view of a 2-D particle simulation. Charges are thought of as “rods”.
mean charge density $\rho_0$, all as input variables. Note that by specifying the above parameters, $\rho_0$ determines the size of the simulation particles (number of electron charges per simulation particle) through $q_L$:

$$\rho_0 = n0 * q = n * N_p * q_L / (h_x h_y)$$

where $n0$ is the particle density, and $n$ is the number of physical particles per simulation particle.

In testing the code for plasma oscillation, a physical phenomenon, it is therefore important that the input parameters used in the test are consistent so that the results make sense (physically).

### 3.2.3 The Plasma Frequency, $\omega_p$

The plasma frequency is defined as $\omega_p = \sqrt{q \rho_0 / m \epsilon_0}$. These oscillations are observed in real plasmas. To get a feel for where these oscillations stem from, one can take a look at what happens to two infinite vertical planes of charges of the same polarity when they are parallel to each other.

From Gauss we have that the flux at one of the planes is:

$$\oint E \, dA = \frac{q}{\epsilon_0}$$

Looking at a “pill-box” around one small area on the plane, one notices that all the fields but the vertical ones (in and out of the plane) cancel, so one ends up with:

$$\oint E \, dA = 2 \int E \, dA = 2 E A$$

Knowing that $\sigma = n q / Area$ and $q = \int \sigma \, dA$, and plugging this back to the flux equation one gets:

$$EA = \frac{\sigma A}{\epsilon_0}$$

giving $E = \sigma / 2 \epsilon_0$. Since we have two planes, the total field in our system is:

$$E = \frac{\sigma}{\epsilon_0}$$
Now we want to show that \( x'' + \omega^2 x = 0 \), where \( \omega \) would describe the plasma frequency. From Newton we have:

\[
F = m \cdot a = qE
\]

Looking at the field equation from Gauss in terms of \( x \):

\[
E = \frac{\sigma}{\epsilon_0} = \frac{nq}{\epsilon_0 \text{vol}} x = \frac{\rho}{\epsilon_0} x
\]

Hence, one can write:

\[
x'' = a = \frac{q\rho}{m\epsilon_0} x
\]

If the planes appear periodically, one could then deduce the following oscillatory behavior of the movement of the planes with respect to one another:

\[
\omega_p = \sqrt{\frac{q\rho}{m\epsilon_0}}
\]

This is referred to as the plasma frequency.

In order to verify what our code actually will behave according to the physical laws, an in-depth analysis of one general time step can be used to predict the general behavior of the code. The analytic derivations provided in Appendix B prove that our numerical approach indeed produces the predicted plasma frequency.

### 3.3 Solving for The Field

The fields in which the particles are simulated are typically described by partial differential equations (PDEs). Which numerical method to use to solve these equations depends on the properties of the equations (linearity, dimensionality, separability, variation of coefficients) and the boundary conditions (periodic, mixed or simple, isolated).

Simple, field equations can be solved using methods that compute the exact solution directly (direct methods), whereas the more complicated equation and boundaries require methods that go through a set of iterations improving on an
initial guess (*iterative methods*). For very inhomogeneous and non-linear systems, current numerical techniques may not be able to find any solutions.

For an overview of some of the classical numerical methods used for solving field equations, Chapter 6 of Hockney and Eastwood’s book [HE89] is recommended. Young [You89] gives a nice overview of iterative methods. It should, however be noted that numerical analysis is still is an evolving discipline whose current and future contributions may influence one’s choice of method. Recent interesting techniques include multigrid methods and related multilevel adaptive methods [McC89].

### 3.3.1 Poisson’s Equation

The behavior of charged particles in applied electrostatic fields is governed by the Gauss’ law for charge conservation which can be described by the following Poisson’s equation:

\[
\nabla^2 \Phi = -\frac{\rho}{\epsilon_0}
\]

where \( \Phi \), \( \rho \) and \( \epsilon_0 \) are the electrostatic potential, space charge density, and dielectric constant, respectively. The Laplacian, \( \nabla^2 \Phi \), acting upon \( \Phi(x, y) \), is for the 2-D case using a Cartesian coordinate system in \( x \) and \( y \):

\[
\nabla^2 \Phi = \frac{\partial^2 \Phi}{\partial x^2} + \frac{\partial^2 \Phi}{\partial y^2}
\]

Combining this definition with Gauss’ law, hence yields the following elliptic \(^1\) PDE:

\[
\frac{\partial^2 \Phi}{\partial x^2} + \frac{\partial^2 \Phi}{\partial y^2} = -\frac{\rho}{\epsilon_0}
\]

\(^1\)A linear second-order differential equation, \( L(u) \), is classified as elliptic if

\[
L(\Phi) = a(x, y)\frac{\partial^2 \Phi}{\partial x^2} + 2b(x, y)\frac{\partial^2 \Phi}{\partial x \partial y} + c(x, y)\frac{\partial^2 \Phi}{\partial y^2} + d(x, y)\frac{\partial \Phi}{\partial x} + e(x, y)\frac{\partial \Phi}{\partial y} + f(\Phi) = g(x, y),
\]

where \( a, b, c, d, e, f, \) and \( g \) are given real functions which are continuous in some region in the \((x,y)\) plane, satisfy \( b^2 < ac \). Notice that in the case of the above Poisson’s equations, \( a = c = 1 \) and \( b = d = e = f = 0 \).
which is the second order equation we need to solve in order to determine the potential, $\Phi$, and subsequently the field, $E = -\nabla \Phi$ (hence the name field solver).

The Poisson’s equation is a common PDE which we will show can be solved using FFT-based methods when periodic and certain other simple boundary conditions can be assumed. Our parallel implementation currently uses this type of solver.

### 3.3.2 FFT solvers

Direct solvers based on the FFT (Fast Fourier Transform) and cyclic reduction (CR) schemes compute the *exact* solutions of $N$ difference equations relating to a PDE in $O(N \log N)$ or less operations [HE89]. Since PDE solvers typically are $O(N^2)$ or more operations and the above schemes model elliptic equations, these schemes are commonly referred to as “rapid elliptic solvers” (RES). RES may only be used for some common special cases of the general PDEs that may arise from the field equation. However, when they can be used, the are generally the methods of choice since they are both fast and accurate.

FFT solvers require that the PDEs have constant coefficients (e.g. the Poisson equation $\nabla^2 \Phi = f$) in simple regions (e.g. squares or rectangles) with certain simple boundary conditions.

**The Fourier Transform**

The 1D *Fourier transform* of a function $h$ is a function $H$ of $\omega$:

$$
H(\omega) = \int_{-\infty}^{\infty} h(x) e^{-i \omega x} dx.
$$

The *inverse Fourier transform* takes $H$ back to the original function $h$:

$$
h(x) = \frac{1}{2\pi} \int_{-\infty}^{\infty} H(\omega) e^{i \omega x} d\omega.
$$
If $h$ is a description of a physical process as a function of time, then $\omega$ is measured in cycles per unit time (the unit of frequency). Hence $h$ is often labeled the time domain description of the physical process, whereas $H$ represents the process in the frequency domain. However, in our particle simulation, $h$ is a function of distance. In this case, $\omega$ is often referred to as the angular frequency (radians per second), since it incorporates the $2\pi$ factor associated with $f$, the unit of frequency (i.e. $\omega = 2\pi f$).

**Differentiation via scaling in $\mathbf{F}$**

Notice that differentiation (or integration) of $h$ leads to a mere scaling of $H$ by $i\omega$:

$$\frac{dh}{dx} = \frac{1}{2\pi} \int_{-\infty}^{\infty} i\omega H(\omega) e^{i\omega x} d\omega.$$  
(3.21)

The second derivative can similarly be obtained by a scaling of $H$ by $-\omega^2$:

$$\frac{d^2h}{dx^2} = \frac{1}{2\pi} \int_{-\infty}^{\infty} (i\omega) \frac{dH}{dx} d\omega$$  
(3.22)

$$= \frac{1}{2\pi} \int_{-\infty}^{\infty} (i\omega)[(i\omega)H(\omega)] e^{i\omega x} d\omega$$  
(3.23)

$$= \frac{1}{2\pi} \int_{-\infty}^{\infty} (-\omega^2)H(\omega) e^{i\omega x} d\omega.$$  
(3.24)

Hence, if the Fourier transform and its inverse can be computed quickly, so can the derivatives of a function by using the Fourier (frequency) domain.

**Discretizing for computers**

In order to be able to solve the PDE on a computer, we must discretize our system (or one of its periods in the periodic case) into a finite set of grid points. In order to satisfy the discrete Fourier transform (DFT), these grid points must be equally spaced dimension-wise (we may apply a 1D DFT sequentially for each dimension to obtain a multi-dimensional transform). A 1D DFT maps $N$ complex numbers (the $h(x)$’s) with a spacing $d$ into $N$ complex numbers (the $H(\omega)$’s):

$$H(h(x)) = H(\omega) \approx \sum_{n=0}^{N-1} f_n e^{i\omega n} / N.$$  
(3.25)
Notice that the Fourier terms (harmonics) must also be able to satisfy the boundary conditions. Hence, acceptable boundaries include those that have can be represented by a Fourier sine transform (no cosine terms), i.e. fixed values (Dirichlet); those using a Fourier cosine transform, i.e. slopes (Neumann); or those that are periodic (systems that repeat/“wrap” around themselves) and hence use the full DFT.

Assuming that the system we are modeling has spacings $h_x$ and $h_y$ in $x$ and $y$, respectively, then the Sampling Theorem says that if a function $g$ is bandwidth limited to within the spacings (i.e all frequencies of $g$ must satisfy the bands $-h_x < k_x < h_x$ and $-h_y < k_y < h_y$), then $g$ is completely determined by the grid-point samplings. The frequency at these limits, $f_c \equiv \frac{1}{2h}$, is called the Nyquist critical frequency. If $g$ has frequencies exceeding $f_c$, then the higher order frequencies which are a multiple of $\frac{1}{2h}$ of the lower frequencies will coincide at the grid-points and hence inflate the discretized function. This phenomenon is known as aliasing. They way to overcome aliasing is either to reduce the grid-spacings (increase the sampling rate) or use a low-pass filter on the function before it gets discretized.

2D FFT solver

Going back to the Poisson’s equation related to our 2D electrostatic field:

$$\frac{\partial^2 \Phi}{\partial x^2} + \frac{\partial^2 \Phi}{\partial y^2} = -\frac{\rho}{\epsilon_0} \quad (3.26)$$

Since we assumed that our system is periodic in both $x$ and $y$, we can compute the 2D DFT of $\rho$ for a system lengths $L_x$, $L_y$ with spacings $h_x$ and $h_y$ in $x$ and $y$, respectively.

If we in addition take advantage of the Fast Fourier Transform algorithm (FFT) we can get the following efficient procedure for solving the field:
1) Compute \( \text{FFT}(\rho(x, y)) \):

\[
\hat{\rho}(k_x, k_y) = \frac{1}{L_x L_y} \sum_{x=0}^{N-1} \sum_{y=0}^{M-1} \rho(x, y) e^{2\pi i x k_x / L_x} e^{2\pi i y k_y / L_y} \tag{3.27}
\]

2) Get \( \Phi \) by integrating \( \rho(k_x, k_y) \) twice by dividing by \( k^2 = k_x^2 + k_y^2 \) (see previous comments) where \( k_x = 2\pi / L_x \) and \( k_y = 2\pi / L_y \), and scaling by \( 1/\epsilon_0 \).

3) In order to get back to the grid, \( \Phi(x, y) \) take the corresponding inverse Fourier Transform:

\[
\Phi'(x, y) = \frac{1}{J L} \sum_{k_x=0}^{L-1} \sum_{k_y=0}^{L-1} \Phi(k_x, k_y) e^{-2\pi i x k_x / L_x} e^{-2\pi i y k_y / L_y}. \tag{3.28}
\]

### 3.3.3 Finite-difference solvers

We now include the description of a 5-point finite-difference SOR (Successive Over Relaxation) method which we used in some simple test-cases under Neumann (reflective) and Dirichlet (fixed) boundary conditions.

Applying a 5-point central finite difference operator for the Laplacian, the following is obtained for each coordinate:

\[
\nabla^2 \Phi \approx (\Phi_{i,j+1} + \Phi_{i-1,j} - 4\Phi_{i,j} + \Phi_{i+1,j} + \Phi_{i,j-1}) / h^2, \tag{3.29}
\]

where \( h \) is the grid spacing in \( x \) and \( y \) (here assumed equal).

Combining Gauss’ law and the above approximation, and the solving for \( \Phi_{i,j} \), the potential at a given node \((i,j)\), yields the following 2-D finite-difference formula (Gauss-Seidel):

\[
\Phi_{i,j} = \left( \frac{\Phi_{i,j} + \Phi_{i,j+1} + \Phi_{i-1,j} + \Phi_{i+1,j} + \Phi_{i,j-1}}{\epsilon_0} \right) / 4, \tag{3.30}
\]
This is the finite difference approximation we used in our original Poisson solver for testing some simple cases.

To speed up the convergence of our finite approximation techniques, we used a Successive Over Relaxation (SOR) scheme to update the interior points. The $\Phi_{i,j}$'s were hence updated and “accelerated” as follows:

$$
\begin{align*}
tmp &= \frac{(\Phi_{i,j+1} + \Phi_{i-1,j} + \Phi_{i+1,j} + \Phi_{i,j-1})}{4}, \\
\Phi_{i,j} &= \Phi_{i,j} + \omega(tmp - \Phi_{i,j})
\end{align*}
$$

where $\omega$ is the acceleration factor assumed to be optimal between 1 and 2. Notice that Equation 3.31 is the same as Equation 3.30. In our test implementation, our code was run for several different $\omega$’s, and we determined that for the grid spacings we used, it seemed to optimize (i.e. need fewer iteration before convergence) between 1.7 and 1.85. As mentioned, convergence was here assumed to be an acceptable threshold for round-off (we used $10^{-14}$ for most of our test runs on our Sun Sparc stations) calculated by updating the exit condition as follows:

$$
exit = \max(tmp - \Phi(i,j), exit).
$$

There is an extensive numerical literature discussing how to predict and calculate the optimal acceleration factor $\omega$.

The SOR is a fairly common technique. Other newer and possibly more accurate methods exist such as multigrid methods and techniques using larger template (more points). Since we assumed the field in our final code to be periodic, we opted for the FFT-solver since it is a much more accurate (gives us the direct solution) and also fairly quick method that parallelizes well.

### 3.3.4 Neumann boundaries

Natural Neumann boundaries are boundaries where the interior points are assumed to appear “reflected” across the boundary. In other words,
\[ \frac{d\Phi}{dx} = 0 \]

for points along the Neumann border. For our 2-D case, the finite difference update for border points on a square grid hence become:

\[ \frac{\Phi_{i,j-1} - \Phi_{i,j+1}}{2h_x} = d\Phi_{b,x} \quad \text{(Left or right borders)} \]  

\[ \frac{\Phi_{i-1,j} - \Phi_{i+1,j}}{2h_y} = d\Phi_{b,y} \quad \text{(Top or bottom borders)} \]

where \( d\Phi_{b,x} \) and \( d\Phi_{b,y} \) form the boundary conditions (derivative of the potential) for that boundary. In the case of impermeable Neumann borders, \( d\Phi_{b} = 0 \).

Solving the above equation for the point beyond the border, the result was then plugged into the 2-D central finite difference formula, as shown below (here: \( h = h_x = h_y \)). To simplify notation, we used the following "template" for our finite difference approximations:

\[ k \quad k = (i+1,j) \quad l = (i,j-1) \]
\[ l \quad m \quad n \quad m = (i,j) \quad n = (i,j+1) \]
\[ o \quad o = (i-1,j) \]

The equations for approximating the potentials at the Neumann borders hence become:

\[ \Phi_m = \left( \frac{h_x h_y}{\varepsilon_0} d\Phi_{b,x} + \Phi_k + 2\Phi_n + \Phi_o \right) / 4.0 \quad \text{[Left border]} \]  

\[ \Phi_m = \left( \frac{h_x h_y}{\varepsilon_0} d\Phi_{b,x} + \Phi_k + 2\Phi_l + \Phi_o \right) / 4.0 \quad \text{[Right border]} \]

Similar equations were derived for the top and bottom borders.

Systems with periodic or Neumann boundaries are singular. In order for the SOR to converge, as well as for testing purposes, some of the boundary points may specified as Dirichlet (fixed potential, i.e. electrodes). In our test implementation we used a mask to indicate Neumann or Dirichlet edges.

We tested our SOR code on a grid with part of the bottom left being a 0V electrode, and the upper right border being a 1V electrode. We observed how
the fields contours looked by using our plot facility. As expected, the potentials “smoothed” between the 0V and 1V electrode, i.e. the potentials values at each grid point changed very little locally after convergence. A group of particles of the same charge initialized close to each other at the OV electrode would, as expected, over time move towards the 1V electrode and generally spread away from each other. They would, as expected, also curve back into the system when coming near the Neuman left or right border.

3.4 Mesh–Particle Interactions

The solver provided us with the potential \( \Phi(i, j) \) at each grid-point. The field \( E \) at each corresponding grid-point was then calculated using a first order difference in each direction. The field \( E \) was hence stored in two arrays, \( Ex \) and \( Ey \), for the \( x \) and \( y \) directions, respectively. We used the following 1-D difference equations for the interior points when applying the field to each node on the grid:

\[
Ex_{i,j} = (\Phi_{i,j-1} - \Phi_{i,j+1})/ (2 * h_x) \\
Ey_{i,j} = (\Phi_{i-1,j} - \Phi_{i+1,j})/ (2 * h_y)
\]

where \( h_x \) and \( h_y \) are the grid spacings in \( x \) and \( y \), respectively. The actual field can be viewed as the vector resulting from combining \( Ex \) and \( Ey \).

3.4.1 Applying the field to each particle

A function was then written to calculate the electric field at a given particle’s location \((x_0,y_0)\), given the field grids \( Ex \) and \( Ey \), their size, and the grid spacings \( h_x, h_y \). (All parameters were passed through pointers). The finite-element scheme can best be described through Figures 3.2 and 3.3.
j = (int) \((x0 + h_x)/h_x\);  
\(i = (int) ((y0 + h_y)/h_y)\).

Figure 3.2: Calculation of node entry of lower corner of current cell

The field’s contribution to each particle was hence calculated as:

\[
E_{\text{part}_x} = (Ex(i, j) \ast (h_x - a) \ast (h_y - b)) \tag{3.40}
\]
\[
+ Ex(i + 1, j) \ast (hx - a) \ast b \tag{3.41}
\]
\[
+ Ex(i, j + 1) \ast a \ast (hy - b) \tag{3.42}
\]
\[
+ Ex(i + 1, j + 1) \ast a \ast b)/(hx * hy) \tag{3.43}
\]

\[
E_{\text{part}_y} = (Ey(i, j) \ast (h_x - a) \ast (h_y - b)) \tag{3.44}
\]
\[
+ Ey(i + 1, j) \ast (hx - a) \ast b \tag{3.45}
\]
\[
+ Ey(i, j + 1) \ast a \ast (hy - b) \tag{3.46}
\]
\[
+ Ey(i + 1, j + 1) \ast a \ast b)/(hx * hy) \tag{3.47}
\]

### 3.4.2 Recomputing fields due to particles

To take into account the charge due to all particles, the fields needed to be recomputed after each time step. Particles had to be synchronized in time before
the field update. As before, we assumed an initial charge density per node. This density was updated according to the particles’ location – i.e. the particles influence the field. The fields at each node were updated as in the previous section.

The charge density grid, $\rho$ was hence updated after each time-step using $a$ and $b$ as defined in Figure 3.3. The corresponding equations for the updates were $k = \rho_p / (h_x + h_y)$:

\[
\begin{align*}
\rho(i, j) & = (h_x - a) \cdot (h_y - b) \cdot k; \\
\rho(i + 1, j) & = (h_x - a) \cdot b \cdot k; \\
\rho(i, j + 1) & = a \cdot (h_y - b) \cdot k; \\
\rho(i + 1, j + 1) & = a \cdot b \cdot k.
\end{align*}
\]
3.5 Moving the particles

Using the Poisson solver developed in the previous section, we then developed code to track particles in an electric field. Particle coordinates were read in from a file and tracked until they hit a boundary. The initial conditions were set as described in the last section, except we allowed for a non-zero source term \( \nabla^2 \Phi = S, S \neq 0 \).

A hybrid finite-element/finite difference method and Euler integration were used to track independent particles in the field until they encountered a boundary. I.e. the fields were computed at the nodes using Finite Differences (FD) and interpolated with Finite Elements (FE).

The program assumed an initial charge density per node which got updated according to the particles’ location – i.e. the particles influenced the field. The fields at each node were updated at a fixed time-interval \( t \), whereas the particles were moved according to \( \Delta t \), a variable that got adjusted so that the particles would move no more than \( 1/4 \) of a cell side per \( \Delta t \) time increment. The particles’ trajectories were recorded on output files.

For tracking the particles, the mobility model updates the locations directly. A leap frog method was considered for the acceleration model, where the velocities calculations “leap” over the positions, and vice versa.

