Development of a Parallel Electrostatic PIC Code For Modeling Electric Propulsion

Julien Pierru

Thesis submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Aerospace Engineering

Joseph J. Wang, Chair
Adrian Sandu
Wayne Scales

June 2, 2005
Blacksburg, Virginia

Keywords: Particle in Cell, MPI,
parallel simulation

Development of a Parallel Electrostatic PIC Code and its Use to Model Electric Propulsion

Julien Pierru

(ABSTRACT)

This thesis presents the parallel version of Coliseum, the Air Force Research Laboratory plasma simulation framework. The parallel code was designed to run large simulations on the world fastest supercomputers as well as home mode clusters. Plasma simulations are extremely computationally intensive as they require tracking millions of particles and solving field equations over large domains. This new parallel version will allow Coliseum to run simulations of spacecraft-plasma interactions in domain large enough to reproduce space conditions. The parallel code ran on two of the world fastest supercomputers, the NASA JPL Cosmos supercomputer ranked 37^{th} on the TOP500 list and Virginia Tech's System X, ranked 7^{th}. DRACO, the Virginia Tech PIC module to Coliseum, was modified with parallel algorithms to create a full parallel PIC code. A parallel solver was added to DRACO. It uses a Gauss-Seidel method with SOR acceleration on a Red-Black checkerboard scheme. Timing results were obtained on JPL Cosmos supercomputer to determine the efficiency of the parallel code. Although the communication overhead limits the code's parallel efficiency, the speed up obtained greatly decreases the time required to run the simulations. A speed up of 51 was reached on 128 processors. The parallel code was also used to simulate the plume expansion of an ion thruster array composed of three NSTAR thrusters. Results showed that the multiple beams merge to form a single plume similar to the plume created by a single ion thruster.

Acknowledgements

Working towards a Master's Degree thesis might sound like an individual achievement. The truth is, it is not. I could not have done it without the assistance and encouragements of those around me. Many people contributed to my success wether it was academically, financially or socially. I apologize to anybody whose name does not appear on this list, and should have been, as it is unfortunately virtually impossible to make an exhaustive list of everyone who encouraged me over the years.

First and foremost I would like to thank my graduate advisor Dr Joseph Wang for being a strong supporter of my abilities as well as for his invaluable teachings during my four years at Virginia Tech.

I would also like to thank the members of my comittee Drs Adrian Sandu and Wayne Scales.

This work could not have been possible without all the assistance provided by the folks at the Anantham cluster and System X supercomputer. Particularly my thanks go to Peter Haggerty who had to clean up the mess I left on his supercomputers after I crashed a few of the nodes on several occasions. Their assistance and help to debug and make the code run on the supercomputers is invaluable and greatly appreciated.

I already mentioned that a Master's Degree is a team effort, so it is only natural that I thank my fellow laboratory teammates, Binh Tran, Lubos Brieda, Raed Kafafy, Alex Barrie and Hyunju Jeong for their collaboration. I do have a special mention for Lubos Brieda who greatly helped me understand the Coliseum code and for his important contributions to my research.

I would like to thank Luke Scharf, our system administrator, for doing an excellent job at keeping our computer lab in a perfect working order. From his arrival up to now, his track record is spotless, no data was lost under his supervision and even helped me recover parts of my thesis from his backups after I accidently erased weeks

of strenuous thesis writing.

My friends have always been there for me and cheered me on all along my time at Virginia Tech, to name a few: Cyril Montabert and his wife Molly, Kyle Huston and Matthew Gleed.

Finally I would like to thank my entire family for their love and support, my brother Kevin of whom I am extremely proud and last but not least my parents for their unconditional love and financial support...My parents have been supporting my every decisions even if it meant for them to have 6000 miles between us, so Mom, Dad I thank you so much for everything and all the sacrifices you made for me, I love you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter provides an overview of the thesis and an introduction to ion propulsion and high performance computing. A short introduction of plasma physics and a presentation of plasma characteristic parameters will be given at the end of this chapter.

1.1 Thesis Overview

This thesis presents the development and the results of an ion thruster array plume simulation using parallel programming on massively parallel supercomputers. Chapter 1 briefly introduces the ion thruster and the domain of high performance computing. The end of Chapter 1 presents the governing equations of plasma used in this simulation. A presentation of the Particle In Cell algorithm and a short overview of the Coliseum plasma simulation framework will be presented in Chapter 2. Chapter 3 presents the work done on parallelizing the Virginia Tech plasma simulation module to Coliseum. Chapters 4 & 5 contain the results of the simulation. Chapter 6 contains the summary of the thesis and conclusions.

1.2 Overview of Electric Propulsion

1.2.1 Why Electric Propulsion?

Electric propulsion is a type of space propulsion that uses plasma to produce thrust. The plasma is composed of charged particles that are accelerated by electrostatic or electromagnetic forces. Chemical propulsion systems are capable of producing large amounts of thrust for short amount of time. In chemical propulsion, the energy is stored in the fuel and is released during combustion. In electrical propulsion, the energy comes from a separate source. One of the main attracations of Electric Propulsion is the high specific impulse that can be achieved. The high specific impulse or Isp of electric propulsion reduces the amount of propellant required.

Electric propulsion devices are extremely efficient for use in space propulsion and station keeping. A space mission using electric propulsion typically takes a long time due to the low thrust produced by the thruster. Hence it is important to understand the effects of the propulsion devices on the space environment as well as the effects on the spacecraft itself.

1.2.2 The Ion Thruster

The ion thruster has been in development since the 1960's. NASA's Deep Space 1, launched in 1998, was the first flight of a spacecraft using an ion thruster as the primary propulsion system. The ion thruster used on this particular mission was the 30 cm NASA Solar Electric Propulsion Technology Application Readiness (NSTAR), see Figure 1.1. The NSTAR thruster operates in the low kiloWatt range (0.5kW to 2.3kW)and produces a thrust from 19mN to 92mN. The resulting specific impulse are 1900s at 0.5kW and 3100s at 2.3kW [11]. The propellant used by the NSTAR ion thruster is Xenon. Deep Space 1 flight validated the technology and by September 2001 the thruster had logged over 10,000hrs of operation.

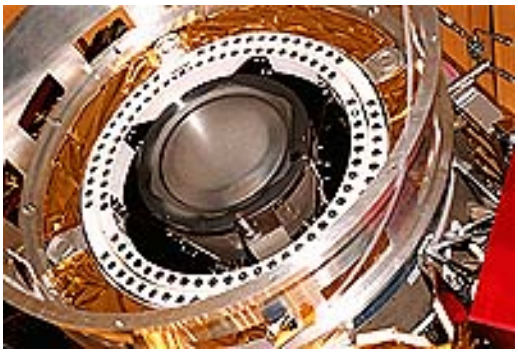

Figure 1.1: The NSTAR thruster mounted on the Deep Space 1 spacecraft.

A schematic of an ion thruster is given in Figure 1.2. The propellant, Xenon, is introduced inside the discharge chamber and is ionized by collisions with electrons emitted by the cathode. The Xenon ions are extracted from the chamber by the high potential difference applied between the screen grid and the accelerating grid. Once the ions have exited the chamber they form a beam that is neutralized by electrons emitted from the neutralizer cathode.

1.3 High Performance Computing

The increasing need for computer simulations has given rise to the rapid development of high performance computing (see Figure 1.3). High performance computing (HPC) involves developping high level parallel programming languages or libraries of parallel functions and supercomputer components.

Until recently supercomputers were mainly available to governement laboratories and large corporations due to their high costs. However with new clustering technologies and high speed networks, world class supercomputers can be build for only a fraction of the cost of supercomputers available from IBM or Cray Inc. In 2003 Virginia Tech unveiled its SystemX which ranked number 3 on the TOP 500 lists of the world's fastest supercomputers. The tag price at the time was only $\$6 million$, 58 times cheaper than the number 1 supercomputer on the list, The Earth Simulator.

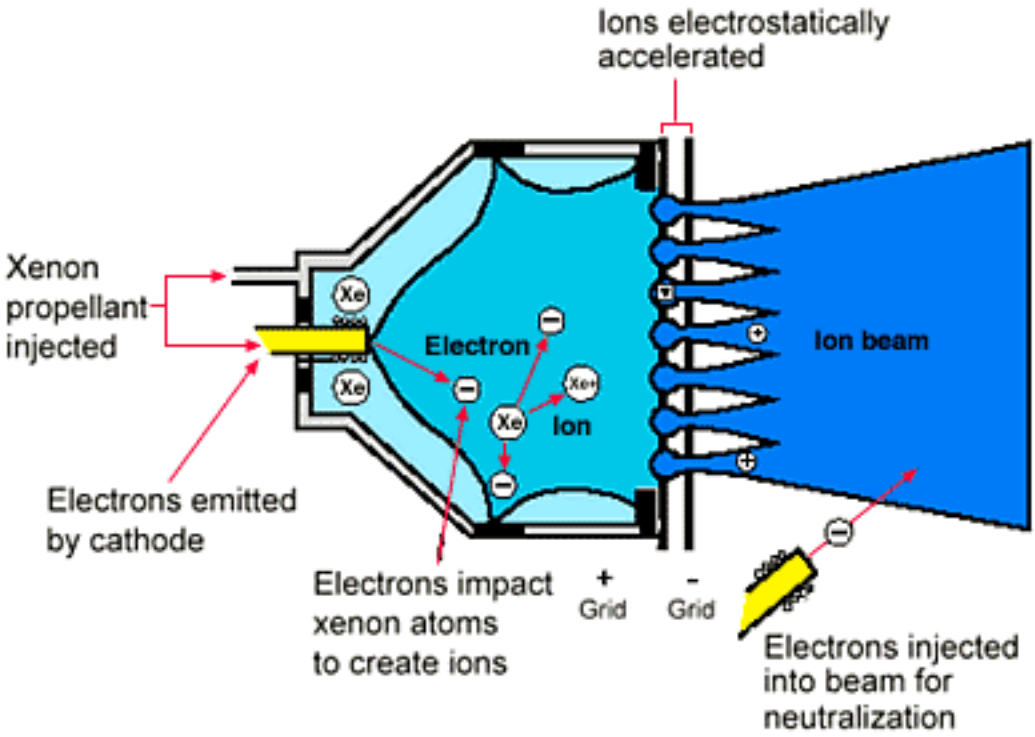

Figure 1.2: Schematic of an ion thruster.

The supercomputers used in this thesis were the NASA JPL's Cosmos supercomputer and Virginia Tech's SystemX. Their characteristics are presented in Table 1.1.

	System X	Cosmos
System	1100 Dual Apple XServe	512 Dual Dell PowerEdge 1750
Processor	PowerPC G5 2.3GHz	Intel Xeon 3.2GHz
Network	Mellanox Infiniband	Myrinet
RPeak	20240 GFlops	6553.6 GFlops
RMax	12250 GFlops	4298 GFlops
Rank	7	37

Table 1.1: System data from the supercomputers used in this thesis.(Data from the TOP500 list)

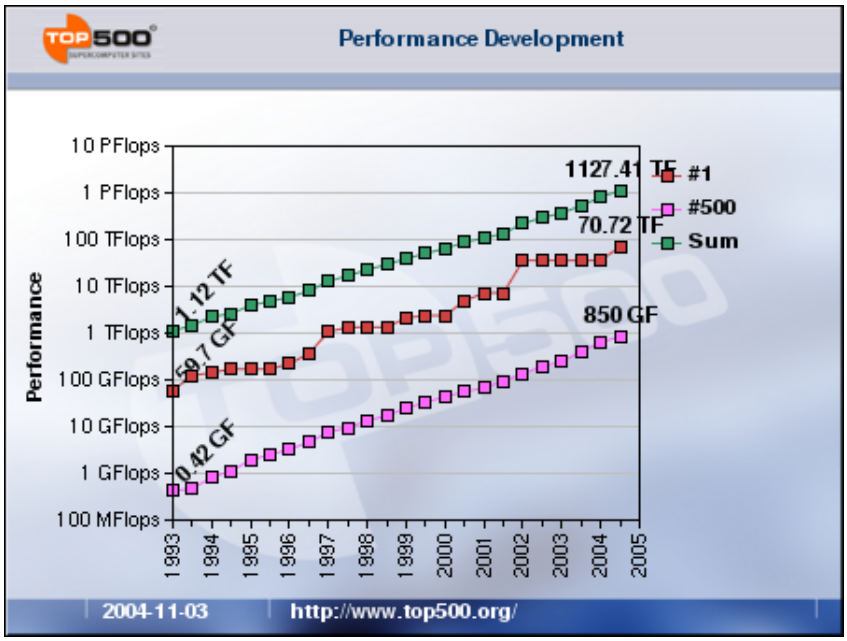

Figure 1.3: Performance evolution of the world's fastest supercomputers over the last 22 years.(Data from the TOP500 list)

1.4 Introduction to Plasma Physics

In 1928 Langmuir, who was experimenting with gas ionization, was the first scientist to use the word plasma. It is now believed that 99% of the universe matter is under the plasma form. The plasma state of any element can be reached at a very specific temperature, thus it is known as the "fourth" state of matter. This section gives a short introduction to plasma physics and the equations used in the simulation.

1.4.1 Governing Equations

The governing equation of a plasma consist of a set of equations for the Electric and Magnetic field as well as a particle motion equation [8][2]. The Lorentz equation for the motion of a particle of mass m, velocity v, charge q moving in an electric field E and magnetic field B.

$$\vec{F} = m\frac{d\vec{v}}{dt} = q\left[\vec{E} + \frac{\vec{v}}{c} \times \vec{B}\right] \tag{1.1}$$

The field is modeled by the Maxwell's equations:

$$\begin{aligned} \nabla \times \vec{E} &= -\tfrac{\partial \vec{B}}{\partial t} \\ \nabla \times \vec{B} &= \epsilon_0 \mu_0 \tfrac{\partial \vec{E}}{\partial t} + \mu_0 \vec{j} \\ \nabla \cdot \vec{E} &= \tfrac{\rho}{\epsilon_0} \\ \nabla \cdot \vec{B} &= 0 \end{aligned} \tag{1.2}$$

However for the scope of this thesis the external magnetic field is assumed to be either negligeable or a constant background, thus the Maxwell's equation simplify to solving the Poisson equation:

$$-\nabla \cdot \vec{E} = \nabla^2 \phi = -\frac{\rho}{\epsilon_0} \tag{1.3}$$

where $\epsilon_0 = 8.854 \times 10^{-12} F/m$ is the dielectric permitivity of the vaccum.

1.4.2 Boltzmann Electrons

The charge density is obtained from computing the difference between the positive charges of the ions and the negative charges of the electrons. However for our applications one may use a simplified model for the electrons under certain conditions. The model is Boltzmann electrons. The Boltzmann Electrons [8] follow a specific distribution function, and this function is used to determine the density of electrons in the domain. The pressure of the electron plasma is given by:

$$p = m_p n(\vec{x}, t)\langle c^2 \rangle / 3 \tag{1.4}$$

Where m_p is the mass per gas molecule and c is the random velocity of the particle. The mean kinetic energy per particle in terms of the Boltzmann constant k is:

$$\langle E \rangle = m_p \langle c^2 \rangle / 2 = 3kT/2 \tag{1.5}$$

From Equations 1.4 and 1.5 we have a new expression for the pressure:

$$p = nkT \tag{1.6}$$

Which is the equation of state for an ideal gas. Considering the plasma is in a conservative force field with Electrostatic field Ψ:

$$\vec{F} = -\nabla \Psi(\vec{x}) \tag{1.7}$$

Then the steady-state momentum balance equation is:

$$n\vec{F} = \nabla p \tag{1.8}$$

With p defined in Equation 1.6, Equation 1.8 integrates to:

$$n(\vec{x}) = n_0 exp[-\Psi(\vec{x})/kT] \tag{1.9}$$

The exponential term is called the Boltzmann factor and can be rewritten in this case as:

$$n(\vec{x}) = n_0 exp[-e\phi(\vec{x})/kT] \tag{1.10}$$

Where e is the electrostatic charge and ϕ is the electrostatic potential. This is the model used in the field solver to obtain the electron distribution when the simulation only uses ions.

1.4.3 Plasma Characteristics

Plasma as a gas can be characterized by many parameters such as its density or temperature. These parameters give indications on the behavior of the plasma. A short definition of these parameters is presented below and table 1.2 sums up these parameters with examples of plasma [2].

