IMPLEMENTATION AND PERFORMANCE ANALYSIS OF A HIGH–ORDER CEM ALGORITHM IN PARALLEL AND DISTRIBUTED COMPUTING ENVIRONMENTS

A Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Olle Larsson

May, 1998

IMPLEMENTATION AND PERFORMANCE ANALYSIS OF A HIGH–ORDER CEM ALGORITHM IN PARALLEL AND DISTRIBUTED COMPUTING ENVIRONMENTS

Olle Larsson

APPROVED:

Dr. Lennart Johnsson
Department of Computer Science, UH

Dr. Olin Johnson
Department of Computer Science, UH

Dr. Martin Herbordt
Department of Electrical Engineering, UH

Dean, College of Natural Sciences and Mathematics

Acknowledgements

I would like to thank all people helping me out with this project. My advisor Dr. Lennart Johnsson, for ideas, the right questions, and support; Dr. Nikos Pitsianis for nice tips in the early stages of the implementation; Keith Crabb, UH, and Lars Malinowsky, PDC, for helping out to find answers to technical questions related to the IBM architecture; William Eaton, BCVL, d.o. for the SGI Origin 2000; my thesis committee, for correcting this document; and finally Maria, for all support despite the hours you have had to spend alone.

Support from DoD AFOSR Grant F49620-96-1-0289 and computing time at PDC, ANL and BCVL is hereby also acknowledged.

IMPLEMENTATION AND PERFORMANCE ANALYSIS OF A HIGH–ORDER CEM ALGORITHM IN PARALLEL AND DISTRIBUTED COMPUTING ENVIRONMENTS

An Abstract of a Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Olle Larsson

May, 1998

Abstract

A parallel implementation of a high–order algorithm for computing the scattering of an electromagnetic wave from geometrically complex objects is presented. Optimizations performed both at the algorithmic level and in the parallel implementation are discussed in detail. The performance analysis includes estimates of the best possible performance with respect to arithmetic capabilities of the processor architectures and operations in the code, as well as the peak performance with respect to memory bandwidth. We show that the SGI Origin2000 architecture is not suited for these types of algorithms.

On IBM SP platforms, a Fortran 90 with MPI implementation achieves a sequential efficiency of about 70% of estimated peak. Communication accounts for less than 10% of the execution time. The performance is about 40% better than the previous best implementation of the same algorithm on SP systems.

The structure of our implementation is also described in some detail. The code is highly modular and includes a flexible domain decomposition routine that as a default, distributes the domain evenly across all processors in a way that minimizes the necessary communication.

We also describe a distributed latency–tolerant IBM SP implementation, using the vBNS backbone and the Globus toolkit.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter we will introduce notation and expressions that will be used throughout this thesis. The characteristics of our problem and basic concepts of parallel computing are explained. Finally, the architectural characteristics of two common parallel platforms, the IBM SP and the SGI Origin2000, are investigated.

1.1 Abbreviations and units

When referring to data size units, b stands for *bit* and B for *byte* throughout this thesis. Power prefixes such as k (kilo, 10^3), M (mega, 10^6) and G (giga, 10^9) are, when written in front of a data size unit (b or B), denoting a power 2 (2^{10}, 2^{20}, 2^{30} for k, M, G, respectively).

The terms *processor* and *CPU* will be used interchangeably.

A *flop* is defined to mean a floating–point operation (multiplication, addition or subtraction). With *flops* we denote the plural of flop, (*not* processing rate, 'flop per second'). Processing rate is denoted *flops/sec* or *flops/cyc*, where *cyc* is short for

machine cycle.

Division and square root operations are often implemented in hardware as iterative methods, thus reducing the computation to a sequence of *mult* and *add* operations. They should therefore account for more than one flop. However, in the work conducted in this thesis, division and square root operations are only performed during initialization and are therefore not considered in any flop rate discussion.

1.2 Our problem

Our problem is to compute the scattering from an object of arbitrary shape, given a known incident electromagnetic wave (see Figure 1.1). The object size might be of much larger size than the wave length, which imposes a large problem. For instance, today's radar antennas operate at frequencies around 2 GHz. That corresponds to a wavelength of about 15 cm. To get an acceptable accuracy of phase shifts and other perturbations in the scattered wave, around 20 grid points per wave length is generally needed, or a grid point distance of less than 10 mm. At the same time, the object which is studied, usually falls in the range of tenths of meters. Considering the 3–D dimensionality of the problem, we see that the problem size gets very large, with billions of grid points.

To accommodate such a large problem, we first reduce the problem size by applying a high–order algorithm. The algorithm of our choice typically needs around 6–8 grid points per wave length, thereby reducing the problem size by a factor 15–40. This will still be a very large problem for one computer to solve, which calls for the need of *parallel computing*, described in the next section.

The high–order algorithm of our choice is discussed in Chapter 2. It is further

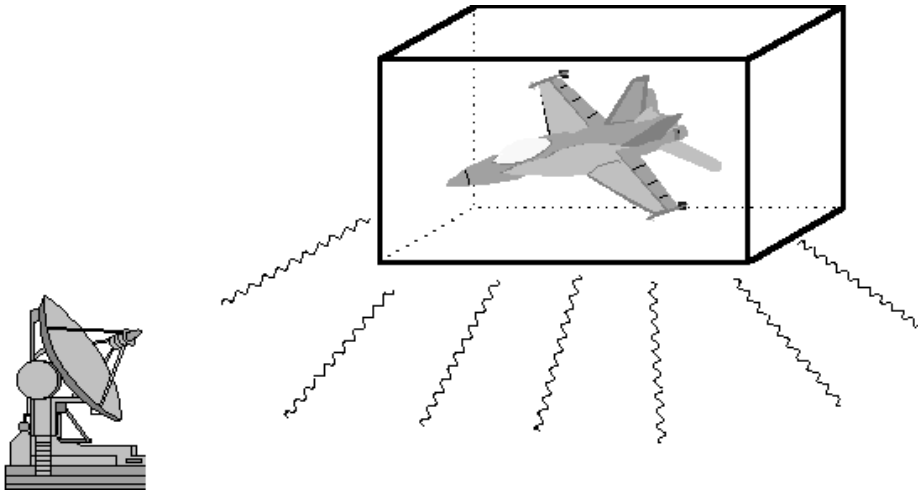

Figure 1.1: A simplified view of our problem.

enhanced in Chapter 3 and parallelized in Chapter 4.

1.3 Parallel machines

A *parallel machine* is a collection of often autonomous *processors* interconnected in some form (by a bus, switch, or network). By *partitioning* a problem, preferably in equal parts with respect to workload, with one partition per processor, a solution to the problem can be computed much faster for most applications.

Three important aspects of a parallel algorithm are:

- *Scalability* — when the number of processors increase, it should be possible to increase the size of the problem proportionally.
- *Speed–up* — a ratio denoting how much faster the parallel algorithm performs on a problem of size N when run on p processors. There are two definitions; S_p,

just scaling the computing power by a factor p, and S_N, where the problem size is also scaled so that the subproblem that each processor handles is constant : Assuming the execution time is T_p on p processors,

$$S_p = \frac{T_1}{T_p} \quad \text{or} \quad S_N = \frac{p \cdot T_1}{T_p}$$

We refer to S_p as *non–scaled speed–up*, and to S_N as *scaled speed–up*.

- *Efficiency* — a number between 0 and 1, denoting how well the algorithm performs as a fraction of the *ideal–case* performance.

  The word *efficiency* is often used in several different ways. We distinguish between *parallel efficiency*, the speed–up over the number of processors, and *hardware efficiency*, the fraction of the available resources actually utilized.

1.4 Parallel architectures

Two different architectures are investigated in this thesis: the IBM SP, and the SGI Origin2000. The systems are described in more detail below. A comparison of major characteristics (used in a detailed performance analysis) can be found in Table 1.1.

1.4.1 IBM SP

The IBM SP is constructed by connecting one or several racks of standard RS/6000 machines with a specially designed high–speed backplane switch. The switch has a peak bandwidth of 110 MB/sec/channel and a hardware latency of 35 μsec. Each processor, or *node*, runs its own copy of IBM's UNIX clone, AIX. A diagram of the architecture can be found in Figure 1.2.

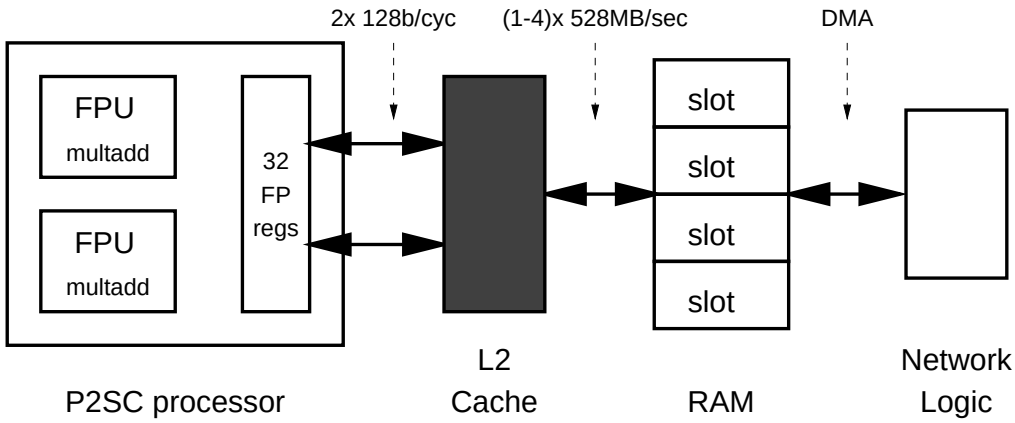

Figure 1.2: Basic architecture of the IBM SP processor.

The CPUs are Power2 Super Chips (P2SC). There are *thin* and *wide* processors, the naming of which originally reflected the differences in the memory bus bandwidth; today, it only reflects the extra I/O bus that the wide processors have. The wide nodes operate at 135 MHz, while there are two flavors of thin processors; one version is operating at 120 MHz, while the newer 160 MHz version (also called T4) also has improved cache hardware. We will call the different processor models *thin*, *wide* and *T4* for short.

The Power2 architecture is well described in [1] and [2]. The processors are equipped with 32 floating–point registers and two floating–point units (FPU), each capable of computing a fused *mult–add* operation ($x \leftarrow a \times b + c$) in one clock cycle. Each FPU can be fed a new instruction once every cycle. Thus, the peak flop rate per processor is four times the clock rate.

The P2SC processors are equipped with a 256–bit wide memory bus, divided in two 128–bit channels. The channels can simultaneously perform a *quad–word*

memory instruction, that is a *load* or a *store* of two 64–bit floating–point numbers in one instruction. As the memory bus operates at clock speed, the peak number of load/store operations is four double–precision numbers per cycle.

The memory bus bandwidth between real memory and secondary cache is dependent on the memory population; there can be up to four 64–bit memory cards in a population, with a theoretical bandwidth of 528 MB/sec each. Twice as many cards yields a twofold increase in bandwidth.

Three different SP systems were used in the work for this thesis. At the time of this writing, the SP at the Texas Center for Computational and Information Sciences (TCCIS), University of Houston, Houston, TX, carried 56 thin 120 MHz processors (half memory population), and 8 wide (full memory population). The SP at Parallelldatorcentrum (PDC), Stockholm, Sweden, carried 128 T4 processors in the parallel pool, all with full memory population.

1.4.2 SGI Origin 2000

The Silicon Graphics Origin2000 (O2K for short) is a *scalable symmetric multiprocessor with distributed shared memory*. A node is made up of two processors (195MHz MIPS R10000) sharing the same piece of RAM, which is accessed through a hub. The hub also connects to I/O ports and a special CrayLink port, which in turn is connected to other hubs via a crossbar switch, for fast remote memory access (see Figure 1.3).

The R10000 processor is well described in [3]. It has two *stripped* FPUs, one multiplier and one adder. The two units can operate independently, or be pipelined. Pipelining the two units for a *mult–add* operation will increase the performance as the number of instruction decodings (which takes one cycle) are reduced.

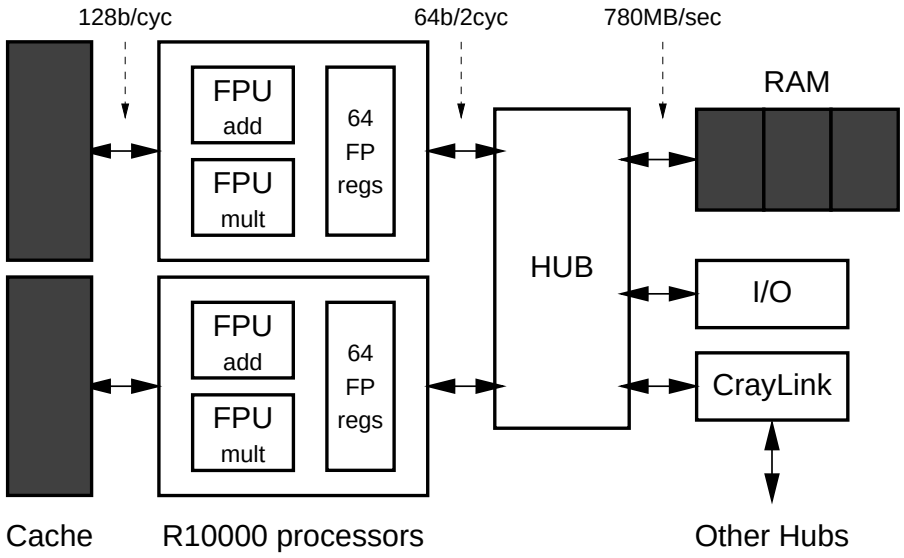

Figure 1.3: Basic architecture of the SGI Origin2000.