3.5.1 The Mobility Model

In the Mobility model, the new location \((x_1, y_1)\) of a particle at location \((x_0, y_0)\) is calculated given the field \((\text{Epart-}x, \text{Epart-}y)\) at that particle, mobility \((\mu)\), time-step \((dt)\), and grid (looking at particle velocity = \(\text{dx}/\text{dt}\)). To make sure reasonable steps were taken, the implementation should check whether the particle has moved more than \(1/4\) of the particle’s cell’s side (\(hx, hy\) passed in) within a time-step. If so, it may reduce the time-step, recompute the location, and return the new time-step. In our test implementation, the particles were stopped when they hit a border, and a flag \texttt{bflag} was set.
Given the previous routines, the following simple equations described the location updates in the mobility model:

\begin{align}
x_1 &= x_0 + (\mu E_{\text{par},x} \Delta t); \\
y_1 &= y_0 + (\mu E_{\text{par},y} \Delta t).
\end{align}

The above equations assume the medium to be isotropic, that is that the mobility of the medium, \(\mu\), is the same in both \(x\) and \(y\). Most fields considered are in such media.

**Testing the Mobility Model**

By the laws of physics, particles will tend to be attracted to Dirichlet boundaries (electrodes) with opposite high charges, but be repelled from Neumann boundaries. The latter happens because as the particles close in on a Neumann border, they see their “image charge” reflected across the border. Since equally signed charges repel each other, the particle steers away and does not cross a Neumann border.

The code was tested by initializing the bottom and top border as 0V and 1V electrode plates, respectively. Nine particles were then started at the bottom plate \((y=0)\) equally spaced between \(x=0.4\) and 0.6 (grid ranging from \(x=0\) to 1). We verified that the particles did not spread out unless the fields were recomputed. A single simulation particle started near the bottom plate \((0\text{V})\), would always go straight up to the top plate \((1\text{V})\), since no other particles would be present to influence its field.

### 3.5.2 The Acceleration Model

In the Acceleration Model for moving the particles, the force on the particle is proportional to the field strength rather than the velocity:

\[ F = q \times E \]
where $\mathbf{E}$ is the field $(\mathbf{E}_x, \mathbf{E}_y)$ at that particle. This is the model we will be using in our parallelized plasma simulation.

The leap frog method

The leap frog method was used for updating the particles’ location according to this model. A drag term proportional to the velocity was also added in.

We used the same functions as described in the last section for computing the fields and then particles’ contribution to them. However, instead of the routine for moving the particle using the Mobility Model, we now used two routines, one for computing the particle’s velocity, the other for updating the new location.

The functions calculate the new speed $(v_{x1}, v_{y1})$ of a particle at location $(x_0, y_0)$ with initial speed $(v_{x0}, v_{y0})$, given the force $\mathbf{F}$, particle mass $m$, time-step $(\Delta t)$, and the grid.

The leap-frog method models a particle’s movement by having the velocity update lag a half time-step behind the update of a particle’s position (position $(x,y)$ “leaping over” the velocity, then the other way around:

$$v^{n+1/2} = v^{n-1/2} + (\mathbf{F}((x^n, y^n))/m) \ast \Delta t$$

(3.55)

Splitting the directional vectors into $x$ and $y$ terms give:

$$F_x = q \ast E_{part,x}$$

(3.56)

$$F_y = q \ast E_{part,y}$$

(3.57)

Adding the drag term, this gave us the following code segment:

$$vx1 = vx0 + ((Fx/mass) \ast del_t) - (drag \ast vx0 \ast del_t);$$

$$vy1 = vy0 + ((Fy/mass) \ast del_t) - (drag \ast vy0 \ast del_t);$$

With the equations for updating the locations:

$$x^{n+1} = x^n + v^{n+1/2} \ast \Delta t$$

(3.58)
or the code segment:

\[
\begin{align*}
x_1 &= x_0 + (v_{x0} * \text{del}_t); \\
y_1 &= y_0 + (v_{y0} * \text{del}_t);
\end{align*}
\]

Notice that at the first time-step, the velocity is "pulled back" half a time-step \((1/2\Delta t)\), and then "leaps" over the location step (Figure 3.4).

When the particle hits a border, the particle re-appears on the other side if a periodic system is assumed. A flag is set if the particle moves more than a system length within a time-step. In this case the time-step will need to be reduced in order to have a viable simulation. Parameterization will be discussed further later in this chapter.

**Testing the Acceleration Model**

Initial testing was done as for the mobility model. Again, we started 5 particles at the bottom plate and showed that they spread out nicely as they moved towards the upper plate, except for cases where the particle came fairly close to their left or right borders. As predicted, the particles heading for the border were then be repelled by their image charges.

The true "acid-test" for our code was, however, to see whether it could correctly simulate the plasma oscillations described in the next section.
3.6 Testing the Code – Parameter Requirements

The parameters $h_x$, $h_y$, $L_x$, $L_y$, $\Delta t$, and $N_p$ need to satisfy the following constraints in order for the plasma waves to be adequately represented and so that the model is collisionless (Hockney and Eastwood [1988] describes these for the 1-D case):

1. $\omega_p \Delta t \ll 2$, where $\omega_p$ is the plasma oscillation frequency. One can usually expect $\omega_p \Delta t$ between .1 and .2 to give the optimal speed versus accuracy.

2. $h_x, h_y \leq \lambda_D$, i.e. that the spacing $\leq \lambda_D$, the Debye length defined to be the characteristic wavelength of electrostatic oscillation ($\lambda_D = v_T / \omega_p$, where $v_T$ is the thermal velocity of the plasma).

3. $L_x, L_y \gg \lambda_D$.

4. $N_p \lambda_D \gg L_x, L_y$, i.e. number of simulation particles per Debye length should be large compared to the simulation area. This generally guarantees that there are a large number of simulation particles in the range of the velocities near the phase velocity of unstable waves.

By analyzing the code carefully from this point, confidence can be achieved regarding both the stability and the correctness of the code.

3.6.1 $\omega_p$ and the time step

To see whether the code could produce the correct plasma frequency, we reformulated the code used in the acceleration model (Section 3.5.2) to have periodic boundary conditions on all boundaries.

The system is then loaded with two bands of particles with charge per unit length $n \times q$ in the direction perpendicular to the simulation plane, and mass per unit length $n \times m$. We are hence now modeling an “infinite system” where the two particle planes will each be seeing another plane of charges across the boundaries.
as it is repelled from the other band in its cell. This system hence corresponds to
the oscillatory system described in Section 3.2.3.

Assuming the system size is $L_x$ by $L_y$, the particle bands are then placed either
close to the boundary or close to the center of the system, aligned in the x or y
direction. As long as the bands are not placed at distance of $\frac{1}{2}L$ from each other,
the system should oscillate.

The leap-frog particle-pusher was used to advance the particle positions and
velocities. Since we now are assuming periodic conditions, we could use the FFT-
solver described in Section 3.3.2 for the Poisson’s equation to obtain a more
accurate result than the SOR method.

If the code was correctly normalized, the particle planes should oscillate back
and forth in the y-direction (or x-direction) through the center of the system at a
frequency of $\omega_0 = \omega_p = 2\pi / \text{1 period}$.

We tested our code for several different time-steps $\Delta t$ and verified that our code
indeed approximated the expected the plasma frequency as long as $\omega_p \ast \Delta t \ll 2$.
Using the following known physical parameters (from Physics Today, Aug ’93):

$$q = -1.6021773 \ast 10^{-19};$$
$$\epsilon_0 = 8.854187817 \ast 10^{-12};$$
$$m = 9.109389 \ast 10^{-31};$$

and the input parameters:

$$\rho_0 = q \ast n0 = -1.602 \ast 10^{-12} \text{ (}n0 = 10^7 \text{ – typical for some plasmas)}$$
$$\text{drag} = 0.0; \Delta t = \text{(varying – see below)}; t_{max} = 0.0002$$

we can calculate the expected plasma frequency:

$$\omega_p = \sqrt{\frac{q \rho_0}{m \epsilon_0}} = \sqrt{\frac{(-1.6021773 \ast 10^{-19})^2}{(9.109389 \ast 10^{-31})(8.854187817 \ast 10^{-12})}} \approx 1.78 \times 10^5$$
To avoid any grid-refinement problems/interpolation errors, we put the bands at $x=0.1875$ and at 0.8125 which is in the center of the bands respective column of cells for an 8x8 system. We were hence able to get the following tests demonstrating the $\omega_p \Delta t$ relationship shown in Table 3.1.

These results, shown in Table 3.1, agree with the theoretical result we obtained above. Notice that for $\Delta t = 1.0 \times 10^{-5}$, $\omega_p \Delta t = 2$, and as predicted by the theory is the turning point for stability.

### 3.6.2 Two-stream Instability Test

To further test whether our codes were able to simulate physical systems, we also performed a two-stream instability test [BL91]. In this case, two set of uniformly distributed particles (in our case 2D particle grids) are loaded with opposite initial drift velocities. Detailed knowledge of the non-linear behavior associated with such simulations were developed in the '60s.

Systems that simulate opposing streams are unstable since when two streams move through each other one wavelength in one cycle of the plasma frequency, the density perturbation (bunching) of one stream is reinforced by the forces due to bunching of particles in the other stream, and *vice versa*. The perturbations hence grow exponentially in time. In order for this test to work, care must be taken in choosing the initial conditions. In our case, we chose to test each dimension separately using an initial drift velocity $v_{drift} = \omega_p * L_x$ for half the particles and $v_{drift} = -\omega_p * L_x$ for the other half of the particles.

As can be seen from Figures 3.6a-c, our code was able to capture the characteristic non-linear “eyes” associated with two-stream instabilities. Our time-step was set to $1.5 \cdot 10^{-7}$. The “eyes” appeared within 10 time-steps of the large wave appearing. Notice that these are distance *versus* velocity plots showing 1D effects.
Table 3.1: Plasma oscillations and time-step

Two-band test

<table>
<thead>
<tr>
<th>$\Delta t$ (seconds)</th>
<th>period ($\times 10^{-5}$)</th>
<th>$\omega_p = 2\pi / period$ ($\times 10^5$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10 \times 10^{-5}$</td>
<td>blows up</td>
<td>— passes borders in 1 $\Delta t$!</td>
</tr>
<tr>
<td>$5.0 \times 10^{-5}$</td>
<td>blows up</td>
<td></td>
</tr>
<tr>
<td>$2.0 \times 10^{-5}$</td>
<td>blows up</td>
<td></td>
</tr>
<tr>
<td>$1.5 \times 10^{-5}$</td>
<td>blows up</td>
<td></td>
</tr>
<tr>
<td>$1.2 \times 10^{-5}$</td>
<td>2.5</td>
<td>2.6, but blows up after 1 period</td>
</tr>
<tr>
<td>$1.0 \times 10^{-5}$</td>
<td>3.0</td>
<td>2.0</td>
</tr>
<tr>
<td>$0.52 \times 10^{-5}$</td>
<td>3.12 - 3.64</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3.38 (avg)</td>
</tr>
<tr>
<td>$0.50 \times 10^{-5}$</td>
<td>3.00 - 3.50</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3.25 (avg)</td>
</tr>
<tr>
<td>$0.48 \times 10^{-5}$</td>
<td>3.36</td>
<td></td>
</tr>
<tr>
<td>$0.40 \times 10^{-5}$</td>
<td>3.20 - 3.60</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3.4 (avg)</td>
</tr>
<tr>
<td>$0.30 \times 10^{-5}$</td>
<td>3.20 - 3.60</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3.4 (avg)</td>
</tr>
<tr>
<td>$0.25 \times 10^{-5}$</td>
<td>3.50</td>
<td></td>
</tr>
<tr>
<td>$0.10 \times 10^{-5}$</td>
<td>3.50</td>
<td></td>
</tr>
<tr>
<td>$0.05 \times 10^{-5}$</td>
<td>3.50</td>
<td></td>
</tr>
<tr>
<td>$0.01 \times 10^{-5}$</td>
<td>3.50</td>
<td></td>
</tr>
</tbody>
</table>
Each dot in Figure 3.6a actually represents all the particles in $x$ mapping to the $y$ position. We obtained similar plots for a corresponding test of $x$ and $v_x$.

3.7 Research Application – Double Layers

Access to a large parallel systems, such as the KSR-1, will allow us to remove the restrictions imposed on us by the speed of the smaller computers we currently use. Our research group hopes, thereby, to be able to clarify a number of interesting plasma physics issues. First, an instability occurs in the present simulations due to an interaction between the accelerated and background electrons. The instability, when present, changes the electron velocity distribution that would be observed by a satellite passing below the plasma. The development of the instability depends on the perpendicular structure of the nonlinear development of both the Alfvén wave and the instability. The perpendicular and parallel resolution required for accurate representation of this effect requires both a high-speed platform and a parallel algorithm appropriate for the problem. Second, we have observed the presence of anomalous resistivity in regions of substantial Alfvén wave-generated electron drift. We would like to pinpoint the cause of this resistivity, but its mechanism has proved elusive due to the presence of substantial noise. Use of parallelism would allow us to employ a much larger number of particles, enabling us to reduce the noise due to very large superparticles. Third, the evolution of an Alfvén wave pulse is currently restricted by the simulation system length. Because of the periodic boundary conditions, employed in our current simulation, an Alfvén wave packet must eventually traverse a region previously crossed, encountering plasma conditions of its own wake. Again, the longer system possible on parallel systems such as the KSR-1, would alleviate this problem. There are also other simulation issues which would benefit from the resources and parallelization possibilities presented in this thesis.

Our initial experiments indicate that the KSR1 is a good match for our problem.
Figure 3.5: Two-stream instability test. a) Initial conditions, b) waves are forming, c) characteristic two-stream eye.
We find the combination of the relative ease of implementation provided by its shared-memory programming environment combined with its significant memory and processor resources very attractive.

Due to the computational and memory requirements of our code, all major arrays need to fit in local memory. Given the current KSR1’s hardware, a conservative estimate would imply we are restricted to 2 GB of memory (the other 2GB used for OS, program, and data storage). Each particle uses 4 double precision quantities (velocity and location in both x and y) and hence occupies 32 bytes. Particle code investigations of auroral acceleration typically employ 10-100 particles per grid depending on the effect being studied. Given the current memory restrictions, we would therefore like to model systems that are 4096-by-256 or larger. This should enable us to study effects currently not seen using current serial codes (using, for example, 256-by-32 grids).
Chapter 4

Parallelization and Hierarchical Memory Issues

"Parallelism is enjoyment exponentiated." – author ca. 1986.

4.1 Introduction

To date, the field of scientific parallel computation has concentrated on optimizing the parallelism for individual modules, such as solvers, matrix transposers, factorizers, particle pushers, etc. Our particle simulation code, however, is fairly complex and consists of several interacting modules. A key point in our work is therefore to consider the interactions between these sub-program blocks and analyze how the interactions affect parallelization.

In particular, we would like to see how the solver partitioning (say for an FFT solver) may impact the overall particle partitioning. Previous work has either ignored this issue, or in the case of Azari et al. [ALO89], used a localized direct solver. The latter, however, only works for very specialized cases. The more general problems usually require the use of some sort of numerical PDE solver.

Traditional parallel methods using replicated and partitioned grids, as well as a novel grid partitioning approach that leads to an efficient implementation facili-
tating dynamically partitioned grids are described. The latter is a novel approach that takes advantage of the shared-memory addressing system and uses a dual pointer scheme on the local particle arrays to keep the particle locations partially sorted automatically (i.e. sorted to within the local grid partition). Load-balancing techniques associated with this dynamic scheme will also be discussed.

Finally, we will investigate how memory hierarchies affect parallelization and show a novel approach using hierarchical data structures for storing the simulation grid. In the context of grid updates of the charge density, we will refer to this technique as cell caching.

4.2 Distributed Memory versus Shared Memory

The primary problem facing distributed memory systems are maintaining data locality and the overhead associated with it. This problem with parallel overhead also extends to the shared memory setting where data locality with respect to cache, is important. The author proposes that one view the KSR as a shared memory system where all memory is treated as a cache (or hierarchy thereof).

Shaw [Sha] points out that his experience with the SPARC-10s shows they have an interesting property which seem highly relevant. In order to achieve their peak speed (17-19 MFLOPs), the data must be in what Sun calls the SuperCache which is about 0.5 MBytes per processor (maybe to increase in future versions of the hardware). This implies that if you are going to partition a problem across a group of SPARC-10s, you have many levels of memory access to worry about:

1. machine access on a network
2. virtual memory access on one machine on a network
3. real memory access on one machine on a network
4. SuperCache access on one processor on one machine on a network
Hence, there is a great deal to worry about in getting a problem to work “right” when you have a network of multiprocessor SPARC-10s.

A network of Sun 10’s will consequently raise a lot of issues similar to that of the KSR in that they both possess several levels of cache/memory. (The KSR also has a 0.5MB local cache on each processor (0.25MB for data, 0.25MB for instructions).) To achieve optimum performance on any given parallel system, no doubt, a lot of fine-tuning is necessary. It is, however, hoped that our work can address the general problems and give some guidelines on the parallelizations of fairly complex physics (and similar) codes.

4.3 The Simulation Grid

When parallelizing a particle code, one of the bottle necks is how to update the grid quantities. The easiest and most common parallel implementations, have a local copy of the grid for each thread (processor).

In the ideal case, the grid is distributed, the processing nodes only share grid quantities on the borders, and the particles remain totally sorted; i.e., all particles within a sub-grid are handled by the same local thread (processing node).

4.3.1 Replicated Grids

To ensure that all grid updates occur without any contention from threads trying to update the same grid node (applying contributions from particles from different threads), one of the most common techniques is to replicate the grid for each thread. Each thread then calculates the contribution of its own particles by adding them up in a local array equal in size to the whole grid. When all the threads are done, the local grids are then added together either by one global master thread or in parallel by some or all threads. Experiments performed by the author on the KSR1 show that, due to the overhead in spawning and synchronizing many threads, a global master approach is in fact faster for smaller grids (say, 32 x 32).
### 4.3.2 Distributed Grids

Even though the particle push phase is parallelized using a replicated grid, the grid may still be *distributed* in a parallelized field solver. For an FFT solver this typically involves block-column and block-row distributions. However, when distributing the grid with particle updates in mind, we shall show that a column or row-oriented partitioning may not be the most efficient.

#### 4.3.3 Block-column/Block-row Partitioning

Assuming, nevertheless, that one tries to stick to the column/row distribution, in the particle update phase, a node would still need to copy a row (or column) to its neighbor (assuming the number of grid-points in each direction = the number of processing nodes available) since the particle-grid and grid-particle calculations are cell-based. This leaves each processor with the grid structure shown in Figure 4.1.

```
 o-----o-----o-----o-----o-----o-----o-----o
 |     |     |     |     |     |     |     |
 |     |     |     |     |     |     |     |
 |     |     |     |     |     |     |     |
 0-----0-----0-----0-----0-----0-----0-----0
```

- o = local grid data; 0 = data copied from neighbor

Figure 4.1: Grid point distribution (rows) on each processor.

### 4.3.4 Grids Based on Random Particle Distributions

Square subgrids give a much better border/interior ratio than skinny rectangles for *random* distributions of particles in an electrostatic field. The particles then tend to leave their local subdomains less often, and less communication is hence needed.
We investigated several other regular polygons (see following Tables 4.1 and 4.2), but concluded that the square is indeed a good choice since its border/area ratio is reasonable and its implementation simpler.

Table 4.1: Boundary/Area ratios for 2D partitionings with unit area.

<table>
<thead>
<tr>
<th>Polygon</th>
<th>Boundary/Area Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Circle:</td>
<td>3.54</td>
</tr>
<tr>
<td>Regular Hexagon:</td>
<td>3.72</td>
</tr>
<tr>
<td>“Uniform Grid” Hex:</td>
<td>3.94</td>
</tr>
<tr>
<td>Square:</td>
<td>4.00</td>
</tr>
<tr>
<td>Regular Triangle:</td>
<td>4.56</td>
</tr>
</tbody>
</table>

Table 4.2: Surface/Volume Ratios for 3D partitionings with unit volume.

<table>
<thead>
<tr>
<th>Polygon</th>
<th>Surface/Volume Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sphere (optimum):</td>
<td>4.84</td>
</tr>
<tr>
<td>Cylinder w/ optimum height and radius:</td>
<td>5.54</td>
</tr>
<tr>
<td>Staggered regular hexagon with depth = 1 side:</td>
<td>5.59</td>
</tr>
<tr>
<td>Cube:</td>
<td>6.00</td>
</tr>
</tbody>
</table>

4.4 Particle Partitioning

4.4.1 Fixed Processor Partitioning

The easiest and most common scheme for particle partitioning is to distribute the particles evenly among the processors and let each processor keep on tracking the same particles over time. This scheme works reasonably well when each processor maintains a local copy of the grid that, after each step is added to the other local grids to yield a global grid which describes the total charge distribution.
Unfortunately, replicating the grid is not desirable when the grid is large and several processors are used. In addition to the obvious grid summation costs, it would also consumes a great deal of valuable memory and therefore hampers our efforts to investigate global physical effects – one of the prime goals of our particle simulation.

Since the particles will become dispersed all over the grid over time, a fixed particle partitioning scheme would also not fare well in combination with a grid partitioning.

An alternative would be to use a hybrid partitioning like the one described by Azari and Lee [AL91,AL92] for a distributed memory hypercube (still has problems for very inhomogeneous cases), or to sort the particles according to local grids periodically. The latter would also require a dynamic grid allocation if load balance is to be maintained. We will get back to these combined schemes later in this chapter.