We can rewrite Equation 1.3 by definition of the charge density:

Plasma	n (m^{-3})	T (keV)	B (T)	ω_{pe} (s^{-1})	λ_D (m)	$n\lambda_D^3$	ν_{ei} (Hz)
Interstellar	10^6	10^{-5}	10^{-9}	$6 \cdot 10^4$	0.7	$3 \cdot 10^5$	$4 \cdot 10^8$
Solar wind (1AU)	10^7	10^{-2}	10^{-8}	$2 \cdot 10^5$	7	$4 \cdot 10^9$	10^{-4}
Ionosphere	10^{12}	10^{-4}	10^{-5}	$6 \cdot 10^7$	$2 \cdot 10^{-3}$	10^4	10^4
Solar corona	10^{12}	0.1	10^{-3}	$6 \cdot 10^7$	0.07	$4 \cdot 10^8$	0.5
Ion thruster	10^{15}	10^{-3}	-	$2 \cdot 10^9$	$4 \cdot 10^{-4}$	$7 \cdot 10^4$	10^3
Arc discharge	10^{20}	10^{-3}	0.1	$6 \cdot 10^{11}$	$7 \cdot 10^{-7}$	40	10^{10}
Tokamak	10^{20}	10	10	$6 \cdot 10^{11}$	$7 \cdot 10^{-5}$	$3 \cdot 10^7$	$4 \cdot 10^4$

Table 1.2: Plasma characteristic parameters.

$$\nabla^2\phi = e(n_e - n_i)/\epsilon_0 \tag{1.11}$$

If both the ions and the electrons follow a Maxwell distribution in the plasma and have the same temperature kT:

$$\begin{aligned} n_e &= n_{e0} exp(-e\phi/kT) \\ n_i &= n_{i0} exp(e\phi/kT) \end{aligned} \tag{1.12}$$

However the potential is defined to be zero at infinity thus the respective densities for electrons and ions are equal at infinity hence Equation 1.11 becomes:

$$\nabla^2\phi = en_0 \left[exp(-e\phi/kT) - exp(e\phi/kT)\right]/\epsilon_0 \tag{1.13}$$

We normalize this equation by defining the potential $\Phi = e\phi/kT$ and the length λ_D in meters as:

$$\lambda_D = \sqrt{\frac{\epsilon_0 kT}{n_0 e^2}} \tag{1.14}$$

With $X = x/\lambda_D$ Equation 1.13 becomes:

$$\frac{d^2\Phi}{dX^2} = [exp(\Phi) - exp(\Phi)] \tag{1.15}$$

The solution to this equation is:

$$\phi = \phi_w exp[-x/(2\lambda_D)] \tag{1.16}$$

Where ϕ_w is the potential at the reference point. The length λ_D is called the Debye Length and is the length of the screening effect of the electrons beyond which the plasma can be concidered quasi-neutral. From this characteristic length we can define the plasma parameter as:

$$g = \frac{1}{n_e \lambda_D^3} \tag{1.17}$$

which is a measure of the dominance of collective interactions over collisions.

The plasma oscillations are an example of such collective interactions. In a plasma the neutrality is assured by the mobility of the electrons that move in the plasma to compensate regional charge imbalance. As such they create an oscillatory motion that sligthly deviates from the equilibrium state and its frequency is an important parameter known as the plasma frequency and is defined as the ratio of the electron thermal speed over the Debye length:

$$\omega_e = \frac{\sqrt{kT/m_e}}{\lambda_D} \tag{1.18}$$

These parameters help characterized the different types of plasma present in the universe. The plasma used on Ion thruster is a mid range plasma with a density higher

than any space plasma but not high enough to be a collision driven plasma. Thus the only effect that has to be taken into account in the simulation is the collective behavior due to the self created and external fields present in the simulation domain.

Chapter 2

The Particle In Cell Code

This chapter first introduces the Particle In Cell (PIC) algorithm and then briefly presents the DRACO module developped at Virginia Tech for the Coliseum project.

2.1 The PIC Algorithm

PIC is an algorithm used to determine the motion of the particles forming the plasma. A PIC code moves millions of particles on a mesh. This mesh represent the computational domain. The PIC code moves the particles according to the equation of motion given by Equation 1.1. The particles have the size of the mesh cell and the motion is tracked for the centroid of the particle. Once the particles have been moved, the charge density is computed at the mesh nodes. This is done by weighing the particles in a cell onto the nodes of that cell. The weighing scheme is presented in Figure 2.2 and will be explained in the next paragraph. The field can then be computed from the charge density according to the Poisson equation (Equation 1.3). The force exerted on the particle is computed from the value of the field on each of the eight nodes of the cell. This PIC cycle is depicted in Figure 2.1 [1]. To weigh either the charge density or the force a linear weighing method is used and follows the scheme in Figure 2.2. Since a particle has the size of the cell the centroid of the particle is used

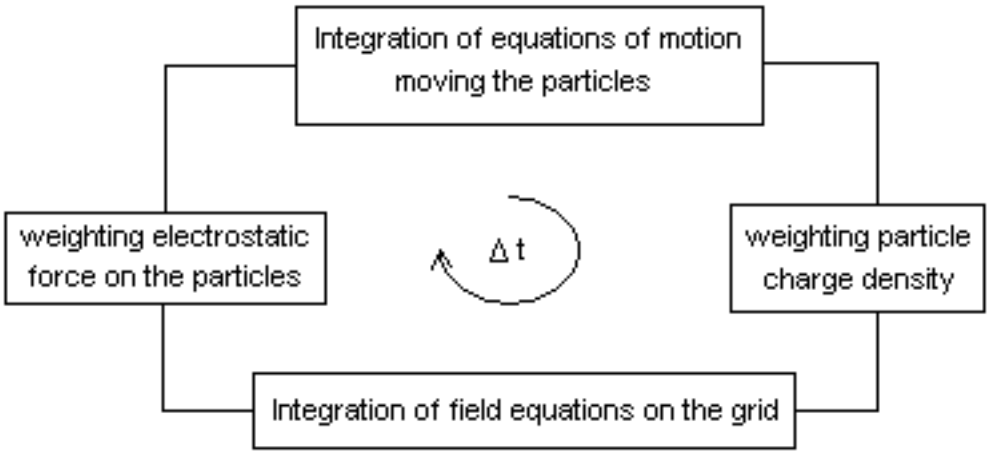

Figure 2.1: The PIC algorithm cycle.

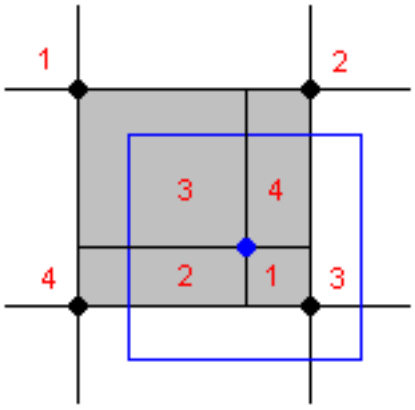

Figure 2.2: The PIC weighing method.

to weigh the particle on the surrounding nodes. In 3 dimensions the centroid divides the cell into eight smaller volumes. The value of these volumes are the weighting coefficient to the opposite node. The equation that relates the position of the particle centroid to the positions of the nodes is:

$$\vec{c} = \frac{\sum_{i=1}^{8} \vec{n_i} v_i}{dxdydz} \tag{2.1}$$

In the following equations let us assume that the volume v_i is normalized by the volume of the cell. Thus when weighing the charge density Equation 2.1 becomes:

$$\rho_i = \frac{1}{v_{ij}} \sum_j Z_j \tag{2.2}$$

Where Z_j is the charge of the j^{th} particle in the cell and v_{ij} is the volume made by the j^{th} particle opposite the i^{th} node. The force exerted on the j^{th} particle by all the surrounding nodes is:

$$\vec{F_j} = \sum_i Z_j \vec{E_i} v_{ij} \tag{2.3}$$

PIC codes can be either electrostatic or electromagnetic. In an electrostatic PIC code, the electric field is solved from the Poisson equation. In an electromagnetic PIC code, both the electric and magnetic field are solved from the Maxwell's equations. The PIC code in this thesis uses an electrostatic PIC method.

The particles moved by the PIC code can be of different species. If these species are ions and electrons the PIC code is said to be a Full PIC code, since it is modelling all the species in the plasma. A fluid model is used in this thesis to model the electrons. The fluid model used is the Boltzmann electrons and was described in section **??**.

2.2 Introduction to Coliseum

Coliseum is a spacecraft plasma interactions simulation software developed by the Air Force Research Laboratory at Edwards AFB [6]. This software contains different modules developed by teams from MIT and Virginia Tech. Virginia Tech is responsible for the DRACO module and its cartesian mesh VOLCAR. DRACO contains all the tools to run a PIC simulations. Indeed DRACO has different types of sources plus the sources from Coliseum, different field solvers including the new Immersed Finite Element Solver [9] developed here at CapLAB and a full PIC algorithm. The user can choose from all these options in the input files and run plasma simulations ranging from ray tracing to Full PIC.

2.3 DRACO

DRACO uses an Electrostatic PIC code with linear weighting scheme originally developped by Dr Joseph J. Wang and ported from FORTRAN 77 to C by Lubos Brieda. DRACO is a powerful and nearly self sufficient module of Coliseum, it has its own field solvers, sources to emit both ions and electrons, and many diagnostics features. DRACO uses the VOLCAR Cartesian mesh also developped at Virginia Tech by Lubos Brieda [5][4].

2.3.1 Solvers

DRACO [3] is equipped with several field solvers with different capabilities, one Boltzmann solution, three Finite Difference solvers, and one Finite Element solver (Table 2.1). The Boltzmann *solver* does not actually solve the elliptical partial differential equation for the field, it only inverts Equation 1.10 to obtain the potential. As such it should not be used for computations but only for testing purposes.

The Finite Difference solvers are the fastest real solvers in DRACO. They all have

Solver	Type
$BOLTZMANN$	N/A
$DADI$	Finite Difference ADI
GS	Finite Difference Gauss-Seidel + SOR
GS_MPI	Finite Difference Gauss-Seidel + SOR
IFE	Immersed Finite Element

Table 2.1: Available solvers in DRACO.

both the linear and non-linear capability. When non-linear they use Equation 1.10 to obtain the electron distribution. The first solver came with Dr Wang's legacy PIC code, it is called $DADI$ for Dynamic Alternating Direction Implicit and uses the ADI scheme. The two other FD solvers are very similar to each other except than one was designed for parallel computations. GS and GS_MPI both use the Gauss-Seidel method with a SOR acceleration. The acceleration parameter ω is currently hardcoded to the value 1.6. The parallel Gauss-Seidel solver uses a Red-Black cherckerboard scheme to allow for a parallel use. The scheme will be desribed in chapter 3. Finally DRACO has the possibility to be used with an Immersed Finite Element solver developped by Raed Kafafy [9].

2.3.2 Sources

Coliseum uses sources to insert particles into the simulation domain. Sources are attached to specified surfaces loaded along the geometry. Several type of sources are available, they range from simple background sources to complex thruster sources (Table 2.2). The source used in this simulation is $ION\_THRUSTER$. A source emits a particle from an element of the surface of which it is attached. This source has the particularity of changing the mass flow rate of any of those small elements to create a distribution of the mass flow over the whole surface. This is important because an ion thruster emits more particles from its center than it does from the off center region. This source was written to better model the current density distribution at

Source	Type
$MAXBACK$	background
$LINE$	beam
$UNIFORM\_BEAM$	beam
$MAXSTREAM$	beam
$NSTAR\_CEX$	thruster
$ION\_THRUSTER$	thruster

Table 2.2: Available source models in DRACO.

the exit of the thruster. Indeed injecting the particles following a specific distribution is critical to the accuracy of the simulation.

The source takes the usual parameters as input, such as average mass flow rate, particle velocity and plasma temperature. However it requires a few more inputs to create the proper distribution. This is done by adding the coefficients of a polynomial fitting the current density profile we want to obtain.

This source in addition to the use of curved emitting surface dramatically increases the accuracy of the simulation.

2.4 Conclusions

Coliseum with the DRACO-VOLCAR module is the plasma simulation framework on which the parallel code for this thesis was written. The PIC algorithm is simple enough to be modified to work on parallel and the availability of different solvers and sources rendered the development easier. The actual parallel code will be presented in the following section.

Chapter 3

Parallelization

This chapter present the parallel version of Coliseum. The first section of this chapter gives a short introduction to MPI, the second section presents the algorithms used in the parallelization of Coliseum and finally the last section gives an in depth explanation of the parallel field solver.

3.1 The Message Passing Interface Library

3.1.1 Introduction to MPI

The Message Passing Interface or MPI [10] is a set of API functions that enables progammers to develop high performance parallel programs. The low level routines contained in MPI are communication functions. These functions pass messages between serial processes to create a parallel job.

MPI was designed for high end parallel commercial machines such as IBM SP, SGI Origin, but is now at the front end of most *off the shelf* clusters. MPI has support for both shared and distributed memory which is the reason of its success, the same code can be used on different architectures with little or no modifications.

3.1.2 MPI Communications

MPI provides programmers with a wide range of functions, from basic blocking communications to building your own MPI data types. However only basic communications were required for this project.

Two kinds of communication exist in MPI, blocking and non-blocking. A blocking *send* communication will pause the execution of the code as long as the data stored in the sending buffer has not been completely send to the recipient, the *receive* works the same way. This method allows for synchronization of the code since every process will wait for one another.

First consider 2 processors. Processor 1 is trying to send data to processor 2, and both processors execute the code until they reach a communication command. Once the processors reach the command, processor 1 will send its data to processor 2, while processor 2 will halt the execution of the code until it has received the data from processor 1(see Figure 3.1). If the data send by processor 1 is smaller than 4096 bytes then the data will be buffered in the MPI layer and processor 1 can continue the execution of the code, however if it is above 4096 bytes then processor 1 will halt the execution of the code until the data has been send. This limit is known as the *eager limit* [13] and decides the behavior of the Send command. If the message size is small enough then the cost of copying the message into the buffer is less than the synchronization overhead resulting from the blocking of the sending task. When the message size increases above the limit, then situation is the opposite. The limit exists to provide the best compromise, even though this behavior might be the source of deadlocks. The pseudocode presented in Figure 3.1 will work no matter what the size of data is. However the parallel code presented in this thesis is written so that every processor will execute the same code as presented in Figure 3.2. Once the two processors reach the send command, they will thus try to send data to each other. Again two scenarios occur, first the data in each processor is smaller than 4096 bytes

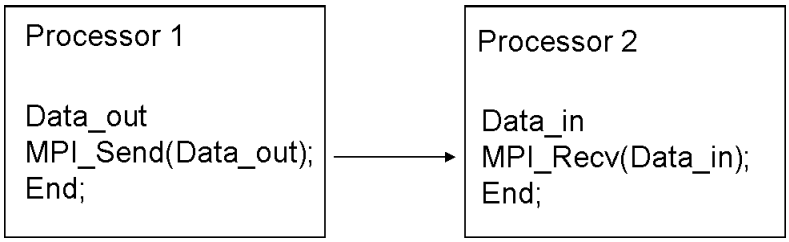

Figure 3.1: Basic MPI communications between 2 processors.

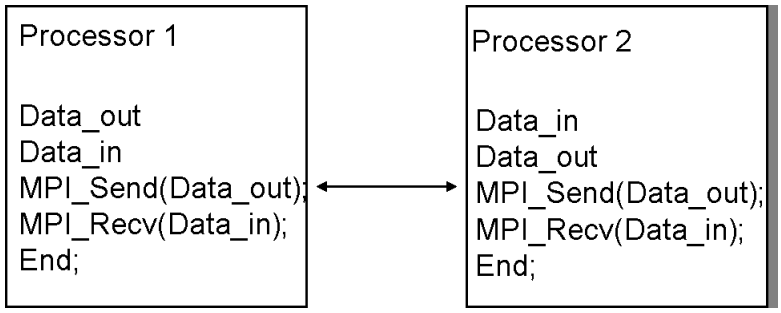

Figure 3.2: MPI communications between 2 processors using $MPI\_Send()$ and $MPI\_Recv()$.

or second it is larger. If the data is smaller then both processors will buffer the data to be send and carry on with the execution of the code which is to receive the data, in this case all is well. Now, if the data is larger than 4096 bytes then both processor will stop at the sending command since none of them can buffer the data and thus carry to the next line of code to receive the data. In this case all the processors try to send but none of them is trying to receive, the situation ends up in a deadlock. A way to solve this problem would be to have the second processor to receive the data first and then try to send its own data, however the code is written the same for each processor, hence another solution needs to be found.

The solution is to use another type of communication. The pseudocode for the solution is presented in Figure 3.3, it is the communication scheme used in the Coliseum parallel code. Here the code remains the same for both processors but the function called $MPI\_Sendrecv()$ will allow the processor to send and receive data at the same

time, thus avoiding a deadlock.