Both the multiplier and the adder have a latency of two machine cycles. (The *mult–add* instruction therefore has a latency of four cycles.) Both units can be fed a new instruction once every cycle. Thus, the peak floprate is 2 flops/cyc.

There are 32 logical floating–point registers in the R10000 processor, plus another 32 temporary registers used for register renaming, to store intermediate results and to minimize the stalls due to instruction dependencies. The temporary registers are handled by hardware and are not accessible.

Each processor has a large secondary cache, to which it connects through a 128–bit memory bus, running at clock speed. The bus connecting to the hub is 64–bit wide and runs at half the clock speed. The throughput of the hub, as well as the transfer rate from memory to the hub, is 780 MB/sec. This is consistent with the 64–bit width of the system bus operating at 97.5 MHz for one R10000 processor.

	IBM SP	**SGI O2K**
CPU clock speed (MHz)	120,135,160	195
Floating point units	2 multadd	1 mult, 1 add
FPU latency (cycles)	1	4
Peak flops / clock cycle	4	2
CPU $\longleftrightarrow$ Cache bandwidth (bits/cycle)	256	128
Cache $\longleftrightarrow$ Memory bandwidth (MB/s)	2112	780 ($\times$0.5)
Cache size	512 kB	4MB

Table 1.1: Hardware comparison *per processor*, IBM SP and SGI Origin 2000.

However, there are two CPU's using the same hub and sharing the bandwidth. A peak bandwidth for large memory transactions when using both processors in a pair will therefore drop to 390 MB/sec per processor.

In the work for this thesis, the 24–node SGI O2K system at the Baylor College Virtual Laboratory (BCVL), Baylor College of Medicine, Houston, TX, was used.

1.5 Programming paradigms

When constructing a parallel program, various levels of abstraction can be used. *Fine–grained parallel programming* is close to the underlying hardware, where every packet sent and received by the processors is explicitly managed in the code. In *course–grained parallel programming*, the programmer codes in a traditional programming language and does not have to deal with whether data is local or remote in the memory associated with another processor — the compiler and the run–time system determine where data are located, and generate required communication actions and synchronization.

When using a fine–grained model, one approach is to use asynchronous communication packages like *Active Messages (AM)* [4][5] or *Fast Messages (FM)* [6]. These two low–overhead packages are tailored for small messages — both packages essentially contain a *Get* and a *Put* command. An incoming message will interrupt the receiving processor, which will take appropriate measures (send back the data that was requested by a *Get*, or store the data that was received with a *Put*).

A disadvantage of these packages is that interrupts and context switches are quite expensive to perform in many architectures. Also, when trying to minimize the overhead, *threads* (processes sharing the same memory space) are used to reduce the cost of copying messages between buffers. Not all computer architectures and operating systems provide support for light–weight threads, however.

A second approach is to use synchronized communication methods like *MPI* [7] or *PVM* [8], where a *send* command must be paired with a *receive* command on the other processor in order for a communication to succeed. The major drawback is that this generalized paradigm imposes more overhead, as less is assumed of the underlying architecture. The major benefit is that the program code becomes more easily portable, as MPI and PVM are standardized and wide–spread. Proprietary efficient implementations of MPI and to some extent PVM exist as well.

Course–grained programming means programming in an ordinary sequential fashion (preferably using *Fortran 77* or *Fortran 90*) with a small set of extra primitives. For instance, a *DO* loop can also be expressed as a *FORALL* loop, to emphasize the independence and parallelism of the loop execution. The programmer may also guide the compiler how to partition arrays etc. by giving guidelines through specially formatted comments in the code. It is the most intuitive form of parallel programming.

The most widely used course–grained programming languages today are *High Performance Fortran* (HPF) [9], and *SGI Power Fortran*. Both are enhancements and/or variations of older, similar programming languages like CM Fortran and Cray Fortran.

Unfortunately, much work still remains in constructing good course–grained compilers. It is today still very hard to get a high efficiency out of a HPF code, which is mostly due to the challenges in writing a good run–time system that includes features such as prefetching techniques for acquisition of remote data so that it arrives before it's needed in the execution. More intelligence in the compiler is also needed in order to better foresee remote accesses in the code, and schedule the sequence of operations appropriately.

Chapter 2

Algorithms

In this chapter we will make a small survey of existing numerical methods for calculating electromagnetic fields in complex geometries. We end with a more elaborate description of the algorithm of our choice for the parallel implementation.

2.1 Governing equations and terminology

As described in Section 1.2, our problem is to compute the scattering from an object of arbitrary shape, given a known incident electromagnetic wave. The physical relations are given by the four electromagnetic equations due to James Clerk Maxwell (1831–1879). The equations can be formulated in various forms, and each form suits a special numerical technique. The diversity in techniques and implementations is quite large since this area is of significant research interest, with hundreds of new papers published every year on the subject. The main techniques are explained and discussed with their pros and cons in [10]. A few of the most widely used techniques are described below.

The scattering in different directions is often characterized by the *Radar Cross Section (RCS)*, a measure of the far field electromagnetic energy from the scattering object at a spherical angle θ, ϕ relative to the incident wave direction;

$$rcs_E(\theta,\phi) = \lim_{r\to\infty} 4\pi r^2 \left|\frac{E_{sc}}{E_{in}}\right|^2 \qquad \text{or} \qquad rcs_H(\theta,\phi) = \lim_{r\to\infty} 4\pi r^2 \left|\frac{H_{sc}}{H_{in}}\right|^2 \tag{2.1}$$

The scattering intensities next to the object can be used to compute $E_{sc}(\theta,\phi)$ and $H_{sc}(\theta,\phi)$ by a near–field to far–field transformation. Thus, it suffices to determine the solution locally next to the object.

2.2 Finite Element Method

When using the *Finite Element Method (FEM)*, the solution y to a problem is approximated on a set of *base functions* $u_1 \ldots u_n$. The base functions are locally defined functions with some overlapping in the functional space S that they span [11]. Moreover, the base functions are often chosen to be simple (low–order polynomials or trigonometric functions), so that integrals and derivatives of the composite form $(u_i \cdot u_j)$ can be computed easily.

From functional analysis, a set of weights $\omega_1 \ldots \omega_n$ can be found such that $\hat{y} = \sum \omega_i u_i$ will be the *best–fit* approximation in S to the solution y. The w_i values are obtained by solving a system of linear equations, $A\omega = b$, where the elements a_{ij} in the matrix A are functions of the contributions of the base functions u_i and u_j to each component of b_i (as we approximate y with $\hat{y}$). The vector elements in b are known, being functions of boundary conditions and non–trivial parts of the differential equation to solve. As the base functions are defined locally over only a few elements, the matrix in A is sparse — but large (quadratic product of the number of elements and the number of base functions). Several iterative methods

exist to solve the system of equations, but such methods are not considered in this brief overview.

Applying FEM to electromagnetic problems has its major disadvantage in that it works best for a closed system with known boundaries. Several techniques exist to overcome this problem, such as *absorbing boundaries*, which work well for 2–D problems but not (yet) so well in 3–D [10].

The main advantage of FEM is that the elements are defined individually, and resolution can shift widely depending on the complexity of the geometry (implying the complexity of the solution).

2.3 Moment method

The Moment Method (MM) resembles FEM in that it reduces a complex integral equation formulation of a problem into solving a sparse system of linear equations. The integral equation is usually either an *Electric Field Integral Equation (EFIE)* or a *Magnetic Field Integral Equation (MFIE)*. In both cases, the equations physically represent the surface or volume integral of an object under influence of an incident electromagnetic field; they take the form

$$E_i = f_E(J) \qquad M_i = f_M(J),$$

where E_i/M_i is the incident field and J denotes the (unknown) induced current in the object. From J other quantities can be inferred, such as the scattered field.

By approximating J using a set of base functions, $\hat{J} = \sum w_i u_i$, the problem is converted into a system of linear equations, as for FEM; solving the system will give us the unknown parameters w_i. Several iterative methods exists, among them [12] and [13].

MM performs well for antenna and electromagnetic scattering problems, but it depends heavily on the formulation of the integral equation — which varies with the problem geometry, and it is therefore not ideal as a general solver method.

2.4 Finite–Difference Time–Domain

In the *Finite–Difference Time–Domain* method (FDTD), the time–variational formulation of Maxwell's equations is used:[14]

$$\frac{\partial \vec{B}}{\partial t} + \nabla \times \varepsilon \vec{D} = 0 \tag{2.2}$$

$$\frac{\partial \vec{D}}{\partial t} - \nabla \times \mu \vec{B} = -\vec{J} \tag{2.3}$$

$$\nabla \cdot \vec{B} = 0 \tag{2.4}$$

$$\nabla \cdot \vec{D} = \rho \tag{2.5}$$

$\vec{B}$ denote the magnetic flux density, $\vec{D}$ the electric displacement. ε and μ are the electromagnetic constants (succeptability and permeability), and ρ is the charge density (on the object's surface).

The computational space is often discretized in two interleaved grids such that the discretized B solution is evaluated on the cell faces of the discretized E solution, and vice versa.

By approximating the space with cubic cells, equations Eq. (2.2) — Eq. (2.5) can be discretized using a variety of numerical approximation schemes that result in equations of the form

$$\begin{cases} B(t + \Delta t) + f\left(D(t)\right) = 0 \\ D(t + 2\Delta t) + f\left(B(t + \Delta t)\right) = 0 \end{cases}$$

Through time stepping, a steady–state or periodic solution is obtained. Runge–Kutta and Leapfrog methods are examples of often used time stepping methods.

The main advantage of FDTD is the low computational expense for each time step, and that the locality in space of common numerical differentiation schemes makes it easy to parallelize the algorithm. The major disadvantage is that the grid size determines the *granularity* in an object's shape, as well as the size of the time step. As all curvatures must be discretized in steps of a cell side's length, the scattering from an object with a well–rounded surface will easily generate a very large problem to solve.

To overcome this problem of a fine discretization of space and time, the *Finite–Volume Time–Domain* method (FVTD) has evolved, such that each cell can have its own shape and volume, but the basic concept is the same.

FDTD and FVTD methods are widely used for electromagnetic problems, primarily to solve radar scattering problems.

2.5 Finite–Volume Time–Domain

By combining the time–dependent Maxwell equations Eq. (2.2) and Eq. (2.3), we get

$$\frac{\partial u}{\partial t} + R(u,t) = \hat{J} \tag{2.6}$$

where the solution vector $u = (B_x, B_y, B_z, D_x, D_y, D_z)^T$ — hereby called *flux*. $R(u,t)$ denotes the residual effect of the curl operators, boundary conditions and surface currents, and $\hat{J} = (0,0,0,J_x,J_y,J_z)^T$.

In the FVTD approach of Shang et al. [15] [16] [17] [18] [19] [20] [21], which is the focus of the rest of this thesis, the scattering object is assumed to be perfectly

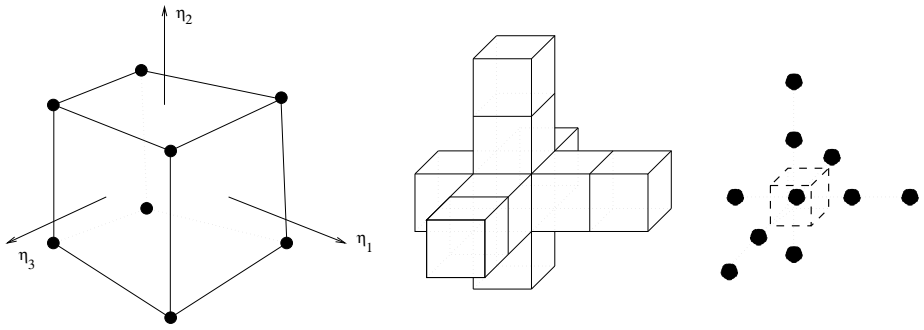

Figure 2.1: The volume cell used in FVTD, along with MUSCL dependencies for computing $R(u,t)$ using the surrounding cell–centered solutions.

---

conducting (also called totally reflecting), which means that $\hat{J} = 0$ in Eq. (2.6). Discretizing the problem with six–sided finite volume cells yields an equation of the form

$$\frac{\Delta u}{\Delta t} + R(u,t) = 0 \tag{2.7}$$

Let the volume cells be described in the curvilinear coordinate system (η_1, η_2, η_3). Call the positive directed surface normals of each volume cell $\vec{\xi}_{1\ldots3}$ (See Figure 2.1).