4.4.2 Partial Sorting

One way to reduce memory conflicts when updating the grid in the case where the grid is distributed among the processors, is to have the particles partially sorted. By partial sorting we mean that all particle quantities (locations and velocities) within a certain subgrid are maintained by the processor handling the respective subgrid. In this case, memory conflicts (waits for exclusive access to shared variable – here: grid locations) are limited to the grid points on the borders of the subgrids. This method can be quite costly if one it is necessary to globally sort an array with millions of particles fairly often.

“Dirty” bits

An alternative approach is to maintain local particle arrays that get partially sorted after each time-step. (This would be the equivalent of sending and receiving particles that leave their sub-domain in the distributed memory setting.)
A fairly common technique from the shared-memory vector processing world is to add a "dirty-bit" to each particle location and then set or clear this bit depending on whether or not the new location is within the local subgrid. Cray programmers worrying about using too much extra memory for the "dirty" bits have been known to use the least significant bit in the double-precision floating point number describing the locations as a "dirty" bit!

If the location is on a new grid, the locations may still be written back with the appropriate "dirty-bit" setting. This would, however, require a following search through the "dirty-bits" of all local particles by all processors and a corresponding fill-in of "dirty" locations locally on each processor, (assuming local memory should not be expanded and wasted).

4.4.3 Double Pointer Scheme

A more elegant way that achieves partial sorting of local particle arrays automatically during particle updates is to maintain two pointers, a read and a write pointer, to the local particle arrays. If the new particle location is still within the local subgrid, then both pointers get incremented, otherwise only the read pointer gets updated and the exiting particle is written to a scratch memory. Notice how this automatically "sorts" the particles back into the local array.

After the thread (processor) is done, it could then go through the global scratch array and fill in incoming particles by updating the write pointer.

It should be pointed out that this dual pointer techniques does not lend itself as well to vectorization as the "dirty-bit" approach, unless a set of pointers is used for each vector-location. However, in our work we are concentrating on parallelizations across scalar processors.

Load balancing information

This dual pointer scheme the write pointers automatically tells you how load balanced the computation is after each time-step. If some thread (processor) suddenly
gets very many particles or hardly any, flags could be raised to initiate a repartitioning of the grid for load balancing purposes. This would also be useful in the extreme case where most of the particles end up in a small number of subgrids and causing memory problems for the local particle array.

4.5 Load Balancing

4.5.1 The UDD approach

Load-balancing ideas stemming from the author’s work on fault-tolerant matrix algorithms [EUR89] can also be applied to load balancing particle codes. There, algorithmic fault-tolerant techniques were introduced for matrix algorithms that had been especially tailored for efficient multi-processing on hypercubes. The hypercube algorithms were based on an interconnection scheme that involved two orthogonal sets of binary trees. By focusing on redistributing the load for each processor to minimize the effect on the remaining partial orthogonal trees, low communication overhead was maintained.

The optimum re-distribution of particles should be similar to that shown for the UDD (Uniform-Data-Distribution) approach for matrices [EUR89] – i.e. a uniform distribution of the particles over currently available processors (assuming a homogeneous system). Elster et al. analyzed several re-distribution techniques, of which the Row/Column UDD (Uniform Data Distribution) [Uya86,UR85a,UR85b, UR88] method proved to be the most interesting. First a column-wise UDD has performed on the row with the faulty processor. This involved distributing the data points of the faulty processor equally among the other processors residing in the same row as the healthy ones by *rippling* the load from processor to processor so that only near-neighbor communication was needed. Then, by shrinking the y-direction (height) of the sub-matrices on the remaining processors in the row of the faulty processor, while increasing the height of the sub-matrices on the remaining processors correspondingly (row-wise UDD), load balancing was achieved with both
In our particle sorting setting similar ideas might prove useful in re-distributing the grid when opting for load balancing.

4.5.2 Load balancing using the particle density function

One way to do the partitioning is to maintain a watch on the particle density functions of the x and y directions. One would in this case periodically calculate the density “profile” of the system. For example if the grid had the distribution as shown in Figure 4.2, it would give a x-profile as shown in Figure 4.3. The x-profile could then be used to partition the grid in the x-direction (see Figure 4.4). The same could the be done for the y-direction, giving uneven rectangular sub-grids.

This is reminiscent of adaptive mesh refinement, so there are surely ideas to be
used from that area. The scheme is also similar to the one recently proposed by Liewer et al. [LLDD90] for a 1D code. They use an approximate density function to avoid having to broadcast the particle density for all grid points.

4.5.3 Load and distance

Techniques based purely on particle load (as outlined in the previous section) would only achieve load balance for homogeneous systems. Many parallel systems, including the KSR, has nodes with uneven loads due to either time-sharing, the presence of I/O processors on a subset of the processors, etc.

One idea we are considering is to use run-time information to aid us in achieving load balance. By incorporating the use of tables maintaining “load” (how busy are the individual processing nodes) and “distance” (how far away are the other nodes with respect to communication time) into the code, one could make the implementations more generally suitable, not only for the KSR, but also for a distributed system environments such as a network of Suns.

Thoughts to consider when searching for the right approach particle sorting and grid re-assignment in this context, are what input distributions and run-time distributions can commonly be expected.
If the input data is uniform, and one can expect it to remain so during the computations, then it is reasonable to assume a static allocation of sub-grids. However, if the system has one limited area of particles migrating around the system, it is reasonable to more seriously consider a dynamic grid approach.

Either way, “load” and “distance” information could then be used to determine particle/grid partitionings, and when to sort (update the partitioning). How one actually sorts (re-arrange partitionings) will actually depend on: 1) the network topology; which indicates which processors are neighbors, and hence dictates how to minimize communication, and 2) the memory model; which affects how passing and caching occurs.

Granted, the KSR tries to hide these two dependencies with the aid of its clever operating system, but if these two points are ignored, one is still likely to end up with an inefficient implementation. Communication overhead is, however, substantial for distributed workstations, so tailoring the implementations to minimize now becomes crucial.

4.6 Particle sorting and inhomogeneous problems

Azari and Lee [AL91] addressed the problem that results when several particles end up in one processor, by assigning each part of the grid to a group of processors (hybrid partitioning). This works reasonably for fairly homogeneous problems, but would take a serious performance hit for strongly inhomogeneous problems where most of the “action” takes place in a small region of the system (one processor “group”). Unfortunately, there are several such cases in plasma physics, so developing algorithms to handle these inhomogeneous cases is definitely worth investigating.
4.6.1 Dynamic Partitionings

The way we see the problem of inhomogeneous problems solved, is to use both a dynamic grid and a dynamic particle partitioning (well, the particle partitioning is basically implied), i.e. what Walker refers to as an adaptive eulerian decomposition. The number of grid elements per processor should here reflect the concentration of particles, i.e. if a processor does computations in an area with a lot of particles, it would operate on a smaller grid region, and vice versa. Grid quantities would then need to be dynamically redistributed at run-time as particle congregate in various areas of the grid.

On page 49 of his thesis, Azari [Aza92] indeed mentions re-partitioning of the grid as a possible attempt at load balancing. To quote him:

One possible attempt for load balancing could be to re-partition the grid space using a different method such as bi-partitioning. These methods have been designed for non-uniform particle distribution on the grid. However, the grid partitioning unbalanced the grid-related calculations since the number of grid points in each subgrid would be different. Also, the re-partitioning task itself is a new overhead.

The overheads will be further analyzed in this thesis. Since we are using an FFT solver, the grid will need to be re-partitioned regularly regardless of the particle pusher in order to take advantage of parallelism in the solver.

4.6.2 Communication patterns

The communication cost for the transpose associated with a distributed 2D FFT should be similar to that of going from a row distribution to a dynamic sub-grid (for the particle phase). Another argument for doing a re-distribution of the grid is that the row or column distribution used by the FFT is not as suitable for the particle stage, since one here would generally prefer square subgrids on each processor.
If, however, for some reason a more block-column or block row partitioning isn’t desirable also for the other stages, then the FFT solver’s order of the 1D-FFTs should match this partitioning in order to avoid an extra transpose. We will get back to this idea in the next chapter.

Azari-Bojanczyk-Lee [ABL88] and Johnson-Ho [JH87] have investigated matrix transpositions for meshes and hypercubes, respectively.

4.6.3 N-body/Multipole Ideas

The author has also considered some parallel multipole/N-body ideas [ZJ89,BCLI92]. Multipole methods use interesting tree-structured approaches such as the Barnes-Hut and ORB (Orthogonal Recursive Bisection) trees to subdivide particles. One idea would be to use a tree-structure similar to that described by Barnes and Hut [BH19] to organize the particles for each "sort" [BH19].

Barnes-Hut tree

The BH (Barnes-Hut) tree organizes the particles by mapping them on to a binary tree, quad-tree (max. 4 children per node), or oct-tree (max. 8 children per node) for 1-D, 2-D or 3-D spaces, respectively. Considering particles distributed on a 2-D plane (with more than one particle is present), its quad-tree is generated by partitioning the space into 4 equal boxes. Each box is then partitioned again until only one particle remains per box. The root depicts the top level box (the whole space), each internal node representing a cell, and the leaves particles (or empty if the cell has no particle).

The BH algorithm traverses the tree for each particle approximating whole subtrees of particles for boxes containing particles sufficiently far away from the present particle during force calculations. (Since it is the data structure I am interested in, I will not go into the detail of what the calculations actually estimate physically.)
Tree structures and particle sorting

How does this relate to particle sorting for PIC codes? Since particles invariably wander off in different directions, particles that were local to the grid cells (bound to processor nodes) at the start of the simulations, after a while are no longer near their origins. This causes a lot of communication traffic for distributed memory machines. One approach to overcome this is to sort the particles to the processors containing their current (and possibly neighboring) cells.

The first idea is to map the new particle locations to the BH tree and then assign the nodes according to the sub-tree distribution this gives. This will yield a nicely balanced distribution of particles, but since the tree structure itself does not account for cell-cell interaction (neighboring cells), a more clever approach is required for PIC codes since they are heavily “neighbor-oriented”.

The Fast Multipole Method (FMM) uses a similar recursive decomposition of the computational space into a tree structure. Unlike the BH method which only involves particle-particle and particle-cell interactions, the FMM also allows cell-cell interactions. How it goes about the actual computations and the different types of interactions is fairly complex, but not that interesting for our purposes.

In order to make use of the structures these methods present, the author believes that a hybrid BH-FMM approach could prove the most useful. Since we only care about neighboring cells, it should be possible to simplify the calculations for cell-cell interactions.

4.7 The Field Solver

Noticing that different implementations use different solvers, it is useful to see what impact choosing an FFT solver would have on the particle-pushing stages of our code, compared to other solvers, such as multigrid.

It is also worth noting that an FFT solver and periodic boundary conditions would generally not be used for very inhomogeneous systems. Otani [Ota] has
pointed out, however, that it is possible for very large waves to exist in an otherwise homogeneous system which, in turn, lead to significant bunching of the particles. It is predicted that one would not have less than, say, half of the processors handling most of the particles.

4.7.1 Processor utilization

However, using only 50% of the processors effectively on a parallel system is indeed a significant performance degradation. It is “only half” of the parallel, but with respect to the serial speed, which is the important measure, it is very significant, especially for highly parallel systems. For example on a 128-processor system a 50% utilization/efficiency gives us a 64-times speed-up (max. theoretical limit) versus a 128-times speedup for a fully used system. I.e. there is a loss of resources with respect to a single node of a factor of 64!

4.7.2 Non-uniform grid issues

Ramesh [Ram] has pointed out that keeping a non-uniform grid in order to load-balance the particle push stage may lead to problems with the field solver.

The grid could alternatively be partitioned uniformly (equal grid spacing) on a system-wide basis. The number if many grid points that get stored by each processor would vary according to non-uniform distribution. Ramesh’s point, however, does still come into play in that we now have a solver that will have an uneven load across the processors, a performance hit, also pointed out by Azari and Lee, that will have to be considered.

Alternatively, a solver for non-uniform meshes could be used. It, however, may lead to another “can of worms” with respect to accuracy, etc.

4.7.3 FFT Solvers

A parallel FFT is considered quite communication-intensive, and because of the communication structure, isis best suited for hypercubes which have a high degree
of communication links. Doing a 2D FFT usually also implies doing a transpose of the rows/columns. G. Fox et al. [FJL+88] point out that for this approach the overhead is functionally the same as the more direct 2D approach, but by performing all the communication at once (transpose), some start-up-message overhead is saved.

Claire Chu, a student of Van Loan, wrote a PhD thesis on parallel FFTs for the hypercube. Sarka et al. have investigated parallel FFTs on shared memory systems. Interestingly enough, the only reference I seem to have encountered for parallel FFTs on array processors, is Maria Guiterrez's, a student of my MS advisor who did some image processing work on the MPP (bit-serial array proc.).

Van Loan's book [Loa92]. It includes both Clare Chu's work on the hypercube and a section on FFTs for shared memory systems. All the algorithms in the book are written in a block-matrix language. As mentioned in Section 2, G. Fox et al. also covers parallel FFTs. Both point out the use of a transpose.

Most distributed parallel algorithms involving tree structures, FFTs or other highly connected communication topologies assume a hypercube interconnection network (it fits so perfectly for the FFT!). In addition, most distributed memory and shared memory approaches found in numerical texts tend to look at algorithms with only one data point per processor.

In our case, this is not a good model. For instance, if we were to use 16 processors, this would only leave us a 4-by-4 grid. Besides, having only 4 grid points per processor (assuming the neighboring column gets copied) seems very inefficient for the particle phase – and certainly does not allow for much dynamic grid allocation.

Consequently, it would be a lot more reasonable to assume that we have, say, a $n \times n$ grid mapped on to $p$ processing elements (PEs) with $n = O(p)$. In this case, if $n = p$, we will have $p$ 1D-FTTs to solve in each direction on $p$ PEs. The question is, of course, how to redistribute the grid elements when going from one
Let us first consider a standard grid and ring topology shown in Figure 4.5.

Assume that both the processor array and the field grid dimensions are a power of 2 (simplifying the FFTs). For the first FFT, it is clear that it would be most advantageous to have one (or more) columns (or rows) per processor.

Assuming only one column per processor, it would then take $O(N \log N)$ execution time to perform an N-point FFT. One would then need to "transpose" the grid entries in order to perform the 1D-FFTs in the other direction. Hopefully, such packages could be obtained from the vendor since this is obviously a time-consuming task given all the communication involved.

It is not obvious how fast such an algorithm would be on the KSR given the underlying token-based communication structure. There is also the problem with contention from other ring cells that may be excluded from one’s current processor set. The reason the latter is likely to happen is that the KSR has some processors with I/O attachments that significantly degrade their computational use, causing
an unbalanced system. Given this, it is unlikely that one could get the full use of a 32-processor ring – the ideal for an FFT. If only 16 processors are used and one does not explicitly request exclusive access to the whole ring, other user’s processes running on the remaining 16 nodes could cause extra communication traffic on the ring affecting one’s performance. Further details on the KSR’s architecture is given in Chapter 6.

The “pool-of-task” approach that Van Loan mentions in his book may be a reasonable approach if one considers running on the full “unbalanced” 32-processor ring (or 64 or more). Notice here the potential contention between load-balancing the 1D-FFTs and the transpose.

**Matching Grid Structures and Alternating FFTs**

As mentioned in the introduction, the local memory on the KSR = (128 sets) x (16-way associativity) x (16Kb (pagesize)) = 32 Mb. In fact, all physical memory comes in powers-of-two. On the KSR local memory thrashing hence occurs when a processor repeatedly references more than 16 addresses with a stride of 128 pages (or 32 addresses with a stride of 64 pages or 64 addresses with a stride of 32 pages). One should hence take care to ensure that strides are a non-power-of-two of the multiple of a page size (16kB).

Notice how this conflicts with the implementation of 2D FFTs which tend to operate on arrays that indeed are powers-of-two. It is for this reason and the fact that one usually can make use of fast unit-stride 1-D FFT routines provided by the manufacturer (often hand-coded in assembler), that 2D FFT implementations often involve an actual transpose.

During our 2-D FFT field solver, there is a reordering (transposition) of the grid between each set of 1-D FFT calls in order to be able to use contiguous vectors for the 1-D FFT computations. This reordering of the grid can be quite costly if the grid is fine and if there are on the average a few particles per cell, respectively. If we keep the particles in block-vector grids during the particle push, both re-ordering
steps can be saved, and we hence get the following:

1. Column-wise – FFT
2. Transpose
3. Row-wise FFT
4. Row-wise inverse FFT
5. Transpose
6. Column-wise – inverse FFT
7. Particle Push

Notice that this block-column grid partitioning conflicts with the optimum square-shaped grid partitioning for uniform problems.

4.7.4 Multigrid

In noting how similar our dynamic grid-partitioning approach for the particle-grid/grid-particle steps is to that of adaptive grid schemes one finds in multigrid (MG) methods, one could ask, would it be reasonable to consider using an actual parallel MG method for the field solver? Good Multigrid references include [Bri87, McC89]. The problem seems, however, to be that these methods are still not fully developed for parallel systems. The examples in the references focus on problems with Neumann and Dirichlet boundary conditions.

4.8 Input Effects: Electromagnetic Considerations

For electromagnetic codes, the magnetic field lines will for certain criteria tend to cause particles to move in a direction primarily along the field. In this case,
one would indeed like the subpartitions to be aligned as flat rectangles along this
direction.

Electrons tend to be tied to the field lines and move huge distances along the
magnetic field, but have difficulty moving perpendicular to it. Ions behave a lot like
the electrons, but since they have more mass, they tend to make larger excursions
across the fields. If the frequency regime is high, one will only see a part of the
ions spiraling around the magnetic field $B$, and the ions behave as if they were
unmagnetized.

If the frequency regime is way above the cyclotron frequency (how many times
a particle can go around a magnetic field line per second), one may want to model
motion where the electrons are unmagnetized. For these unmagnetized cases, there
is no preferred general directions of the particles, so square partitionings would be
preferable.

4.9 Hierarchical Memory Data Structures: Cell Caching

Grids are typically stored either column or row-wise. However, in a system with
memory hierarchies, especially systems with caches, we shall show that these are
not necessarily the best storage schemes for PIC codes.

When the particle’s charge contributions are collected back on the grid points,
each particle will be accessing the four grid points of its cell (assuming a quadrangle-
gular grid).

If the local grid size exceeds a cache line (which is typically the case since cache
lines tend to be small – 16 words on the KSR1), and column or row storage of the
grid exits, each particle will need at least two cache lines to access the four grid
points it is contributing to.

Since a significant overhead is paid for each cache hit, we hence propose that
one instead stores the grid according to a cell caching scheme. This means that
instead of storing the grid row or column-wise during the particle phase, one should store the grid points in a 1-D array according to little subgrids that may fit into one cache-line. On the KSR1 where the cache line is 16 words, this means that the grid is stored as a sequence of either 8x2 or 4x4 subgrids. The latter would minimize the border effects. However, even the 8x2 case shows a 22% reduction in the number of cache-line accesses. Further analysis of this scheme will be presented in Chapter 5.

Row-storage of 2-D mxn array:

\[ a_{00}a_{01} \cdots a_{0n}a_{10} \cdots \cdots a_{mn} \]

4x4 cell-caching storage:

\[ a_{00} \cdots a_{03}a_{10} \cdots a_{13}a_{04} \cdots a_{07} \quad a_{14} \cdots a_{17} \cdots \cdots a_{mn} \]

Figure 4.6: Row storage versus cell caching storage

In order for the cell-caching scheme to be effective, the grid needs to be subpage (cache-line) aligned. The KSR automatically does this when using malloc in C.

Notice that this cell-caching technique is really a uni-processor technique. Therefore, both parallel and serial cell codes for any system with a cache would benefit from using this alternative block storage.

It would, however, affect the alternating FFT approach we will be introducing since cell-caching storage would not preserve the column/row storage in the FFTs.
Chapter 5

Algorithmic Complexity and Performance Analyses

“Life is short, the Art long, opportunity fleeting, experience treacherous, judgment difficult.” – Hippocrates [ca. 460-357 B.C.], Aphorisms.
Proverbial: Ars longa, vita brevis.

5.1 Introduction

In order to get a better understanding of how various parallelization approaches would affect performance, this chapter includes a complexity analysis of the approaches we think are the most reasonable. These analyses consider both computational requirements and memory traffic. By combining the complexity results with the timing results from serial codes and known parallel benchmarks, one can estimate how effective the parallelizations will be on a given parallel system, given certain chosen problem parameters such as grid size, and the number of simulation particles used. A fine-tuning of these results is possible after fully implementing and testing the chosen parallel algorithm(s) on a chosen architecture. Our test-bed was the KSR1 (see Chapter 6).
5.2 Model

When moving from sequential computer systems to parallel systems with distributed memory, data locality becomes a major issue for most application. Since intermediate results and data need to be shared among the processing elements, care must be taken so that this process does not take an inordinate amount of extra run-time and memory space. After all, the purpose of parallelizing the codes is so they can take advantage of the combined speed and memory size of parallel systems.