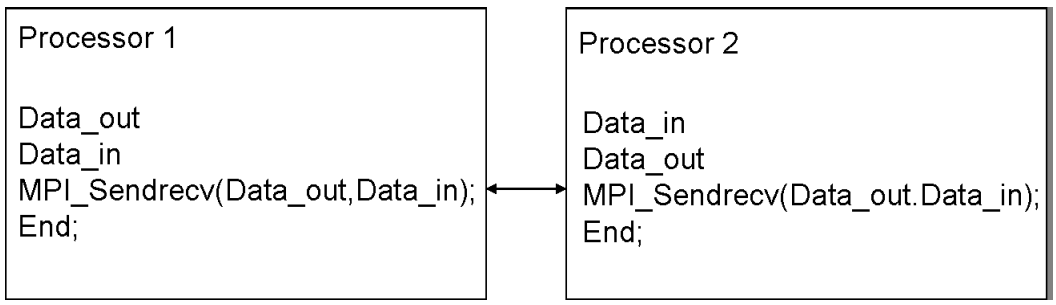

Figure 3.3: MPI communications between 2 processors as used in the Coliseum parallel code.

3.2 The Parallel PIC

This section presents the parallel algorithms used in Coliseum. The Virginia Tech Draco module to the Coliseum software was originally written as a single processor code, however it became evident that a higher performance version of this module was needed. Indeed the simulations ran with Coliseum are extremely computationaly intensive and require large amounts of memory to store both the domain and the particles.

The Draco module is in constant development, it was then decided to write the parallel version of it not as a separate module but as a transparent layer to the existing code. As such this parallel version could benefeciate of any improvement made to the code, it was also easier since only a few functions needed updating. The whole parallel code is encapsulated within the original code and can be either actived or deactivated at compile time by defining the variable MPI (-DMPI).

3.2.1 Mesh Setup

One of the most important part of the parallel code is the decomposition of the domain. The domain is composed of different objects, first the mesh of the domain on

which the simulation variables will be weighted, then the geometry contained within the domain; both the geometry and the mesh have to be divided among the processors. The process was rather easy to implement on a cartesian mesh.

Processor Coordinates

The first step for parallel domain decomposition is to provide each processors with data about their neighbours. The function called $GetProcCoord()$ will determine the location of the processor in the entire domain. In 2-Dimensions the processor layout is presented in Figure 3.5. If there is no processor on a face neighbour is set to -1 otherwise it is set to the rank of the processor. The number of cell in each directions (n_x, n_y, n_z) is given as input to the simulation as well as the number of processors in each directions (np_x, np_y, np_z). When the parallel code is initialized through the command $MPI\_Init()$ each processor is assigned a unique number called rank. this number can be viewed as an ID number used to differentiate the processors and as a mean to communicate with each other. This processor rank is used to determine its location and thus its coordinates. The algorithm in 1D is as follow, as long as the rank is smaller than the number of processor in this direction the rank of the neighbours in this direction are simply rank+1 and rank-1. If the rank is equal to one of the extremums, 0 or n_x, then this processor only has one neighbour in this direction. In 3Dimensions the checks in pseudocode are:

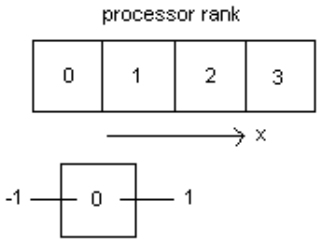

Figure 3.4: Processor Layout

$$
\begin{aligned}
&if((rank+1)mod(np_x) \neq 0) \Rightarrow neighbour_{x+} = rank+1 \\
&if((rank)mod(np_x) \neq 0) \Rightarrow neighbour_{x-} = rank \\
&if((rank/np_x+1)mod(np_y) \neq 0) \Rightarrow neighbour_{y+} = rank+np_x \\
&if((rank/np_x)mod(np_y) \neq 0) \Rightarrow neighbour_{y-} = rank-np_y \\
&if((rank/(np_x*np_y)+1)mod(np_z) \neq 0) \Rightarrow neighbour_{z+} = rank+(np_x*np_y) \\
&if((rank/(np_x*np_y))mod(np_z) \neq 0) \Rightarrow neighbour_{z-} = rank-(np_x*np_y)
\end{aligned}
\tag{3.1}
$$

If the $if()$ statement fails the neighbour is set to -1 as the processor is a boundary processor. The coordinates of the processor, or more precisely the relative coordinate of the processor with respect to its neighbour are stored in a simple data structure. Each processor has an array containing the rank of each processor adjacent to each of the 6 faces of its local domain.

$$
Proc \rightarrow neighbour[0-5] = rank_{adjacent\,processor} \tag{3.2}
$$

Processor Layout and Local Domain

The processors will be layed out starting at the origin then filling the first row along the x-direction then the y-direction and finally the z-direction. The process is the same as the one to fill a box of square size chocolate candy. You would start at one corner of the box putting one candy next to the other to form a column, then when that column is full you form another column until you totally fill the bottom of the box, then you start over for the second layer of chocolate and so on until you fill up the entire box. The result can be seen in Figure 5.4. This operation is done by the function called $SetLocalDomain()$. The algorithm is simple and similar to the one used to determine the processor coordinates.

The local number of cells is set to:

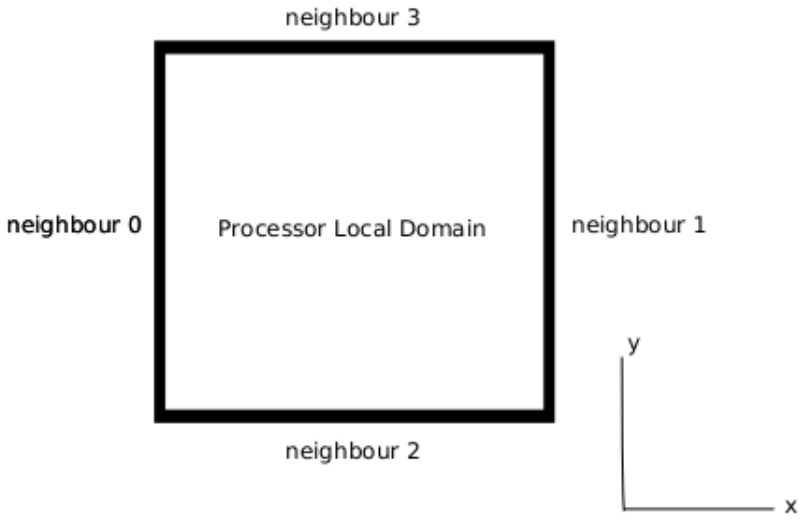

Figure 3.5: Processor Layout

$$\begin{aligned} n_{xlocal} &= n_x/np_x \\ n_{ylocal} &= n_y/np_y \\ n_{zlocal} &= n_z/np_z \end{aligned} \tag{3.3}$$

The local origin is obtained from:

$$\begin{aligned} i_{start} &= [(rank)mod(np_x)] * n_{xlocal} \\ x_{start} &= i_{start} * dx + x_0 \\ j_{start} &= [(rank/np_x)mod(np_y)] * n_{ylocal} \\ y_{start} &= j_{start} * dy + y_0 \\ k_{start} &= [(rank/(np_x * np_y))mod(np_z)] * n_{zlocal} \\ z_{start} &= k_{start} * dx + z_0 \end{aligned} \tag{3.4}$$

3.2.2 Particle Push and Adjust

Once the domain has been decomposed among the processors the simulation can start injecting particles in the domain. This part of the code was not changed to work on

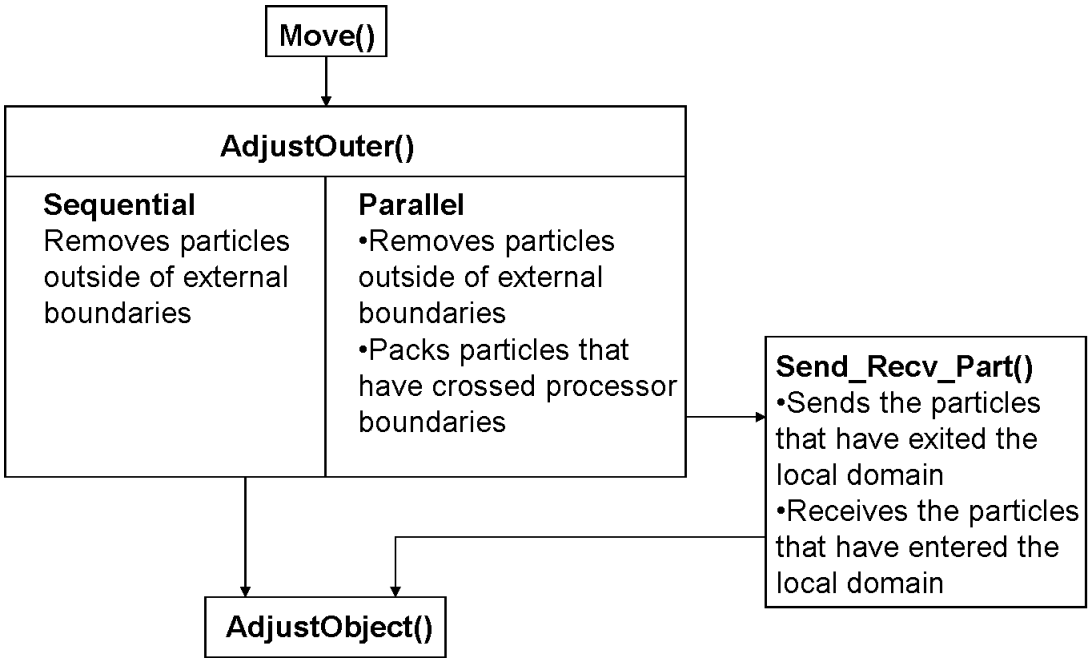

Figure 3.6: Flowchart of the particle adjust process.

parallel. However adjusting the particle to the local domain, $AdjustOuter()$, required quite a bit of work. In the sequential version of the code a particle could only be either inside or outside the domain, with the parallel version a particle can be outside the local domain but still inside the computational domain. This particularity occurs when a particle crosses a domain boundary which is a processor boundary, it means then that a particle has to be shipped to that processor and removed from the local domain. A flowchart of the process is shown in Figure 3.6. This part of the parallel code is one of the most network intensive. At every time step, all the particles of the domain have their location checked against the domain boundaries. Each particle that crosses over a processor boundary is packed into a buffer and at the end of the check each of the six buffers associated with each face is send to the corresponding processor. The processor also receives a particle buffer array from each face and adds these new particles into its particle array. A bug in this version of the code was found, it turns out that from time to time a particle during its push can move across two

processors and thus be send to the wrong one. This problem occurs when a particle is inside one of the corner cell of the domain, it is rare but not uncommon. The bug was temporarily fixed by adjusting the particle twice at each time step, this solution is obviously not computationaly performant and will need fixing in the future.

3.2.3 Adjusting the Processor Boundary

The code uses shared node between processors, which means that at a processor boundary the nodes belong to both the processor on the left and the one on the right, see Figure 3.7. Thus the values computed at those particular nodes are incomplete within the processor, since the adjacent cell from the next processor is also a contributor to the value being computed. For example in the case of the charge density, a boundary node sits between two cells that belong to two different processors. The charge density at this boundary node is the sum of the contribution from the left and right cell:

$$\rho_{boundary} = \rho_{left} + \rho_{right} \tag{3.5}$$

To perform this update, boundary nodes from an adjacent face are copied into a buffer and send to the neighbour. The neighbour unpacks the buffer and adds the value from the buffer into its own boundary nodes. This operation is repeated for each face. This algorithm inherently takes care of boundary nodes that are on edges and corners. Those nodes have the particularity of having contribution from more than two processors.

3.3 The Field Solver

The parallel field solver was one of the latest addition to the parallel code. The method employed by the solver had to be an easy one to implement so that it could

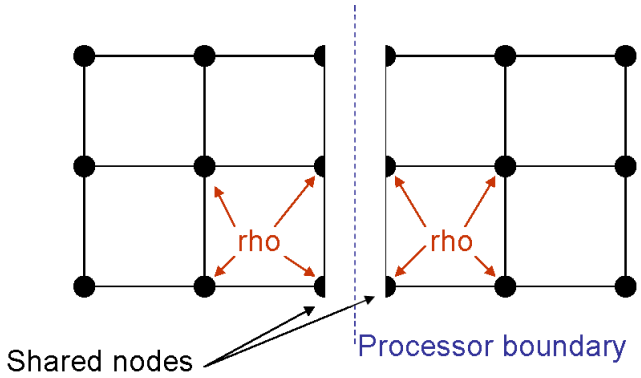

Figure 3.7: Schematic of the shared nodes update.

written and debugged in a short amount of time. Draco is already equipped with a Gauss-Seidel solver so it was decided to use this method with a Red-Black checkerboard scheme to build a parallel version of it. The Red-Black checkerboard allows half of the nodes in the domain to be computed from the other then allowing the two types of nodes (red and black) to be updated independently. This scheme is very popular in parallel solvers.

3.3.1 The Gauss-Seidel Method with SOR Acceleration

In order to numerically solve the Poisson equation, a finite difference representation of the equation is needed. Equation 1.3 can be rewritten:

$$\frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} + \frac{\partial^2\phi}{\partial z^2} = -\frac{\rho}{\epsilon_0} \tag{3.6}$$

Using a central differentiation for second order partial derivatives gives a 7-point finite difference formula [12]:

$$\frac{\phi_{i+1,j,k} - 2\phi + \phi_{i-1,j,k}}{\Delta x^2} + \frac{\phi_{i,j+1,k} - 2\phi + \phi_{i,j-1,k}}{\Delta y^2} + \frac{\phi_{i,j,k+1} - 2\phi + \phi_{i,j,k-1}}{\Delta z^2} = -\frac{\rho}{\epsilon_0} \tag{3.7}$$

Which can be rearranged to:

$$\begin{aligned} C &= \frac{\Delta x^2 \Delta y^2 \Delta z^2}{2(\Delta y^2 \Delta z^2 + \Delta x^2 \Delta z^2 + \Delta x^2 \Delta y^2)} \\ \phi &= C \left[\frac{\rho}{\epsilon_0} + \frac{\phi_{i+1,j,k} + \phi_{i-1,j,k}}{\Delta x^2} + \frac{\phi_{i,j+1,k} + \phi_{i,j-1,k}}{\Delta y^2} + \frac{\phi_{i,j,k+1} + \phi_{i,j,k-1}}{\Delta z^2} \right] \end{aligned} \tag{3.8}$$

Equation 3.8 is the finite difference representation of the Poisson equation. However the simulation uses a fluid model for the electrons, thus the charge density term in Equation 3.8 needs to be changed to:

$$\rho = \rho_{ions} - n_e \tag{3.9}$$

Where n_e is obtained from Equation 1.10, which gives:

$$\rho = \rho_{ions} - n_0 exp[-e\phi/kT] \tag{3.10}$$

But adding Equation 3.10 to Equation 3.8 makes this last equation non linear. To solve this non linear equation one can either linearize the equation using a Newton iteration [14] or update the right hand side of the equation at every solving iterations. The second solution is implemented in the parallel solver.

Equations 3.8 and 3.10 are the equations used by the solver to update the value of the potential at a node. The code is an SOR acceleration scheme to speed up the convergence of the solution. SOR stands for Successive Over Relaxated, which means that the solver over estimates the current solution by a small fraction. This procedure reduces the number of steps required to reach the solution. This method is implemented in one line:

$$\phi = \phi_{old} + \omega(\phi_{new} - \phi_{old}) \tag{3.11}$$

Where ϕ_{new} is the value obtained from Equation 3.8. The value of ω changes the convergence rate of the solver, however it is currently hardcoded to 1.6.

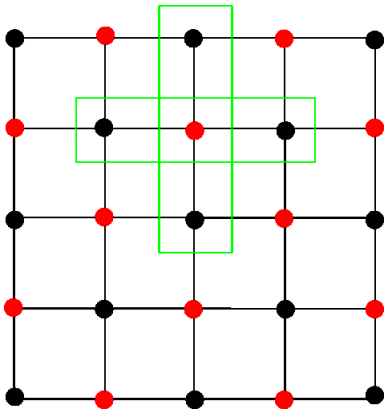

Figure 3.8: Schematic of the Red-Black checkerboard scheme.

3.3.2 The Red-Black Checkerboard Scheme

The Red-Blacks checkerboard scheme divides the nodes from each local domains into two kinds, the Red nodes and the Black nodes. Such division can be seen on Figure 3.8. The idea of this scheme is to have two different populations of nodes, and these two populations can be solved independently from each other. Recalling Equation 3.8, one can see that all is needed to obtain the new value of the potential at a node is the value from the six surrounding nodes. If the node being computed is a red node, the six surrounding nodes are black by definition, this proves that to update the nodes from one population, only the values of the nodes from the other population are required. The corresponding algorithm will first update the Red nodes from the values of the Black nodes using Equation 3.8, and then update the Black nodes using the newly computed values from the Red nodes.

