A Parallel Programming Approach for Scientific Applications

A Dissertation Presented to the Faculty

of the

Department of Computer Science

University of Houston

In Partial Fulfillment of the Requirements for the

Degree Doctor of Philosophy

By

Terry W. Clark

December, 1996

A Parallel Programming Approach for Scientific Applications

Terry W. Clark

Approved:

Dr. L. Ridgway Scott, Chairman

Dr. Olin Johnson

Dr. S. Lennart Johnsson

Dr. Charles Koelbel

Dr. Jehan–François Pâris

Dean, College of Natural Sciences
and Mathematics

Acknowledgments

This dissertation is an outgrowth of the unique, interdisciplinary Texas Center for Advanced Molecular Computation (TCAMC). The TCAMC serves as a meeting ground for computer scientists, computational chemists, mathematicians, engineers, and physicists, all with the common goal to advance the capabilities of the computer simulation of "*Real world problems.*" This work and the TCAMC owe their existence in large part to my advisor, Ridgway Scott. Together with Olin Johnson, S. Lennart Johnsson, Chuck Koelbel, and Jehan–François Pâris, a committee at once both critical and supportive resulted. Andy McCammon continued to play an extraordinary role in this work, as he had in my work in the past. Bogdan Lesyng provided a unique opportunity to visit twice the Interdisciplinary Center for Mathematical and Computational Modeling at Warsaw University where I conducted the first IPfortran workshop.

I am fortunate to have had the opportunity to work with Reinhard von Hanxleden, a collaborator on much of the work here. (More than once Reinhard's power editing scorched deadlines while core dumping emacs editors.) Babak Bagheri was an early inspiration in the beginning of my work with massively parallel computers, an exercise I continue to this day. I am grateful to Rick Kufrin and Jim Lupo, both users and developers of the IPfortran compiler. Piotr Bala, another collaborator, showed me the true meaning of portability with his Quantum Classical Dynamics Code. In spite of our short time together at the University of Houston, Ernesto Gomez provided near record numbers of stimulating conversations per wall clock interval.

I acknowledge those institutions and the people making them what they are who supported my work in a variety of ways. These include the University of Houston Department of Computer Science, the Institute for Molecular Design, the Center for Research on Parallel Computation at Rice and Cal Tech, and the San Diego Supercomputing Center. Finally, I commend the Free Software Foundation and the Linux community who made available the software that converted my home computer into a machine capable of preparing this dissertation.

A Parallel Programming Approach for Scientific Applications

An Abstract of a Dissertation

Presented to the Faculty of the

Department of Computer Science

University of Houston

In Partial Fulfillment of the Requirements for the

Degree Doctor of Philosophy

By

Terry W. Clark

December, 1996

Abstract

In the 1980s the development of distributed memory parallel computers using replicated "off–the–shelf" components connected by a network gave computational scientists cost–effective alternatives to vector processing architectures. Scientific applications for the distributed memory computer are suited to the replicated architecture with processors operating on separate portions of the data set. Yet to harness these architectures has proven to be far more challenging than their vector processing predecessors. In an attempt to address the difficulty in engineering scientific applications for massively parallel computers, a variety of paradigms have been developed including parallel programming languages such as High Performance Fortran, standard interfaces such as MPI, and runtime systems such as PARTI. Parallel languages attempt to automate to the extent possible low–level details in decomposing a calculation, mapping it to the hardware, and satisfying the interprocess data dependencies. To automate these operations for many complex scientific applications is not within the grasp of existing compilers. Serial programs, also known as *dusty decks*, further test a compiler's capabilities.

To address the software engineering problem for parallel scientific programs, we developed the IPfortran language and compiler. This thesis discusses the design, development and testing of the IPfortran compiler. IPfortran grew out of an effort to ease the task of parallelizing complex scientific codes, especially pre-existing ones which are often poorly structured. IPfortran has been found to be very successful in parallelizing molecular dynamics, quantum dynamics and Brownian dynamics applications, providing a low–level, but machine independent paradigm for performing operations that routinely arise in parallel scientific programming. We discuss the molecular dynamics algorithms in general and in the specific context of IPfortran and Fortran D, a prototype for the HPF languages. We demonstrate that the complex interplay from high–level details of problem decomposition down to the low–level details in machine mapping are addressable with IPfortran. Through our experiences in parallelizing some of the molecular dynamics algorithms with Fortran D, we provide guidelines for structuring Fortran programs to facilitate parallelizing them with HPF–like languages.