Typically, communication overhead is modeled as follows (see Hockney and Jessup [HJ81]):

\[ t_{\text{comm}} = \alpha + N\beta \]

where \( t_{\text{comm}} \) is the communication time for an N-vector. Here \( \alpha \) is the start-up time and \( \beta \) a parameter describing the bandwidth on the system.

If one then assumes that the computations do not overlap with communication, the total time a given parallel approach would take (assuming the process does not get swapped out by the operating system at run-time) would then be:

\[ T_{\text{total}} = t_{\text{comp}} + t_{\text{comm}}, \]

where \( t_{\text{comp}} \) is the time the algorithm spends on computations and \( t_{\text{comm}} \) is the communication overhead described above.

Modern parallel systems with distributed memory, starting with the Intel iPSC 2, however, have independent I/O processors that let the users overlap the time spent sending data between processors, i.e. communication, with computation time. How efficient the implementations are then also becomes related to how well the program can request data in advance while doing large chunks of computations. How much the application programmer can take advantage of this overlap feature is hence also strongly application-dependent.
On hierarchical memory systems, $t_{comm}$ will be a function of which levels of the memory hierarchy are accessed for the requested data. For simplicity we will consider a model with only two hierarchical parameters, $t_{lm}$, which denotes time associated with local memory accesses and $t_{gm}$, time associated with global memory:

$$t_{comm} = t_{lm}(N) + t_{gm}(M),$$

It should be noted that $t_{lm}$ covers both items within a cache-line and items within a local subcache. For a vector of $N$ local data elements, $t_{lm}(N)$ is hence not a simple constant, but rather a function of whether the individual data elements are 1) within a cache-line, 2) within cache-lines in cache and 3) in local memory. Similarly, $t_{gm}$ is a function of whether a vector of $M$ data elements accessed are all within 1) a communication packet (equal to a cache-line on the KSR), 2) in some distant local memory, or 3) on some external storage device such as disk. Since access to external memory devices typically is several orders of magnitude slower than access to system memory, we will assume that this is so undesirable during computation that the problems in this case are tailored to fit within the system memory.

Notice that serial computer systems with local caches also involve $t_{comm} = t_{lm}$ of the same complexity.

### 5.2.1 KSR Specifics

The KSR also has other hardware features that hamper the accuracy of the above distributed memory model. Among these features is its hierarchical token ring network and the local sub-caches associated with each processor.

Each KSR processor cell includes of a 0.5MByte sub-cache, half of which is dedicated to instruction, the other half to data. The performance of the system will hence also depend on how efficiently the local sub-cache gets utilized. Similarly, the performance of distributed memory systems with vector processors, will depend
on how efficiently the vector units at each processing node are utilized.

The KSR1's interconnection network consists of a ring of rings rings. At Cornell, we currently have 128 node KSR1 system made up of 4 32-node rings that are connected to a top-level communication ring. The “message passing” (actually hidden due to the shared memory programming environment) is further limited in that there may be no more than approximately 14 messages (tokens) on one ring at any given time; i.e. a maximum of 14 messages can fly among 32 processors simultaneously. This is a more severe limit than what is normally seen on distributed machines with I/O chips (e.g. standard rings, meshes and hypercubes). One normally expects a minimum of N messages (near-neighbor) to be handled simultaneous by an N-processor ring. The KSR does, however, have the advantage that the communication bandwidth of its ring is significantly higher than what one sees on typical ring architectures. Also, since the ring is unidirectional and each message-header has to make it back to the sender, there is virtually no difference in communication with the nearest neighbor versus communication with a node connected at the opposite end of the ring. However, major communication delays are experienced when having to access data located on other rings.

The calculations of the expected performance of each node is also complicated by the fact that 3 processors in each KSR ring have I/O devices attached making these nodes slower than the others. When running an application on all 32 ring nodes, this must hence be considered. KSR fortunately provides an environment variable that lets users avoid these special cells when performing benchmarks.

When analyzing various approaches to our problem, we shall start with the simplest model, i.e., one which assumes uniform processors and no communication overlap. Refinements will then be made as we build up the model and incorporate feed-back from our test runs. Our prototype implementations benchmarked in Chapter 6 did not take advantage of pre-fetching features.
5.2.2 Model parameters

The following parameters will be used in our model:

- “xx” below are replaced by one of the following:
  - \textit{part-push-v} : Update of particle velocities (includes charge gather).
  - \textit{part-push-x} : Update of particle positions.
  - \textit{scatter} : Calculation of each particle’s contribution to the charge density \( \rho \) based on its fixed simulation charge and location.
  - \textit{fft-solver} : Field solver using 2 FFT. May be implemented using 1D FFTs along each of the two simulation axes.
  - \textit{transpose} : Reordering of grid data points from being stored among the PEs as block-columns to block-rows, or \textit{visa versa}. In the case of 1 vector/PE, this is a straightforward vector transpose.
  - \textit{grid-sum} : Add up local grid copies into a global grid sum.
  - \textit{border-sum} : Add temporary border elements to global grid.
  - \textit{part-sort} : Procedure for relocating particles that have left portion of the grids corresponding to their current node. This involves transfer of \( v \) and \( x \) data for these particles to PEs that have their new local grid quantities.
  - \textit{loc-ptcls} : accessing local arrays of particle quantities.
  - \textit{find-new-ptcls} : Search through out/scratch arrays for incoming particles.

- \( T \) : Total time – describes times which involve both computation time and communication time (where needed). The communication time is considered to be the time spent exchanging needed data among processing units.

- \( T_{xx} \) : Total time for the specific algorithm/method “xx”.

• $t_{comp}$ : Computation time of algorithm “xx”.

• $t_{comm}$ : Communication time of algorithm “xx”.

• $\alpha, \beta$ : Start-up time and bandwidth parameter, respectively, as defined above in Section 5.1.

• $t_{lm}(N)$ : Local Memory communication time involved in accessing a vector of length $N$ stored on a local node (see Section 5.1)

• $t_{gm}(N)$ : Global memory communication time involved in accessing a vector of length $N$ stored on a parallel system (see Section 5.1). Includes $t_{lm}(N)$ for the specified vector.

• $P$ : Number of processing nodes available.

• $P_x$ : Number of processing nodes in the $x$-direction when using a sub-grid partitioning.

• $P_y$ : Number of processing nodes in the $y$-direction when using a sub-grid partitioning ($P = P_x * P_y$).

• $N$ : Generic integer for algorithmic complexity. E.g. $O(N)$ implies a linear-time algorithm.

• $N_p$ : Number of super-particles in simulation.

• $max_{loc}N_p$ : Maximum number of particles on any given processor.

• $N_{p_{\text{moved}}}$ : Total number of particles that moved within a time-step.

• $N_x$ : Number of grid points in the $x$-direction (rectangular domain).

• $N_y$ : Number of grid points in the $y$-direction (rectangular domain).

• $N_g$ : Total number of grid-points in the $x$ and $y$-directions. $N_g = N_x * N_y$.

• $c_{calc}$ : Calculation constant characterizing the speed of the local node.
5.2.3 Result Summary

table 5.1 Show a summary of the performance complexity figures for the three approaches analyzed. These approaches are 1) serial PIC on a system with a local cache, 2) parallel algorithm using a fixed particle partitioning and a replicated grid with a parallel sum, and 3) a parallel algorithm using a fixed grid partitioning that automatically partially sorts the local dynamic particle arrays.

From this table one can see that given that one is limited in how much fast local cache memory one can have per processor, the serial code not only suffers from having only one CPU, but also suffers when the data sets get large and no longer fit in cache ($t_{lm}$ arguments are large). One can also see that the replicated grid approach will suffer from the $t_{grid-sum}$ computation and communication overhead for large grids ($N_g$) on a large number of processors ($P$). The fact $max_{loc} N_p$ will clearly hamper the grid partitioning approach for load-imbalanced systems with $max_{loc} N_p \gg \frac{N}{P}$ for a good portion of the time-steps. Chapter 4 discussed ways to compensate for this imbalance by re-partitioning the grid (and re-sorting the particles).

The following sections describe how we arrived at the equations in Table 5.1 and discusses further issues associated with each approach.

5.3 Serial PIC Performance

Following is an analysis of the serial algorithm. Since it is assumed to run on only one node, “communication” time will here be assumed limited to local memory accesses.

In large simulations that use more than the the 32Mb local memory, this will not be the case. The shared memory programming environment will then store some of the data on other nodes. This can be considered equivalent to swapping data to disk on smaller serial systems, except that the KSR uses fast RAM on a high-speed ring instead of slow I/O devices.
Table 5.1: Performance Complexity – PIC Algorithms

<table>
<thead>
<tr>
<th></th>
<th>Serial w/ cache</th>
<th>Particle Partitioning w/ parallel sum</th>
<th>Grid Partitioning with automatic partial particle sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clutter field at particles and update velocities</td>
<td>$t_{comp} = C(N_p)$</td>
<td>$t_{comp} = C(N_p)$</td>
<td>$t_{comp} = C(\max_{i=1}^n N_p)$</td>
</tr>
<tr>
<td></td>
<td>$t_{comm} = Ct_{tm}(N_g) + Ct_{tm}(N_p)$</td>
<td>$t_{comm} = Ct_{tm}(N_p)$</td>
<td>$t_{comm} = Ct_{tm}(N_p)$</td>
</tr>
<tr>
<td>Update article positions</td>
<td>$t_{comp} = C(N_p)$</td>
<td>$t_{comp} = C(N_p)$</td>
<td>$t_{comp} = C(\max_{i=1}^n N_p)$</td>
</tr>
<tr>
<td></td>
<td>$t_{comm} = Ct_{tm}(N_p)$</td>
<td>$t_{comm} = Ct_{tm}(N_p)$</td>
<td>$t_{comm} = Ct_{tm}(N_p)$</td>
</tr>
<tr>
<td>Scatter particle charges to grid (charge densities)</td>
<td>$t_{comp} = C(N_p)$</td>
<td>$t_{comp} = C(N_g \log P)$</td>
<td>$t_{comp} = C(\max_{i=1}^n N_p + N_g \log P)$</td>
</tr>
<tr>
<td></td>
<td>$t_{comm} = Ct_{tm}(N_p)$</td>
<td>$t_{comm} = Ct_{tm}(N_p)$</td>
<td>$t_{comm} = Ct_{tm}(N_p)$</td>
</tr>
<tr>
<td>Need to re-arrange grid for FFT!</td>
<td>Not if start in same dimension; otherwise $C(t_{tm}(N_g))$</td>
<td>Not if do series sub-grids; otherwise $C(t_{tm}(N_g))$</td>
<td>Not if partitioned grid in only one dimension; otherwise $C(t_{tm}(N_g))$</td>
</tr>
<tr>
<td>3D FFT solver</td>
<td>$t_{comp} = C(N_g \log N_g)$</td>
<td>$t_{comp} = C(N_g \log N_g)$</td>
<td>$t_{comp} = C(N_g \log N_g)$</td>
</tr>
<tr>
<td></td>
<td>$t_{comm} = Ct_{tm}(N_g)$</td>
<td>$t_{comm} = Ct_{tm}(t_{comp}) + Ct_{tm}(N_g)$</td>
<td>$t_{comm} = Ct_{tm}(t_{comp}) + Ct_{tm}(N_g)$</td>
</tr>
<tr>
<td>Field- Grid calculations (Finite Differences)</td>
<td>$t_{comp} = C(N_g)$</td>
<td>$t_{comp} = C(N_g)$</td>
<td>$t_{comp} = C(N_g)$</td>
</tr>
<tr>
<td></td>
<td>$t_{comm} = Ct_{tm}(N_g)$</td>
<td>$t_{comm} = Ct_{tm}(N_g)$</td>
<td>$t_{comm} = Ct_{tm}(N_g)$</td>
</tr>
</tbody>
</table>
5.3.1 Particle Updates – Velocities

The particle routine for pushing the velocities gathers the fields at each particle and then updates their velocities using this field. The field gather is a bilinear interpolation that consists of 7 additions and 8 multiplications. The number of additions can be reduced by 2 by using temporary variables for the weighing sums \((h_x-a)\) and \((h_y-b)\). The velocity update \(v = v + [q*E_{part}*(1/m)*\Delta t - (drag*v*\Delta t)\)

can similarly be optimized to use only 2 multiplications and 2 additions for each dimension. We hence have:

\[
T_{\text{serial-part-push-v}} = t_{\text{comp-gather}} + t_{\text{comp-part-push-v}} \quad (5.1)
\]

\[
+ t_{\text{comm-gather}} + t_{\text{comm-part-push-v}} \quad (5.2)
\]

\[
= N_p * c_{1\text{calc}} + N_p * c_{2\text{calc}} \quad (5.3)
\]

\[
+ t_{\text{lm}}(N_g) + t_{\text{lm}}(N_p) \quad (5.4)
\]

\[
= O(N_p) + O(t_{\text{lm}}(N_g) + t_{\text{lm}}(N_p)) \quad (5.5)
\]

where \(c_{1\text{calc}}\) and \(c_{1\text{calc}}\) is the time spent doing the multiplicationss and additions associated with gather and the velocity updates, respectively.

Notice that if the particles are not sorted with respect to their grid location, \(t_{\text{lm}}(N_g)\) would imply a lot of local cache hits (assuming the grid is too large to fit in cache). If the grid is stored either row or column-wise, each gather will also imply a read from two different grid cache-lines when the rows or columns exceed the cache-line size, unless a reordering scheme such as cell-caching is done. Cell-caching was introduced in Chapter 4 and its complexity is discussed later in this chapter.

5.3.2 Particle Updates – Positions

The particle routine for pushing the locations updates each location with the following 2D multiply and add: \(x = x + v \cdot \Delta t\), again giving:

\[
T_{\text{serial-part-push-x}} = t_{\text{comp-part-push-x}} + t_{\text{comm-part-push-x}} \quad (5.6)
\]
\[ N_p \times c_{\text{calc}} + t_{\text{lm}}(N_p) \]
\[ = O(N_p) + O(t_{\text{lm}}(N_p)) \]

where \( c_{\text{calc}} \) is the time spent doing the above multiplications and additions.

### 5.3.3 Calculating the Particles’ Contribution to the Charge Density (Scatter)

Here each particle’s charge gets scattered to the 4 grid corners. This operation requires 8 additions and 8 multiplications, where, as in the gather case, 2 additions can be saved by using temporary variables for the weighing sums \((h_x - a)\) and \((h_y - b)\). We hence have:

\[
T_{\text{serial-charge-gather}} = t_{\text{comp-charge-gather}} + t_{\text{comm-charge-gather}}
\]
\[
= N_p \times c_{\text{calc}} + t_{\text{lm}}(N_p)
\]
\[
= O(N_p) + O(t_{\text{lm}}(N_p))
\]

where \( c_{\text{calc}} \) is the time spent doing the above multiplications and additions.

Like the gather case, each particle is likely to cause two cache-hits regarding grid points unless a reordering scheme such as cell-caching is employed.

### 5.3.4 FFT-solver

1D FFTs are known to take only \( O(N \log N) \) computation time. Our current serial code solves for the field according to the following FFT-based algorithm:

- **Step 1:** Allocate temporary array for storing complex column before calling FFT routine provided by Numerical Recipes.
- **Step 2:** Copy matrix into complex array. \( O(N_g) \) (memory access)
- **Step 3a:** Call complex FFT function row-wise. \( O(N_g \log N_x) \)
- **Step 3b:** Call complex FFT function column-wise.

(Now have FFT(charge density), i.e. \( \rho(kx, ky) \)) \( O(N_g \log N_y) \)
Step 4: Scale in Fourier mode (i.e. divide by $k^2 = kx^2 + ky^2$ and scale by $1/\text{eps}$ to get $\Phi(kx, ky)$). $O(N_g)$ additions and multiplications.

Step 5a: Call complex FFT function doing inverse FFT column-wise. $O(N_g \log N_g)$

Step 5b: Call complex inverse FFT function row-wise. $O(N_g \log N_g)$

Step 6: Transfer result back to real array. $O(N_g)$ (memory access)

Step 7: Obtain final result by scaling (multiplying) the result by $1/(N_X \times N_Y)$. $O(N_g)$ multiplications

The memory copying (including transposing) and scaling operations (Steps 1, 2, 4, 6 and 7) are all considered to be local and linear in number of grid points, i.e. they require $O(N_g)$ operations with no inter-processor data transfers taking place (hopefully).

The memory copying could be avoided by implementing a more tailored FFT. However, since we strongly recommend that vendor-optimized FFT routines be used (and these often use proprietary algorithms), we will stick to the simplified notion that a 2-D FFT generally takes $O(N^2 \log N)$ on a serial computer.

In the above algorithm there is an implicit transpose between steps 3a and 3b as well as 5a and 5b. Such transposes generally generate a lot of cache-hits. We hence have:

$$T_{\text{serial--fft--solver}} = t_{\text{comp--fft--solver}} + t_{\text{comm--fft--solver}}$$

$$= O(N_g \log N_g) + O(t_{\text{lm}}(N_g)),$$

where $O(t_{\text{lm}}(N_g))$ signifies the cache-hits associated with the transpose. The 2D FFT will also have some cache-hits associated with it.

### 5.3.5 Field-Grid Calculation

The field-grid calculation determines the electric field in each direction by using a 1-D finite difference equation of the potentials (on the grid) calculated by the field
solver. This involves 2 additions and two multiplications for each grid point for each direction, giving the following computation time:

\[ T_{\text{serial-field-grid}} = t_{\text{comp-field-grid}} + t_{\text{comm-field-grid}} \]  
\[ = N_g * c_{\text{calc}} + t_{\text{bm}}(N_g) \]  
\[ = O(N_g) + O(t_{\text{bm}}(N_g)) \]

Notice that since \( N_p \) generally is an order of magnitude or so larger than \( N_g \), we can expect this routine to be fairly insignificant with respect to the rest of the computation time. This is verified in Chapter 6.

### 5.4 Parallel PIC – Fixed Particle Partitioning, Replicated grids

As mentioned in Chapter 4, one way of parallelizing a PIC particle code is to replicate the grid array(s). This is done to avoid the write conflicts that may occur when particles in neighboring cells try to contribute to the same charge density grid points.

In the shared memory setting, we only need to physically replicate the charge density array. The other grids (the potential and the field grids) get copied in on read-access, and since they are only written to once for each time-step, we do not have to worry about write-conflicts causing erroneous results.

#### 5.4.1 Particle Updates – Velocities

The particle push routines are ideal candidates for parallelization since they perform independent calculations on each particle. Since we assume the same nodes always process the same particles, parallelization is achieved by parallelizing the global particle loop. We hence have

\[ t_{\text{comp-replicated-grid-push-v}} = \left( \frac{N_p}{P} \right) * c_{\text{calc}} \]  
\[ = O\left( \frac{N_p}{P} \right). \]
Notice that there is no inter-process communication associated with writes. However, since the particles tend to become randomly distributed in space (if they were not already), and since a processing element is always responsible for updating the same group of particles, regardless of the particle grid location, the reads associated with gathering the field at the particles most likely will cause cache-hits for large grids since it is unlikely the whole field grid will fit in the subcache.

Regardless, the replicated grid method is likely to cause the entire grid to get copied into local memory as particles are scattered all over the system. This can be viewed as another level of cache hit:

\[
T_{\text{comm-replicated-grid-push-v}} = t_{\text{grid}} + t_{\text{loc-pcels}} \\
= O(t_{gm}(N_g)) + O(t_{lm}(\frac{N_p}{P})).
\]

Assuming the grid was distributed by the solver and field-grid phases into block-columns or block-rows, this means that when using large systems spanning several rings, these copies will occur as a hierarchical broadcasts incorporated in the function \(t_{gm}\). However, since we can assume the entire grid eventually will be needed, we can take advantage of pre-fetching if such operations are available, and if the grid will fit in local memory.

5.4.2 Particle Updates – Positions

Since the particle position updates are completely independent both of one another and the grid, this routine can be “trivially” parallelized, and near-perfect linear speed-up expected:

\[
T_{\text{replicated-grid-push-loc}} = t_{\text{comp-push-loc}} + t_{\text{comm-push-loc}} \\
= O(\frac{N_p}{P}) + O(t_{lm}(\frac{N_p}{P})).
\]
\((O(t_{\text{lm}}(\frac{N_p}{P})))\) are here minimal since the particle partitioning is fixed. The particle arrays can hence be sequentially accessed.

### 5.4.3 Calculating the Particles’ Contribution to the Charge Density

Once the particle positions are determined, one can calculate the charge density by accumulating the charges each particle represent on the grid partitioning the simulation space. Hence, each particle’s charge gets scattered to the 4 grid corners of the grid cell containing the particle. Since there are four cells associated with each grid point, write-conflicts are likely to occur since the particle partitioning is fixed. These write-conflicts are, however, avoided by replicating the grids in the computation phase. However, after these local charge density arrays have been computed, they all need to be summed up into a global array for the solver. This sum could be done either serially or in parallel using a tree structure.

The serial sum would require \(O(N_g \cdot p)\) additions plus \(P - 1\) block-transfers of \(N_g\) elements. These transfers, like the ones happening in the push-v phase may occur as *hierarchical gathers*. Again, since we know in advance which data we will be needing, we can take advantage for pre-fetching if such operations are available.