Supported by eigenvalue analysis and by the axiom regarding superposition of waves, the calculation of $R(u,t)$ can be split in three separate calculations of *flux balance* differences, ΔF, one calculation for each of the three dimensions. For a volume cell at coordinates i,j,k, this can be expressed as

$$R(u_{i,j,k},t) \quad = \quad \frac{1}{V_{i,j,k}} \left(F^1_{i,j,k} - F^1_{i-1,j,k} + F^2_{i,j,k} - F^2_{i,j-1,k} + F^3_{i,j,k} - F^3_{i,j,k-1} \right) \tag{2.8}$$

where $1/V_{i,j,k}$ is the determinant of the discretized Jacobian describing the coordinate transformation between the Cartesian and the curvilinear frame; this scalar value is actually identical to the inverse of the cell volume.

Each flux balance $F^d_{i,j,k}$ is calculated on the positively directed cell interface in coordinate dimension d; it is a discretized formulation of the surface area times the flux density parallel to the surface plane (that is, perpendicular to the surface normal). This is described by introducing the operator T:

$$F^d_{i,j,k} = \sigma_d T(\vec{n}_d) u_{i,j,k} \tag{2.9}$$

where σ_d is the area of the positive directed surface and $\vec{n}_d$ the normalized outward surface normal in dth dimension for cell i, j, k. The operator $T(\vec{n})$ is defined as a 6×6 matrix, but can also be expressed as a 2×2 block matrix of the form below due to its symmetry:

$$T = \begin{pmatrix} \frac{1}{\sqrt{\mu\varepsilon}} A & \frac{1}{\varepsilon} B \\ -\frac{1}{\mu} B & \frac{1}{\sqrt{\mu\varepsilon}} A \end{pmatrix} \tag{2.10}$$

$$A = \begin{pmatrix} 1 - n_x^2 & -(n_x n_y) & -(n_x n_z) \\ -(n_x n_y) & 1 - n_y^2 & -(n_y n_z) \\ -(n_x n_z) & -(n_y n_z) & 1 - n_z^2 \end{pmatrix} \tag{2.11}$$

$$B = \begin{pmatrix} 0 & -n_z & n_y \\ n_z & 0 & -n_x \\ -n_y & n_x & 0 \end{pmatrix} \tag{2.12}$$

Note that the two block matrices defined in Eq. (2.11) and Eq. (2.12) are operators: applied on the vector $\vec{s} = (s_x, s_y, s_z)$,

$$A\vec{s} = \vec{s} - (\vec{n} \cdot \vec{s})\, \vec{n} \qquad B\vec{s} = \vec{n} \times \vec{s}$$

In order to achieve a higher order of accuracy in the space domain Shang now applies the *Monotonic Upwind–centered Scheme for Conservation Laws* (MUSCL)

due to van Leer:

$$\begin{cases} v_i^+ = u_i + \dfrac{\phi}{4}\left((1-\kappa)\Delta u_i + (1+\kappa)\Delta u_{i+1}\right) \\ v_i^- = u_{i+1} - \dfrac{\phi}{4}\left((1+\kappa)\Delta u_{i+1} + (1-\kappa)\Delta u_{i+2}\right) \end{cases} \tag{2.13}$$

where $\Delta u_i = u_i - u_{i-1}$ and ϕ and κ are constants; $\phi = 1$, $\kappa = \frac{1}{3}$ yield third–order accuracy. v^+ and v^- can be thought of as 'outgoing' and 'incoming' flux, traversing through the cell.

Applying the MUSCL scheme to the operator T, we can now compute the flux balance with third–order accuracy;

$$F^d = \sigma_d \left(T^+ v_d^+ + T^- v_d^-\right) \tag{2.14}$$

The modified operators T^+ and T^- are

$$T^+ = \begin{pmatrix} \frac{1}{\sqrt{2\mu\varepsilon}}A & \frac{1}{2\varepsilon}B \\ -\frac{1}{2\mu}B & \frac{1}{2\sqrt{\mu\varepsilon}}A \end{pmatrix} \tag{2.15}$$

$$T^- = \begin{pmatrix} -\frac{1}{2\sqrt{\mu\varepsilon}}A & \frac{1}{2\varepsilon}B \\ -\frac{1}{2\mu}B & -\frac{1}{2\sqrt{\mu\varepsilon}}A \end{pmatrix} \tag{2.16}$$

Special care must be taken at the boundaries. At the outer boundary where the computational domain ends — but not the real world — we define a simple non–reflecting condition, setting incoming flux contribution $v^- = 0$ for the outermost cell layer. For the innermost cell layer (the object surface), $v^+ = u_0 = (\vec{B}_0\ \vec{D}_0)^T$, where $\vec{B}_0$ and $\vec{D}_0$ are defined as

$$\begin{cases} \vec{B}_0 + \vec{b}_0 = \left(\vec{B}_1 + \vec{b}_1\right) \perp \vec{n} \\ \vec{D}_0 + \vec{d}_0 = \left(\vec{D}_1 + \vec{d}_1\right) \,//\, \vec{n} \end{cases} \tag{2.17}$$

$\vec{B}_1, \vec{D}_1$ denote the flux in the cell layer adjacent to the object, and $\vec{n}$ is the unity normal vector of the face that separates the two layers. $\vec{b}_0, \vec{d}_0$ and $\vec{b}_1, \vec{d}_1$ denote the

contribution from the incoming electromagnetic beam that is scattered (this is a known quantity).

Finally, to get an accuracy in the time domain that matches that of the space domain, Shang chooses to solve Eq. (2.7) using the standard fourth–order Runge–Kutta scheme, Eq. (2.18).

$$\left\{ \begin{array}{rcl} u_{t+\Delta t} & = & u_t - \dfrac{\Delta t}{6}\left(u'_1 + 2u'_2 + 2u'_3 + u'_4\right) \\ u'_j & = & R\left(\; u_t - \alpha_j \Delta t \cdot u'_{j-1} \; , \; t + \alpha_j \Delta t \; \right) \quad , \quad j = 1 \ldots 4 \\ \alpha_{1\ldots4} & = & (0, 0.5, 0.5, 1) \end{array} \right. \tag{2.18}$$

In Chapter 3, we will discuss modifications to reduce the number of operations and memory requirements for this algorithm.

Chapter 3

Algorithm analysis

In this chapter, we analyze and modify the algorithm given by Shang et al. (see Section 2.5). By transforming the scheme described in Section 2.5 with elementary algebra, the required number of arithmetic operations and memory can be reduced compared to a direct implementation.

We then discuss a few characteristics of the transformed algorithm to provide insights into the hardware efficiency that can be expected from sequential implementations on the IBM SP and SGI Origin2000.

3.1 Detailed algorithm (re)formulation

Before implementing the algorithm described in Section 2.5, an attempt was made to reduce the number of operations to be performed. This was done by considering different approaches to carry out matrix–vector multiplications and the MUSCL scheme.

3.1.1 Consolidating arithmetic operations

In its original form, the MUSCL scheme defined in Eq. (2.13),

$$\begin{cases} v_i^+ &= u_i + \dfrac{\phi}{4}\left((1-\kappa)\Delta u_i + (1+\kappa)\Delta u_{i+1}\right) \\ v_i^- &= u_{i+1} - \dfrac{\phi}{4}\left((1+\kappa)\Delta u_{i+1} + (1-\kappa)\Delta u_{i+2}\right) \end{cases}$$

needs 120 flops to compute v^+ and v^- for each volume cell and face direction (10 flops for each vector element, two vectors of length six). Instead, we exploit the fact that

$$\alpha = \frac{\phi}{4}(\kappa - 1) \qquad \beta = \frac{\phi}{4}(\kappa + 1) \qquad \gamma = 1 - \frac{\phi\kappa}{2} \tag{3.1}$$

are constants, which by direct substitution gives the scheme

$$\begin{cases} v_i^+ = \alpha u_{i-1} + \gamma u_i + \beta u_{i+1} \\ v_i^- = \alpha u_{i+2} + \gamma u_{i+1} + \beta u_i \end{cases} \tag{3.2}$$

Thus, v^+ and v^- can be calculated in 60 flops — a reduction by half.

The computation of the flux balance $F = \sigma(T^+v^+ + T^-v^-)$ can be consolidated into two 3×6 matrix–vector operations

$$F_B = \begin{pmatrix} & \\ a\sigma A & b\sigma B \\ & \end{pmatrix} \begin{pmatrix} w_B^- \\ \\ w_D^+ \end{pmatrix} \tag{3.3}$$

$$F_D = \begin{pmatrix} & \\ c\sigma B & a\sigma A \\ & \end{pmatrix} \begin{pmatrix} w_B^+ \\ \\ w_D^- \end{pmatrix} \tag{3.4}$$

where

$$a = \frac{1}{2\sqrt{\mu\varepsilon}}\ ,\ b = \frac{1}{\varepsilon}\ ,\ c = -\frac{1}{\mu}\ ,\ w^+ = v^+ + v^-\ ,\ w^- = v^+ - v^-.$$

Taking the zero diagonal in B into account, this will only require 6×11 flops.

The linear combinations $w^+ = v^+ + v^-$ and $w^- = v^+ - v^-$ can be calculated directly in 60 flops instead of first computing v^+ and v^- and then adding/subtracting them — a total of 72 flops per cell and face. Computing the linear combinations w^+ and w^- is done by defining the constants $\delta_\pm = \gamma \pm \beta$:

$$\begin{cases} w^+ = v^+ + v^- = \alpha\,(u_{i-1} + u_{i+2}) + \delta_+\,(u_i + u_{i+1}) \\ \\ w^- = v^+ - v^- = \alpha\,(u_{i-1} - u_{i+2}) + \delta_-\,(u_i - u_{i+1}) \end{cases} \tag{3.5}$$

Following the modified scheme outlined above, computing F for each cell and face direction is reduced to calculating w^+ and w^- ($30 + 30$ flops) and size inner–product–like computations (6×11 flops), a total of 126 flops. The original scheme demands 120 flops for v^+ and v^-, two 6×6 matrix–vector multiplications (132 flops) and one length–6 vector addition (6 flops), totaling 258 flops.

Note that in the analysis above, the computation of the scale factors and the matrix elements from $\vec{n}$ and σ have not been taken into account. These operations require an additional 12 flops for both the original and the modified scheme. Another 12 flops will be needed as well to compute the contribution to the cell residual of the flux balance in the dth dimension :

$$R(u_i, t) = R(u_i, t) + F_i^d - F_{i-1}^d$$

When computing in the third and last dimension, the residual is scaled by the inverse of the cell volume, at a cost of another 6 flops; this also concludes the computation of $R(u_i, t)$ for one Runge–Kutta step.

To summarize our transformations, we remark that the original scheme, if implemented, would require $(3 \times 282 + 6) = 852$ flops, while ours requires only $(3 \times 150 + 6) = 456$ flops — a reduction of 47%.

3.2 Performance issues

Compiler optimization is a field in Computer Science that still evolves. Hard problems to solve include scheduling (reordering the instructions to avoid stalls) and resource management (deciding what values to put in the processor registers and when to load them from main memory). As these types of problems are NP–complete, we cannot demand that the compiler does a perfect job and uses the hardware optimally.

In this section, we will perform a theoretical analysis of the IBM SP and SGI O2K architecture, and derive upper limits for the expected performance of the enhanced algorithm in Section 3.1. We will conclude that the program performance will be limited by the memory bandwidth and never perform better than about 70% of peak for the IBM architecture. For the SGI, blocking may reduce memory bandwidth and thereby significantly improve peak performance from 35% to approach 60%, however this is an ideal case and an implementation can be expected to perform worse.

3.2.1 Performance degradation due to arithmetics

As both RISC processors considered (Power2 and R10000) have 32 (accessible) floating–point registers, the reformulated algorithm turns out to be problematic to implement. The restriction on the number of values that can be kept in registers forces us to abandon a further optimized algorithm which consists of reusing computed values, as v^-_{i+1} can be easily computed from v^+_i. The implemented version of the algorithm computes the flux balance for one face and adds it to the residual $R(u_i, t)$ in 162 flops, which is only 12 flops more than for the optimal scheme in Section 3.1.1. The extra flops come from a small change (w^+ and w^- are not calculated directly) which make it easier to incorporate irregularities at inner and outer

Task	Operation	Iter	Mult	Add	1–op	2–op
v^+, v^-	12 L-3 IP's	12	36	24	12	24
w^+, w^-	$w^{\pm} = v^+ \pm v^-$	12	0	12	12	0
$n_i n_j$, $a\sigma$, $b\sigma$, $c\sigma$	mult	12	6	0	6	0
$1 - n_i^2$	mult–subtract	12	3	3	0	3
F_B, F_D	6 L-3 IP, 12 L-2 IP	12	42	24	18	24
update R	$R + (F_i - F_{i-1})$	12	0	12	12	0
scale R with $1/V_{ijk}$	mult.	4	6	0	6	0
update u_{New}	$u_{New} + \alpha R$	4	6	6	0	6
update u	$u_{Old} + \beta R$	4	6	6	0	6
		Sum	1116	948	744	660

Table 3.1: Machine instructions required for one full time step. "L-n IP" means a length–n inner product calculation. "Iter" refers to the number of times the computation is iterated.

boundaries.

As the Power2 floating–point units are fully utilized only when performing *mult–add* instructions, the IBM implementation cannot achieve 100% of the processor peak on our algorithm. As for the SGI architecture, the multiplier and adder units are able to work independently, and therefore the unit that does the most work determines the number of cycles required. We assume a perfect scheduling for the other unit.