A problem arises for the processor boundary nodes. A processor boundary node can have one or more of its surrounding nodes in a different processor. This problem is solved by adding guard cells to the buffer used for solving the field. DRACO does not use guard cells, hence the field solver will allocate a larger buffer array containing the local mesh and temporary guard cells. These guard cells are used to simplify the computations and are updated between all the processors at each solving iteration.

Boundary nodes can also be of Dirichlet or Newmann types. A Dirichlet node will not be updated as its value is fixed and specified from the input file, however a Newmann node needs to be updated according to the following equation:

$$\nabla\phi = \vec{0} \tag{3.12}$$

At each iteration the solver will first update the Red nodes, then the Black nodes, once the new potential distribution in the domain has been computed the solver updates the guard cells one face at a time. This procedure can be time consuming and will be investigated in Chapter 4. Finally the solver checks if the solution has converged by using the L2 norm of the residue vector. The residue is obtained from the difference:

$$\vec{R} = \vec{b} - A\vec{x} \tag{3.13}$$

in which $\vec{b} = \rho_{i,j,k}$, $\vec{x}$ is the solution vector (ϕ) and A is the finite difference scheme matrix. The L2 then is obtained from:

$$L2 = \sqrt{\sum_{i,j,k} R^2_{i,j,k}} \tag{3.14}$$

In $GS\_MPI()$ the solution has converged if two successive L2 norms have their difference smaller than the prescribed tolerance. This method is used since only the local residue is computed:

$$\begin{aligned} &if(L2_n - L2_{n-1} < tolerance)\, then\, local\_convergence = 1 \\ &MPI\_Allreduce(local\_convergence, convergence, MPI\_PROD) \\ &if(convergence == 1)\, then\, solution\, has\, converged \end{aligned} \tag{3.15}$$

Once all the local L2 norms have converged then the solver exits as the solution computed is within the tolerance specified.

3.4 Conclusions

This chapter gave a short introduction to the MPI library used to develop the parallel version of Coliseum. The algorithms implemented in the parallel code were then presented. These algorithms are important in a parallel code since they form the core of the code, indeed in a domain decomposition parallel simulation the computational domain has to be divided among the processors, and the processors have to be able to communicate with one another. The simulation moves the particles across the entire computation domain which sits across several processors, hence the code needs to be capable of exchanging particles between processors. Finally the code has to find a single solution to the field using results from different processors, the difficulty resides in the fact that the solution has to be continuous across the processors boundaries.
It is obvious that a parallel code performs more computations than a sequential code and spends a lot of time in communications with other processors, this in return can affect the performance of the simulation. This problem will be addressed in the following chapter, benchmarks and timing results were obtained on supercomputers for different number of processors and domain decompositions.

Chapter 4

Timing Results

This chapter presents the timing results obtained for the parallel code on supercomputers.

4.1 Parallel Efficiency

Supercomputers come in two different kinds regardless of their architectures, Shared memory and Distributed memory. In a shared memory system all the nodes share the same memory block whereas on a distributed memory system every node has its own memory block containing a part of the problem. However since each processor can not work independently of each other,they have to regularly send data to other nodes to keep the simulation consistent. The stream of data between processors is accomplished over a TCP/IP network using either Gigabit Ethernet or a high speed Myrinet network. These communications have to be made at least at every time iteration. These communications can become very time consuming when many processors are involved. Hence the more processor you have the longer the code will spend in communications thus relatively increasing the iteration time. The parallel efficiency measures the effect of increasing the number of processors on the computation time. The parallel efficiency is computed from a reference case, more precisely the itera-

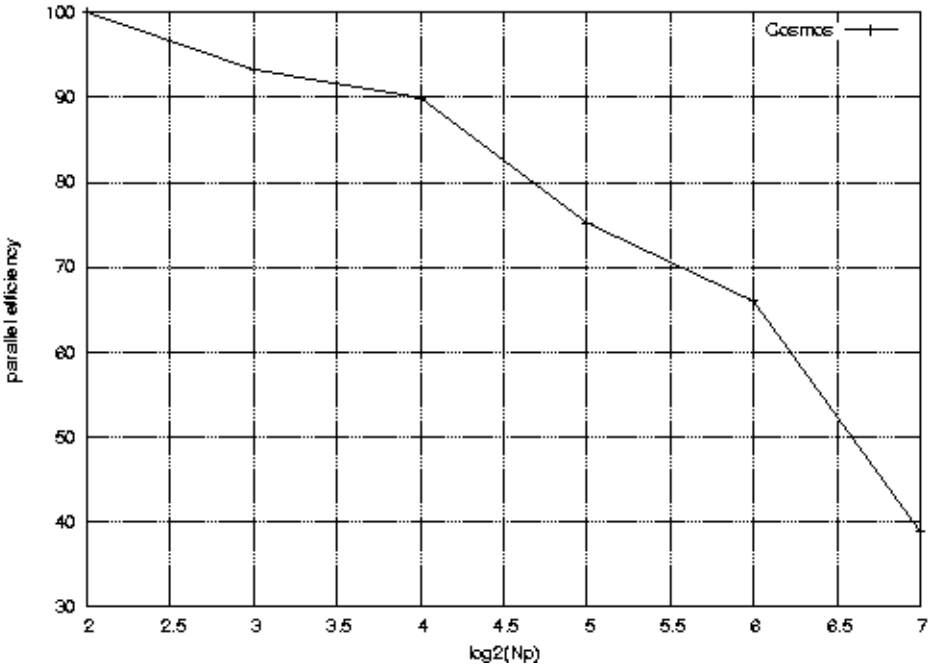

Figure 4.1: The parallel efficiency on JPL's Cosmos Supercomputer.

tion time at a specific time step. The reference case is chosen after a steady state is reached for the simulation. The parallel efficiency is the ratio of that reference value over the tested value [15]:

$$\epsilon = \frac{(t_{it})_{Ref}}{N_p t_{it}} \tag{4.1}$$

However if the reference case uses more than one processor, the equation becomes:

$$\epsilon = \frac{(N_p t_{it})_{Ref}}{N_p t_{it}} \tag{4.2}$$

Figure 4.1 shows the variation of the parallel efficiency on the JPL Cosmos supercomputer as I increase the number of processor from 4 to 128. The efficiency monotonically decreases from a 100% to about 40% for 128 processors. MPI uses blocking Send and Receive, so only one communication can occur at a time per processor. The simulation runs faster on more processors as it does on one but not at the maximum

theoretical speed up. For example from 1 to 2 processors the maximum theoretical speed up is 2, which means that the code can run twice as fast, but because of inter processor communications, less than a 100% of the CPU time is spend on computations, thus the speed up might be 95% of 2, depending on the quality of the code and the load balance.

The parallel efficiency is a good quantifier of the *balance* of the parallel code. Load balancing is an important concept in parallel computations. A well balanced code would have all of the processors working at peak load at all time. However because we are using domain decomposition, there are parts of the domain where not much work is required. For the moment the local domain size is fixed on a processor which means that the only way to balance the code is to chose the most efficient domain decomposition possible. The parallel efficiency can be modified greatly by the domain decomposition, it depends on the geometry and the specificities of the problem. Some cases are more sensitive than others. For example I found out that for 8 processors the convergence of the field solver depended on the distribution of processors along the domain dimensions, some distributions were more performant than others. The tested ditributions were in x y z: 2x4x1 4x2x1 and 2x2x2. After several tests it was found that the most performant combination for this test case (Figure 5.13) was 2x4x1. Indeed having 2 processors in the z-direction which is the flow direction is useless half of the time since for the first half of the simulation no particles have reached the second part of the domain. The geometry also has a symetry along the x-plane, thus having 4 processors along the y-direction yielded the best results.

4.2 CPU Hours

Most supercomputers return the diagnostics on the simulation runs, one of them is the total CPU time (see Table 4.1) which can be converted in units of total CPU

$\#Processors$	$t_{it}(s)$	$t_{field}(s)$	CPU hours	time	efficiency (%)
4	11.19	10.33	14	3.5h	100.00
8	6.00	5.3	18.6	2h20min	93.25
16	3.11	2.82	17.4	65min	89.95
32	1.86	1.62	15.2	28.5min	75.20
64	1.06	0.83	19.8	18.6min	65.98
128	0.9	0.44	17.1	8min	38.85

Table 4.1: The timing results on JPL's Cosmos Supercomputer.

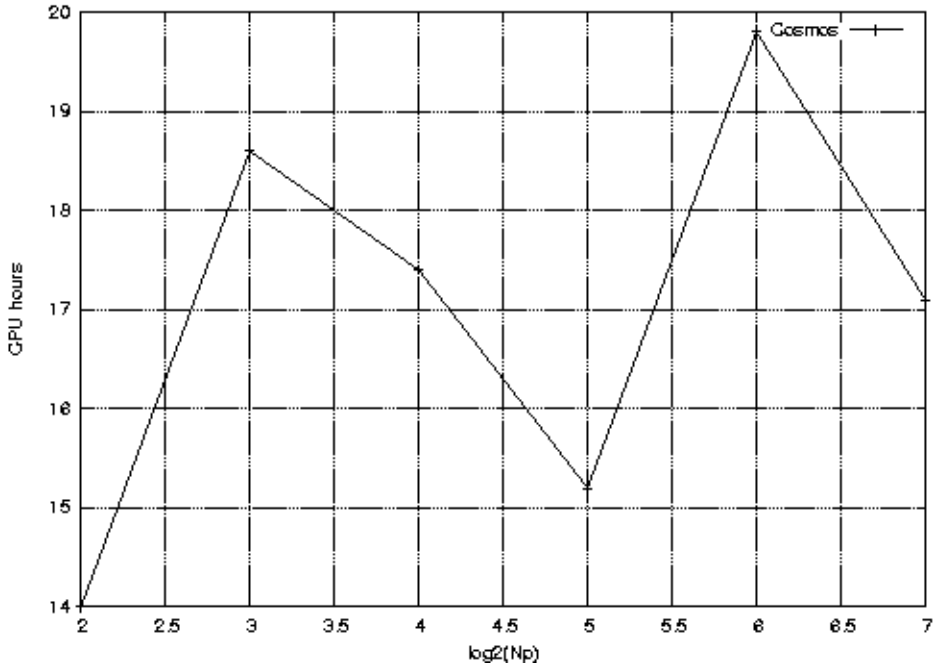

Figure 4.2: The CPU time evolution on JPL's Cosmos Supercomputer.

hours added over all the processors. It is interesting to see (Figure 4.2) that the plot of CPU hours versus the number of processors used is everything but monotone. This plot can be used to determine the most cost effective solution since the pricing of supercomputers is calculated from the CPU hours. A good solution would be a compromise between the smallest CPU hours and the shortest computation time.

4.3 Cost of Communications

The results in this section were obtained from System X. The previous two sections presented the efficiency of the parallel code. This section will highlight the reason for the decrease in parallel efficiency as the number of processors is increased.

The case ran to evaluate the communication cost is the same as in section 4.1. The number of processors tested ranged from 8 to 128. The value of the timing were taken at steady state. The results are presented in Table 4.2 and 4.3 and in Figures 4.3 and 4.4.

Figure 4.3 shows the evolution of the ratio of the time spend in solving the field over the total iteration time. This ratio decreases as the number of processors increases, this means that for higher number of processors some computation time is spend in another part of the code. However since these values are obtained for a same simulation at the same iteration, the total number of particles is the same. As a result the time must be spend in packing and distributing the particles, a communication overhead that becomes less negligeable with the number of processors used.

Figure 4.4 shows the evolution of the communication overhead within the field solver. The ratio increases with the number of processors. However it is interesting to note that even though the solving time and communication time both decrease, the ratio increases with the number of processors. This peculiar behavior is due to the fact that by increasing the number of processors on a fixed domain, the local domain keeps getting smaller, thus the solver takes less time to converge. Also as the local domain becomes smaller the size of the guard cells in the solver becomes smaller, thus the communications take less time to complete. However the computer takes less time to solve the field than it does performing the communications. Indeed the speed at which the communication is done is not dependent on the computer but on the speed and type of the network. As a result the ratio of the communication overhead within the solver increases.

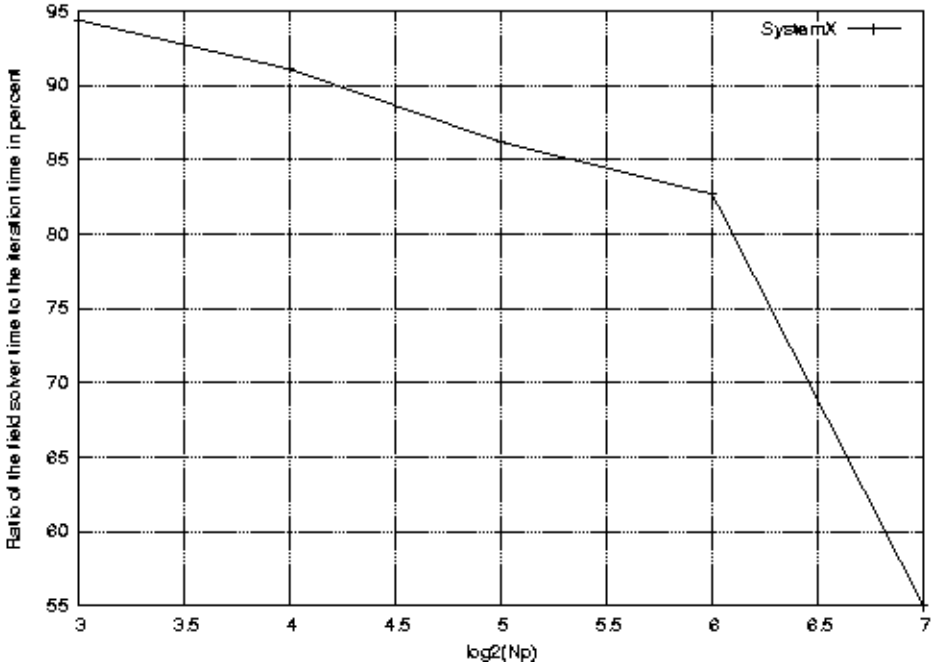

Figure 4.3: The relative time of the field solver per iteration.

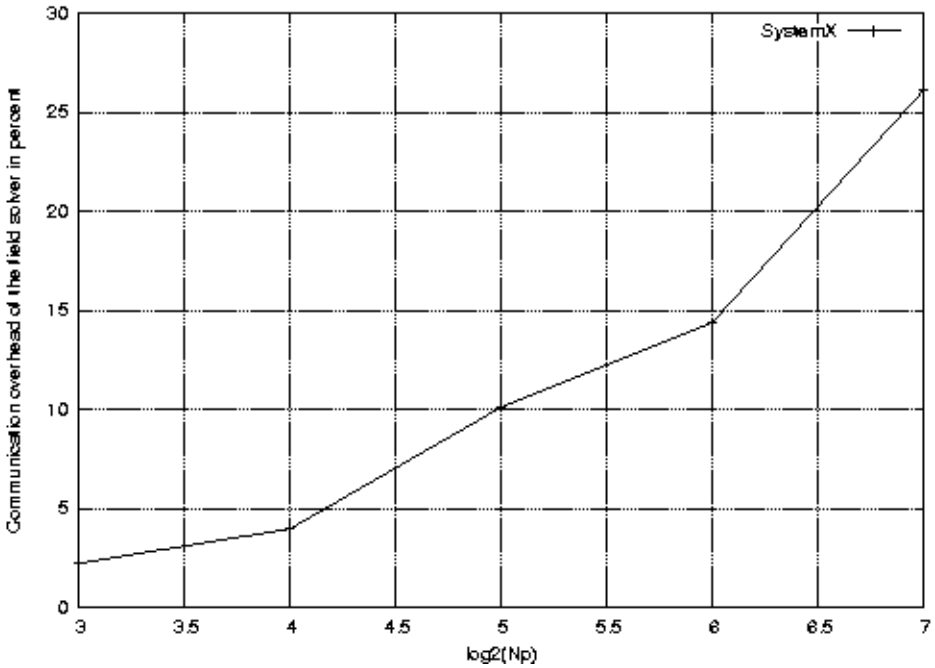

Figure 4.4: The communication overhead of the field solver per iteration.

$\#Processors$	$t_{it}(s)$	$t_{phi}(s)$	Ratio
8	10.11	9.51	94.1%
16	4.7	4.28	91.1%
32	1.81	1.56	86.2%
64	1.1	0.91	82.7%
128	0.6	0.33	55%

Table 4.2: The relative time of the field solver per iteration.

$\#Processors$	$t_{phi}(s)$	$t_{comm\,phi}(s)$	Ratio
8	9.51	0.214	2.25%
16	4.28	0.171	3.99%
32	1.56	0.158	10.13%
64	0.91	0.131	14.4%
128	0.33	0.086	26.1%

Table 4.3: The communication overhead of the field solver per iteration.

4.4 Conclusions

This chapter gave presents the efficiency of the parallel code and explains the decrease in performance as the number of processors used is increased. The time spend in communications is the overhead imposed on a parallel code. The performance of a parallel code depends on how well the code is written to limit inter processor communications.