If a global parallel tree sum was used, \(O(N_g \cdot \log p)\) additions would be required. Notice, however, that if \(p = 128\), \(\log p = 7\), which means \(O(N_g \cdot \log p)\) may approach \(N_p\) (assuming 10 or so particles per cell)! Our benchmarks in Chapter 6 illustrate this phenomenon.

If one processing node were to gather the whole result, this would cause network traffic for the solver since the solver would be using a distributed grid. This could be avoided if the sums were arranged to be accumulated on subgrids. The root of each summation tree will here correspond to the final destination of the subgrid.

Notice that this split still causes the same sum complexity:

\[
t_{\text{comp-parallel-sum}} = O(P \cdot N_g / P \cdot \log P) = O(N_g \cdot \log P)
\]
We hence have:

\[
T_{\text{replicated-grid-scatter}} = t_{\text{comp-scatter}} + t_{\text{comp-grid-sum}} \\
+ t_{\text{comm-scatter}} + t_{\text{comm-grid-sum}} \\
= O\left(\frac{N_p}{P}\right) + O(N_g \log P) \\
+ O(t_{\text{lm}} \left(\frac{N_p}{P}\right)) + O(t_{gm} (M_g \log P))
\]

\[\text{(5.24)}\]
\[\text{(5.25)}\]
\[\text{(5.26)}\]
\[\text{(5.27)}\]

### 5.4.4 Distributed memory FFT solver

Since the FFT solver is completely independent of the particles, its parallelization can also be done separately. In fact, it is highly recommended that one takes advantage of vendor-provided parallel libraries whenever available for this case. Notice that the communication involved in going from one dimension to the other basically involves transposing a distributed grid.

**Transpose**

Assuming one vector per processor, the most obvious way to do a ring transpose is to at the first step, send out the whole vector except the entry that the node itself will be using in the row (column) operation. If a full I/O ring was available, this would imply taking communication time \(= \alpha + (N - 1)\beta\) on a N-by-N grid. There should be no conflicts since the nodes will be sending this data to their nearest neighbor.

For each step, one entry then gets “picked off” until every node has received the intended row (or column) (see example below). The average message length would be \(N/2\), but the start-up time would each time be \(\alpha\). This gives \(N\alpha + \left(\frac{N^2}{2}\right)\beta\) total communication time. Figure 5.1 depicts the data movements in a transpose of a 4-by-4 matrix on a 4-processor ring.

I.e. \(N = P = 4\), giving us a communication time of \(4\alpha + 4 \cdot 2\beta\) (or \(4\alpha + 8\beta\)) i.e the transpose is \(O(N^2)!!\) This is worrisome with respect to the \(O(N \log N)\) FFTs
Figure 5.1: Transpose of a 4-by-4 matrix on a 4-processor ring.

computational time, especially since computational time is usually a lot less than communication time.

Over-all 2D-FFT complexity

Total communication costs for “transpose” by following the ring approach above, we get:

\[ t_{\text{comm, transpose}} = N\alpha + \frac{N^2}{2} \beta \]

This is the communication time it would take to re-shuffle N-vectors. Note the above equations assume that there is exactly one N-vector per processor. This gives a total communication time for the FFT solver \((P = N_x = N_y)\):
\[ t_{\text{comm-fft}} = p \alpha + \frac{N_g^2}{2} \beta \]

Total time for a 2D parallel FFT with one \( N \)-vector per processor will then be:

\[ T_{\text{parallel-2D-fft}} = t_{\text{comp-2D-fft}} + t_{\text{comm-2D-fft}} \]
\[ = 2(N^2 \log N) \ast c_{\text{calc}} + N_g \alpha + \frac{N^2}{2} \beta \]

assuming the data-blocks are bunched together so that the number of vectors within a computation phase does not affect the start-up time. Here \( c_{\text{calc}} \) is a constant indicating the speed of the processor. Note that since this analysis did not take into consideration the number of memory accesses made \textit{versus} the number of floating point calculations made, this parameter would have to be some statistical average of the two in order to be meaningful. Local caching considerations could also be figured into this parameter.

Assume now a \( N \times N \) grid on a \( P \) processor ring, where \( \frac{N}{P} = k \), \( k \) an integer. This means that each processor has a computation phase that performs \( \frac{N}{P} (N \log N) \) 1-D FFTs.

Each communication phase then transfers ("transposes") on the average \( \frac{N}{P} \frac{N}{2} \) blocks of data \( P \) times.

Each processing unit in the computation phase does:

\[ t_{\text{comp-fft}} = \frac{N}{P} (2 \ast N \ast \log N) \]

computations.

Each communication phase then transfers ("transposes") on the average \( \left( \frac{N}{P} \right) \ast \left( \frac{N}{P} \right) \) blocks of data during the \( P \) phases. This gives total communication time (multiplying above with \( \frac{N}{P} \)):

(Note: \( \left( \frac{N}{P} \right) \ast P = N_g \))

\[ t_{\text{comm-fft}} = N \ast \alpha + \frac{N^2}{2} \ast \beta. \]
The total time estimate for a 2D FFT is hence:

\[
T_{2D-fft} = O \left( P \cdot \frac{N_g}{P} \cdot \frac{N_g}{2} \cdot c_{calc} \right) + O \left( N_g \cdot \alpha + \frac{N_g^2}{2} \cdot \beta \right) \quad (5.30)
\]

\[
= O \left( N_g \left( \frac{N_g}{P} \log N_g + \alpha + N_g \beta \right) \right) \quad (5.31)
\]

When several rings are involved, another memory hierarchy gets added. Similarly, for truly large systems, one 1D FFT may not fit in a cache line and additional performance hits will hence be taken.

**Parallel 2D FFT Solver**

A parallel 2DD FFT solver involves two of the above parallel 2D FFTs (counting both the regular FFT and the inverse FFT) and a scaling of $1/(N^2)$ for each grid quantity. A factor of 2 does not show in the complexity equations, in fact, even the scaling of $O(N^2)$, can be considered included in with the 2D FFT ($O(N^2 \log N)$).

Assuming we have a grid of size $N_g = N^2$, we hence have the following complexity equations for the 2D FFT solver:

\[
T_{parallel-fft-solver} = t_{comp-fft-solver} + t_{comm-fft-solver} \quad (5.32)
\]

\[
= O(N_g \log N_g) + O(t_{lm}(N_g \log N_g)) + O(t_{gm}(N_g)) \quad (5.33)
\]

where $O(t_{gm}(N_g))$ signifies the cache-hits and memory traffic associated with the transpose. The 2D FFT will also have some cache-hits associated with it.

**5.4.5 Parallel Field-Grid Calculation**

The finite difference equations associated with the field-grid calculations parallelize fairly straightforwardly. Since the input grid is the potential grid supplied by the solver, it makes sense to partition the interior loop into comparable block-vectors. The border cases may be parallelized separately. One should hence be able to achieve:

\[
T_{parallel-field-grid} = t_{comp-field-grid} + t_{comm-field-grid} \quad (5.34)
\]
The field-grid calculations may cause some read cache hits for the next phase if the pusher is not going to be using block-vector partitioning.

5.5 Parallel PIC – Partitioned Charge Grid Using Temporary Borders and Partially Sorted Local Particle Arrays

In this section we will analyze the grid partitioning approach outlined in Chapter 4 which replicates the grid borders and uses dual pointers on the local particle arrays in order to maintain an automatic partial sorting of the particles.

5.5.1 Particle Updates – Velocities

The total time of the update of the local particle velocity arrays is equal to the maximum time required on one node, i.e.:

\[ t_{\text{comp-serial-part-push-v}} = \max_{\text{local \_Np}} N_p * c_{\text{calc}} \]
\[ = O(\max_{\text{local \_Np}} N_p). \]  

Adding in the communication time associated with the gather and the velocity updates gives us the following equation:

\[ T_{\|\text{-push-v}} = t_{\text{comp-gather}} + t_{\text{comp-part-push-v}} \]
\[ + t_{\text{comm-gather}} + t_{\text{comm-part-push-v}} \]
\[ = \max_{\text{loc \_Np}} N_p * c_{\text{calc}} + \max_{\text{loc \_Np}} N_p * c_{2\text{calc}} \]
\[ + t_{\text{lm}}(N_g) + t_{\text{lm}}(N_p) \]
\[ = O(\max_{\text{loc \_Np}} N_p) + O(t_{\text{lm}}(N_g) + t_{\text{lm}}(\max_{\text{loc \_Np}})) \]

where \( c_{\text{calc}} \) and \( c_{2\text{calc}} \) is the time spent doing the multiplications and additions associated with gather and the velocity updates, respectively. Notice that this
version suffers a lot fewer cache-hits in the gather phase than the replicated grid method. Since the particles are already partially sorted, only local grid points or those right on their borders, contribute to the field at each particle ($t_{versus gm}$).

### 5.5.2 Particle Updates – Positions

If the particles are stored in local arrays using a read and write pointer on the local position array(s) and a write pointer for an output/scratch array, then one can check whether the particles’ new locations are still within the local grid, and based on this check, either write them back in to the local array or out to the global scratch array. These operations may be completely local if each node has its own scratch area to write to – hence the use of an extra write pointer per node for the out/scratch array. (The nodes may otherwise have to spin-lock on a global write pointer.) This part of the codes hence takes:

$$t_{push-local-pts} = \max_{local} N_p \cdot \text{calc}$$  \hspace{1cm} (5.44)

$$= O(\max_{local} N_p).$$  \hspace{1cm} (5.45)

If the particle arrays are perfectly load-balanced, $O(\max_{local} N_p) = O(N_p/P)$.

It is highly unlikely that a large number of particles will leave on any given time-step, since most particles should be moving within each subdomain if accurate (and efficient) results are to be obtained. How bad $O(\max_{local} N_p)$ will get depends on how inhomegenous the problem being simulated gets and whether dynamic grid-reordering (global sorts needed) is done to compensate to these imbalances.

The above figures do not include the testing of the new particle location or the cost in also having to auto-sort the velocities so that they correspond to the particle locations. The latter would involve $O(\max_{local} N_p)$ local read/writes.

Notice that if the grid is divided up along only one dimension, then only one particle location index has to be tested.

To account for possible incoming particles, the nodes now have to check each
other’s out/scratch arrays. This will involve:

\[
    t_{\text{find-new-particles}} = t_{\text{tm}}(\max_{local} N_p)[\text{writes}]
    + t_{\text{gm}}(N_p, \text{move})[\text{reads}]
\]

(5.46)  

(5.47)

Notice that the reads involve read-transfers from other nodes. How much time
these updates will take depends on how many particles leave their local domain.

The number of particles leaving their local domain each time-step depends on
the grid partitioning, the distribution of the particles and their initial velocities.
If the particles are uniformly distributed, the optimal grid partitioning would be
square subgrids (see Chapter 4, Section 3.4). Notice that this partitioning is con-
flicting with the one desirable for the FFT solver which requires a block-vector
partitioning (Figure 5.2).

![Grid Partitioning](image)

a) block-vector  

b) subgrid

Figure 5.2: Grid Partitioning: a) block-vector, b) subgrid

A small savings will be achieved when using block-vector partitioning in the
test for particle locations since only one dimension has to be checked. However,
this may be far out-weighed by the penalty for, in the block-vector case, having
larger borders with fewer interior points than the subgrid partitioning.

Assume that the average probability that the local particles on a processing
node leave their cell after each time-step is \( P(\text{loc}_N_p, \text{leave}_N) \). Assuming one row or column of grid-cells per processor, then about 50% of those particles that leave their local cell, will still remain on the local processor (ignoring the ones leaving diagonally off the four corners). See Figure 5.3.

![Figure 5.3: Movement of particles leaving their cells, block-vector setting.](image)

This means that \( P(\text{loc}_N_p, \text{leave}_N) / 2 \) particles must be transferred to another processor.

However, if a sub-matrix partitioning was chosen for the grid points in the particle push phase, then assuming the particles will not move more than one cell over, only the border cells will be contributing particles to be moved to another processing node.

The probability of particles leaving is a function of the initial velocities and positions of the particles and the size of each cell. The question then becomes: how “bad” is the distribution of particles, and how likely is this “bunch” to leave its current processor?

Notice also that if one could anticipate the over-all drift of the particles, the grid partitioning could adapt to the drift. E.g. if the particles are generally drifting in the x-direction, then it would be very advantageous to partition the grid as block-rows and thereby significantly reducing the need for particle sorting (re-location of particles) after each time-step. However, in plasma simulations there is usually considerable up-down motion, even though there is left-right drift present.

The total time taken, aside from local read/writes and dynamic load-balancing
steps, will be:

\[ T_{\text{push-z}} = t_{\text{comp-push-z}} + t_{\text{comm-loc-ptcls}} + t_{\text{comm-find-new-ptcls}} \]  
\[ \text{where } c_{\text{calc}} \text{ includes the additions and multiplications associated with the particle position updates as well as the tests for whether the particles' new positions still are within the local grid domain of their processing node. The searches through the output/scratch arrays are included as } t_{\text{gm}}(N_{\text{p,moved}}). \]

5.5.3 Calculating the Particles’ Contribution to the Charge Density

In this routine each particle’s charge is scattered to the four grid points surrounding its cell, where the charge is then added to the charge density. Since the particles share the grid points with particles in the nearest neighboring cells, some overhead must be expected in overcoming the potential write-conflicts that will result when the grid is distributed. (See Figure 5.4)

A standard approach would be to use a global lock on the border elements, however, this will cause several busy-waits unless the particles are completely sorted.

Another approach would be to use the idea from replicated grids, i.e. replicated borders. Notice that only one border for each dimension needs to be replicated since the other local border may as well be the master copy. When calculating the particles’ contribution to the overall charge density, we first calculate them as if the extra border array was the top border (simplifies implementation for periodic systems). These border arrays then get added to the master copy. Since only one node has an extra copy of any given border, there will be no write conflicts, so these can be done in parallel. However, there will be network traffic involved getting the corresponding master copy values to be added in.
Figure 5.4: Parallel Charge Accumulation: Particles in cell ‘A’ share grid points with the particles in the 8 neighboring cells. Two of these grid points, ‘a’ and ‘b’, are shared with particles updated by another processing node.

\[
T_{\text{parallel-scatter}} = t_{\text{comp-scatter}} + t_{\text{comp-border-sum}} + t_{\text{comm-scatter}} + t_{\text{comm-border-sum}}
\]

\[
= O(\max{\text{loc}}_N) + O(\max{\left(\frac{N_x}{P_x}, \frac{N_y}{P_y}\right)})
\]

\[
+ O(t_{\text{lm}}(\max{\text{loc}}_N)) + O(t_{\text{gm}}(\max{\left(\frac{N_x}{P_x}, \frac{N_y}{P_y}\right)}))
\] (5.54)

Notice that the size of the border(s) depends on the partitioning. If the domain is only partitioned in on direction (block-vector), say in y, then the border will be \(\max{\left(\frac{N_x}{P_x}, \frac{N_y}{P_y}\right)} = N_x/1 = N_x\) and the total extra space allocated for these borders is \(P \times N_x\). Notice how even this is considerably less than what the replicated grid uses \((P \times N_y \text{ when } N_y \gg P)\).

As in the particle push phase, the optimal case for simulations with uniform particle distributions, will be square sub-partitioning.
5.5.4 FFT solver

Despite the fact that the block-vector partitioning is far from optimal for the other parts of the code, it is probably inadvisable to try to any other partitioning for the FFT phase. Most vendors provide very efficient hand-coded FFT routines that would probably be a lot faster than user-coded versions. Ideally parallel 2D FFTs and inverse FFTs are available and all one has to worry about is accessing/arranging the data in the appropriate order. If only 1D FFT routines are available, the serial local version may be used as a building block as described in Section 5.4. Regardless, the overall time for a parallelized solver should be that of the solver used in the replicated grid case.

A re-arrangement of the grid from a subgrid partitioning to a block-vector partitioning (or vice versa) will require time $t_{gm}(N_g)$.

5.5.5 Parallel Field-Grid Calculation

The finite difference equations associated with the field-grid calculations parallelizes as for the replicated grid case:

$$T_{parallel-field-grid} = t_{comp-field-grid} + t_{comm-field-grid}$$

$$= \left(\frac{N_g}{P}\right) \times c_{calc} + t_{lm}(\frac{N_g}{P})$$

$$= O\left(\frac{N_g}{P}\right) + O(t_{lm}(N_g))$$

As before, the field-grid calculations may cause some read cache hits for the next phase if the pusher is not going to be using block-vector partitioning.

5.6 Hierarchical Data structures: Cell-caching

Grids are typically stored either column or row-wise. However, in a system with memory hierarchies, especially a systems with caches, we showed in Chapter 4 that these are not necessarily the best storage schemes for PIC codes. Instead,
one should try to build a data structure that minimizes the number of cache-lines needed for each scatter (calculation of a particle’s contribution to the charge density).

When the particle’s charge contributions are collected back on the grid points, each particle will be accessing all four grid points of its cell (assuming a quadrangular grid structure). If one uses a column or row storage of the grid, one can hence expect the need to access two cache-lines of grid quantities for each particle. E.g. for a row-wise stored grid there will typically be one cache-line containing the top two corners and one containing the bottom two corners.

If one instead uses the cell-caching strategy of Chapter 4, where one tries to fit as many cells as possible within a cache line, the number of cache-hits can be reduced. In this case, square subdomains of the grid are fitted into the cache rather than rows or columns (or block-rows and block-columns). For 3D codes, sub-cubes should be accommodated as well as possible. In other words, whatever access pattern the code uses, the cache use should try to reflect this.

Since a cache-line on the KSR is 128 bytes, i.e. 16 64-bit floating point numbers, each cache-line can accommodate a 4-by-4 subgrid. Figure 5.5 shows the number of cache-hits associated with each cell in this case.

This means that a column/row storage approach would use 2 x 16 cache-hits whereas cell-caching would use \([3x3]*1 + (3+3)*2 + 4 = 25\) cache-hits, an improvement of more than 25%!

In general, if the cache size is \(C_x \times C_y\), then the total number of cache-hits per cell is:

\[
Total_{\text{cache\_hits\_per\_cell}} = (C_x - 1)(C_y - 1) \times 1 \text{ (interior points)}. \quad (5.58)
\]
\[
+ (C_x - 1) + (C_y - 1) \times 2 \text{ (borders)} \quad (5.59)
\]
\[
+ 1 \times 4 \text{ (corner)} \quad (5.60)
\]

If \(C_x = C_y = C\), the optimal situation for cell-caching, the above equation
becomes:

$$Total_{cache\_hits\_per\_cell} = C^2 + 2C + 1 \quad (cell – caching) \quad (5.61)$$

In order not to have to change all the other algorithms, cell-caching does, however, require more array indexing. To access array element $A(i,j)$, we need:

$$A(i, j) = A \left[ \left( \left( \frac{i}{C} \right) \times \left( C^2 \times C^2 \right) \right) + \left( \frac{j}{C} \times C^2 \right) \right] + \left( j \mod C \right) + \left( i \mod C \right) \times C \right] \quad (5.62)$$

Although this requires many more operation than the typical $A(i, j) = A[i \times N_x + j]$, most of these operation can be performed by simple shifts when $C$ is a power of 2. Compared to grabbing another cache-line (which could be very costly if one has to get it from across the system on a different ring) this is still a negligible cost.

It should be noted that the cell-cashed storage scheme conflicts at the local level with the FFT solver. The over-all grid could still be stored block-column, or rather block-column-cell-caches to avoid communication costs when re-ordering the grid.

○ = grid points stored in cache-line

Figure 5.5: Cache-hits for a 4x4 cell-cached subgrid
In the replicated grid case, the re-orderings could be incorporated in the final stage of the grid summation and as part of the field-grid calculations with the help on a temporary grid array.

It should be emphasized that cell-caching is a local memory construct that benefit *serial* as well as parallel machines with local caches.
Chapter 6

Implementation on the KSR1

“One must learn by doing the thing; though you think you know it, you have no certainty until you try.” – Sophocles [495-406 B.C.], Trachiniae.

This chapter describes our implementations of some of the ideas presented in Chapters 4 and 5. Our test-bed is the Kendall Square Research machine KSR1 currently available at the Cornell Theory Center.

Although the KSR1 physically has its chunks of 32 Mb main real memory distributed among its processors, to a programmer it is addressed like a shared-memory machine. However, serious performance hits can be experienced if one does not pay careful attention to memory locality and caching. For optimal performance one would hence have to consider it a distributed memory machine with respect to data locality. By also having a good understanding for how the memory hierarchy works, delays due to swapping and data copying can be minimized. Many of these issues were discussed in Chapter 4 and 5.

The configuration in this study has 128 processor nodes with 32Mb of real memory on each processing node.
6.1 Architecture overview

Each KSR processor cell consists of a 0.5Mb sub-cache, half of which is dedicated to instruction, the other half to data. Its 64-bit 20 MIPS processing units execute two instructions per cycle giving, according to the manufacturer, 40 peak MFLOPs per cell (28 MFLOP FFT and 32 MFLOP matrix multiplication).

Communication between the processing cell and its local memory (cache) is said to perform at 1 Gb/second, whereas the standard i/o channel are listed as 30 Mb/second. The cells experience the following memory latencies:

- sub-cache (0.5 Mb): 2 cycles or .1 micro second;
- local cache (32 Mb): 20-24 cycles or 1 micro second
- other cache on same ring (992 Mb): 130 cycles or 6.5 micro second;
- cache on other ring(s) (33,792 Mb): 570 cycles or 28.5 micro second;
- external disks (variable): 400,000 cycles! or approximately 20 msec.