The 2064 operations required for computing one full time step (four Runge–Kutta iterations) for one cell are listed in Table 3.1; a total of 1116 *mult* and 948 *add*, which can be combined into 744 single–flop instructions and 660 *mult–add* instructions. For the IBM platform, this yields 1.47 flops per instruction on average, or 73% of peak (2 flops per instruction). The SGI platform will have to wait for the multiplier unit to complete its operations. Assuming a perfect scheduling of the adder (which is not too far–fetched), the 2064 operations will be performed in 1116 cycles, or 92% of peak (2 flops per cycle).

Task	Variables	Elem.	Iter.	Load	Store
w^+, w^-	$u_{i-1} \ldots u_{i+2}$	24	12	288	0
matrix elements	$\vec{n}$, σ	4	12	48	0
store result	F	6	12	0	72
compute ΔF	F_i, F_{i-1}	12	12	144	0
update R	$R(u_i, t)$	6	12	72	72
scale R	$\frac{1}{V_i}$	1	4	4	0
next R–K step	$R(u_i, t)$	6	4+0	24	0
	u_{Old}	6	3+1	18	6
$u_i = u_{Old} + \alpha R$	u_i	6	0+4	0	24
$u_{New} = u_{New} + \beta R$	u_{New}	6	4+4	24	24
			Sum	622	198

Table 3.2: Total number of *load* and *store* operations required per cell for a full Runge–Kutta iteration.

So, assuming the memory access is ideal, the reformulated scheme will not achieve higher efficiency than 73% of the processor peak for the IBM platform, and 92% for the SGI.

3.2.2 Performance degradation due to memory access

To estimate the impact of memory operations on the performance, we studied required memory access in a way similar to the analysis in Section 3.2.1. For one full time step (four Runge–Kutta iterations), our transformed algorithm requires 622 load operations and 198 store operations per cell (Table 3.2); a total data transfer of 6560 B (using 64–bit values).

If there is no memory bottleneck, the 2064 flops per time step and cell can be carried out in 1116 cycles on the SGI architecture, as mentioned earlier. The average sustained memory transfer from main memory to processor would be $\frac{6560}{1116} = 5.87$ bytes/cycle, or 1093 MB/sec, as the processor speed is 195 MHz. This memory

transfer rate is 1.4 times higher than the possible 780 MB/sec between memory and processor, or 2.8 times the 390 MB/sec achieved when using both processors in a pair. We therefore conclude that a straightforward implementation will be heavily limited by memory access on the SGI O2K, and at best perform close to $\frac{1}{2.8} = 35\%$ of peak in multiprocessor mode.

This performance can however be enhanced by *blocking* the code and utilize the secondary cache, as the cache bus is much wider and faster than the system bus (see Figure 1.3 in Section 1.4). By prefetching, a cell block will be in the cache and when performing the whole computation of $R(u,t)$ for the cells in that block, a substantial increase in performance can be expected due to the quick loads and reusage of the elements.

Assuming a blocked scheme such as this will eliminate the memory bottleneck for the computation of $R(u,t)$, we can estimate the effect in performance by looking at the 10.7% of the arithmetical operations that do not benefit from a blocking scheme — the Runge–Kutta update. Combining the results in Table 3.1 and Table 3.2, the operations associated with the update need to load/store 992 B per 72 cycles, or 6.8 times the theoretically possible peak transfer from main memory in multiprocessor mode. A slowdown of 10% of the computation by a factor 7 will yield a theoretical hardware efficiency of 60%.

We have so far not taken into account the extra operations needed to store temporary results in the computation of $R(u,t)$. Even if the cache could be fully utilized for this purpose, a slowdown from the ideal case will occur. We therefore conclude that a peak efficiency of a blocked implementation on the SGI Origin2000 will be well below 60%, as the memory bus on the SGI platform is simply too narrow for this type of application.

Routine	L1	L2	Leff	S1	S2	Seff
F, update $R(u_i, t)$	39	34	0.732	9	11	0.775
R–K update of u_i, u_{New}	9	94	0.956	0	66	1.000

Table 3.3: Statistical load/store efficiency analysis of IBM compiler–generated output.

The rest of this section is dedicated to the theoretical performance of the implementation on the IBM architecture. This is more difficult to derive because we can choose to load/store one or two values per instruction.

In an ideal case, the memory operations in table Table 3.2 can be done in $\frac{622+198}{4} = 205$ cycles, with two 128–bit load/store instructions issued per cycle. However, the fact that all data is not always perfectly aligned and resource constraints (such as not enough available registers) may force the processor to load data one value at a time — thus discarding half of the information coming through the memory bus.

Analysis was therefore conducted on the compiler–generated assembler code. The two routines considered were the one that computes $R(u_i, t)$ and the one that updates the Runge–Kutta summations. The resulting efficiencies are shown in Table 3.3.

Adjusting the ideal–case calculations in Table 3.2 with the newly obtained efficiencies, we find that the number of instructions increases from 205 to 270. This corresponds to a memory bus load of 8640 B per cell and time step.

For the IBM architecture, the 2064 flops per cell and time step are carried out in 1404 instructions, or 702 machine cycles (there are two FPUs). This means that in order to not be constrained by the memory, the minimum sustained bandwidth requirement between CPU and memory is $\frac{8640}{702} = 12.31$ bytes/cycle.

For the three possible processor speeds (120,135 and 160 MHz), this yields minimum transfer rates of 1409, 1585 and 1878 MB/sec. This is fully within theoretical limits, provided a full memory population.

However, taking into account the degradation due to the algorithm frequently switching between storing and loading, and the fact that the actual memory bandwidth is far less than the theoretical for sustained memory scans, we conclude that the algorithm, when running on the IBM architecture, will be somewhat bounded by memory bandwidth — but by far not as much as for the SGI architecture.

For comparison, we conduct the same special analysis of the Runge–Kutta update as in the SGI case. We reveal a need of loading 1152 B per 60 cycles, or 1.04, 1.16 and 1.38 times the theoretical bandwidth for 120, 135 and 160 MHz processors, assuming a full memory population. This does not affect the overall peak performance by more than 4%.

Chapter 4

Parallel implementation

In this chapter, we discuss implementation details and parallelization of the enhanced FVTD algorithm. An automatic partitioning scheme is described, as well as the modularity and structure of the code.

Partly due to the results in Section 3.2.2, the implementation work was mainly focused on the IBM SP. The resulting code, however, assumes nothing about the underlying architecture and is fully portable.

4.1 Performance issues

To boost performance, several small investigations were conducted to see what scheme would perform best. As an example, one such investigation found out that the computation at the end of each Runge–Kutta iteration, updating u and u_{New} by the operations

$$\begin{cases} u_i = u_{Old} + \alpha R(u_i, t) \\ u_{New} = u_{New} + \beta R(u_i, t) \end{cases} \tag{4.1}$$

is done 25% faster if first setting $u_i = uOld$, then performing the operations

$$\begin{cases} u_i = u_i + \alpha R(u_i, t) \\ u_{New} = u_{New} + \beta R(u_i, t) \end{cases} \tag{4.2}$$

in the same loop, reusing the loaded value of $R(u_i, t)$.

4.1.1 Dimension reordering

In partitioning a problem, the volume parts assigned to each processor may have very high aspect ratios with respect to the extent in the different dimensions. (For example, this is often the case if not all dimensions are split.) To ensure best possible performance, an automatic dimension reordering was implemented, such that the dimensions are in the order of decreasing extent with the largest dimension innermost. Experiments showed that for a problem with uneven dimension extents, a reordering in this manner could yield significant performance gain.

The major drawback with the dimension reordering is that it creates a more complex implementation, so the peak performance will degrade. However, tests show that for interesting problem sizes, this degradation is less than a couple of percent compared to an implementation with fixed, ideal loop order.

The dimension reordering also has the advantage that the implementation is forced to become coordinate system independent.

4.2 Communication issues

4.2.1 Analysis of communication patterns

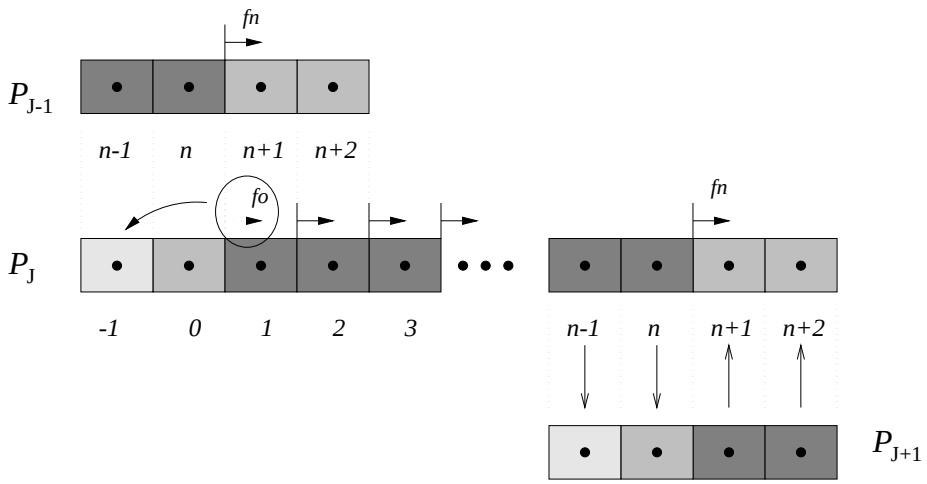

Figure 4.1: Communication dependences in one dimension for processor P_J.

Due to the MUSCL scheme, a 'two cell layers up, one cell layer down' dependence is imposed along each of the coordinate axes. But, to be able to compute $R(u_1, t)$, F_0 ($= F_n$ in the eyes of the other processor) is needed in addition to the local F_1 (see Figure 4.1).

Instead of communicating twice, first getting u_0 and in a second step F_0, a two–layer package (u_{-1}, u_0) is sent, whereby the computation of F_0 is overlapped and done locally. Sending two layers instead of one also balances the communication and makes the time between each communication longer, which is good with respect to latency hiding.

A good scheme for parallelism in the communication is to impose a *red–black* communication scheme (see Figure 4.2). Studies have shown that sequencing the communication in this manner outperforms the somewhat anarchistic scheme of starting all communication simultaneously [22]. The benefits from red–black communication

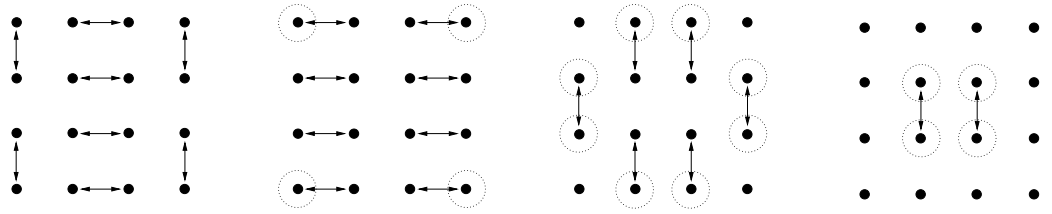

Figure 4.2: Communication using a red–black sequence. This method is generally faster than starting all communication simultaneously, as all nodes communicates with at most one neighbor at a time. The nodes are circled to denote the last communication step in which they participate.

---

mainly lies in avoiding network adapter limitations and minimizing buffer maintenance — the true bottlenecks of networked systems these days — than minimizing the network load.

But, considering the irregular communication patterns that arises from the curvilinear coordinate system, it is not obvious how to devise an algorithm generating an optimal sequence in the general case. The choice therefore initially fell on the anarchistic approach, starting all data transfer simultaneously by using the non–blocking communication primitives in MPI — a simple, yet the second most effective method. We will see in Chapter 6 that the performance of this method suffices.

4.2.2 Partitioning issues

Given the number of processors, the problem size, and the curvilinear coordinate dependences, our implementation partitions a given grid into equally large subgrids at run–time. The implementation minimizes the communication by trying a number of different splitting options. In most cases, minimizing the border area of the subgrids will yield the best result. However, that is not the case in a curvilinear frame. An

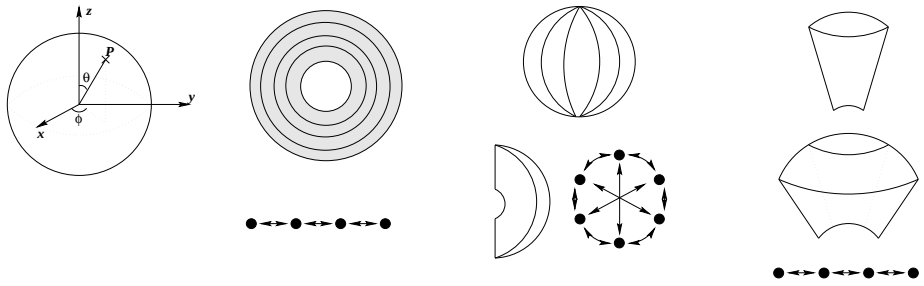

Figure 4.3: Partitions and communication dependences for different cuts in spherical coordinates, from left to right: r, ϕ, θ.

example involving spherical coordinates is given in Figure 4.3.