Chapter 5

Physics Results

This section presents and discusses the results obtained with the parallel code. The code was first validated with a simple test case, then the code was used to simulate the plasma plume of an array of ion thrusters.

5.1 Plasma Flow around a Sphere

A validation of the parallel code was required before the code could be used for engineering applications. A simple test case was used to verify that the results obtained on multiple processors agree with those obtained on a single processor. The test case chosen was a plasma flow around a sphere. The set up is simple, a charged conducting sphere is immersed in a plasma flow (Figure 5.1). A stream source of Oxygen ions is used for the simulation, the plasma flows in the Z-direction. This test simulates a spherical object orbiting the earth in the upper atmosphere. The velocity of the ions is the orbiting velocity at 100 kilometers. The plasma density is chosen to match the conditions at this altitude. In order to validate the code, the code was run 3 times in parallel with different domain decompositions as the number of processors was increased. With two processors the domain was divided once longitudinally, with 4 processors the domain was divided twice longitudinally (once in x and once in y)

Parameter	Value
$\dot{m}$	$8.2 \cdot 10^{-12}$ kg/s
T	509K
v_i	$7.7 \cdot 10^3$ m/s
dt	$5 \cdot 10^{-8}$ s
T_{eRef}	1 eV
ϕ_{Ref}	5 V
n_{eRef}	$1 \cdot 10^{12}$ $part/m^3$
spwt	$5 \cdot 10^3$
ϕ_w	-100V

Table 5.1: Input parameters to the plasma flow around a sphere case.

and finally with 8 processors the domain was divided once in each directions. Figures 5.2, 5.3, 5.4 show the decompositions. In each case the potential and the charge density are compared with the one processor case, in order to be consistent the one processor case used the sequential Gauss-Seidel solver present in DRACO while the parallel cases used the parallel Gauss-Seidel solver. The results for the electrostatic potential are presented in Figures 5.5-5.8 while those for the charge density are presented in Figures 5.9-5.12. The Figures show that indeed the parallel results agree with the sequential results, the potential contour lines are identical and the charge density values and contours are the same in all the cases. The most critical part in parallel computations is handling the inter processor boundaries. In fact this can lead to serious problems such as discontinuity across the boundaries and resulting in totally different solutions in each local domain. However it is not the case here, the solution is definitely continuous across the processor boundaries.

The results from Chapter 4 and from this section validate the parallel code as being accurate and efficient. The solution obtained with the parallel code is identical to the one obtained with the sequential code and there is a significant speed up when using the multiprocessor code.

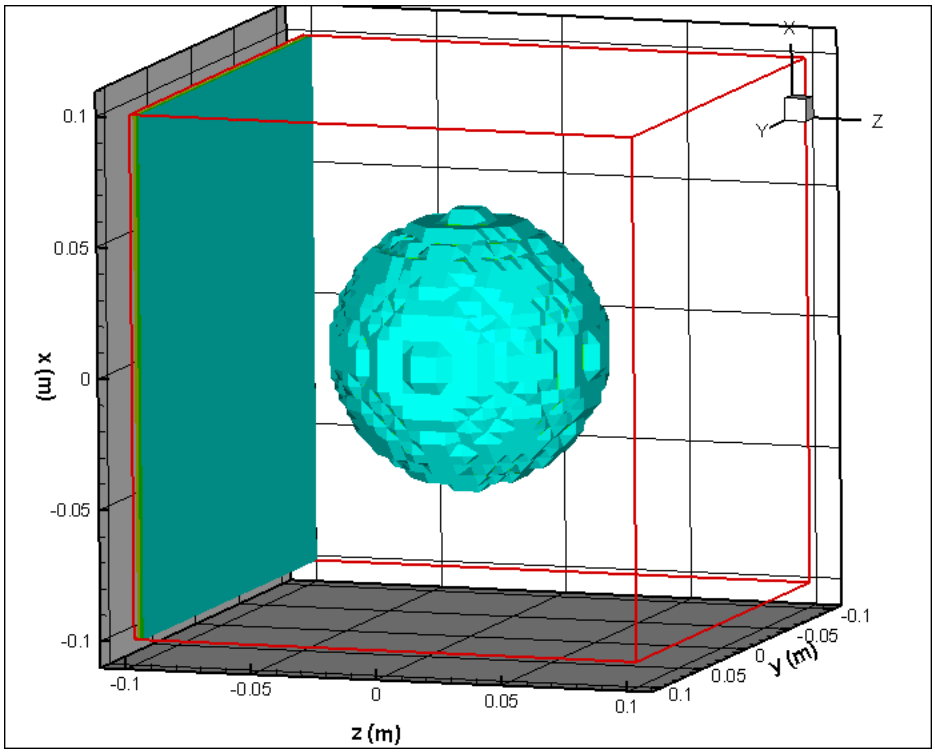

Figure 5.1: The sphere setup.

5.2 Ion Thruster Array

This section will present and discuss the effects of using an array of ion thruster on the plume expansion. These results were obtained from the parallel version of Coliseum.

5.2.1 Simulation Setup

The parallel code was run to simulate the plasma plume created by the exhaust of an array of three NSTAR ion thrusters. The geometry used in this simulation is shown in Figure 5.13. The three thrusters are mounted equidistant of each other on a cylidrincal frame. Since we are only modeling the beam ions in this simulation, the exit plate of the thruster is given a curvature similar to the one existing on the NSTAR thruster, about 15 degrees. The curvature is important in order to obtain the correct beam divergence. The source $ION\_THRUSTER$ was used with input parameters

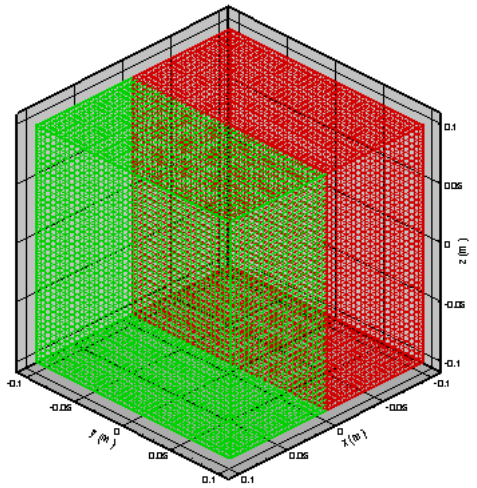

Figure 5.2: The domain decomposition for 2 processors.

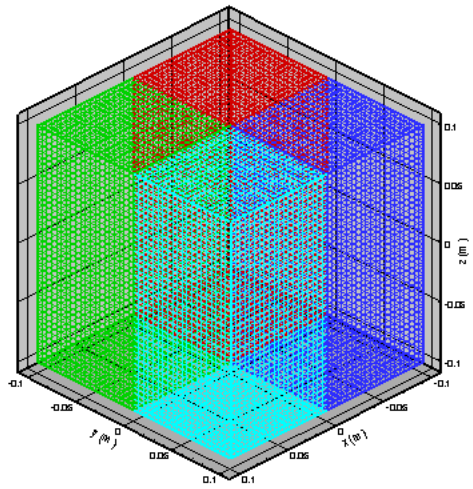

Figure 5.3: The domain decomposition for 4 processors.

obtained from a curve fitting of the NSTAR thruster current density profile [7]. The result is a 6^{th} degree polynomial and is shown in Figure 5.14.

$$j_z(r) = -222.69r^6 + 342.15r^4 - 182.22r^2 + 61.941 \tag{5.1}$$

The characteristics of the NSTAR thruster used in the simulation are summed up in Table 5.2. The propellant used by the NSTAR thruster is Xenon. It is important to note that the mass flow rate given in this table has been scaled down 10,000 times to match the simulation scaling of a hundred applied in this case. The simulation uses

Parameter	Value
$\dot{m}$	$3.3776 \cdot 10^{-10}$
T	35000K
v_i	$2.724 \cdot 10^4$ m/s

Table 5.2: NSTAR characteristic parameters.

an accurate model for the ion source, it is particularly important when simulating beam ions as nothing but the initial conditions have an effect on these fast moving ions. The input parameters of the simulation are presented in Table 5.3.

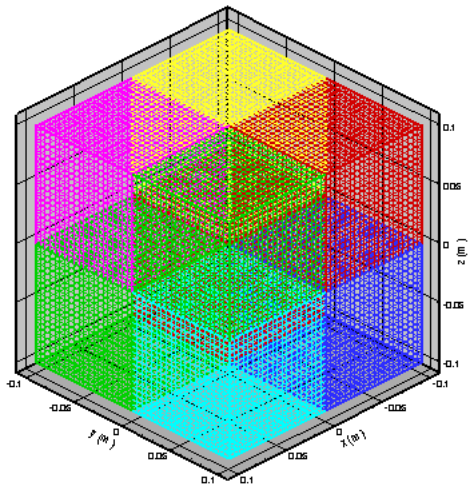

Figure 5.4: The domain decomposition for 8 processors.

Parameter	Value
dt	$3 \cdot 10^{-9} s$
T_{eRef}	$3.2eV$
ϕ_{Ref}	5V
n_{eRef}	$1.5 \cdot 10^{16}$
spwt	10^3
ϕ_w	0V

Table 5.3: Ion thruster array simulation input parameters.

5.2.2 Simulation Results

The simulation was run on three different supercomputers for different domain sizes and domain decompositions. The machines used were the JPL Cosmos supercomputer, the Anantham cluster from the Computer Science department of Virginia Tech and finally the System X supercomputer from the Terascale Facility at Virginia Tech. The runs are detailed in Table 5.4. The number of cells in each domain is chosen to respect the cell length of one Debye length, here $\lambda_D = 1.08579 \cdot 10^{-4}$ m. The goal of this simulation is to determine the effects of multiple thrusters on the plasma plume. The shape of an ion thruster plume is well known, however it has been difficult until now to physically test or even simulate a multiple thruster plume. Each ion thruster

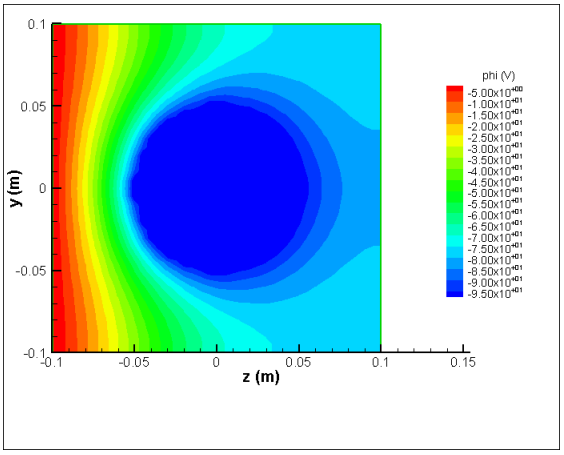

Figure 5.5: The electrostatic potential distribution on 1 processor.

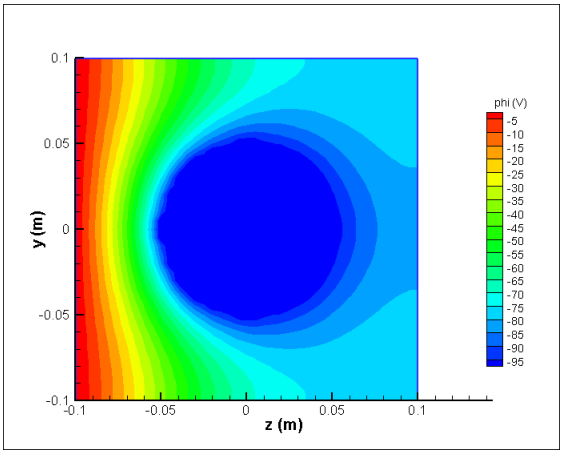

Figure 5.6: The electrostatic potential distribution on 2 processors.

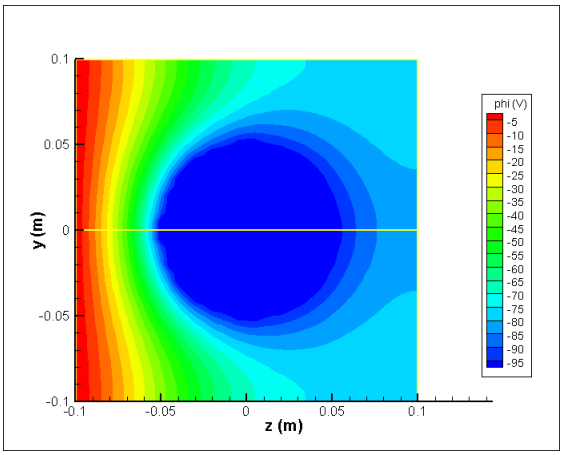

Figure 5.7: The electrostatic potential distribution on 4 processors.

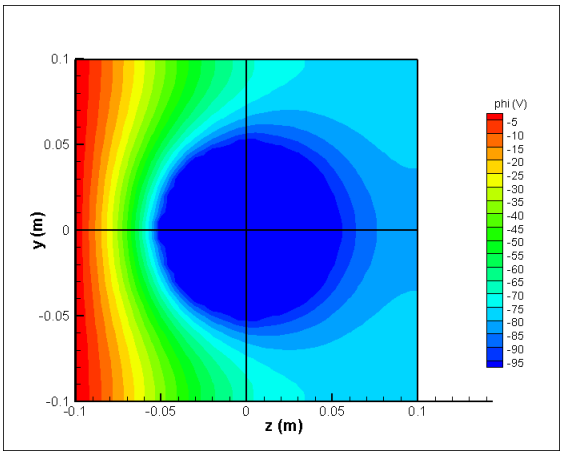

Figure 5.8: The electrostatic potential distribution on 8 processors.

exhaust in the array will produce an ion beam, but the close proximity of multiple beams has consequences on the shape of the plasma plume as the ions move further downstream. This change in shape is the result of peculiar potential and charge density distributions. The following subsection will present in details these distributions and their effects on the plasma plume.

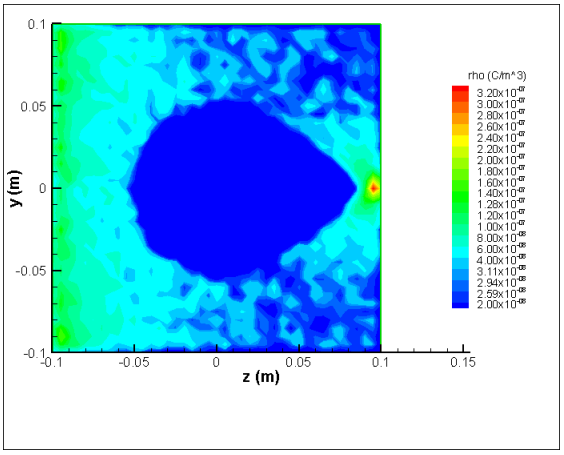

Figure 5.9: The charge density distribution on 1 processor.

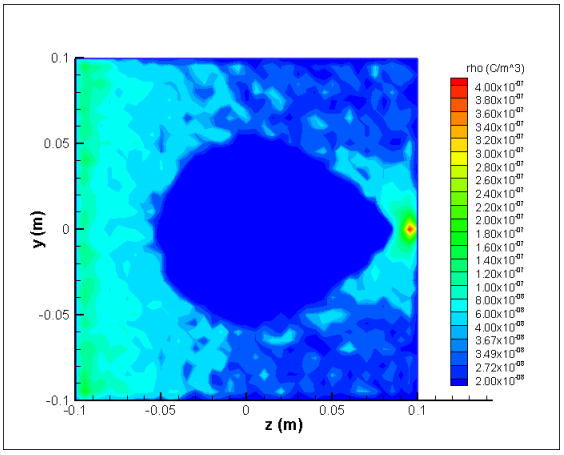

Figure 5.10: The charge density distribution on 2 processors.

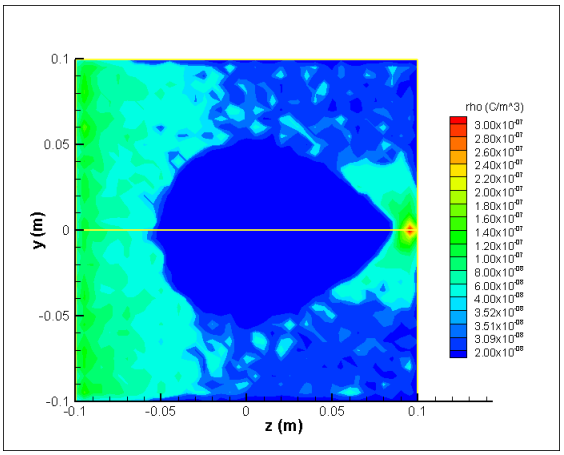

Figure 5.11: The charge density distribution on 4 processors.