Our test-bed currently has 10 nodes with i/o devices. Care must hence be taken when timing these nodes. For the benchmarks presented in this chapter, a system variable was set so that the threads only ran on nodes without i/o devices.

The long disk latency is due almost entirely to the disk-access speed. It should be clear from this figure that programs than need so much memory that frequent disk I/O is necessary during computation, will take an inordinate amount of time. In fact, one of the strengths of highly parallel computers is not only their potentially powerful collective processing speed, but the large amounts of real memory and cache that they provide.

6.2 Some preliminary timing results

To get a better feel for the performance of the KSR, we first ran a serial C-implementation of the standard BLAS routine SSCAL (vector-times-scalar) on a
series of local platforms. It should be noted that since we performed this test with unit stride, this lets the KSR take advantage of caching. However, no cache re-use happened as each data element was used only once.

As can be seen from the results in Table 6.1, the KSR’s scalar units out-perform the Sun SPARC 4m/670 MPs (although not the DEC Alpha\(^1\)) as long as the problem was small enough. For larger problems, the load seemed to play a role giving us wide-ranging timing results for the same problem. This could be due to caching as well as the KSR OS’s attempt to ship our process dynamically to another node. These are clearly issues that also will need to be considered for parallel implementations.

\(^1\)The DEC Alpha results where obtained on a loaner machine from Digital Equipment Corporation with no other users present. Along with later results, the timings obtained seem to indicate that its scalar speed is about 3 times that of a KSR node and about 4 times that of a Sun 4m/670MP—very impressive indeed.

<table>
<thead>
<tr>
<th>Arch.:</th>
<th>Sun 4c Sparc 1 (ottar.cs)</th>
<th>Sun 4c Sparc 2 (ash.tc)</th>
<th>Sun 4m/670 MP (leo.cs)</th>
<th>DEC Alpha (sitar.cs)</th>
<th>KSR-1 (homer.tc)</th>
</tr>
</thead>
<tbody>
<tr>
<td>load:</td>
<td>0.08 - 0.39</td>
<td>0.00 - 0.06</td>
<td>1.00 - 1.30</td>
<td>0.0 - 0.1</td>
<td>2.23 - 3.0</td>
</tr>
<tr>
<td>v-length</td>
<td>(secs)</td>
<td>(secs)</td>
<td>(secs)</td>
<td>(secs)</td>
<td>(secs)</td>
</tr>
<tr>
<td>100,000</td>
<td>0.2499</td>
<td>0.1000</td>
<td>0.1000</td>
<td>0.0167</td>
<td>0.0000</td>
</tr>
<tr>
<td>0.2666</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>200,000</td>
<td>0.500</td>
<td>0.217</td>
<td>0.200</td>
<td>0.050</td>
<td>0.120</td>
</tr>
<tr>
<td>0.500</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>400,000</td>
<td>0.98</td>
<td>0.42</td>
<td>0.38</td>
<td>0.07</td>
<td>0.22</td>
</tr>
<tr>
<td>0.98</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1,000,000</td>
<td>2.43</td>
<td>1.06</td>
<td>1.02</td>
<td>0.20</td>
<td>0.56</td>
</tr>
<tr>
<td>2.45</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2,000,000</td>
<td>4.85</td>
<td>2.12</td>
<td>1.97</td>
<td>0.40</td>
<td>1.10</td>
</tr>
<tr>
<td>5.07</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4,000,000</td>
<td>12.87</td>
<td>4.25</td>
<td>4.07</td>
<td>0.82</td>
<td>5.44</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8,000,000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.65</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.67</td>
</tr>
</tbody>
</table>
6.3 Parallel Support on the KSR

The KSR offers a parallelizing Fortran compiler based on Presto for standard parallelizations such as tiling. Presto commands then get converted to Pthreads which are then again sitting on top of Mach thread. The C compiler uses Pthreads (Posix P1003.4 thread) directly leaving it to the programmer to do the actually thread manipulations. One may also use SPC (Simple Presto C) which provides a simplified mechanism when using functions that get called by a team of threads (generally, one per processing node). The SPC teams provides an implicit barrier right before and after each parallel call.

The KSR’s programmer’s memory model defines all threads in a single process to have a shared data area. Each thread also has a private data area. The memory system takes care of data replication, distribution, and movement, though it is highly useful to make note of how this is handled so that programs can be optimized to minimize memory transfers.

The KSR offers four Pthread constructs for parallelization: single Pthreads, barriers, mutex locks, and condition variables. Only one thread can call “fork & execute” for a given set of parallel threads. All subsequent threads are equal hierarchically, and any of these may again call a fork.

The barriers are implemented through calls to Barrier check_in and Barrier check_out. The former causes all slaves to wait for the designated master, the latter makes the master wait for all slaves to finish. The explicit KSR Pthread barrier construct hence operate as a “half” barrier. In order to obtain the full synchronization usually associated with the term barrier, a pthread_checkout (master waits for slaves) needs to be issued right before a pthread_checkin (slaves wait for master). SPC calls implicitly provide full barriers, i.e. a check-out followed by a check-in.
6.3.1 C versus Fortran

Fortran may appear to be the most natural language for implementing numerical algorithms. Although at present only the KSR Fortran compiler offers automatic parallelization, we have chosen to implement our code in C since it allows for more flexibility. C, with its powerful pointer constructs for dynamic memory allocation and strong link to UNIX, is also rapidly becoming more popular. Since we want to use some of the C pointer features in the implementation, C indeed becomes a natural choice. (The author expects the dynamic memory allocation feature to appear in future Fortran standards. If that happens, Fortran may again appear more attractive.) C interfaces well with both Assembly Language and Fortran, and is along with Fortran 77 and C++, the only languages currently available on our KSR1.

6.4 KSR PIC Code

6.4.1 Porting the serial particle code

The first thing we did after exploring some the Pthread features on test cases, was to port our serial particle code developed on the local Sun workstation to the KSR1. The only code change required to get it running correctly (aside from editing include-statements pointing to local library files) was to change the `printf` statements formatting argument “\%lf” and “\%le” to plain “\%f” and “\%e”. This is the result of moving from a 32-bit machine to a 64-bit machine and hence what the implementors notion of `double` is.

In order to get a better notion of what kind of performance we could hope for, benchmarks were obtained for the serial version of our particle code on the KSR. See Table 6.2. Notice that the difference between the 16 x 16 entries and the corresponding test-runs for 32 x 32 grid points with the same number of particles, is around 12-13 seconds for all of them. This should hence indicate the increased
solver time for going from a 16 x 16 grid to a 32 x 32 grid. Notice how this figure proves significant compared to the overall time for the runs with an average of 1-2 particles per cell, but becomes more negligible as the number of particles per cell increases.

Table 6.2: Serial Performance on the KSR1
100 time-steps, non-optimized code
Load = 1.5-3.0

<table>
<thead>
<tr>
<th>Run no.</th>
<th>Grid size</th>
<th>No. of particles</th>
<th>time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8 x 8</td>
<td>64</td>
<td>1.88</td>
</tr>
<tr>
<td>2a</td>
<td>16 x 16</td>
<td>1024</td>
<td>15.68</td>
</tr>
<tr>
<td>2b</td>
<td>16 x 16</td>
<td>2048</td>
<td>27.86</td>
</tr>
<tr>
<td>2c</td>
<td>16 x 16</td>
<td>4096</td>
<td>51.81</td>
</tr>
<tr>
<td>2d</td>
<td>16 x 16</td>
<td>8192</td>
<td>100.42</td>
</tr>
<tr>
<td>3a</td>
<td>32 x 32</td>
<td>1024</td>
<td>28.50</td>
</tr>
<tr>
<td>3b</td>
<td>32 x 32</td>
<td>2048</td>
<td>39.90</td>
</tr>
<tr>
<td>3c</td>
<td>32 x 32</td>
<td>4096</td>
<td>63.76</td>
</tr>
<tr>
<td>3d</td>
<td>32 x 32</td>
<td>8192</td>
<td>113.30</td>
</tr>
</tbody>
</table>

To gain further insight on how much each part of the code contributes to the overall time of the computations, benchmarks were obtain for runs simulating 16,384 particles over 100 time steps on both a Sun Sparc 1, Sun Sparc 460/MP, and the KSR1 (Table 6.2). In the optimized case, we removed divisions from particle loops by globally defining the appropriate inverses, and replacing the divisions with multiplications. As one can see from the results, the default KSR division in C
is really slow. By setting the -qdiv flag when compiling, we obtained significant speed-ups, but not quite as good as our optimized code (except for the solver, which we did not optimize).

Table 6.3: Serial Performance of Particle Code Subroutines
Sun 4c/60 Sparc 1, Sun Sparc 670/MP, and the KSR1
128 x 128 = 16K particles, 32-by-32 grid, 100 time-steps

<table>
<thead>
<tr>
<th></th>
<th>Sparc 1 slow div (sec)</th>
<th>670/MP slow div (sec)</th>
<th>KSR1 slow div (sec)</th>
<th>Sparc 1 optimized (sec)</th>
<th>670/MP optimized (sec)</th>
<th>KSR1 optimized (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>4.53</td>
<td>-</td>
<td>9.9</td>
<td>5.37</td>
<td>5.92</td>
<td>9.84</td>
</tr>
<tr>
<td>Pull-back</td>
<td>2.71</td>
<td>-</td>
<td>1.5</td>
<td>2.32</td>
<td>0.95</td>
<td>0.52</td>
</tr>
<tr>
<td>Scatter (Part-rho)</td>
<td>85.45</td>
<td>-</td>
<td>35.1</td>
<td>75.75</td>
<td>31.4</td>
<td>11.2</td>
</tr>
<tr>
<td>FFT Field solve</td>
<td>34.03</td>
<td>-</td>
<td>8.9</td>
<td>33.58</td>
<td>13.9</td>
<td>8.8</td>
</tr>
<tr>
<td>Update particle velocities (incl. gather)</td>
<td>152.4</td>
<td>-</td>
<td>108.20</td>
<td>120.70</td>
<td>49.4</td>
<td>32.8</td>
</tr>
<tr>
<td>Update particle locations</td>
<td>47.93</td>
<td>-</td>
<td>32.2</td>
<td>31.47</td>
<td>17.7</td>
<td>26.2</td>
</tr>
<tr>
<td>Field.grid (F to E)</td>
<td>1.70</td>
<td>-</td>
<td>1.66</td>
<td>1.08</td>
<td>0.5</td>
<td>0.22</td>
</tr>
<tr>
<td>Simulation</td>
<td>314.82</td>
<td>-</td>
<td>188.3</td>
<td>264.94</td>
<td>113.3</td>
<td>79.77</td>
</tr>
</tbody>
</table>

Like the vector code, these KSR figures also compare favorable to those obtained on Sun workstations. The Sparc 1 took, for our application, about 1.5 times longer, on the average, than the KSR serial runs with a moderate load.
These results were used when developing our strategies for Chapter 4 and 5. It is worth noting how little time the field-grid calculations takes compared with the rest of the code. Since this routine is a strictly grid-dependent (i.e. $O(N_g)$), this matches our analysis in Chapter 5 since there are a lot fewer grid-points than number of simulation particles, $N_p$, i.e. $N_g \ll N_p$, and all the other routines are dependent on $N_p$.

## 6.5 Coding the Parallelizations

Our first effort was to parallelize our code by partitioning the code with respect to particles. Since each of the particle updates are independent, this is a fairly “trivial” parallelization for these routines. For the charge collection phase, however, this means blocking and waiting for grid elements or replicating the grid.