If, for instance, the third coordinate dimension ϕ is split, making a 'peeled orange' distribution, a communication dependence will also arise in the θ dimension, as the MUSCL dependences in the θ dimension for $(\theta = 0)$ and $(\theta = \pi)$ require the solution u at the opposite side of the z axis. However, when splitting the θ coordinate, these boundary dependences will be local. The time to complete an extra local copying of data is neglible compared to the communication time.

4.2.3 Latency hiding improvements

The possibility to overlap the communication time by computation was investigated. As the domain is partitioned into 3–D volumes, the amount of data that needs to be sent increases with $\mathcal{O}(N^{2/3})$, while the computations increase with $\mathcal{O}(N)$, where N is the number of cells. It therefore suffices that the partition assigned to each node is sufficiently large to completely hide the communication.

In the original IBM implementation, attempts were made to minimize the latency by constructing special stencils using user–defined MPI data types, masking the

memory of the different face elements in the u structure. By using these stencils, the copying of the border elements are avoided. However, tests show the CPU time needed to evaluate the stencils causes the communication to require much longer time when overlapping it with a heavily loaded floating–point unit.

The execution flow of our implementation is as follows: communication is started in the background, followed by boundary updates; then, when the communication is finished in a certain dimension, the contribution to $R(u_i, t)$ for that dimension is computed.

Two approaches to improve performance were investigated on the IBM SP. The two alternative schemes are enabled by setting a preprocessor directive. They both take advantage of the fact that sending or receiving a linear array can be completely handled by the network logic, fetching and storing the elements with Direct Memory Access transfer (DMA). The processor is never involved in any way; tests showed that floating–point performance does not degrade at all when performing communication simultaneously in this fashion. We note a slight performance degradation due to the extra copying of the border elements in the u structure.

In addition to exploiting the DMA transfer, our first alternative loops over the dimensions such that a dimension with no communication is processed first, giving some higher degree of latency hiding.

The second alternative is more effective. It confines the copying of border elements to linear arrays and computes $R(u_i, t)$ for all cells except the cells closest to a communication border. These are computed later in a second step. A 2% performance penalty is noted due to the break down of the computation into smaller parts.

In Chapter 6 we will show that both alternative 1 and 2 generate a better performance then the original communication scheme. The second approach shows very high latency tolerance.

4.3 Code organization

The implementation is divided up in Fortran90 *modules* to easily incorporate updated software. To enhance code structure and to minimize errors in the code, macros are extensively used. The code therefore needs to be preprocessed by a C compiler before piping it into the Fortran compiler.

The program is written with an intent to support high quality visualization; the solution at each border of each partition can periodically be flushed to disk. An accompanying grid definition file (automatically generated) makes it possible to view the solution in 3-D and also to make MPEG-1 movies of the solution process. The output generated is made so that it can be directly visualized using IBM's Data Explorer. The movies can be produced using the Berkeley MPEG encoder software [23].

A chart of the different modules and their dependences can be seen in Figure 4.4. The command line arguments understood by the implementation are listed in Table 4.1.

4.3.1 Generic modules

A set of generic modules were created to support the implementation. Each generic module is designed for a specific task such as communication, file I/O, command line argument parsing, time and performance measurement, etc. All modules are

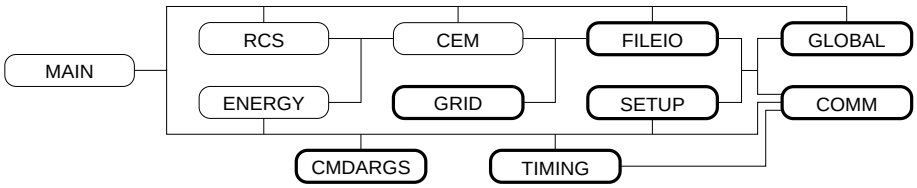

Figure 4.4: Module dependences. The dependences are directed out from the modules' right side. The modules with thick borders are generic or derivatives of generic modules.

Option	**Action**
`-a` ϵ	Sets numerical epsilon to ϵ.
`-d D1 D2 D3`	Sets the dimension reordering to (`D1`, `D2`, `D3`), the first dimension being the innermost.
`-f flags`	Toggles a number of flags, enabling or disabling different program outputs (for instance, a memory dump after the last time step, making it possible to resume a simulation).
`-g`	Dumps a grid definition to disk (for visualization).
`-i I`	Dumps border solutions with time step interval `I`.
`-l file`	Loads a grid definition from files with name `file.###`, where `###` is the processor number.
`-n N`	Sets the time step limit to `N`.
`-p P1 P2 P3`	Sets the partition layout, where `P`x is the number of processors in each dimension.
`-r B E`	Performs a Fourier transform on the solution between time steps `B` and `E`, and compute a RCS.
`-s N1 N2 N3`	Sets the problem size to (`N1`,`N2`,`N3`).
`-w` ω	Sets the angular frequency of incoming electromagnetic beam to ω.

Table 4.1: Command line arguments for our parallel implementation.

constructed with code reusability in mind.

The *global* module contains the general data structures, for easy access. It also provides a set of utility functions, such as transforms between real and reordered dimensions.

The *communication* routines are designed so that the underlying communication package (in this case MPI) can be easily exchanged. Each function is written to be general (`send_int()`, `recv_real()`, ...). MPI specific details, such as providing a message tag (a unique integer) for all concurrent messages exchanged between two processors, is hidden from the outside world within the communication routines, and managed automatically to ensure robustness.

The *file I/O* routines use the communications routines, so that only one processor will actually perform the physical read and write operations. By doing this, we do not have to send all our data sets to remote sites when computing in a distributed environment; all data are read and written by the 'master' processor (the processor with lowest rank in the MPI hierarchy).

The *grid generation* routines provide suitable parts of the model problem (see Section 4.2.2 and Section 4.4), given a partition and a processor number. The subgrid is then sent out to the appropriate processor. There is also an option of reading the grid parts from file, thus making it easy to change the code to compute the scattering of some geometric object other than the model sphere. The grid generation routines also determine the status of the different grid faces, if it is an inner boundary, outer boundary or if communication is necessary (and if so, with whom).

The *setup* module converts the grid information by applying the dimension reordering (Section 4.1.1). It also computes surface areas, normal vectors, and the inverse volume for each cell.

The *timing* module keeps track of a set of stop watches. It also contains a report generator, producing a table with min, max, average and variance. Providing the number of operations, there is also an option to get the results in flop rates.

4.3.2 Problem–specific modules

Three modules were written specifically for our algorithm. The *cem* module contains the subroutines for computing $R(u_i, t)$, update the Runge–Kutta solutions, initial setup, and inner and outer boundary conditions.

The *RCS* module filtrates the solution u over time, then using the Fourier transformed solution to compute a near–field to far–field transformation and get an estimated radar cross section of the object.

The *energy* module periodically computes the total energy in the system and reports it; this is used mostly to get an early warning if there are flaws in the model or the setup.

4.3.3 Execution flow

The execution flow of our implemented program can be divided in six major parts, see Table 4.2. A detailed description of the loop part can be found in Table 4.3.

4.4 The model problem

For the implementation verification and pre–study, a model problem was used to prove the correctness of the code; calculating the scattering from a perfectly conducting (PEC) sphere (sometimes also called totally reflecting), centered at the origin.

Part	Description
init	Communication initialization, command arguments
allocate	Data structure allocation
setup	Grid distribution, surface vector calculation
loop	The simulation
postprocess	RCS computation, performance analysis
exit	Deallocation, communication shutdown

Table 4.2: Execution flow of our implementation.

```
do step=1,timesteps
     do rkstep=1,4
          start_comm(u);
          update_inner_and_outer_boundaries(u);
          if (latency_hiding)
               compute_R(u,R,core);
               foreach face
                    wait_comm(face);
                    compute_R(u,R,face);
               end
          else
               wait_comm(all);
               compute_R(u,R,all);
          end
          update_RK_solution(u,R,u_New,u_Old,rkstep);
     end
end
```

Table 4.3: Loop execution flow.

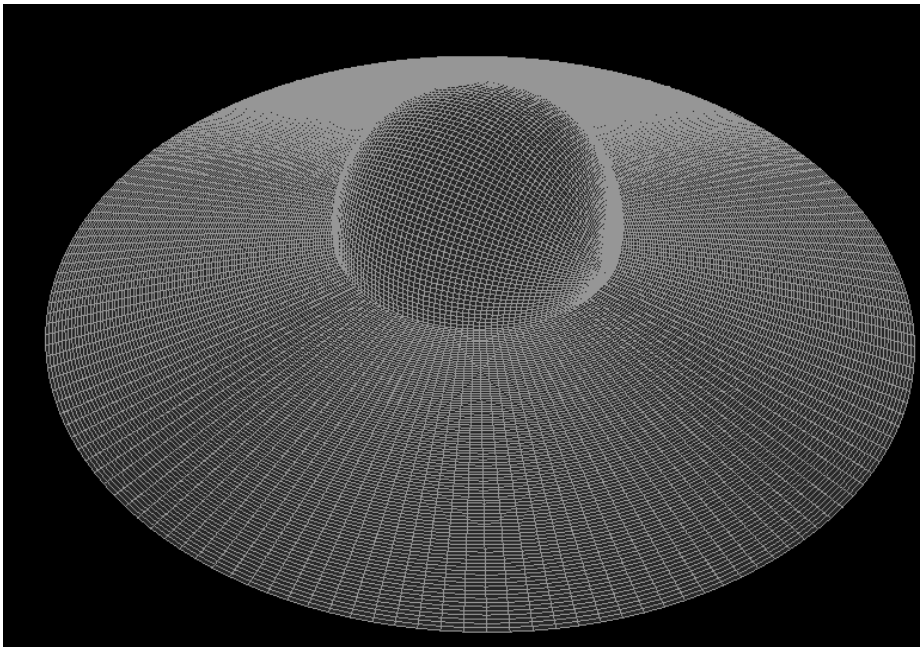

Figure 4.5: Model problem grid (automatically generated).

With a plane, polarized, sinusoidal incident wave, the Radar Cross Section (RCS) of the sphere will have an analytical solution, due to Gustav Mie [24]. The solution is not obtainable in a closed form, but can be expressed as infinite sums of Bessel and Hankel functions, which are invoked when expressing the equation of a plane wave in spherical coordinates. A large set of programs exist to compute approximations of these infinite sums; one such program, MIEV [25] was used to verify the output of the new implementation.

Scattering problems are often characterized by the number $K\alpha$, where $K = \frac{2\pi}{\lambda}$ is the wave number of the incident wave and α the dimensionality of the object – in this case the radius of the PEC sphere.

Previously, solving scattering problems for $K\alpha > 20.0$ with a high accuracy has

not been feasible, since it requires very large (dense) meshes and extensive computer power. Recent efforts have reported successful computations for $K\alpha = 50.0$ [26].

As mentioned briefly in Section 1.2, we benefit from parallelizing a high–order algorithm in several ways:

- We can utilize more computer power and memory
- We can increase the cell sizes for the same error bound compared to a low–order algorithm, thereby reducing the problem size
- High–order algorithms have the potential to be computationally more efficient due to the increased number of operations. Therefore the penalty of parallelization may be much smaller compared to a low–order algorithm.

Chapter 5

Distributed Computing

In this chapter, we give a brief overview of how the need for distributed computing has arisen. The Globus toolkit [27] is explained in some detail. We end with a few important observations made when porting MPI programs to the GUSTO testbed.

5.1 Background and overview

The ever improving networking infrastructure and increasing collaboration between sites have enabled high–performance applications requiring capabilities not found in a single computer. The increasing amount of collaboration between research centers also calls for a more shared view of the resources. *Metacomputers* are virtual cluster of supercomputers and other rare hardware, connected by high–speed networks.

The metacomputer concept supports *shared scheduling*. In an ideal future, one will be able to log in to one giant metacomputer, start a process and retrieve the result from the computation, not caring or even knowing whether the process actually ran on a local or remote computer, or as a distributed process on a collection of

computers.

Early development solutions had a specific target in mind. Condor [28] and Nimrod were primarily created to locate available computing power among a set of workstations. Both had to be used together with some other software like AFS and DFS for the remote file access, or Kerberos [29] for authentication. (Recent development of CORBA will eliminate this dependence.) The object–oriented Legion project [30] takes another viewpoint and does not adhere to the widespread standards; Legion programming is done in the Mentor programming language.

The Globus project [27] tries to create a full–feathered metacomputing toolkit for dynamic environments, including information services, remote authentication, I/O and resource allocation. It incorporates many of the ideas from the I–WAY project [31]. A more detailed description follows in Section 5.2.

5.2 Globus

The Globus toolkit is still at the time of writing in beta version. The University of Houston participates in the project's test bed, GUSTO.

A hierarchal overview of the Globus components can be seen in Figure 5.1. The underlying communication engine is Nexus [32], a general–purpose communications package. Nexus is implemented on top of numerous network architectures, including TCP, ATM adaption layer 5, IBM MPL, etc.