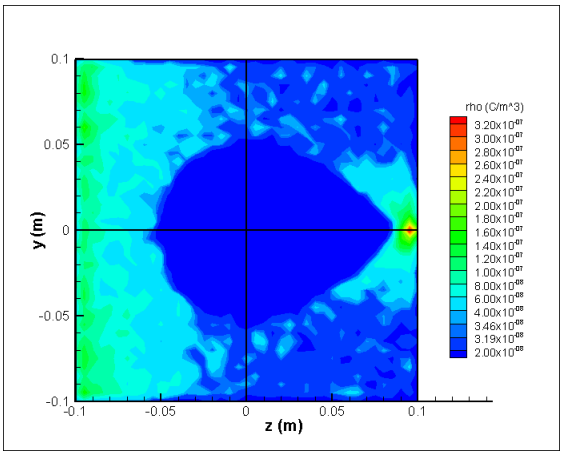

Figure 5.12: The charge density distribution on 8 processors.

Plume Mixing

The divergence of the ion beams at their exit of the ion thrusters forces the three beams to overlap. The divergence angle is large enough that the beams overlap close to the thrusters exit, hence the simulation domain can be kept small to focus on that region. The domain size was 120x120x120. The three thrusters only emit beam ions, they are the fast moving ions directly accelerated by the thruster grids. Their velocity is such that their path is not affected by any change in the electric field, at least not over a short domain length such as the one used in this case. Thus the beam ions

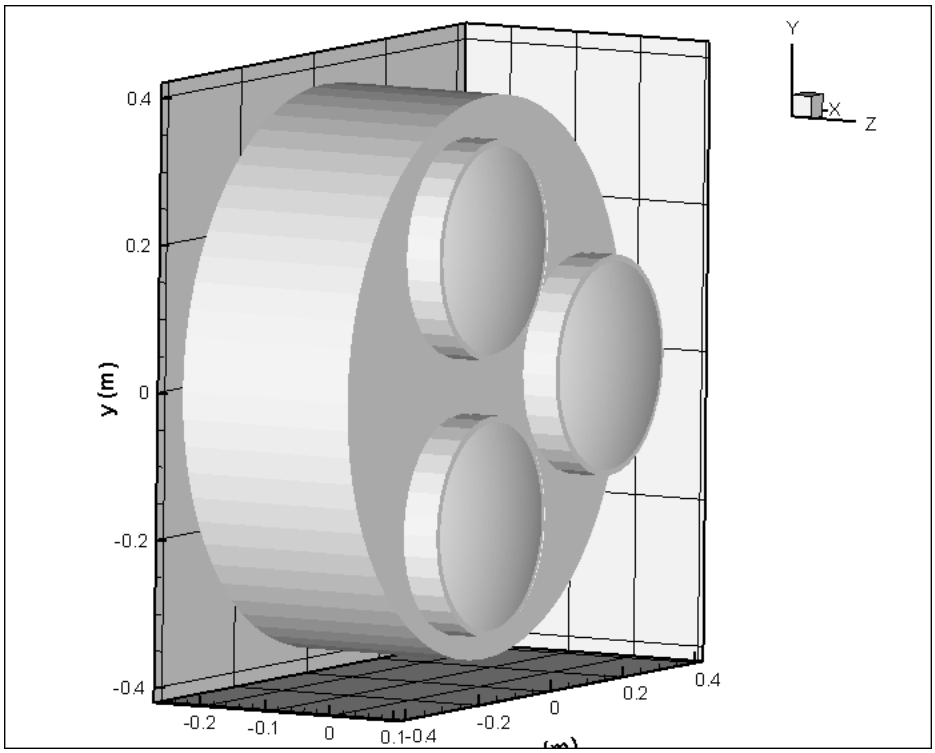

Figure 5.13: The ion thruster array geometry.

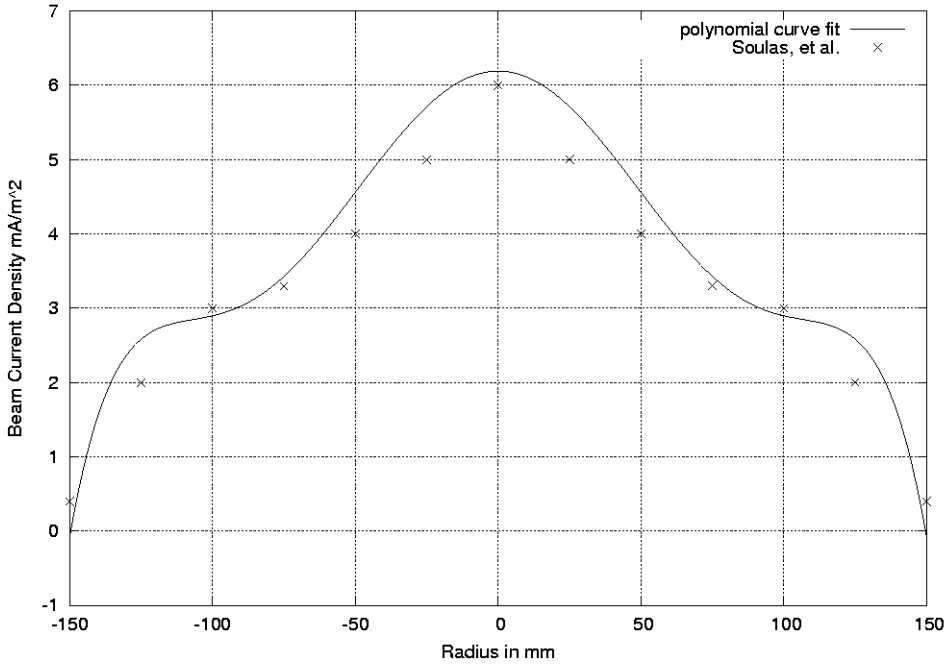

Figure 5.14: Profile of the NSTAR beam current.

	Anantham	Cosmos	SystemX
Domain size	120x120x120	120x120x120	160x160x600
Number of processors	16	8	128
Domain decomposition	2x2x4	4x2x1	8x8x2
Number of particles	1.75 million	1.75 million	26 millions
Maximum number of macroparticles per cell	19	19	36

Table 5.4: Details of the simulation runs.

travel straight, they exit the thruster following the local normal of the accelerating grid. The curvature of the grid spreads out the ions to form a diffused beam as shown in Figure 5.15. When two thrusters are operating close to each other, the plumes

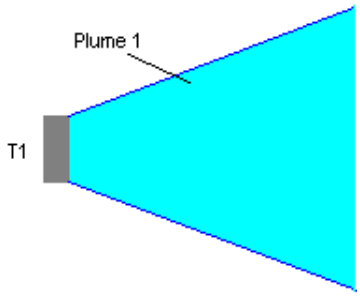

Figure 5.15: Schematic of plume created by one thruster.

from both thrusters will overlap and form a region where the number density of ions is higher than usual on the edge of the beam, see Figure 5.16. The density profile of the current density (Figure 5.14) at the exit of the thruster dictates the number density of ions in the beam. There are more ions in the middle of the beam than on the peripherical region. If one were to look at a beam expansion from the thruster's position directly downstream, one would see a denser nucleus at the middle of the beam that would fade out the farther away from the beam center line. However when two beams connect the merging region of the beams sees a net increase in number density. This is visible in Figure 5.27, the charge density increases in the *connection* regions of the beams. As the ions move further downstream the three beams finally

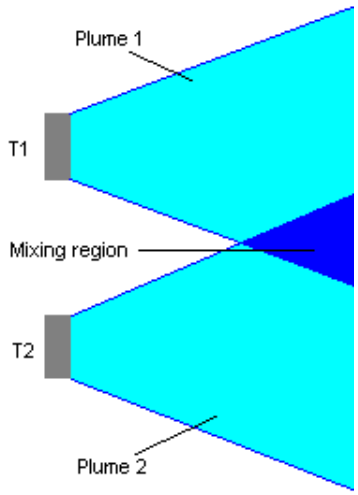

Figure 5.16: Schematic of plume created by two thrusters.

connect (Figure 5.17), so far the beams were only connected two by two. In the small

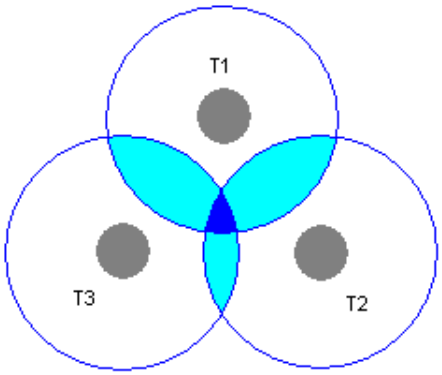

Figure 5.17: Schematic of plume created by three thrusters.

region that the three beams have in common the density is even higher, see Figure 5.28. In Figure 5.28 the density looks the same in the beam and in the mixing regions, but it is due to the fact that the farther away from the beam center line the smaller the density and the mixing regions increase this density to reach densities similar to the center of the beam. The evolution of the mixing regions through the beam is shown in Figures 5.26-5.29. Finally the central mixing region becomes the region with the highest density as the ions reach the end of the domain. This in return gives

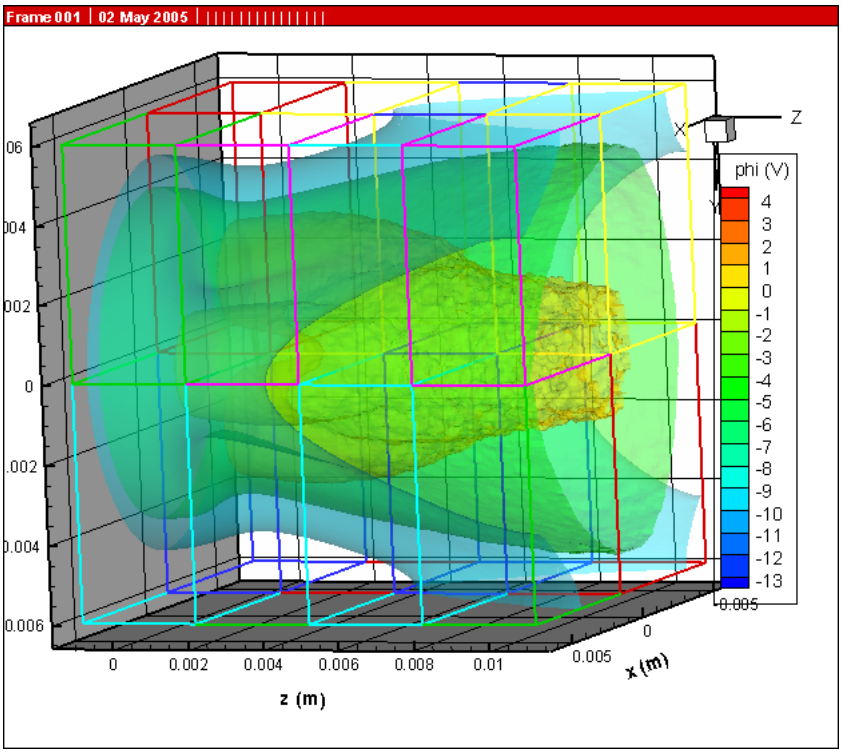

Figure 5.18: Potential distribution isosurfaces with domain decomposition.

the impression that the three beams focused into one, see Figure 5.19.

A region where the charge density is high would see its electrostatic potential increased, thus any mixing region would be made evident by an increase in potential, see Figure 5.20. The evolution of the mixing regions (Figures 5.22-5.25) is more evident when looking at the potential since the charge density data is noisy due to the fact that not enough particles were used in the simulation. The maximum number of macroparticles per cell was found to be 19 where a value of at least 50 is required to overcome the computational noise.

Particle Coupling

In the previous section a mixing region clearly appeared in the core of the beams. Mixing regions are usually the results of two or more beams colliding with each other. These mixing regions can produce particle coupling, which means that the particles

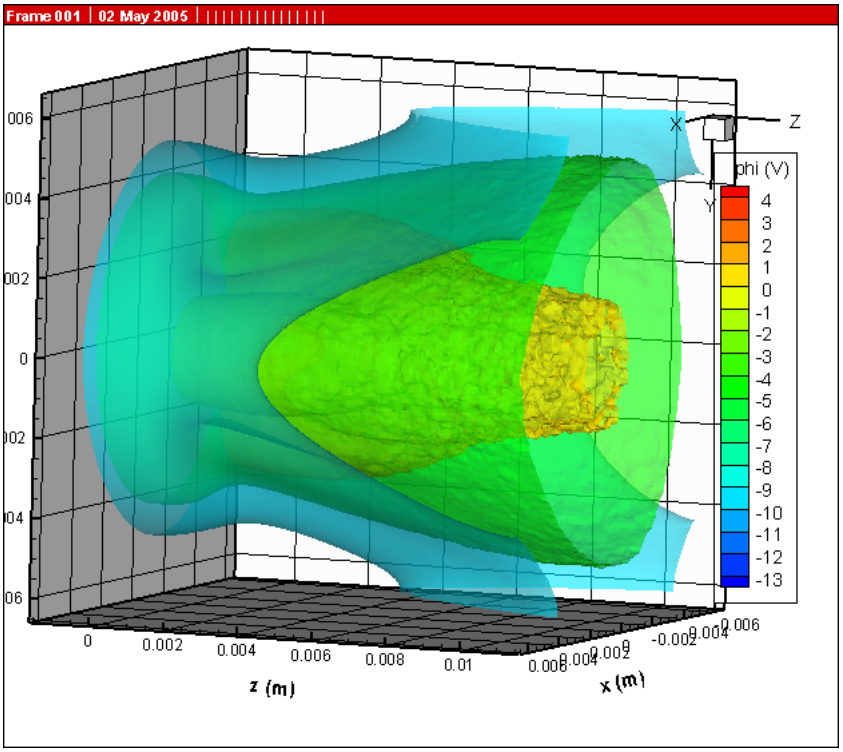

Figure 5.19: Potential distribution isosurfaces.

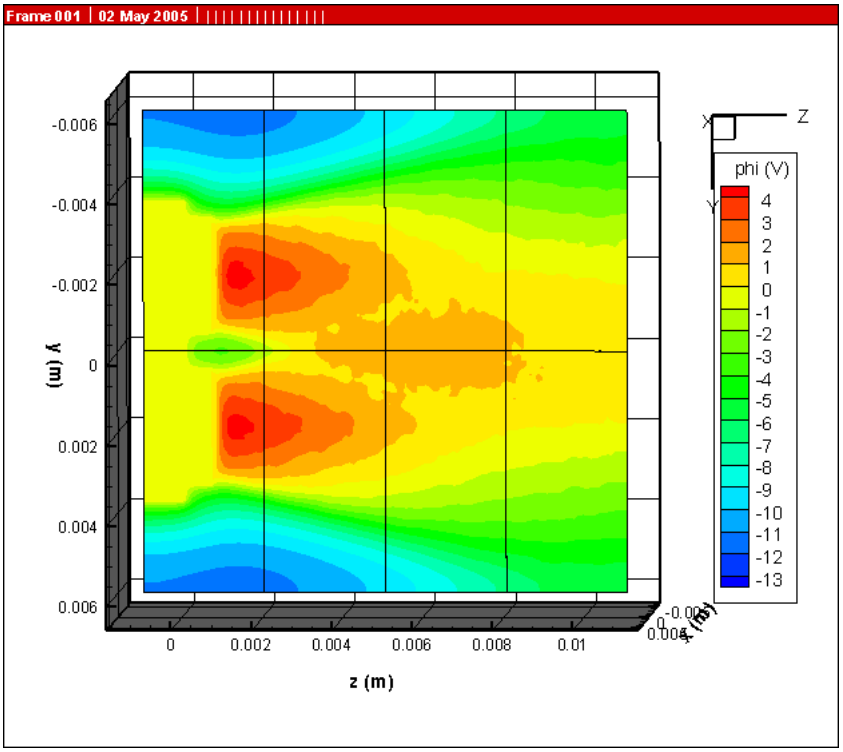

Figure 5.20: Cutting plane of the potential through two of the thrusters.