### 6.5.1 Parallelization Using SPC

In order to achieve the particle parallelization, we first spawned a team of Pthreads using SPC (Simple Presto C) for each available processor. Our code hence uses implicit barrier constructs as depicted below through the `pr_call` calls. Although Pthreads are light-weight constructs producing very little overhead on creation, it is rather wasteful to spawn a thread for thousands of particles on only 64-256 processors. We hence group the particles using the `modulus` and `ceiling` functions when calculating the pointers to the particle number each thread starts at (similar to the partitioning of a random-sized matrix across a set of processors). This can either be done within a single system call using globals or using local system calls in each subroutine. We opted for the latter for the sake of modularity. Figure 6.1 shows the code segment for our main particle loop after adding SPC calls to parallelized particle update routines.
```c
/* *********************** main simulation loop: *********************** */
t = 0.0;
while (t <= t_max)
{
    /* printf("Apply field to particles and \n\n"); */
    /* printf("Calculate velocities ... \n\n"); */
    /* In parallel version: Use all teams to execute */
    PUSH_V(Ex, Ey, Np, Part_x, Part_y, Nx, Ny, hx, hy, drag, q, mass,
    t, Vx, Vy); in parallel.
    * External parameters are passed as arguments using pr_call. */
    pr_pcall( team_id, PUSH_V, copyargs(Ex, Ey, (double) Np, Part_x,
        Part_y, (int) Nx, (int) Ny, (double) hx, (double) hy,
        (double) drag, (double) q, (double) mass, (double) t, Vx, Vy ));
    /* Calculate new positions for each particle: */
    pr_pcall( team_id, PUSH_LOC, copyargs(Part_x, Part_y, Vx, Vy,
        (double) Lx, (double) Ly, (double) hx, (double) hy, (double) del_t ));

    /* Reset the charge desity array to zero and */
    /* printf(" figure in what the particles contribute to Rho: \n\n"); */
    PART_RHO(rho_k, Np, Nx, Ny, hx, hy, Rho);

    /* printf(" Solve the field equation given charge densities, etc.\n\n"); */
    FFT_SOLVE(Phi, Rho, eps, Lx, Ly, Nx, Ny);

    /* Calculate the field at each node -- store in array Ex and Ey for */
    /* x- and y-direction, respectively: */

    PERIODIC_FIELD_GRID(Phi, Nx, Ny, hx, hy, Ex, Ey);
    t = t + del_t;
}
/* end while t */
/************************ end main simulation loop ***************************/

Figure 6.1: KSR SPC calls parallelizing particle update routines
6.6 Replicating Grids

The most popular way to handle the parallel writes associated with the scatter (part-rho) routine is to replicate the charge density grid for each processors and have the nodes (threads) update local copies and then add them together in the end. In a shared memory addressed system like the KSR, this can be fairly easily implemented by adding an extra dimension to the grid array.

Figures 6.2 shows how great speed-ups were achieved for the particle pushers for velocities and positions, respectively, when using these replicated grid for charge accumulation. However, as predicted, the charge accumulation routine really takes a major performance hit as the number of processing nodes increases. In fact, for the non parallelized sum, the overhead in adding the local grid copies together causes so much overhead that this routine experiences a worse than linear slow-down for 8 or more processors! A tree-based parallelized sum is not much better in that it also actually slows down when more than 16 processing nodes are used. This behavior can clearly been seen from the Scatter curve in Figure 6.2. (uses the parallel sums).

As expected, the particle position updates scale fairly linearly in the number of processors since these updates are independent and hence involve no inter-node memory traffic. The velocity updates need to read in grid quantities and since these accesses are fairly random, global read copies need to be made by the system underneath, and this routine hence takes a performance hit over the position update routine.

Since the field-grid routine was already fast and only parallelized in one dimension, it also did not benefit much from parallelization. This is not surprising since at 118 nodes, our maximum test case, each thread typically handled only one row of the grid in this phase.

The scatter and gather/push-v routines also show interesting “kinks” in their curve around 32 processors. This effect can probably be attributed to the KSR1’s
Figure 6.2: Parallel Scalability of Replicated Grid Approach
32-node ring size. Since we eliminated I/O nodes in our runs, this meant that when we requested 32 nodes, those nodes will no longer be on the same ring, and the longer access times for inter-ring memory access starts playing a role.

6.7 Distributed Grid

One of the reasons for distributing the grid was to alleviated the scatter overhead associated with replicated grids. Figure 6.3 gives scatter routine benchmarks obtained from these two approaches for runs with the same problem size of comparable optimization. As expected, the the distributed grid case shows continued speed-up with increasing number of processors, whereas the replicated grid approach behaves as in Figure 6.2, by actually showing slow-down when using more than 16 processors.

Figure 6.4 shows the same benchmarks for the partitioned grids case, but now including particle push results as well. It is harder to compare this data to the replicated grid case, since the efficiency of the push routines are affected by the initial conditions as demonstrated by the results in Table 6.4. This table compares a simulations with relatively large initial velocities such as the plasma instability test case (shown in column 1), to one with initial velocities only one-hundredth of that and one with none. These benchmarks are fairly comparable to each other except for the routine updating the position. Since the global scratch arrays grow with the number of particles leaving at each time-step, this routine takes a large performance hit (almost a factor of 2 in our test case).

Benchmarks for the corresponding replicated grid case are included in the right-most column for comparisons. Although these benchmarks are done on only 4 processor, they already show that the distributed grid scatter routine is faster than the replicated grid scatter routine. However, these results also show the overhead associated with partially sorting and maintaining dynamic local particle arrays.
Figure 6.3: Scatter: Distributed Grid versus Replicated Grids
Figure 6.4: Distributed Grid Benchmarks
However, since the distributed particle update routines also scale linearly in time (see Figure 6.4), this approach becomes favorable as the scatter routine starts to dominate the replicated grid case.

Table 6.4: Distributed Grid – Input Effects
128 × 128 grid; 32,768 particles; 4 processors.

<table>
<thead>
<tr>
<th>Vdrift:</th>
<th>(\omega_p \cdot 2L_y)</th>
<th>(1/100(\omega_p \cdot 2L_y))</th>
<th>0.0</th>
<th>N.A.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (secs)</td>
<td>Time (secs)</td>
<td>Time (secs)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scatter</td>
<td>12.35</td>
<td>11.77</td>
<td>11.83</td>
<td>19.33</td>
</tr>
<tr>
<td>Gather +</td>
<td>23.86</td>
<td>22.53</td>
<td>22.43</td>
<td>19.20</td>
</tr>
<tr>
<td>Push-v</td>
<td>55.14</td>
<td>29.35</td>
<td>29.13</td>
<td>16.67</td>
</tr>
</tbody>
</table>

6.8 FFT Solver

We did not implement an optimized 2D FFT solver on the KSR. The KSR has a vendor provided 2D FFT routine that is supplied in Fortran that we hope to be able to utilize in our production runs. Since these routines are developed by the vendors to show off their hardware, these are often highly optimized routines that use assembly language programming, caching and every other feature the vendor can think of taking advantage of. It is therefore highly unlikely than users will be able to achieve similar performance with their Fortran or C codes, no matter how well they are coded.
6.9 Grid Scaling

From Table 6.3 one would hardly think it worthwhile to parallelize the Field-Grid routine since it takes such an insignificant amount of time compared with the rest of the serial code. However, one does actually start noticing its presence as the grids get larger and the rest of the code parallelized. Figure 6.4 illustrates how the parallelized code, as expected, scales linearly with the grid-size for fixed number of processors and particles.

As expected, the particle push routines for positions showed no effect of the varying grid-size since this routine is independent of the grid, whereas the gather/push-velocity routine and the scatter routine show the grid effects as predicted.

6.10 Particle Scaling

We also included benchmarks where the number of grids and processors were fixed, and the number of particles varied. (See figure 6.5)

As expected, calculating the field at each grid-point based on the charge distribution (field-grid) shows no effect of this scaling since it only involves grid quantities. The “kink” in the particle update curves experienced in our benchmarks at at 4096 particles/node (4 particles per grid) can be attributed to local paging/caching. Since the replicated grid approach accesses the same local particle quantities in sequential memory locations, cache-hits will occur when the local cache-size is exceeded.

6.11 Testing

To assure that the codes were still modeling the expected equations, the output of each new parallelized version was checked for validity. The tests described in Chapter 3 for checking for plasma oscillations, symmetry and two-stream instabilities played an important role in this effort. Unfortunately, we did not have access to the
Figure 6.5: Grid-size Scaling. Replicated grids; 4 nodes; 262,144 Particles, 100 Time-steps
Figure 6.6: Particle Scaling – Replicated Grids
HDF package used to produce the two-stream instability graphs in Chapter 3. We did, however, compare output files of the charge distribution and particle position with our serial code’s corresponding output. We limited the number of iterations to about 10 since these codes produce non-linearities and round-off effects that as time progresses.
Chapter 7

Conclusions and Future work

“\textquote{I know that you believe you understand what you think I wrote, but I am not sure you realize that what you read, is not what I meant...}”

– author’s version of saying first seen on a plaque at Glennwood Pines Restaurant, Rt 89, Ithaca, NY; original by Richard Nixon.

By bringing together knowledge from the fields of computer science, physics, and applied mathematics (numerical analysis), this dissertation presented some guidelines on how to fully leverage parallel supercomputer technology in the area of particle simulations.

This work highlighted several relevant basic principles from these fields including a summary of the main references related to the area of parallel Particle-in-Cell codes. Our code modeled charged particles in an electric field. We analyzed this fairly complex application which has several interacting modules.

Our framework was a physically distributed memory system with a globally shared memory addressing space such as the KSR supercomputer. However, most of the new methods developed in this dissertation hold generally for high-performance systems with either hierarchical or distributed memory.

By studying the interactions between our application’s sub-program blocks, we showed how the accompanying dependencies affect data partitioning and lead to
new parallelization strategies concerning processor, memory and cache utilization.

We introduced a novel approach that lead to an efficient implementation facilitating dynamically partitioned grids. This novel approach takes advantage of the shared-memory addressing system and uses a dual pointer scheme on the local particle arrays that keeps the particle locations automatically partially sorted (i.e. sorted to within the local grid partition). Complexity and performance analyses was included for both this approach and for of traditional replicated grids approach.

We also introduced hierarchical data structures that were tailored for both the cache size and our problem’s structure of memory access. By reordering the grid indexing, we aligned the storage of neighboring grid-points with the local cache. This saved us us 25% of the cache-hits for a 4-by-4 cache.

We showed that further improvements can be made by considering the input data’s effect on the simulation. E.g. in the case of mean particle drift, it may be advantageous to partition the grid primarily along the direction of the drift.

Our analyses and implementation benchmarks demonstrate how the replicated grid approach is appropriate for problems run on a limited number of processors. Since the particles are always processed by the same computational node, an initially load-balanced simulation will remain perfectly load balanced. However, the replicated grid fails to scale properly both with respect to computation and memory requirements, for large problems (say, grids larger than 128-by-128) running on several processors (in our case, greater than 8 with respect to processing time). In this case, the extra storage and computation time associated with adding the local grid copies together, was shown to be significant. Our dual pointer scheme for grid partitioning, on the other hand, is more difficult to implement, but scales well for large problems on highly parallel systems with several dozen or more processors, since it does not need to replicate and summarize the whole grid.
The particle-in-cell codes for this study were tested using physical parameters, which lead to predictable phenomena including plasma oscillations and two-stream instabilities.

7.1 Future Work

“I like the dreams of the future better than the history of the past.”

The field of computational science, a term first cast by physics Nobel Laureate Ken Wilson, is clearly a growing field that will remain in focus for years to come. During the past 10 years computing has become an essential tool for many scientists and engineers, and we expect this trend to continue as technology provides us with more computational power through the advances in semiconductor technologies as well as the use of parallel and distributed computer systems. To illustrate the importance of several universities have recently initiated graduate programs specifically aimed at Computational Science and Engineering [SRS93].

This dissertation focused on analyzing the parallelization of Particle-in-Cell methods, but it is clear that the main techniques developed to parallelize complex computer programs, such as particle simulations, can be applied to many other scientific and engineering codes. Current research areas that take advantage of computational science and engineering include astronomy, biology and virology, chemistry, electromagnetics, fluid dynamics, geometric modeling, material science, medicine, numerical algorithms, physics, problem-solving environments, scientific visualization, signal and image processing, structures technology, symbolic computing, and weather and ocean modeling. The fun has merely begun!

“Man wants to know, and when he ceases to do so, he is no longer man”. – Nansen [1861-1930], on the reason for polar explorations.
Appendix A

Annotated Bibliography

“Nothing has really happened until it has been recorded.”
– Virginia Woolf, quoted in Harold Nicolson, Diaries.

A.1 Introduction

This section give a detailed description of some of the main references related to parallel particle codes. Sections A.2 and A.3 cover the main books and general articles, respectively, whereas Sections A.4 and A.5 cover the PIC references most central to our work. An overview table of these references was provided in Chapter 2.

Since the field of computational science is relatively new and on the intersection of several science disciplines, references are often hard to find using only the Science Citation Index. INSPEC is recent library utility that the author found extremely useful (it not only list authors and titles from published journals, but includes recent conference papers and abstracts as well.) However, to really be on top of this rapidly moving field, it is important to follow the main conferences in high performance computing (e.g. Supercomputing ’9x, SHPCC ’9x, SIAM conferences, The Colorado Conference on Iterative Methods (previously called The Copper Mountain Conference on Iterative Methods), etc.), keep in touch with the main
actors of the field (have them send you technical reports), and look out for new journals such as *Computational Science & Engineering* due out by IEEE in the spring of 1994.

### A.2 Reference Books

#### A.2.1 Hockney and Eastwood: “Computer Simulations Using Particles”

This book [HE89] is a solid reference that concentrates on particle simulations with applications in the area of semiconductors. They include a historical overview of PIC codes and give a fairly thorough example of 1-D plasma models. They covers several numerical methods for solving the field equation (including the SOR and FFT methods we used), and describe several techniques related to semiconductor device modeling, astrophysics, and molecular dynamics. The FFT is described in the appendix.

#### A.2.2 Birdsall and Langdon: “Plasma Physics Via Computer Simulation”

This text [BL91], concentrates on describing the general techniques used in plasma physics codes and hence include fairly detailed descriptions of 2 and 3-D electrostatic and electromagnetic programs. Their book also include a diskette with 1-D electrostatic code complete with graphics for running under MS-DOS and X-windows (X11).

#### A.2.3 Others

Tajima [Taj89] and Bower & Wilson [BW91] are also good generic references for plasma simulation codes with slants towards astrophysics.

For those interested in correlated methods – statistical mechanics and molecular dynamics involving numerical techniques such as the Monte Carlo method
(named so due to the role random numbers play in this method) – Allen & Tildesley [AT92] and Binder [Bin92] are considered some of the best references on the subject. Although these methods are quite different from the PIC method, they may provide interesting algorithmic ideas that could be used when parallelizing PIC codes.

A.2.4 Fox et. al: “Solving Problems on Concurrent Processors”

This 2-volume set [FJL+88,AFKW90] covers the most common parallel techniques used in parallelizing physical codes. The material centers around the authors experience on the Caltech hypercube, and is hence biased towards hypercube implementations. Volume I, subtitled General Techniques and Regular Problems, gives a theoretical overview of the algorithms used and their underlying numerical techniques, whereas Volume II, Software for Concurrent Processor, is a primarily compendium of Fortran and C programs based on the algorithms described in Volume I.

A.3 General Particle Methods

In April 1987 a workshop entitled “Particle Methods in Fluid Dynamics and Plasma Physics” was held at Los Alamos in New Mexico. The proceedings from this workshop was published in Computer Physics Communications the following year and contains several interesting papers in the area:

A.3.1 F.H. Harlow : ”PIC and its Progeny”

[Har88] is a survey article covering the origin of PIC and related techniques as methods for exploring shock interactions with material interfaces in the ’60s. Harlow’s PIC work for fluid dynamics simulations (’64) was the foundation for Morse and Nielson’s (’69) work on higher-order interpolation (CIC) schemes for plasmas.
According to Hockney and Eastwood, they and Birdsall’s Berkeley group (’69) were the first to introduce CIC schemes.

The bulk of Harlow’s paper is a list of 143 references, mostly papers on fluid dynamics by the authors and his colleagues at Los Alamos (1955-87).

A.3.2 J. Ambrosiano, L. Greengard, and V. Rokhlin: The Fast Multipole Method for Gridless Particle Simulation

[AGR88] advocates the use of modern hierarchical solvers, of which the most general technique is the fast multipole method (FMM), to avoid some of the local smoothing, boundary problems, and aliasing problems associated with PIC methods when used to simulate cold plasmas and beams, and plasmas in complicated regions. The paper describes the FMM method and how it fares with respect to the aforementioned problems associated with PIC methods.

A.3.3 D.W. Hewett and A.B. Langdon: Recent Progress with Avanti: A 2.5D EM Direct Implicit PIC Code

[HL88] describes the direct implicit PIC method and some relativistic extensions. The code uses an iterative solution based on ADI (alternating direction implicit) as a 2-D direct field solver. The paper does point out that only minimal consideration was given to algorithms that may be used to implement the relativistic extensions. Some concepts were tested on a 1-D code.

A.3.4 S.H. Brecht and V.A. Thomas: Multidimensional Simulations Using Hybrid Particle Codes

[BT88] describes the advantages and disadvantages in using a hybrid particle code to simulate plasmas on very large scale lengths (several \(\lambda_D\)). By treating the electron as a massless fluid and the ions as particles, some physics that magneto-
hydrodynamics (MHD) codes do not provide (MHD assumes charge neutrality, i.e. \( \rho = 0 \)), can be included without the costs of a full particle code. They avoid solving the potential equations by assuming that the plasma is quasi-neutral \( (n_e \approx n_i) \), using the Darwin approximation where light waves can be ignored, and assuming the electron mass to be zero. They hence use a predictor-corrector method to solve the simplified equations.

**A.3.5 C.S. Lin, A.L. Thring, and J. Koga: Gridless Particle Simulation Using the Massively Parallel Processor (MPP)**

[LTK88] describes a gridless model where particles are mapped to processors and the grid computations are avoided by using the inverse FFT to compute the particle positions and the electric field. However, since the MPP is a bit-serial SIMD (single instruction, multiple data) architecture with a grid topology and not much local memory, they found that the overhead in communication when computing the reduction sums needed for this technique when computing the charge density was so high that 60% of the CPU time was used in this effort.

(see also description of other articles by Lin in Section A.xx)

**A.3.6 A. Mankofsky et al.: Domain Decomposition and Particle Pushing for Multiprocessing Computers**

[M+88] describes parallelization effort on two production codes, ARGUS and CANDOR for multiprocessing on systems such as the Cray X-MP and Cray 2. ARGUS is a 3-D system of simulation codes, the paper paying particularly attention to those modules related to PIC codes. These modules include several field solvers (SOR, Chebyshev and FFT) and electromagnetic solvers (leapfrog, generalized implicit and frequency domain), where they claim their 3D FFT solver to be exceptionally fast.

One of the most interesting ideas of the paper, however, is how they use the
cache as storage for particles that have left their local domain whereas the local particles get written to disk. The cached particles then get tagged onto the local particles in its new cell when they get swapped in. Their experience with CANDOR, a 2.5D electromagnetic PIC code, showed that it proved efficient to multi-task over particles (or a group of particles) within a field block. They note the trade-off due to the parallelization efficiency increasing with the number of particle groups, whereas the vectorization efficiency increasing with the number of particles per group. The speed-up for their implementation on the Cray X-MP/48 showed to reach close to the theoretical maximum of 4.

A.4 Parallel PIC – Survey Articles

A.4.1 John M. Dawson

John Dawson’s 1983 Rev. of Modern Physics paper entitled “Particle simulation of plasmas” [Daw83], is a lengthy review of the field modeling charged particles in electric and magnetic fields. The article covers several physical modeling techniques typically associated with particle codes. It is a serial reference that sites more than 100 related papers.

A.4.2 David Walker’s survey Paper

David Walker’s “Characterizing the parallel performance of a large-scale particle-in-cell plasma simulation code” from G. Fox’s journal Concurrency: Practice and Experience, Dec.’90 issue [Wal90] gives a nice survey of vector and parallel PIC methods. It covers the most basic parallel techniques explored, with an emphasis on the importance of load balancing. The paper stresses that there is a strong need for future work in the Multiple Instruction Multiple Data MIMD, i.e. distributed memory computing arena:

“... For MIMD distributed memory multiprocessors alternative decomposition schemes need to be investigated, particularly with inhomoge-
neous particle distributions."

The paper cites 54 references.

A.4.3 Claire Max: “Computer Simulation of Astrophysical Plasmas”

Max [Max91] gives a nice general description of plasma codes, and how in the coming decade, sophisticated numerical models and simulations will play an important role in the field of plasma astrophysics. She points to current effort on the CRAY machines and how plasma astrophysics has a genuine need for the governmental supercomputer resources potentially provided by the NSF, DOE, and NASA Supercomputing Centers.

A.5 Other Parallel PIC References

A.5.1 Vector codes – Horowitz et al

Nishiguchi (Osaka Univ.) et al.’s 3-page note [NOY85] describes how they bunch particles in their 1-D code to utilize the vector processor on a VP-100 computer. Horowitz (LLNL, later Univ. of Maryland) et al. [Hor87] describes a 2D algorithm with timing analyses done on a similar 3D code for a Cray. Unlike Nishiguchi et al. who employs a fixed grid, Horowitz’s approach requires sorting in order to vectorization. This scheme is a bit slower for very large systems, but requires much less storage.

Horowitz et al. later [HSA89] describe a 3D hybrid PIC code which is used to model tilt modes in field-reversed configurations (FRCs). Here, the ions are modeled as collisionless particles whereas the electrons are treated as an inertialess fluid. A multigrid algorithm is used for the field solve, whereas the leap-frog method is used to push the particles. Horowitz multi-tasks over 3 of the 4 Cray 2 processors in the multigrid phase by computing one dimension to each processor. The interpolation of the fields to the particles was found computationally intensive