Among the other low–level services are

- GSS, a security and encryption package
- GRAM, the local gatekeeper that authenticates and handles remote resource allocation requests

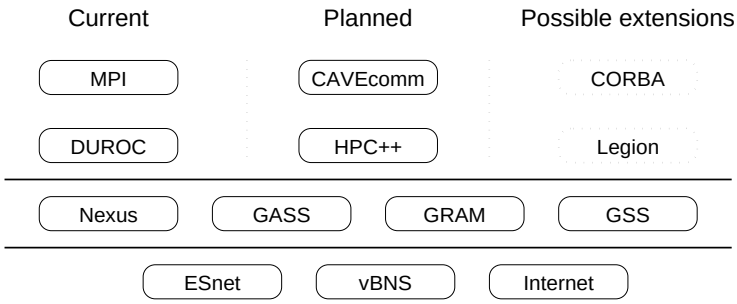

Figure 5.1: Overview of the Globus toolkit. The top–most application level is divided in three regions, illustrating existing services, planned services, and possible future extensions, from left to right, in that order.

- GASS, with support for remote file I/O access and caching.

5.3 Code modification issues

Experiments were conducted with a very early version of the Globus–MPI package, to evaluate what bottlenecks (if any) arise when porting the parallel CEM implementation to a distributed computing environment. The application was run using the SP systems at University of Houston and Argonne National Labs, using the vBNS ATM backbone. As the tests were made using debug–enabled evaluation code, it is at this stage hard to justify comparisons with the implementation running on a single supercomputer. Only main results and experiences are therefore commented upon below.

As the messages take of the order of hundreths of a second to reach their destination, as opposed to microseconds over the backplane switch, it is important to address the issue of good parallelization and latency tolerance. Also, care has to

be taken with the way in which the partitions are layed out across the sites, so that the communications over the *weak links* (in this case the vBNS connection) are minimized.

Another important area is collective communication, such as broadcast and reduction operations. These are most often implemented in a tree fashion, which is undesirable in the case of a distributed environment – preferably only a single message shall traverse the weak link. Tests conducted during and after SC97 [33] showed that by clustering the nodes in groups with one group per site plus another group with the root nodes from the local groups, a three–step rewriting of the collective operations (local group, root group, all other local groups) could reduce the communication time sixfold.

Chapter 6

Results

In this chapter, we present results of our parallel implementation. Due to the low performance bounds estimated for the SGI Origin2000, the work was concentrated on the IBM SP architecture.

6.1 IBM SP

6.1.1 Accuracy issues

The output of our implementation was verified against an existing sequential Fortran77 program, verified and provided by Dr. Joseph Shang [17].

In order to keep control of the numerical accuracy, an error of size $\epsilon = 10^{-12}$ is introduced. This is larger than the errors imposed by machine roundoff.

Due to successive iteration and major differences in operations performed, the two solutions will not be identical. A maximal difference for a single vector element is 10^{-10} after 200 time steps on a $30 \times 40 \times 50$ grid. However, as the solution is $\mathcal{O}(1)$ in magnitude, this divergence is not significant. No difference in sign between the

two solutions has been detected.

The physical correctness of the computed result is illustrated in Figure 6.1, where the total energy in the magnetic field and electric field is plotted over time. The fluctuations between magnetic and electric energy fields responds well to the theory of electromagnetic waves.

6.1.2 Hardware efficiency

The hardware efficiency was measured on the different Power2 processors with a full memory population, and compared to the hardware peak. The problem size ranges from 32K to 456K cells with a cubic partition (all dimensions equally large).

The peak performance was measured per time step and also for the computation of $R(u,t)$ which accounts for 90% of the total arithmetic work. Results are listed in Table 6.1.

The peak performances listed are around 2% higher than the average performance. This can be expected, as the operating system will interrupt the program and occasionally longer execution times will be observed. We note that the performance is independent of problem size, provided the dimension extents are the same.

For an older 77 MHz wide Power2 node, for which the memory bandwidth requirement should not impose a limit, a performance of 52% of hardware peak is achieved when computing R. That is 70% of the theoretical peak derived in Section 3.2. On the P2SC processors, performance drops to around 40% of hardware peak, or 55% of the theoretical estimate.

The overall performance is lower by 8% for the 77 and 120 MHz processors, and by 5% for the 135 and 160 MHz processors. That yields an overall performance of

59% of theoretical peak for the 77 MHz processor, and 44% for the other processors.

We conclude that the implementation on the P2SC processors is limited by memory bandwidth, as predicted from the theoretical analysis. The T4 nodes maintain the same overall performance as the thin nodes, even though the memory bandwidth requirement is 33% higher. It is reasonable to assume that factors such as the improved cache enables this.

6.1.3 Performance impact of vector lengths

When measuring the hardware efficiency, the vector lengths become an issue, as programs in general benefit from looping over longer arrays.

Our implementation was tested on wide nodes, with the innermost dimension extent ranging from 50 to 9000 cells. Performance was measured for the $R(u,t)$ computation and time step. Results are seen in Table 6.2, Figure 6.4 and Figure 6.5.

We see that the flop rates increase by 20 Mflops/sec, or a performance gain by 4%. We conclude therefore that our implementation is fairly insensitive to vectorizing effects.

6.1.4 Whole program analysis

We also studied the efficiency of the different parts of our implementation. The execution time consumed was compared to the arithmetical work. This was done for the original communication scheme and for the two latency hiding schemes outlined in Section 4.2.3. Results are shown in Table 6.3.

The second alternative approach displays a very good latency tolerance. For a partition size of 150K cells per processor, it allows the communication routines

to carry out their task for up to 0.55 seconds, or 80% of the total time between communications.

6.1.5 Scalability

Scalability was measured for both scaled and non–scaled problems on the IBM SP. Timings were collected for the implementation using the best latency hiding alternative (see Section 4.2.3) for up to 96 nodes on the PDC SP Table 6.6. All three communication schemes were timed on the UH SP, where limitations in the queue configurations restrained the scaling to 32 nodes.

For the problem size chosen ($60 \times 48 \times 96$), the latency hiding schemes perform about equal (Figure 6.6). The parallelization maintains 80% and 95% efficiency for non–scaled and scaled problems, respectively. The communication time is completely hidden, except for the of the extra copying to and from linear buffers — and for the performance of the original communication scheme, which non–scaled parallel efficiency drops to about 70%.

We also see from the PDC runs (Figure 6.7) that the implementation is indeed scalable, with more than 90% parallel efficiency for both a scaled and a non–scaled problem.

6.2 SGI Origin2000

An efficient implementation on the SGI O2K requires that the code be blocked for memory bandwidth conservation. Otherwise, the theoretical peak performance is only 35%, as derived in Section 3.2. A blocked version of the code was not developed as part of this thesis.

size	77 MHz		120 MHz		135 MHz		160 MHz	
	R	tot	R	tot	R	tot	R	tot
32^3	0.510	0.429	0.391	0.319	0.355	0.313	0.360	0.316
48^3	0.512	0.428	0.395	0.319	0.351	0.309	0.372	0.320
64^3	0.515	0.429	0.398	0.318	0.368	0.314	0.374	0.321
77^3	0.517	0.431	0.400	0.317	0.370	0.309	0.370	0.322

Table 6.1: Peak hardware efficiency for the Power2 processor family. All measurements were done on systems with full memory population.

length	Non–scaled		Scaled	
	R	tot	R	tot
50	185.96	188.13	180.29	163.09
100	192.09	186.65	188.76	166.88
200	194.42	187.09	192.66	171.03
400	193.61	184.10	194.77	172.40
800	198.95	182.38	198.18	168.33
1000	199.14	181.22	198.53	168.05
2000	195.86	180.09	199.04	170.34
3000	197.01	179.99	198.13	174.09
4000	199.98	180.12	196.57	173.88
5000	200.19	179.37	196.02	176.64
6000	195.79	180.19	198.04	179.22
7000	197.16	179.82	198.74	179.51
8000	199.78	180.27	199.71	180.32
9000	200.17	180.74	200.13	180.81

Table 6.2: Performance versus vector lengths. Flop rates in Mflops/sec on 135 MHz Wide nodes, full memory population. Non–scaled problem size = (length $\times$ 8 $\times$ 8), scaled problem size = (length $\times\, n \times m$), n and m chosen to get a total close to 576000 cells.

Task	Original		LatHid 1		LatHid 2	
	time	arit	time	arit	time	arit
boundary updates	0.6	0.0	2.1	0.0	6.2	0.0
waiting for communication	12.0	0.0	3.5	0.0	2.9	0.0
computing $R(u_i, t)$ (core)	73.1	90.5	78.9	90.5	67.1	74.2
computing $R(u_i, t)$ (borders)	0.0	0.0	0.0	0.0	8.2	16.2
updating R–K aggregates	14.3	9.5	15.5	9.5	15.6	9.5

Table 6.3: Execution time versus arithmetic workload (percentage), using the original communication scheme and the two latency hiding schemes on the IBM SP, 16 thin processors. Problem size 256K cells per processor.

Task	Original		LatHid 1		LatHid 2	
	time	arit	time	arit	time	arit
boundary updates	1.6	0.0	4.7	0.0	7.9	0.0
waiting for communication	20.8	0.0	7.2	0.0	6.6	0.0
computing $R(u_i, t)$ (core)	65.1	90.5	73.9	90.5	57.2	60.6
computing $R(u_i, t)$ (borders)	0.0	0.0	0.0	0.0	14.4	29.9
updating R–K aggregates	12.5	9.5	14.2	9.5	13.9	9.5

Table 6.4: Execution time versus arithmetic workload (percentage), using the original communication scheme and the two latency hiding schemes on the IBM SP, 16 thin processors. Problem size 30K cells per processor.

	Original		**LatHid 1**		**LatHid 2**	
nodes	non–sc.	scaled	non–sc.	scaled	non–sc.	scaled
1	1.000	1.000	1.000	1.000	1.000	1.000
2	0.928	0.958	0.986	0.991	0.984	0.990
4	0.885	0.911	0.978	0.959	0.962	0.972
8	0.846	0.859	0.987	0.936	0.908	0.946
10	0.787	0.857	0.964	0.962	0.939	0.959
12	0.763	0.866	0.904	0.962	0.886	0.958
16	0.770	0.858	0.897	0.934	0.863	0.940
20	0.771	0.823	0.925	0.929	0.907	0.931
24	0.719	0.841	0.863	0.929	0.826	0.933
32	0.698	0.849	0.873	0.948	0.817	0.943

Table 6.5: Parallel efficiency, 120MHz Thin nodes. Problem size $60 \times 48 \times 96$ (275K cells).

	Scaled problem size			Non–scaled problem size		
nodes	HW eff.	speed–up	// eff.	HW eff.	speed–up	// eff.
1	0.317	1.00	1.000	— —	— —	— —
2	0.323	2.04	1.019	— —	— —	— —
4	0.316	3.99	0.997	0.315	3.97	0.993
8	0.307	7.75	0.969	0.304	7.68	0.960
12	0.310	11.73	0.978	0.304	11.52	0.960
16	0.311	15.73	0.983	0.314	15.89	0.993
24	0.301	22.79	0.949	0.299	22.66	0.944
32	0.305	30.82	0.963	0.298	30.13	0.941
64	0.311	62.75	0.980	0.297	59.92	0.936
96	0.308	93.45	0.973	0.289	87.63	0.913

Table 6.6: Hardware and parallel efficiencies on 160 MHz T4 nodes, using the second latency hiding scheme. Scaled problem size is 576000 cells/node. Non–scaled problem size is 4×576000 cells.

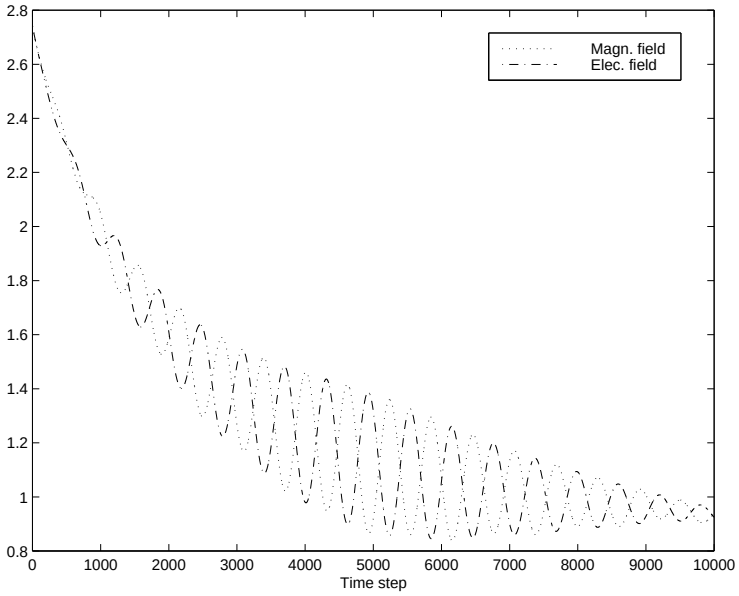

Figure 6.1: Total energy in system, model problem.