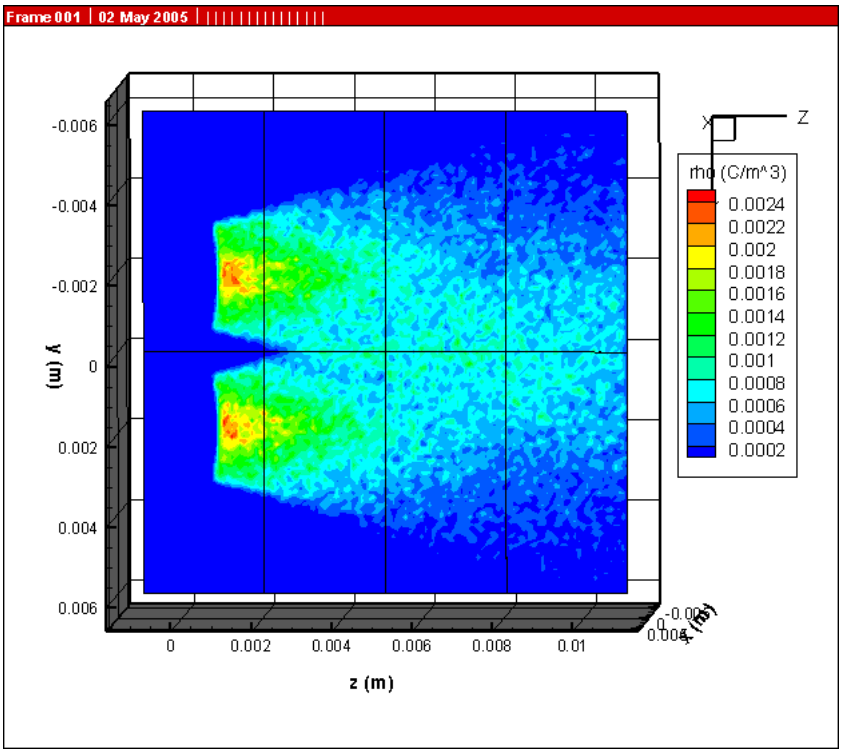

Figure 5.21: Cutting plane of the charge density through two of the thrusters.

are trapped in this region in a gyratory motion. In order to investigate wether or not this behavior occurs for beam ions in such regions we plot the tangential velocity along with the transversal direction x. The phase plot for the velocity v_x versus x is shown in Figure 5.30. The plot shows a typical velocity distribution from a thruster, the reason for the presence of the two distribution is that the plot shows the particles emitted from all three thrusters. However there are no evidence of any coupling involving the beam ions since there is no sign of a gyratory motion that would show as a vortex forming in the middle of the plot. From Figure 5.31 we can see that the beam ions keep a constant velocity all along the domain and are not influenced by any electromagnetic effects. This is due to the high velocity of the beam ions.

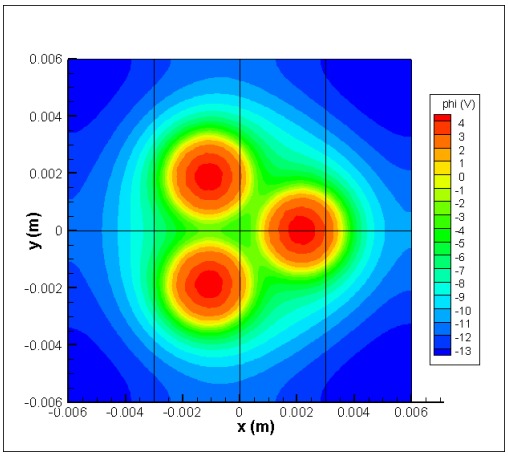

Figure 5.22: Cutting plane of the potential transversially through the beam at z=0.0012.

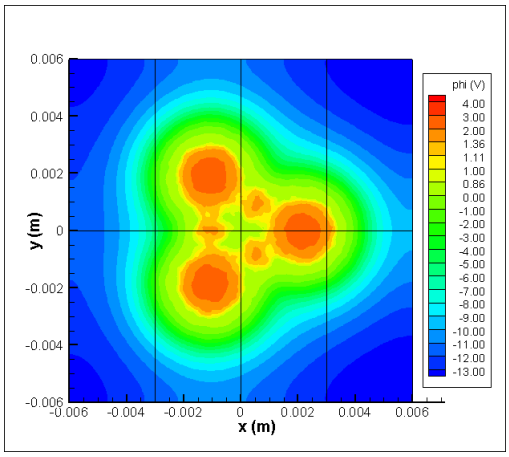

Figure 5.23: Cutting plane of the potential transversially through the beam at z=0.0036.

Plume Expansion

The previous sections only focused on the plume close to the thrusters. This section will show the plume far away from the thrusters. The domain size was thus extended in all directions but especially in the flow direction. The simulation ran was the same as in section 5.2.2 but with a 160x160x600 domain and 15 times as many particles. The maximum number of macroparticles in this case was 36. Figures 5.32 and 5.33 show the electrostatic potential isosurface of the plume. Far downstream the the three thrusters produce a single plume similar as the plume produced by a single thruster. The charge density drops by almost an order of magnitude by the end of the domain, see Figure 5.36.

5.3 Conclusions

This chapter presented the results obtained with the parallel version of Coliseum on several different supercomputers. The code was validated by comparing parallel results to sequential results of a simple test case. The code was then used to produce results on a larger scale. First the simulation ran an array of ion thruster to study the

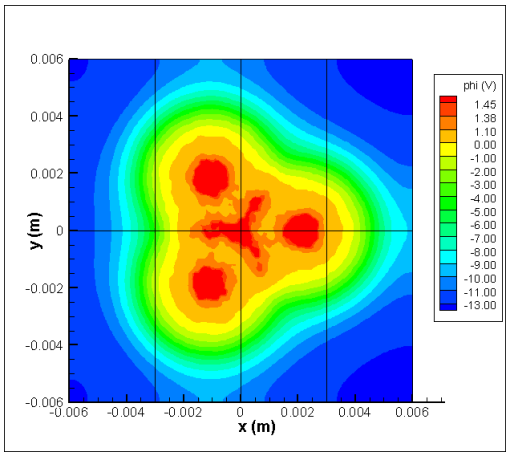

Figure 5.24: Cutting plane of the potential transversially through the beam at z=0.0045.

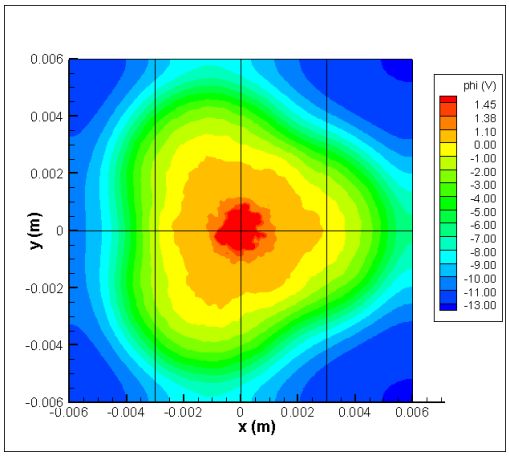

Figure 5.25: Cutting plane of the potential transversially through the beam at z=0.0066.

plume mixing and the effects of mixing on the beam ions. The mixing regions were found to be the results of the ion beams overlaping and that these mixing regions have no effects on the beam ions. If the simulation was running a Full PIC case the electrons would be greatly affected by the high potential regions.

A larger simulation was finally run to show the plume expansion far downstream. This simulation was by far larger than any other ran with Coliseum. It ran in only five hours on System X with 128 processors.

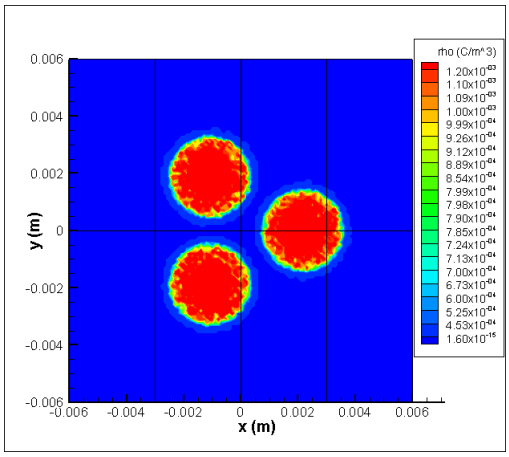

Figure 5.26: Cutting plane of the charge density transversially through the beam at z=0.0012.

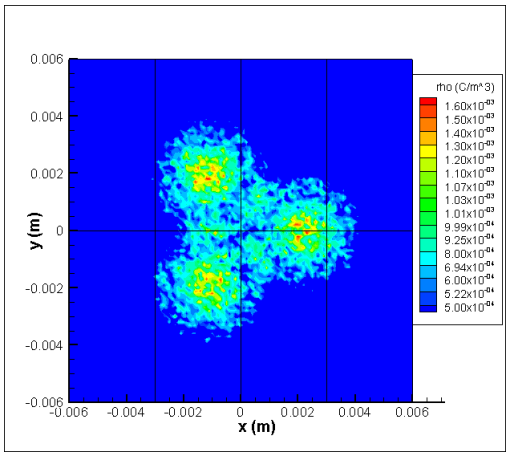

Figure 5.27: Cutting plane of the charge density transversially through the beam at z=0.0036.

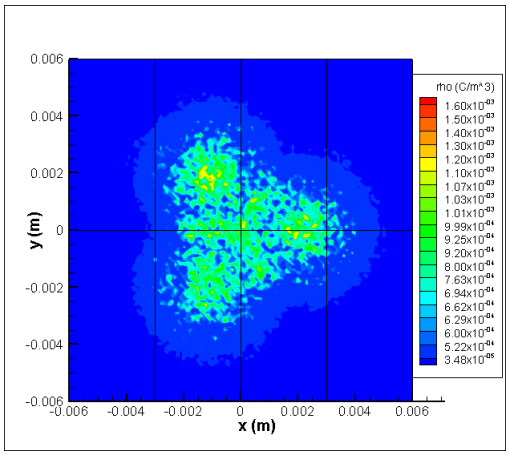

Figure 5.28: Cutting plane of the charge density transversially through the beam at z=0.0045.

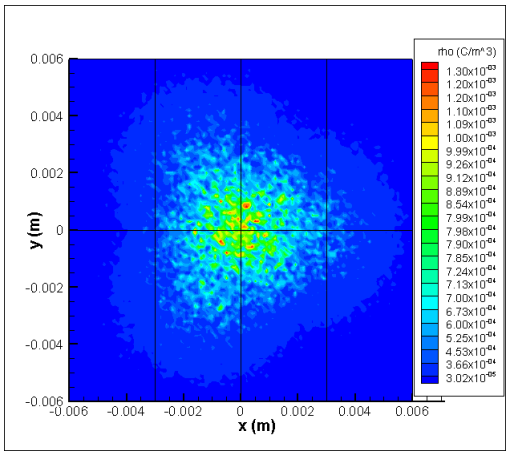

Figure 5.29: Cutting plane of the charge density transversially through the beam at z=0.0066.

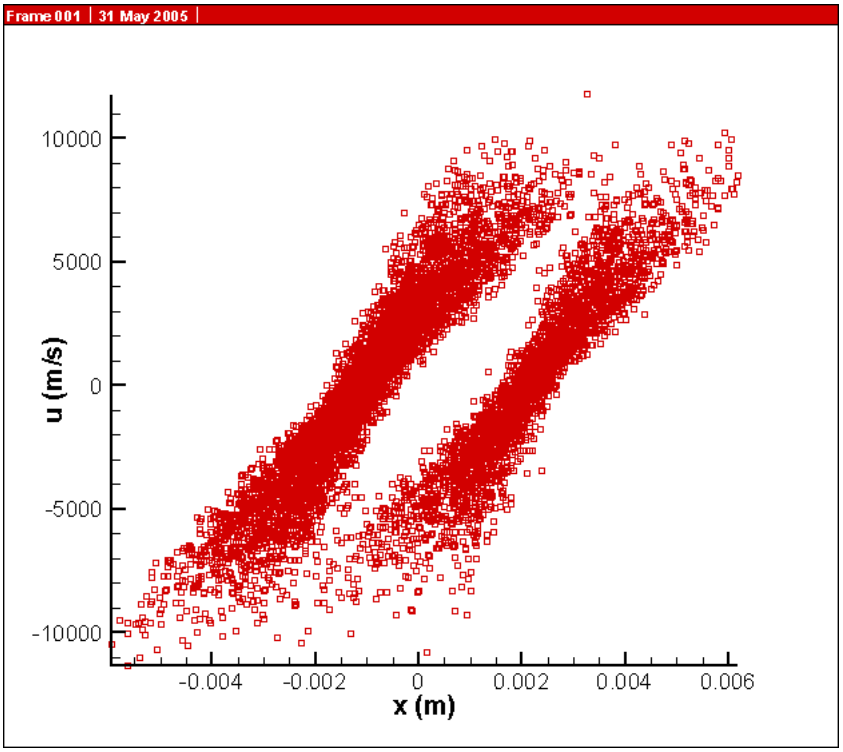

Figure 5.30: Phase plot of the velocity with respect to x.

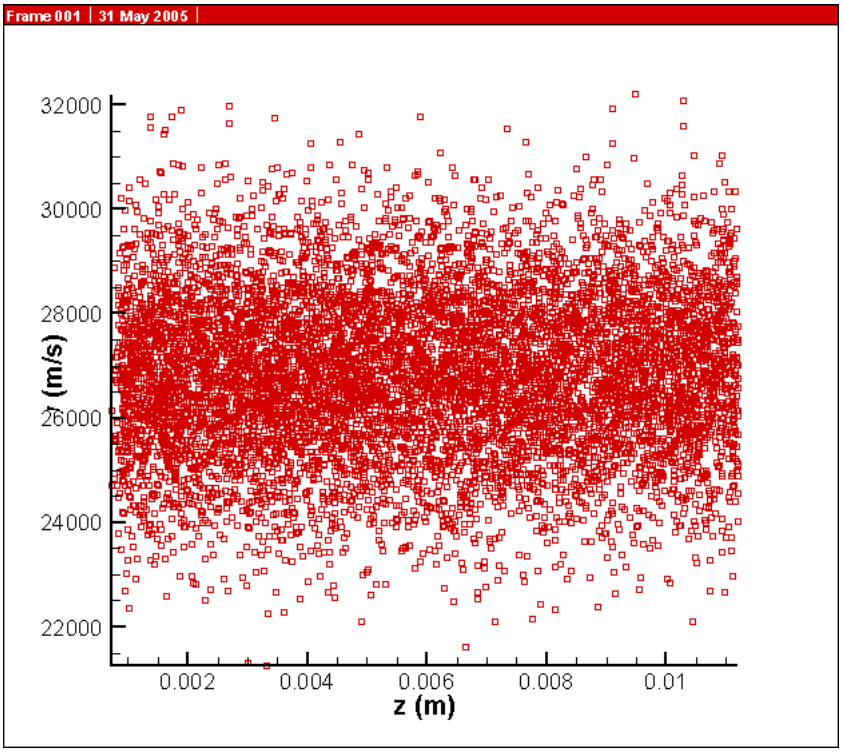

Figure 5.31: Phase plot of the velocity with respect to z.

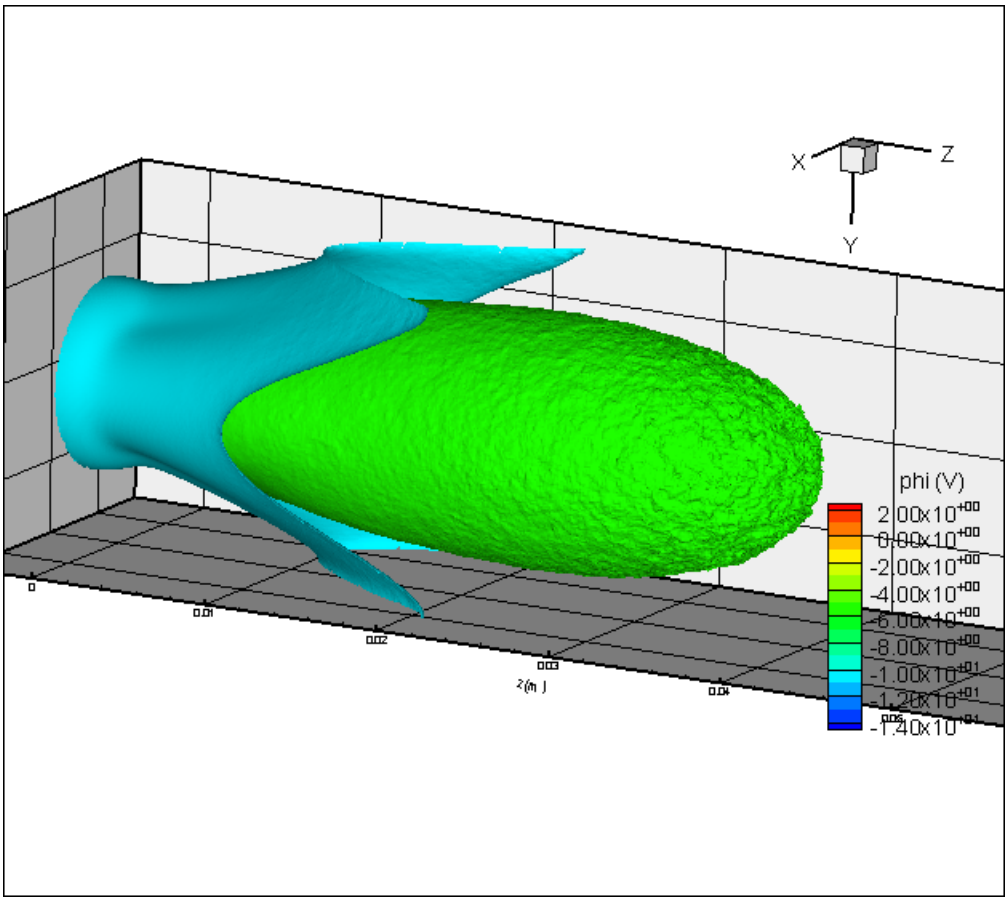

Figure 5.32: Electrostatic potential isosurfaces of the thruster array on a 160x160x600 grid.