Contents

List of Figures

Chapter 1

Introduction

In engineering a technical application for parallel execution one confronts a series of common issues. First, one decides upon a model for the problem at hand. An atmospheric model may contain components that include the fluid–like flow of gases in addition to molecular details that model the formation of new chemical species. A kinetic model for a drug substrate model may contain details about the polarization of atoms, or may take a coarser approach with charge distributions fixed. The detail one incorporates into the model depends on the questions of interest, available theory, and available computing resources.

With a model in hand, one must determine parallel algorithms to implement. The choice of the algorithm can be influenced by the target architecture. Often interprocessor communication plays a critical role in the viability of an approach. A low latency intercommunication network could favor a fine granularity with frequent data exchange. On the other hand, with a high latency data exchange fewer messages with more buffering may be required to get the job done efficiently. In tandem with choosing an algorithm one must make the choice as to how the parallel components are mapped to the target architecture.

There is interaction from the model down to mapping the problem to the hardware that direct the approach to take. In addition, the method available to write the software can influence choices in the process. Furthermore, the anticipated use of a parallel application can influence the programming paradigm of choice. If portability is not an issue, one may use low–level, machine specific primitives for coordinating interprocessor data exchange. On the other hand, if portability is important, a high level, general purpose language may be desirable. The methodology of choice for parallelizing an application is often difficult to determine.

1.1 Software engineering and parallelizing scientific applications

In the 1980s the development of distributed–memory parallel computers using replicated "off–the–shelf" components connected by a network gave computational scientists cost–effective alternatives to vector–processing architectures [95, 57, 62, 94]. Scientific applications for the distributed–memory computer were found to be well adapted to a replicated architecture where a processor operates on a portion of the data set. The programs execute in an address space accessible only by a single processor, exchanging data with other processors using messages exchanged through the interconnection network. The program style is often loosely referred to as message–passing, distributed–memory or the single–program multiple data application (SPMD) approach. Many of the methods and issues applicable to distributed memory architectures apply to shared memory architectures. This thesis will focus, however, on the distributed memory paradigms, especially in the area of interprocessor–communication analysis.

This thesis discusses the design, development and testing of the IPfortran language and compiler. IPfortran grew out of an effort to ease the task of parallelizing complex scientific codes, especially pre-existing ones often poorly structured. A thorough testing of IPfortran addresses issues common to software engineering including software design, prototyping, data abstraction, portability, validation, efficiency, and maintenance.

1.1.1 Target applications

IPfortran targets pre-existing technical applications, as well as parallel applications written from scratch. Due to the complexity of modeling of scientific and engineering phenomena, programs are routinely tens of thousands of lines in length and have been developed by more than one individual over several to many years. It is not uncommon for the application to develop simultaneously with the theory implemented there, making for less than ideal software. Furthermore, it is natural that applications are often written by non-computer scientists programming out of necessity, mainly concerned with correctness. It sometimes happens that efficiency is left as an after thought. Sequential prototypes become production codes that are used for a decade or more. The *dusty deck* is born.

Many phenomena require a program structure with irregular data accesses, which pose problems of their own, "dusty decks" aside. From an operational point of view, irregular accesses are indeterminate at compile time. Irregular accesses are not uncommon in scientific applications. In a survey by Shen, Li and Yew, the linearity of subscripts in six numerical packages using the Paraphrase system were examined [97]:

- Linpack: linear system package (10175 lines)
- Eispack: eigensystem package (11700 lines)
- Nasa: a program from NASA (780 lines)
- Baro: shallow water atmospheric model (1052 lines)
- Itpack: sparse matrix solvers using iterative methods (5591 lines)
- Old: Checon Chebyshev economization program (3218 lines).

They found that without the help of user assertions in performing the dependence analysis, 47% of the one-dimensional array references and 45% of the two-dimensional array references were nonlinear. Using user assertions, 28% of the one-dimensional array references and 15% of the two-dimensional array references remained nonlinear.