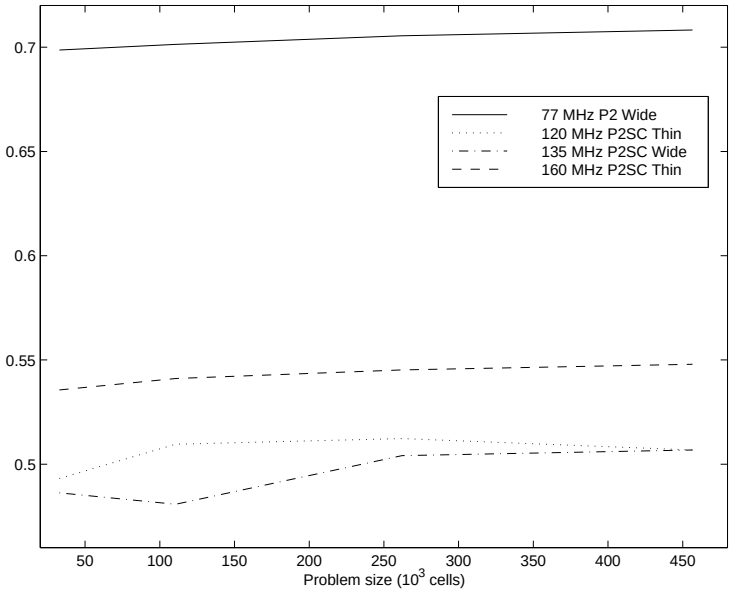

Figure 6.2: Efficiency of theoretical peak computing $R(u,t)$, Power2 processor family.

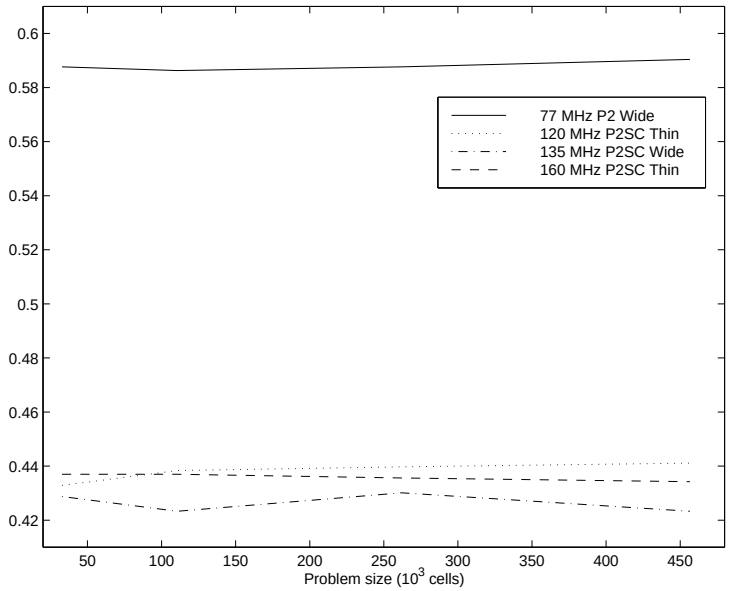

Figure 6.3: Efficiency of theoretical peak for a full time step, Power2 processor family.

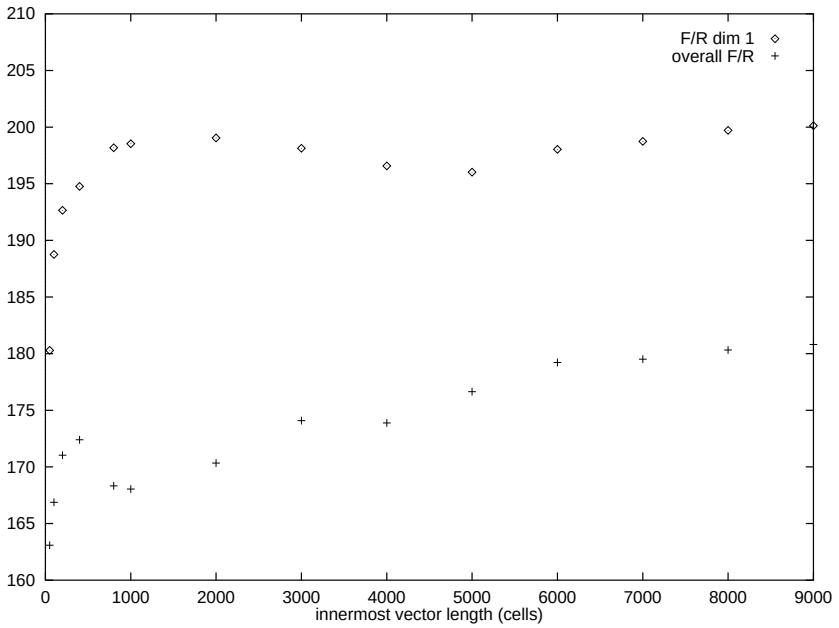

Figure 6.4: Scaled vector efficiency study, 135 MHz Wide nodes.

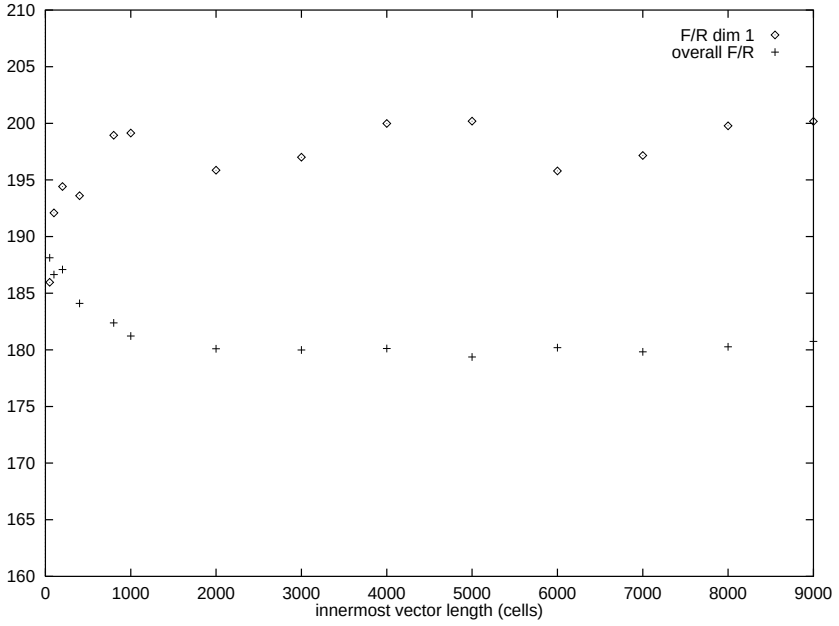

Figure 6.5: Non–scaled vector efficiency study, 135 MHz Wide nodes.

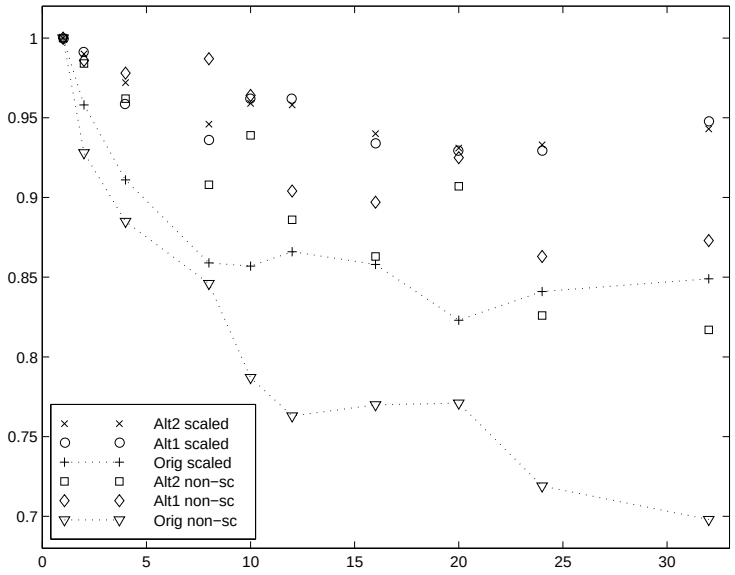

Figure 6.6: Parallel efficiency, 120 MHz Thin nodes.

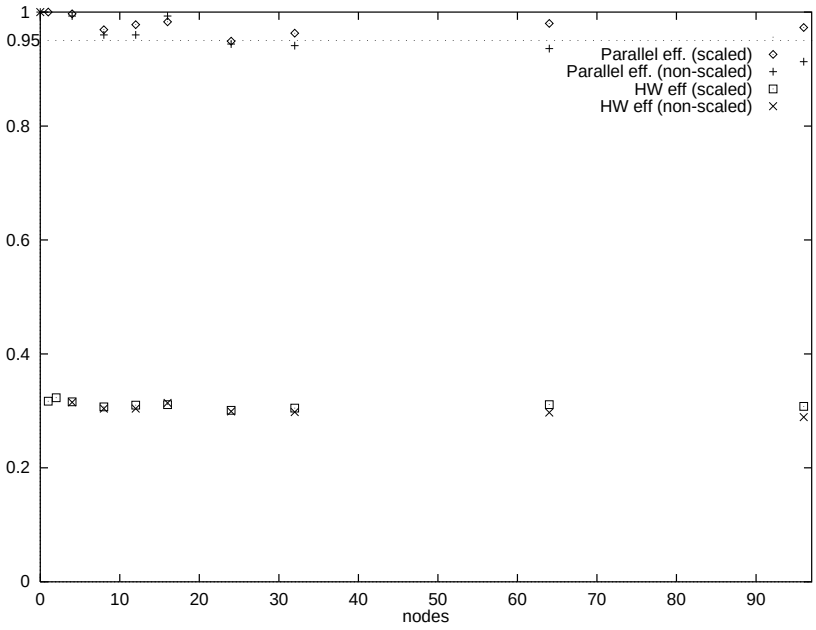

Figure 6.7: Hardware and parallel efficiencies on 160 MHz T4 nodes.

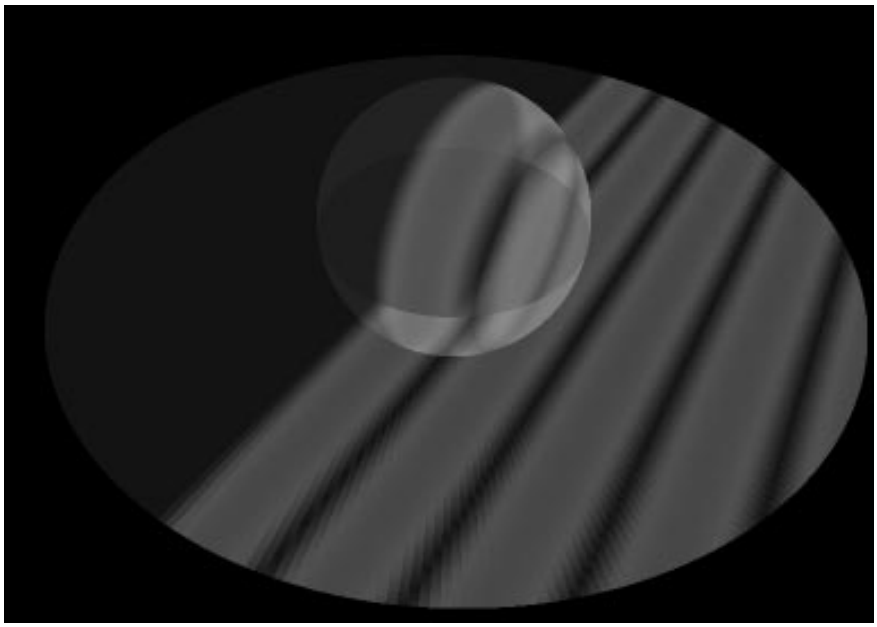

Figure 6.8: Model problem. Initial setup

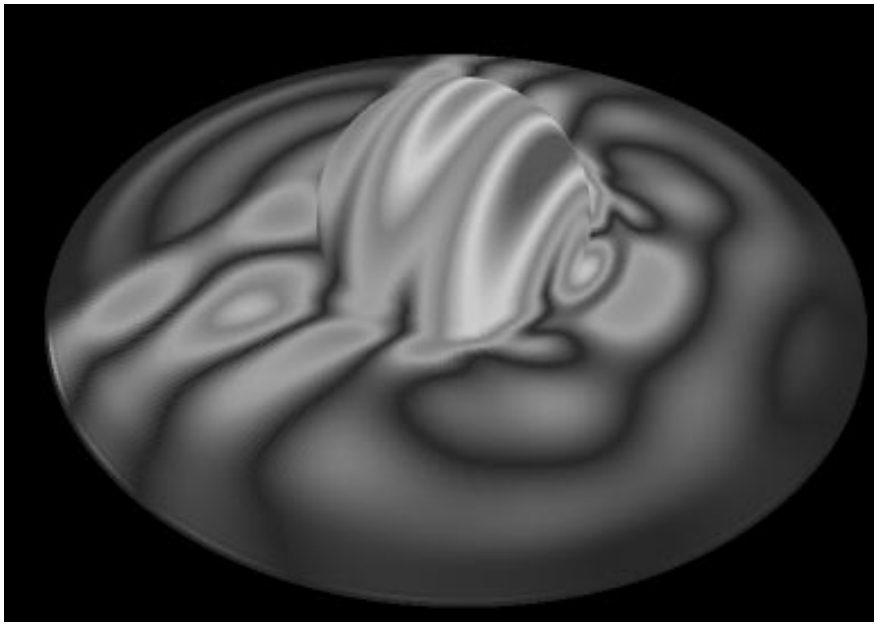

Figure 6.9: Model problem. Snapshot from near the end of a simulation.