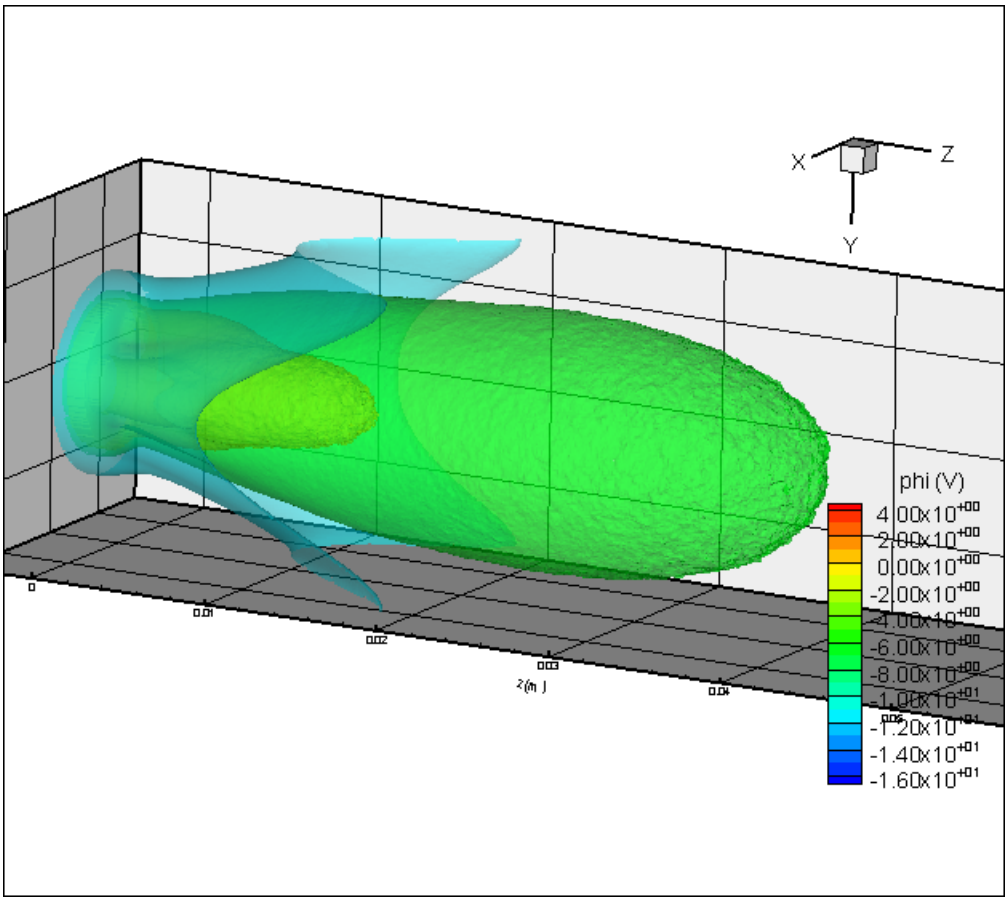

Figure 5.33: Electrostatic potential isosurfaces of the thruster array on a 160x160x600 grid.

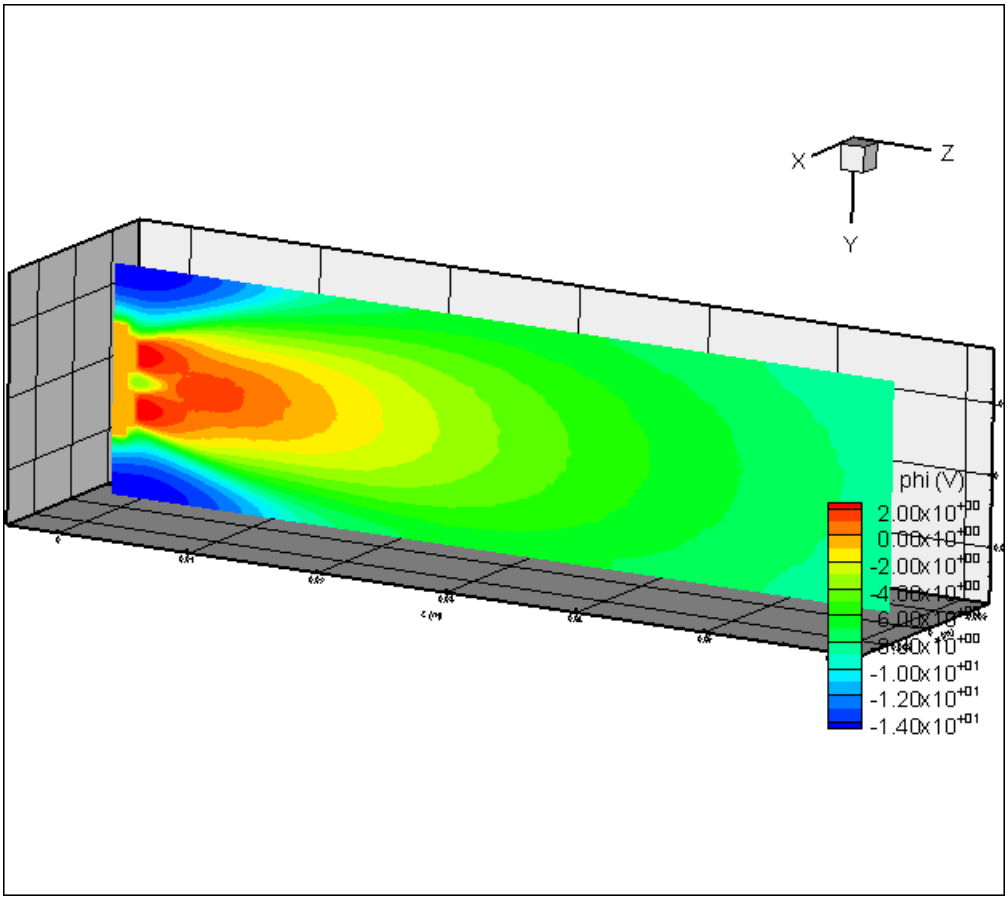

Figure 5.34: Electrostatic potential contours of the thruster array on a 160x160x600 grid.

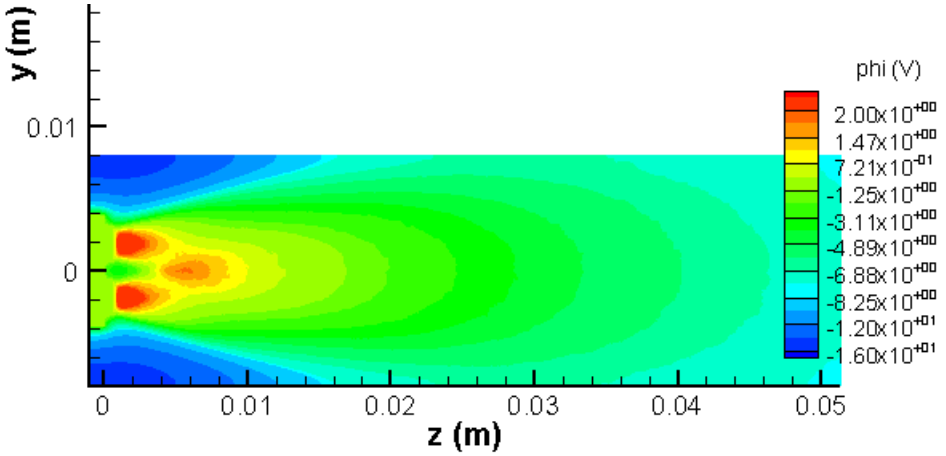

Figure 5.35: Electrostatic potential contours of the thruster array on a 160x160x600 grid.

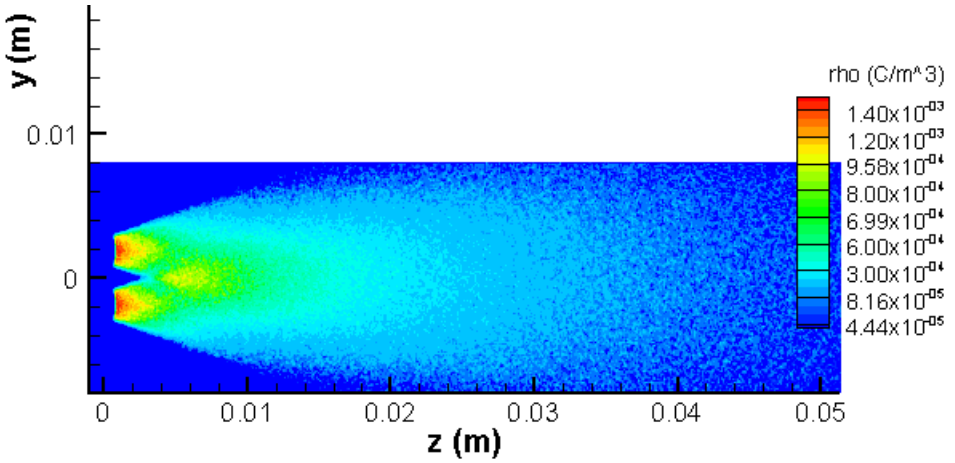

Figure 5.36: Charge density contours of the thruster array on a 160x160x600 grid.

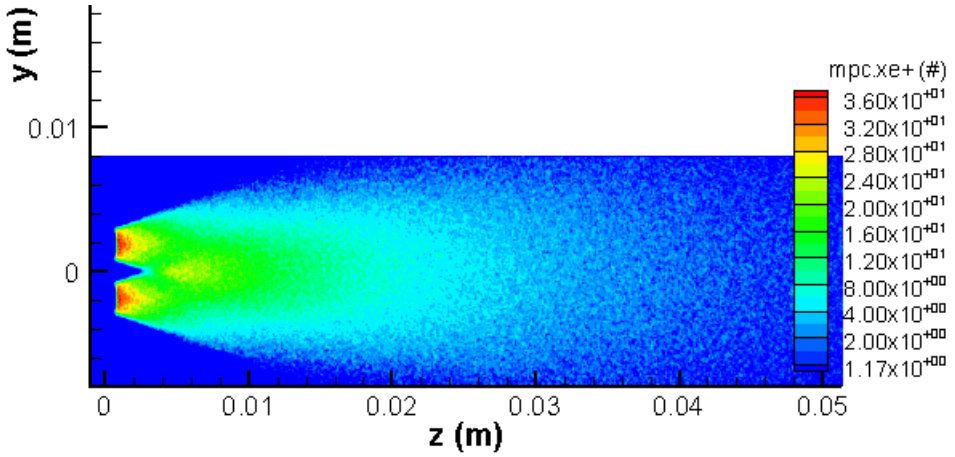

Figure 5.37: Number of macroparticles per cell contours of the thruster array on a 160x160x600 grid.

Chapter 6

Conclusions

6.1 Code Development Conclusions

A parallel Particle in Cell code was written in C using the MPI library. The code was designed to run on the world largest supercomputers. It was successfully tested on the JPL's Cosmos supercomputer ranked 37^{th} on the TOP500 list and Virginia Tech Terascal Facility System X, ranked 7^{th}. The code uses the Coliseum framework developed by the Air Force Research Laboratory and the DRACO module to Coliseum developed by the Computational Advanced Propulsion Laboratory at Virginia Tech for AFRL.

The parallelization of the PIC code was achieved by using a domain decomposition method in which all the processors work on a fraction of the regular domain. The code was also provided with a truly parallel field solver using a Gauss-Seidel method with SOR acceleration on a Red-Black checkerboard scheme. The checkerboard scheme is the algorithm used for a parallel implementation.

6.2 Simulation Run Conclusions

The code was validated by testing it with a simple case. The test case was a plasma flow around a charged sphere. The test consisted in testing the parallel code with multiple domain decomposition to make sure that the communications between the processor were working. The domain decompositions were made in each directions. The results obtained each matched the results from the single processor case. Timing results were obtained for the code running on NASA JPL's Cosmos supercomputer. The efficiency goes down as the number of processor increases due to the augmentation of inter processor communications. However the values of the efficiencies were satisfactory and allow for large speed ups. It was also found that the resulting CPU hours from the total computation time are different for each runs, indicating that some of the configurations and number of processors used can be more advantageous on a cost related matter. The price of the supercomputers are given in dollars per CPU hours we can find a compromise between the total simulation time and the CPU hours to get a simulation that finishes relatively fast while choosing the most cost efficient solution.

Finally the code is applied to run a simulation of an array of ion thrusters. The simulation goal was to investigate the effects of multiple beams on the plume expansion. It was found that the beams merge into one focused beam, creating regions of high electrostatic potential and charge density. These regions however did not influence the ions much as their velocities are too high. Furthermore the parallel code ran a large simulation (160x160x600 and 26 million particles). This simulation aimed at showing the capacity of the parallel version of Coliseum to run large simulations. This run showed the plume expansion of multiple ion thrusters in the far field region. The different ion beams merge to form a single plume similar to that of a single ion thruster.

In Conclusion the current status of the parallel code is satisfactory and capable of

running large commercial simulations on supercomputers.

6.3 Suggestions for Future Work

Future work includes the development of a better load balancing scheme to reach higher parallel efficiencies and a new field solver capable of better resolving geometries. Also better communication algorithms should be found, particularly for particle communications. The code can be used to investigate the effects of domain decompositions on parallel efficiencies as well as the cost/CPU Hours relations. Finally the code should be used to run a Full PIC case to investigate the electron behavior on multiple thruster arrays.

Bibliography

[1] C.K. Birdsall and A.B. Langdon. *Plasma Physics Via Computer Simulation*. Institute of Physics Publishing, Bristol, UK, 2000.

[2] T.J.M. Boyd and J.J. Sanderson. *The Physics of Plasma*. Cambridge University Press, Cambridge, UK, 2003.

[3] L. Brieda. Draco/volcar user's guide. Technical report, Virginia Tech Computational Advanced Propulsion Laboratory, Blacksburg, VA, 2004.

[4] L. Brieda. Development of the draco es-pic code and simulations of ion beam neutralization. Master's thesis, Virginia Polytechnic Institute and State University, 2005.

[5] L. Brieda, R. Kafafy, J. Pierru, and J. Wang. Development of the *draco* code for modeling electric propulsion plume interactions. In 40^{th} *AIAA/ASME/SAE/ASEE Joint Propulsion Conference & Exhibit*, AIAA 04-3633, Fort Lauderdale, FL, July 2004.

[6] J.M. Fife et al. The development of a flexible, usable plasma interaction modeling system. In 38^{th} *AIAA/ASME/SAE/ASEE Joint Propulsion Conference & Exhibit*, AIAA 02-4267, Indianapolis, IN, July 2002.

[7] J.E. Foster, G.C. Soulas, and M.J. Patterson. Plume and discharge plasma measurements of an nstar-type ion thruster. In 36^{th} *AIAA/ASME/SAE/ASEE*

Joint Propulsion Conference & Exhibit, AIAA 00-36940, Huntsville, AL, July 2000.

[8] D. Hastings and H. Garrett. *Spacecraft Environment Interactions*. Cambridge University Press, Cambridge, UK, 1996.

[9] R. Kafafy. *Immersed-Finite-Element Particle-In-Cell Simulation of Ion Thrusters*. PhD thesis, Virginia Polytechnic Institute and State University, 2005.

[10] Peter S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann, San Francisco, CA, 1997.

[11] J.S. Sovey, V.K. Rawlin, and M.J. Patterson. Ion propulsion development projects in us: Space electric rocket test 1 to deep space 1. *Journal of Propulsion and Power*, 17 No. 3:517–526, May-June 2001.

[12] J.C. Tannehill, D.A. Anderson, and R.H. Pletcher. *Computational Fluid Mechanics and Heat Transfer*. Taylor & Francis, Philadelphia, PA, 1997.

[13] University of Minnesota Supercomputing Institute for Digital Simulation and Advanced Computation. *More point to point communication with MPI*. http://www.msi.umn.edu/tutorial/scicomp/general/MPI/content2.html.

[14] J. Wang. *Electrodynamic Interactions Between Charged Space Systems and the Ionospheric Plasma Environment*. PhD thesis, Massachusetts Institute of Technology, 1991.

[15] J. Wang, P. Liewer, and V. Decyk. 3d electrodynamic plasma particle simulations on a mimd parallel computer. *Comp. Phys. Comm.*, 87:35, 1995.