and hence multi-tasked achieving an average overlap of about 3 due to the relationship between task length and the time-slice provided for each task by the Cray. (For the Cray they used, the time-slice depended on the size of the code and the priority at which it ran.) The particle push phase similarly got an overlap of about 2 (many, but simpler calculations). These results are hence clearly depended on the scheduling algorithms of the Cray operating system.

A.5.2 C.S. Lin (Southwest Research Institute, TX) et al

C.S. Lin's paper *Simulations of Beam Plasma Instabilities Using Parallel Particle-in-cell Code on the MPP* (HCCA4)[Lin89b], uses a particle sorting scheme based on rotations (scatter particles that are clustered through rotations). The same approach is described in the author's similar, but longer paper, *Particle-in-cell Simulations of Wave Particle interactions Using the Massively Parallel Processor* [Lin89a].

The papers describe a 1-D electrostatic PIC code implemented on the MPP (Massively Parallel Processor), a 128-by-128 toroidal grid of bit-serial processors located at Goddard. The author simulates up to 524,000 particles on this machine using an FFT solver.

Two previous sorting studies are mentioned in this paper: one where particles are sorted at each time-step (lots of overhead), another where a “gridless” FFT that used more computations was considered.

In the first study, they mapped the simulation domain directly to the processor array and sorted the particles according to their cell every time step. This was found to be highly inefficient on the MPP due to the excessive I/O required between the array processors and the staging memory. They also point out that the scheme would not remain load-balanced over time since the fluctuations in electrical forces would cause the particles to distribute non-uniformly over the processors.

In the other study, they developed what they call a gridless model where they
map particles to random processors. This approach claims to avoid charge collection by computing the electric forces directly using the Discrete Fourier Transform (see Section 2.3.4) However, it was shown to be 7 times as slow as a similar PIC code the CRAY, and is consequently dismissed.

The sorting approach

Designed for the bit-serial MPP, (consisting of the 64-by-64 toroidal grid of bit-serial processors), the implementation uses 64 planes to store 524,000 particles. The approach fills only half the particle planes (processor grid) with particles to make the sorting simpler by being able to shuffle (“rotate”) the data easily in this bit-serial SIMD machine. The spare room was used in the “shuffling” process where congested cells had part of their contents rotated to their northern neighbor, and then west, if necessary during the “sorting” process. The implementation is clearly tied to the MPP architecture. For nodes with more computational power and a different interconnection network, other techniques will probably prove more useful.

Only two other recent papers by C.S. Lin were found using a Science Citation Index search, one by Eliasson from Umeå in Geophysical Research Letters, and a paper by him and his co-authors in JGR-Space Sciences. None of these seem to discuss particle sorting.

A.5.3 David Walker (ORNL)

In “The Implementation of a Three-Dimensional PIC Code on a Hypercube Concurrent Processor” [Wal89] Walker describes a 3-D PIC code for the NCUBE, but does not have a full implementation of the code. He uses the “quasi-static crystal accumulator”, some kind of gather-scatter sorter proposed by G. Fox. et. al. See also Walker’s general reference in Section A.xx.
A.5.4 Lubeck and Faber (LANL)

This 1988 journal paper entitled “Modeling the performance of hypercubes: A case study using the particle-in-cell application” [LF88] is also highly relevant to our work. The paper covers a 2-D electrostatic code benchmarked on the Intel iPSC hypercube. A multigrid algorithm based on Fredrickson and McBryan efforts are used for the field calculations, whereas they considered 3 different approaches for the particle push phase.

The first approach was to assigning particles to whatever processor has the cell information (observing strict locality). The authors rejected this approach based on that their particles tend to congregate in 10% of the cells, hence causing serious load-imbalance.

The second alternative they considered was to relax the locality constraint by allowing particles to be assigned to processors not necessarily containing their spatial region. The authors argue that the performance of this alternative (move either the grid and/or particles to achieve a more load-balanced solution) would be a strong function of the particle input distribution.

The alternative they decided to implement replicated the spatial grid among the processors so that an equal number of particles can be processed at each time-step. This achieves a perfect load balance (for homogeneous systems such as the iPSC).

To us, this does seems to require a lot of extra overhead in communicating the whole grid to each processor at each time-step, not to mention having to store the entire grid at each processor.

This paper does describe a nice performance model for their approach. The authors comment that they found the partitioning of their PIC algorithm for the hypercube “an order of magnitude greater” compared with a shared memory implementation.
A.5.5 Azari and Lee’s Work

Azari and S.Y. Lee have published several papers on their work on hybrid partitioning for PIC codes, [ALO89, AL91, AL90, AL92, Aza92]. Their underlying motivation is to parallelize Particle-In-Cell (PIC) codes though a hybrid scheme of Grid and Particle Partitions.

Partitioning grid space involves distributing particles evenly among processor and partitioning the grid into equal-sized sub-grids, one per available processor element (PE). The need to sort particles from time to time is referred to as an undesirable load balancing problem (dynamic load balancing).

A particle partitioning implies, according to their papers, that all the particles are evenly distributed among processor elements (PEs) no matter where they are located on the grid. Each PE keeps track of the same particle throughout the entire simulation. The entire grid is assumed to have to be stored on each PE in order to keep the communication overhead low. The storage requirements for this scheme is larger, and a global sum of the local grid entries is needed after each iteration.

Their hybrid partitioning approach combines these two schemes with the argument that by partitioning the space one can save memory space on each PE, and by partitioning the particles one may attempt to obtain a well-balanced load distribution which would lead to a high efficiency.

Their hybrid partitioning scheme can be outlined as follows:

1. the grid is partitioned into equal subgrids,
2. a group of PEs are assigned to each block,
3. each grid-block is stored in the local memory of each of the PEs assigned to that block
4. the particles in each block are initially partitioned evenly among PEs in that block.
The papers then go on to describing specific implementations including a hypercube and BBN implementations and corresponding performance evaluations.

A.5.6 Paulette Liewer (JPL) et. al.

Liewer has also co-authored several papers on this topic. Her 1988 paper with Decyk, Dawson (UCLA) and G. Fox (Syracuse) [LDDF88], describes a 1-D electrostatic code named 1-D UCLA, decomposing the physical domain into sub-domains equal in number to the number of processors available such that initially each sub-domain has an equal number of processors. Their test-bed was the Mark III 32-node hypercube.

The code uses the 1-D concurrent FFT described in Fox et. al. For the particle pushing phase, they divide their grid up into \((N - p)\) equal-sized sub-domains. However, the authors point out how they need to use a different partitioning for the FFT solver in order to take advantage of the hypercube connection for this phase. (They need to partition the domain according to the Gray Code numbering of the processors.) The code hence passes the grid array among the processors twice at each time step. In the conclusions, they point out that this may not be the case if a finite difference field solution is used in place of the FFT.

In [LZDD89] they describe a similar code named GCPIC (General Concurrent PIC) implemented on the Mark IIIf6 (64-processors) that beats the CRAY XMP. In 1990 Liewer co-authored a paper [FLD90] describing a 2-D electrostatic code that was periodic in one dimension, and with the option of bounded or periodic in the other dimension. The code used 2 1-D FFTs in the solver. Liewer et al. has recently developed a 3D code on the Delta Touchstone (512 node grid) where the grids are replicated on each processor [DDSL93,LLFD93]. Liewer et al. also have a paper on load balancing described later in this appendix.
A.5.7 Sturtevant and Maccabee (Univ. of New Mexico)

This paper [SM90] describes a plasma code implemented on the BBN TC2000 whose performance was disappointing. They used a shared-memory PIC algorithm that did not map well to the architecture of the BBN and hence got hit by the high costs of copying very large blocks of read-only data.

A.5.8 Peter MacNeice (Hughes/Goddard)

This paper [Mac93] describes a 3D electromagnetic PIC code re-written for a MasPar with a 128-by-128 grid. The code is based on Oscar Buneman’s TRISTAN code. They store the the third dimension in virtual memory so that each processor has a grid vector. They use a Eulerian decomposition, and hence need to sort after each time-step. A finite-difference scheme is used for the field solve, whereas the particle push phase is accomplished via the leap-frog method. Since they assume systems with relatively mild inhomogeneities, no load balancing considerations were taken. The fact they only simulate 400,000 particles in a 105-by-44-by-55 system, i.e. only about one particle per cell and under-utilizing the 128-by-128 processor grid, we assume was due to the memory limitations of the MasPar used (64Kb/processor). The MasPar is a Single Instruction Multiple Data (SIMD) machine.
Appendix B

Calculating and Verifying the Plasma Frequency

B.1 Introduction

In order to verify what our code actually does, an in-depth analysis of one general time step can predict the general behavior of the code. If the code produces plasma oscillations, it is expected that the charge density at a given grid-point behaves proportionally to a cosine wave. That is,

\[ \rho_{ij}^{n+1} = \text{const} \times \cos(\omega_p(t + \Delta t) + \delta) \]  \hspace{1cm} (B.1)

\[ \rho_{ij}^{n} = \text{const} \times \cos(\omega_p(t) + \delta) \]  \hspace{1cm} (B.2)

One cannot expect the code to produce the exact result from the above equation, but rather a Taylor series approximation of them. Looking at the ratio of the above equation, one can hence expect something like:

\[ \frac{\rho_{ij}^{n+1}}{\rho_{ij}^{n}} = 1 - \Delta t^2 \frac{\Delta t}{2} - \Delta t [\tan(\omega_p t) + \delta] \]
B.2 The potential equation

Looking at the potential at each grid point, we assume it is of the form:

$$\Phi_{i,j} = \text{Re}(\hat{\Phi} e^{ik_x x} e^{ik_y y})$$  \hspace{1cm} (B.3)

where $x$ and $y$ are the limits of the grid.

Since $x = (i-1)h_x$, $y = (j-1)h_y$, the above equation can be re-written:

$$\Phi_{i,j} = \text{Re}[\hat{\Phi} e^{i(k_x(i-1)h_x+k_y(j-1)h_y} ]$$  \hspace{1cm} (B.4)

This functions can be thought of as a 2-D plane extending into waves of “mountains” and “valleys” in the third dimension.

B.2.1 The FFT solver

Differentiating $\Phi$ twice directly would give the following expression for $\Phi$:

$$\left( \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \right) \hat{\Phi} (e^{ik_x x} e^{ik_y y}) = -\frac{\rho_{i,j}}{\epsilon_0}$$  \hspace{1cm} (B.5)

$$i^2k_x^2 \hat{\Phi} e^{ik_x x+ik_y y} + i^2k_y^2 \hat{\Phi} e^{ik_x x+ik_y y} = -\frac{\rho_{i,j}}{\epsilon_0}$$  \hspace{1cm} (B.6)

$$-(k_x^2 + k_y^2)\Phi = -\frac{\rho_{i,j}}{\epsilon_0}$$  \hspace{1cm} (B.7)

which again gives us:

$$\Phi = \left( \frac{\rho_{i,j}}{\epsilon_0} \right) \frac{1}{k_x^2 + k_y^2}.$$  

Notice that this is indeed the quantity that we scaled with in the FFT solver.

B.2.2 The SOR solver

Writing Equation 3.46 for $\Phi_{i,j+1}$ in terms of $\Phi_{i,j}$ yields:

$$\Phi_{i,j+1} = \Phi_{i,j} * e^{ik_y h_y}$$

Proceeding in the same fashion fore the other neighboring grid points, one obtains the following expression for the Laplacian $\nabla \Phi^2$:
\[ \Phi_{i,j}[e^{-ik_yh_y} + e^{ik_yh_y} - 4 + e^{-ik_xh_x} + e^{ik_xh_x}]h^2 = -\frac{\rho_{i,j}}{\epsilon_0} \] (B.8)

Using the exponential form of the expression for cosine \( \cos(x) = \frac{1}{2}(e^{ix} + e^{-ix}) \), we get:

\[ \frac{\Phi_{i,j}}{h^2}(2\cos(k_yh_y) - 4 + 2\cos(k_xh_x)) = -\frac{\rho_{i,j}}{\epsilon_0} \] (B.9)

Rewriting the above equation to fit the expression \( 1 - \cos(x) = 2\sin^2(x) \), simplifying using the Taylor series approximation \( \sin^2(x) \approx \frac{x^2}{2} \), and considering \( h = h_x = h_y \), we get:

\[ -\left(\frac{4}{h^2}\right)\left(\frac{(k_x^2 + k_y^2)h^2}{4}\right)\Phi_{i,j} = -\frac{\rho_{i,j}}{\epsilon_0} \] (B.10)

which if we solve for \( \Phi \) gives us:

\[ \Phi \approx \left(\frac{\rho_{i,j}}{\epsilon_0}\right)\frac{1}{k_x^2 + k_y^2} \] (B.11)

Notice how the grid spacing quantities \( h_x \) and \( h_y \) cancel and we obtain the same expression as we did for the FFT solver. This is not surprising given that our finite difference approximation indeed is supposed to approximate the differential.

**B.2.3 The field equations**

A similar result can then be obtained for the field \( \mathbf{E} \) using the above result:

\[ E_{x_{i,j}} = -\Phi_{i,j+1} + \Phi_{i,j-1} \] (B.12)

\[ = -\Phi_{i,j} \left( \frac{e^{ik_xh_x} - e^{-ik_xh_x}}{2h_x} \right) \] (B.13)

\[ = -\left(\frac{\Phi_{i,j}}{h_x}\right) i \sin(k_xh_x) \] (B.14)

Again, using a Taylor series approximation and following the same procedure for \( E_{y_{i,j}} \), we get:

\[ E_{x_{i,j}} \approx -ik_x \Phi_{i,j} \] (B.15)

\[ E_{y_{i,j}} \approx -ik_y \Phi_{i,j} \] (B.16)
Looking at the field at each particle $E_{\text{part}}$:

$$E_{\text{part}}_x = \hat{E}_x[e^{ik_x x + ik_y y}]$$  \hspace{1cm} (B.17)

Whether the $x$ and $y$ are here the coordinates of the grid or the particle, does not really matter; in this case, it is not important where the E-field is being evaluated. We hence approximate the fields at the particles position for $x$ and $y$ to be:

$$E_{\text{part}}_x \simeq E_x$$  \hspace{1cm} (B.18)

$$E_{\text{part}}_y \simeq E_y$$  \hspace{1cm} (B.19)

### B.3 Velocity and particle positions

Recall that we used the following equations for updating the velocity and particle positions:

$$\mathbf{v} = \mathbf{v} + \frac{qE}{m} \Delta t$$  \hspace{1cm} (B.20)

$$\mathbf{x} = \mathbf{x} + \mathbf{v} \cdot \Delta t$$  \hspace{1cm} (B.21)

Here $E = \hat{E}e^{ik_x x_0 + ik_y y_0}$.

In the equation for the particle update, $x$ on the left hand is $x_0 + \delta$(new-time-step), whereas $x$ on the right hand side is $x_0 + \delta$(old-time-step). When using the Leap-Frog method, $\delta t$ in the velocity equation (Equation B.20) was set to $\frac{\Delta t}{2}$ for the first time-step so that the velocity and particle positions would be “leaping” over each other.

### B.4 Charge density

Assuming the cell-size is shrunk down to the point where there is only one particle per cell. Now, looking at a grid point and its 4 adjacent cells, each of the of
the particles in the 4 adjacent cells will have a different \((x_0, y_0)\). These must be considered when updating the charge density.

Looking at the distances the particles in the surrounding cells are from the grid point, we can calculate the charge density at each grid point (Figure 3.5). See also Figure 3.3.

\[
\begin{align*}
  j &= \text{(int)} \left( (x_0 + hx)/hx \right); \\
  i &= \text{(int)} \left( (y_0 + hy)/hy \right);
\end{align*}
\]

![Diagram showing contributions of particles in adjacent cells](image)

\(x\) : particle positions

Figure 3.5: Contributions of particles in adjacent cells with regard to charge density at grid point \((i, j)\).

Here, \(a = x_0 - ((j-1)*hx)\); \(b = y_0 - ((i-1)*hy)\), so plugging these in the equation we used for calculating charge density (see Section 3.3):

\[
\rho_{i,j} = \left( \frac{\rho_0 N_g}{h_x h_y N_p} \right) \left[ (h_x - a)(h_y - b) + (h_x - a)b + a(h_y - b) + ab \right]
\]  

(B.22)

The unperturbed particle positions are hence as shown in Figure 3.6.
x: particle positions

Figure 3.6: Unperturbed positions of particles in adjacent cells with respect to grid point (i, j).

Here, a perturbations of each particle can be viewed as:

\[ a = a_0 + \delta_x \quad b = b_0 + \delta_y \]  

(B.23)

Looking at what \( \delta_x \) and \( \delta_y \) are in terms of, say, \((x_0, y_0)\), we have:

\[ \delta_x = \delta_x e^{ik_x x_0 + ik_y y_0} \quad \delta_y = \delta_y e^{ik_x x_0 + ik_y y_0} \]  

(B.24)

Note that each of the four particles in the adjacent cells have different locations as well as perturbations. Plugging these back into our previous equation:

\[ \rho_{i,j} = [(h_x - a_0 - \delta_x(x_0 + h_x, y_0 + h_y))(h_y - b_0 - \delta_y(x_0 + h_x, y_0 + h_y))] + (h_x - a_0 - \delta_x((x_0 + h_x, y_0))(b_0 + \delta_y(x_0 + h_x, y_0))) + (a_0 + \delta_x(x_0, y_0))(h_y - b_0 - \delta_y(x_0, y_0)) + (a_0 + \delta_x(x_0, y_0))(b_0 + \delta_y(x_0, y_0))] \left( \frac{\rho_o N_p}{h_x h_y N_p} \right) \]  

(B.25)

and expanding the \( \delta \)-terms:
\[ \rho_{i,j} = \left[ ((h_x - a_0) - \tilde{\delta}_x e^{ik_x(x_0 + h_x) + iy(y_0 + h_y)}((h_y - b_0) - \tilde{\delta}_y e^{ik_y(y_0 + h_y)})\right] \]

(B.26)

\[ + \left( (h_x - a_0) - \tilde{\delta}_x e^{ik_x(x_0 + h_x) + iy(y_0 + h_y)}(b_0 + \tilde{\delta}_y e^{ik_y(y_0 + h_y)}) \right) \]

(B.27)

Multiplying out the quantities neglecting the \( \delta_x \delta_y \)-terms since they are \( O(\epsilon^2) \), and hence negligible in this context:

\[ \rho_{i,j} = \left[ (h_x - a_0)(h_y - b_0) + (h_x - a_0)b_0 + a_0(h_y - b_0) + a_0b_0 \right] \]

\[ - \tilde{\delta}_x e^{ik_x(x_0 + h_x) + iy(y_0 + h_y)}((h_y - b_0) - \tilde{\delta}_y e^{ik_y(y_0 + h_y)}(hx - a_0)) \]

\[ - \tilde{\delta}_x e^{ik_x(x_0 + h_x) + iy(y_0 + h_y)}(b_0 + \tilde{\delta}_y e^{ik_y(y_0 + h_y)}(hx - a_0)) \]

\[ + \tilde{\delta}_x e^{ik_x(x_0 + h_x) + iy(y_0 + h_y)}(h_y - b_0) - \tilde{\delta}_y e^{ik_y(y_0 + h_y)}(a_0) \]

\[ + \tilde{\delta}_x e^{ik_x(x_0 + h_x) + iy(y_0 + h_y)}(b_0 + \tilde{\delta}_y e^{ik_y(y_0 + h_y)}(a_0)) \right] \left( \frac{\rho_0 N_q}{h_x h_y N_p} \right) \]

(B.28)

The first term inside the “[“ “]”s is merely \( h_x h_y \). Looking at the \( \tilde{\delta}_x \) terms, we get:

\[ \delta_x terms = (h_y - b_0)[-1 + e^{-ik_x h_x}] \tilde{\delta}_x e^{ik_x(x_0 + h_x) + iy(y_0 + h_y)} \]

(B.29)

\[ + b_0[-1 + e^{-ik_x h_x}] \tilde{\delta}_x e^{ik_x(x_0 + h_x) + iy y_0} \]

(B.30)

We now use the following approximation for \( [-1 + e^{i\tau}] \):

\[ [-1 + e^{i\tau}] \simeq -1 + (1 - i\tau + \frac{x^2}{2} + \cdots) \simeq -ix \]

(B.32)

Hence we can simplify our \( \delta_x terms \) equation to:

\[ \delta_x terms = (-ik_x h_x) h_y \delta_x (x_0 + h_x, y_0 + h_0) + b_0 \] terms

(B.33)
Looking further at the $b_0$-terms:

\begin{align*}
\text{b}_0\text{terms} & = -b_0(-ik_x h_x)\delta(x_0 + h_x, y_0 + h_0) \\
& + b_0(-ik_x h_x)\delta(x_0 + h_x, y_0) \\
& = ((-ik_x h_x)b_0[-1\epsilon i k_y h_y]\delta(x_0 + h_x, y_0 + h_y) \\
& = b_0(-ik_x h_x)(ik_y h_y)\delta(x_0 + h_x, y_0 + h_0) \\
& = b_0 k_x k_y h_x h_y \delta_z(x_0 + h_x, y_0 + h_0) \\
& = b_0 k_x k_y h_x h_y \delta_z(x_0 + h_x, y_0 + h_0) \\
\end{align*}

(B.34)

(B.35)

(B.36)

Since we assume that $\delta_x$ is small so that $b_0 < h_y$ (see Figure 6) and $(k_y h_y)$ or $(k_x h_x) \ll 1$, we can ignore the $b_0$ term and hence end up with:

\begin{align*}
\delta_x\text{terms} & \simeq (-ik_x h_x h_y)\delta_x \\
\end{align*}

(B.37)

Similarly, the $d_y$ terms would simplify to:

\begin{align*}
\delta_y\text{terms} & \simeq (-ik_y h_x h_y)\delta_y \\
\end{align*}

(B.38)

We then get the following equation for charge density. Notice that the “$h_x h_y$”s again cancel.

\begin{align*}
\rho_{i,j} & = \left( \frac{\rho_0 N_q}{h_x h_y N_p} \right) [\delta_x\text{terms} + \delta_y\text{terms}] \\
& \simeq \rho_0 \left( \frac{N_q}{N_p} \right) (-ik_x \delta_x - ik_y \delta_y) \\
\end{align*}

(B.39)

(B.40)

### B.5 Plugging the equations into each other

From the previous sections we have the following 5 sets of equation describing our code at each time-step:

\begin{align*}
\Phi &= \left( \frac{\rho_{i,j}}{\epsilon_0} \right) \frac{1}{k_x^2 + k_y^2} \\
Ex_{i,j} & \simeq -ik_x \Phi_{i,j} \\
Ey_{i,j} & \simeq -ik_y \Phi_{i,j} \\
\end{align*}

(B.41)

(B.42)
\( E_{\text{part}} \simeq E_x \)
\( E_{\text{part}} \simeq E_y \)  \( \text{(B.43)} \)

\[ v = v + \frac{qE}{m} \Delta t \]  \( \text{(B.44)} \)
\[ x = x + v \cdot \Delta t \]  \( \text{(B.45)} \)

\[ \rho_{i,j} \simeq \rho_0 \left( \frac{N_g}{N_p} \right) \left( -ik_x \delta_x - ik_y \delta_y \right) \]  \( \text{(B.46)} \)

In order to see what kind of oscillation the code will produce, one can look at what happens between two time-steps \( n \) and \( n + 1 \). In other words, it would be helpful to know the following term:

\[ \rho_{i,j}^{(n+1)} = \text{term} \ast \rho_{i,j}^{(n)} \]

For the purposes of verifying plasma oscillations, we will assume one particle per cell \( (N_g/N_p = 1) \).

Looking at the \( n \)-th iteration, Equation B.46 gives us:

\[ \rho_{i,j}^{(n)} \simeq \rho_0 \left( -ik_x \delta_x - ik_y \delta_y \right) \]  \( \text{(B.47)} \)

To get \( \rho_{i,j}^{(n+1)} \), we plugged Equation B.46 into Equations B.41-B.46 starting with Equation B.41:

\[ \Phi = \left( \frac{\rho_{i,j}}{\epsilon_0} \right) \frac{1}{k_x^2 + k_y^2} \]  \( \text{(B.48)} \)
\[ E_{x,ij} \approx -i k_x \Phi_{ij} \]
\[ = -i k_x \left[ \frac{\rho_{ij}^{\text{old}}}{\epsilon_0} \left( \frac{1}{k_x^2 + k_y^2} \right) \right] \]
\[ = - \left( \frac{i k_x}{\epsilon_0 (k_x^2 + k_y^2)} \right) \rho_{ij}^{\text{old}} \quad (B.49) \]

Similarly,
\[ E_{y,ij} \approx -i k_y \Phi_{ij} \]
\[ = - \left( \frac{i k_y}{\epsilon_0 (k_x^2 + k_y^2)} \right) \rho_{ij}^{\text{old}} \quad (B.50) \]

We still make the same assumptions about the field at each particle:
\[ E_{\text{part},x}(i,j) \approx E_x(\rho_{ij}) \]
\[ E_{\text{part},y}(i,j) \approx E_y(\rho_{ij}) \quad (B.51) \]

The calculation, however, get a little more tricky when considering the equations for updating particle velocities \( v \) and positions \( x \). One will here have to keep track of which are the new and which are the old values:

\[ v = v + \frac{qE}{m} \Delta t \implies \]
\[ v_0 + \delta v^{\text{new}} = v_0 + \delta v^{\text{old}} + \Delta t \frac{q}{m} E \quad (B.52) \]

Similarly,
\[ x = x + \Delta t \cdot v \implies \]
\[ x_0 + \delta x^{\text{new}} = x_0 + \delta x^{\text{old}} + \Delta t \delta v^{\text{new}} \quad (B.53) \]

We hence have:
\[ \delta v^{\text{new}} = \delta v^{\text{old}} + \Delta t \frac{q}{m} E \quad (B.54) \]
\[ \delta x^{\text{new}} = \delta x^{\text{old}} + \Delta t \delta v^{\text{new}} \quad (B.55) \]
We assume that for the first time-step that $\delta v_x^{old} = \delta v_y^{old} = \delta v_y^{old} = 0$. (They actually cancelled being on both sides of the above equations). Looking at the x-direction (letting $\delta_x \triangleq \delta x_x$) and using Equation B.42:

\[
\delta_x^{new} = \delta_x^{old} + \Delta t \left( \frac{\Delta t q}{m} E_x(\rho_{i,j}) \right) = \delta_x^{old} - \frac{\Delta t^2 q}{m \epsilon_0} k_x^2 + k_y^2 \rho_{i,j}^{old} \quad (B.56)
\]

Similarly:

\[
\delta_y^{new} = \delta_y^{old} - \frac{\Delta t^2 q \cdot q_{iy}}{m \epsilon_0} k_x^2 + k_y^2 \rho_{i,j}^{old} \quad (B.57)
\]

Going back to charge density (Equation B.46) looking at the x-direction:

\[
\rho_x^{new}(i,j) = (-i k_x \delta_x^{new}) \rho_0 = \left[ -i k_x \left( \delta_x^{old} - \frac{\Delta t^2 q \cdot q_{iy}}{m \epsilon_0} \right) \right] \rho_0 = \left[ -i k_x \delta_x^{old} - \frac{\Delta t^2 q}{m \epsilon_0} \left( \frac{k_x^2}{k_x^2 + k_y^2} \right) \rho_{i,j}^{old} \right] \rho_0 \quad (B.59)
\]

We hence have

\[
\rho_x^{new}(i,j) = \left[ -i k_x \delta_x^{old} - K_x \rho_{i,j}^{old} \right] \rho_0 \quad (B.60)
\]

where

\[
K_x = \frac{\Delta t^2 q k_x^2}{m \epsilon_0 (k_x^2 + k_y^2)}
\]

Doing the same for $\rho_y^{new}(i,j)$ and combing the two equations, we get:

\[
\rho_{i,j}^{new} = \left[( -k_x \delta_x^{old} - ik_y \delta_y^{old} ) - (K_x + K_y) \rho_{i,j}^{old} \right] \rho_0 = \left[ 1 - (K_x + K_y) \rho_0 \right] \rho_{i,j}^{old} \quad (B.61)
\]

Looking at the $K_x$, $K_y$ terms together, they simplify as follows:

\[
K_x + K_y = \frac{\Delta t^2 q k_x^2}{m \epsilon_0 (k_x^2 + k_y^2)} + \frac{\Delta t^2 q k_y^2}{m \epsilon_0 (k_x^2 + k_y^2)}
\]
\[ \Delta t^2 q \]
\[ \frac{1}{m\epsilon_0 (k_x^2 + k_y^2)} \]
\[ \frac{\Delta t^2 q}{m\epsilon_0} \] (B.63)

We hence have the following expression for an updated charge density:

\[ \rho_{i,j}^{\text{new}} = \left( 1 - \frac{\Delta t^2 \cdot q}{m\epsilon_0} \cdot \rho_0 \right) \rho_{i,j}^{\text{old}} \] (B.64)

Noting that for the first time-step (where we made the \( \delta v^{\text{old}} = 0 \) assumption), the time-step is actually \( \frac{\Delta t}{2} \) (velocity lags the position update by 1/2 time-step). We hence actually have:

\[ \rho_{i,j}^{\text{new}} = \left( 1 - \frac{\Delta t^2 \cdot q}{m\epsilon_0} \cdot \rho_0 \right) \rho_{i,j}^{\text{old}} \] (B.65)

The plasma frequency is defined as \( \omega_p = \frac{q\mu}{\sqrt{m\epsilon_0}} \).

Assuming our result is the first few entries of the Taylor series approximation of \( \cos(\omega_0 \Delta t) \):

\[ 1 - \frac{\omega_0^2 \Delta t}{2} \cdots \]

we can assume our frequency \( \omega_0 = \frac{q\mu}{\sqrt{m\epsilon_0}} \), and we have shown that our code for this experiment should indeed oscillate at the plasma frequency \( \omega_p \), i.e. \( \omega_0 = \omega_p \).
Bibliography