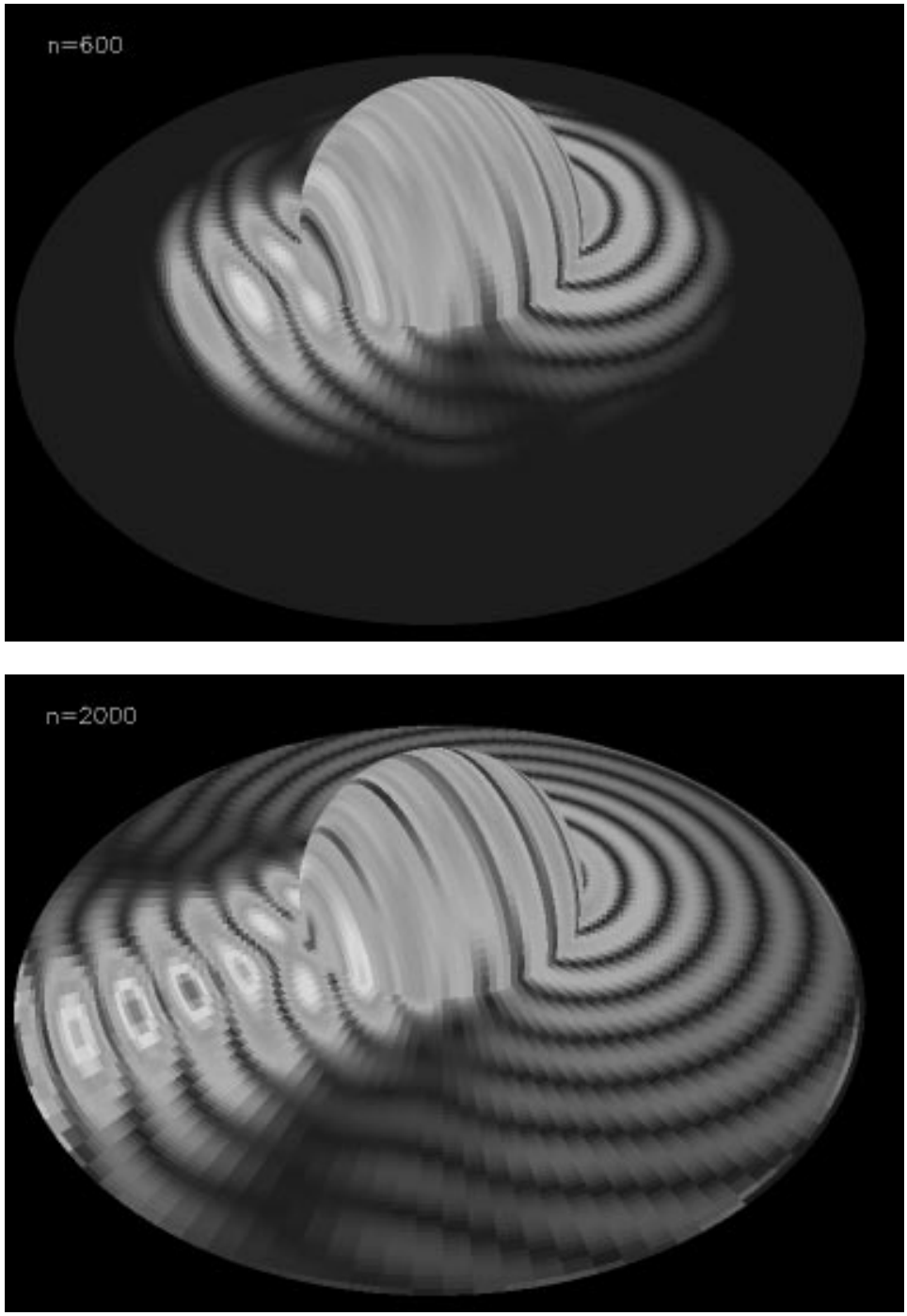

Figure 6.10: Model problem $64 \times 64 \times 96$, $\omega = 20$.

Chapter 7

Summary/Conclusions

In this chapter, we summarize our work and results, and make comparisons with related work.

7.1 Related work

The recent efforts and results of Shang et al. are described in [26], parallelizing the very same algorithm on the IBM SP and SGI Origin2000.

Shang reports performance in terms of DPR, *data processing rate*, or the time to compute one full time step per cell. A DPR of 1.764×10^{-5} sec is reported for a problem size of 420K cells on a wide P2SC node with full memory population. For the same problem size and hardware, our implementation achieves an average DPR of 1.219×10^{-5} sec, an improvement by 44.7%.

The parallelization of Shang's SP implementation degrades to about 70% parallel efficiency for 96 processors, while our implementation maintains an efficiency of 90%.

An interesting observation can be made from their SGI implementation. Shang

reports a highly tailored blocked implementation (parts of it is assembly language) and improved cache utilization techniques, reaching a peak processing rate on the SGI of 161 Mflops/sec, or 41% of hardware peak. This is a result that should be put in contrast to our estimated theoretical peak of 60% for a blocked implementation in multiprocessor mode. This yields an implementation efficiency of 66% of theoretical peak. We conclude that future work, implementing a blocking code on the SGI architecture, may not be particularly successful.

7.2 Summary

A high–order algorithm for computing the scattering of an electro–magnetic wave from geometrically complex objects was revised, optimized, and parallelized. Theoretical analysis was performed on the algorithm, estimating how well it would perform on two common parallel architectures.

A portable implementation was constructed, written in Fortran 90 with C extensions, using MPI as the communications package. Performance exceeding existing implementations by over 40% was achieved on the IBM SP. Our implementation also scales better, with a sustained 95% parallel efficiency using a latency hiding scheme, dimension reordering, and scaled problem size.

The architecture of SGI Origin 2000 requires blocking of loops due to the memory bandwidth constraints. We have shown that cache utilization and architecture–dependent tailoring must be incorporated in order to get acceptable performance.

Chapter 8

Future work

In this chapter, we briefly discuss how the work in this thesis may evolve from its present status.

8.1 Scattering computations

So far, only the solution for a model problem has been computed. Computing the scattering from other geometries like an airplane or missile is possible with the present implementation, given a partitioned grid. Time constraints and obtaining such a grid were the main reasons for not pursuing this here.

To investigate the true coordinate system independence of the code, some simulations involving Cartesian or cylindrical coordinates should be made.

8.2 Multiblock support

The current implementation is only capable of handling one three–dimensional block, which it partitions among the processors. Multiblock techniques would however improve the usability for computing the scattering from complex objects. For instance, a coarse grid over an aeroplane wing might be directly connected to a finer grid around the engine mounted on the wing. The different cell faces do not have to perfectly match each other – multiblock techniques will ensure the numerical stability in the interface zones.

8.3 Blocking memory code

For processor architectures with a need to recognize the limitations of memory hierarchies and bandwidth, such as the SGI Origin 2000 and other symmetric multiprocessing clusters, blocking of loops are necessary. In the future new IBM processors will support higher frequencies, but not necessarily higher memeory bandwidth, and it may be an essential task even for that architecture. Development of codes with additional loop nests and their parametrization for optimal output is an important future task until compilers and run–time systems offer such capabilities.

8.4 Distributed computing

Distributed computing is still a very new and rapidly evolving area. The early MPI support package we used with the GLobus toolkit had many limitations and was not in any way optimized.

We have shown that the collective communication primitives such as synchronization barriers and reduction operations may be significantly improved by instead perform the operations in clusters, minimizing the communication over the weak links between the different participating sites.

But, despite the deficiences, we have concluded that the algorithm is suitable for distributed computing environments.

References

[1] Shlomo Weiss and James E. Smith. *Power and PowerPC – principles, architecture, implementation.* Morgan Kaufmann, 1994.

[2] Cathy May, Ed Silha, Rick Simpson, and Hank Warren. *The PowerPC Architecture – a specification for a new family of RISC processors.* Morgan Kaufmann, 1994.

[3] MIPS Technologies Inc. R10000 Microprocessor Manual Version 2.0, December 1996.

[4] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. *Active Messages: a Mechanism for Integrated Communication and Computation.* Technical Report Report No. UCB/CSD 92/#675, Computer Science Division – EECS, University of California at Berkeley, March 1992.

[5] Chi-Chao Chang, Grzegorz Czajkowski, and Thorsten von Eicken. *Design and Performance of Active Messages on the IBM SP-2.* Technical report, Department of Computer Science, Cornell University, February 1996.

[6] Concurrent Systems Architecture Group. *Fast Messages (FM) 2.0 User Documentation.* Departement of Computer Science, UIUC, December 1996.

[7] Marc Snir, Stewe W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI — The Complete Reference.* MIT Press, Cambridge, MA, 1996.

[8] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine — A User's Guide and Tutorial for Networked Parallel Computing.* MIT Press, Cambride, MA, 1994.

[9] High Performance Fortran Forum. *HPF 1.1 Language Definition*, 1994.

[10] Todd H. Hubing. *Survey of Numerical Electromagnetic Modeling Techniques.* Technical Report TR 91-1-001.3, Dept. of Electrial Engineering, University of Missouri–Rolla, September 1991.

[11] Claes Johnson. *Numerical solution of partial differential equations by the finite element method.* Studentlitteratur, Lund, Sweden, 1987.

[12] Paul Soudais. Iterative Solution of a 3–D Scattering Problem from Arbitrary Shaped Multidielectric and Multiconducting Bodies. *IEEE Transactions on antennas and propagation*, 42(7):954–959, July 1994.

[13] James L. Schmitz. *Dual–Surface Integral Equation using FFT and Conjugate Gradient Methods.* Technical report, Hanscom AFB, MA, December 1994.

[14] Carl Nordling and Johnny Österman. *Physics Handbook.* Studentlitteratur, Lund, Sweden, 1987.

[15] Joseph S. Shang. Characteristics Based Methods for the Time-Domain Maxwell Equations. *29th Aerospace Sciences Meeting*, AIAA 91–0606, 1991.

[16] Joseph S. Shang. A Fractional–Step Method for Solving 3–D Time–Domain Maxwell Equations. *31th Aerospace Sciences Meeting*, AIAA 93–0461, 1993.

[17] Joseph S. Shang and Datta Gaitonde. Characteristic–Based, Time–Dependent Maxwell Equation Solvers on a General Curvilinear Frame. *AIAA Journal*, 33(3):491–498, 1995.

[18] Joseph S. Shang and Datta Gaitonde. Scattered Electromagnetic Field of a Re–entry Vehicle. *Journal of Spacecraft and Rockets*, 32(2):294–301, 1995.

[19] Joseph S. Shang, D.A. Calahan, and B. Vikstrom. Performance of a Finite Volume CEM Code on Multicomputers. *Computing Systems in Engineering*, 6(3):241–250, 1995.

[20] Joseph S. Shang and S.J. Scherr. Time–Domain Electromagnetic Scattering Simulations on Multicomputers. *26th AIAA Plasmadynamics and Lasers Conference*, AIAA 95–1966, 1995.

[21] Joseph S. Shang and Robert M. Fithen. *A Comparative Study of Characteristic–Based Algorithms for the Maxwell Equations. Journal of Computational Physics*, 125:378–394, 1996.

[22] William Gropp and Rusty Lusk. *Tuning MPI Applications for Peak Performance.* Tutorial given at SuperComputing '97, San José, CA, November 1997.

[23] The Plateau Research Group. *Berkely MPEG–1 Video Encoder.* Computer Science Division, UCB, November 1995.

[24] Eric Weisstein. *Eric's Treasure Trove of Physics.* To be published by CRC Press. WWW version obtainable at
`http://www.astro.virginia.edu/~eww6n/TreasureTroves.html`.

[25] Warren J. Wiscombe. Further Improvements in Mie Scattering Algorithms. *Appl. Opt.*, 1988.

[26] Joseph S. Shang, M. Wagner, Y. Pan, and D.C. Blake. Strategies for Time–Domain CEM Computations on Multicomputers. *36th Aerospace Sciences Meeting*, AIAA 98–0979, 1998.

[27] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. To appear in *International Journal of Supercomputer Applications.*

[28] M. Litzkow, M. Livney, and M. Mutka. Condor – a hunter of idle workstations. In *Proc. 8th Intl Conf. on Distributed Computing Systems*, pages 104–111, 1988.

[29] B. Clifford Neuman and Theodore Ts'o. *Kerberos: An Authenication Service for Computer Networks.* Technical Report Report No. ISI/RS-94-399, USC/ISI, 1994.

[30] A. Grimshaw and W. Wolf. Legion – a view from 50,000 feet. In *Proc. 5th IEEE Symp. on High Performance Distributed Computing*, pages 89–99, 1996.

[31] I. Foster, J. Geisler, W. Nickless, W. Smith, and S. Tuecke. Software Infrastructure for the I-WAY High-Performance Distributed Computing Experiment. In *Proc. 5th IEEE Symp. on High Performance Distributed Computing*, pages 562–570, 1996.

[32] I. Foster, C. Kesselman, and S. Tuecke. The Nexus Approach to Integrating Multithreading and Communication. *J. Parallel and Distributed Computing*, 37:70–82, 1996.

[33] Super Computing '97, November 1997. San José, CA.