Irregular accesses are difficult for automatic compilation techniques, typically handled with a combination of compiler and runtime support. Automatic compilation techniques typically focus on regular accesses [22, 108, 3]. Kremer and Kennedy develop a framework that automates data layout in a pre-processing phase for an HPF compiler. Their heuristics solve several NP-complete problems but do not address irregular problems [64, 67]. Automatic compiler support aside, even with annotated languages such as Kali and Fortran D, irregular accesses are difficult to implement efficiently. In studies done with Kali it has been found that the inspector introduces a substantial overhead to the program [66]. The reuse of schedules can ameliorate this overhead making the scheme viable. However, it may not be possible for the runtime system to determine efficient communication schedules for a given processor mapping.

The GROMOS molecular dynamics application is a "dusty deck" with numerous irregular accesses [47]. While a respected and well-organized program, it was written in FORTRAN4, was developed at roughly the same time as the theory, has been modified by a number of individuals, and was likely not intended to be an internationally renowned molecular dynamics code at the outset. Modifications to the code addressing vector processing architectures were made by others, after GROMOS was developed by the principal author; it is doubtful that parallelization was foremost the author's mind. Yet GROMOS was choosen as the starting point for the spatial decomposition EULERGROMOS, which to our knowledge is the first load–balanced spatial decomposition of a biomolecular dynamics application.

In the early 1990s, most popular molecular dynamics programs were of the same vintage as GROMOS.[1] GROMOS was attractive due to its size (the molecular dynamics module is about 10,000 lines), good overall structure, sound science, and a relatively large audience interested in performing computationally demanding simulations with it. To develop and test from scratch a molecular dynamics program for simulating general biomolecular systems would take an expert several years. We are convinced that many scientific applications fit this mold.[2]

The inherent complexity of scientific applications causes problems with automatic compilers and encourages program reuse to avoid re-writing and re-testing. Since "dusty decks" are developed for sequential computers, the algorithms and data structures may not be suitable for an efficient parallel implementation. The parallel programming paradigm suitable for "dusty decks" must deal with unstructured programs, as well as possess the flexibility to deal with structural features inherent to complex technical programs, such as irregular accesses.

While program structure is a nebulous concept, the reducibility of flow graphs is one generous measure. In [1] Aho and Ullman write: *"An equally important fact about reducible flow graphs is that they occur frequently in practice. Allen and Knuth found that in random samples of FORTRAN programs virtually all programs had reducible flow graphs. Moreover, exclusive use of structured flow-of-control statements ... produces programs whose flow graphs are always reducible. These observations indicate that reducible flow graphs occur naturally, even in a language like FORTRAN, which virtually mandates the use of goto's, and even if a programmer has no prior knowledge of structured program design."* During the parallelization described in Chapter 12, the Fortran D compiler encountered an irreducible loop in GROMOS.

1.1.2 Parallel programming paradigms

Parallel software development must take into account a flux in architectures and programming methodologies. For a parallel computer to remain on the market for more

[1] These include CHARMM and AMBER.
[2] In Section 1.5 we introduce the parallelization of the *dusty deck* UHBD.

than five years is exceptional. Moreover, a commercially viable parallel computer will provide a compiler and paradigm for porting applications to the machine. These paradigms can be radically different, and often, machine specific. Consider, for example, the shared address–space, threaded paradigm provided by the KSR-1 and the distributed–address–space paradigm characteristic of the Intel massively parallel computers. An application that uses KSR-1 tiles will need to be completely re-annotated, if not rewritten, for Intel MPPs. It is likely that fundamental data structures in the KSR-1 developed application will require modification just to use the Intel processors, saying nothing about efficiency.

Disparate architectures and programming paradigms can be successfully addressed in several ways. One solution is to use a machine independent, high–level language which is input to a compiler that produces efficient, machine–dependent code. The SUIF compiler system translates Fortran and C input into an intermediate form where compiler analysis looks for loop level parallelism [108]. The SUIF group reports successful compilation and validation of the Perfect Club benchmarks, SPEC92 Fortran floating–point applications, and NAS benchmarks. The details of the parallelization are not reported in [108], however, it is unlikely that irregular accesses are implemented in parallel.

SUIF achieves portability through high–level languages and compiler detected parallelism. As we argued above, however, a "dusty deck" is notorious for poor program structure that complicates compiler analysis. Furthermore, irregular accesses thwart compile time analysis. It remains to be seen whether compiler–only analysis is suitable for complex, production programs. That HPF incorporates a number of user supplied annotations reflects the difficultly inherent to compiling a sequential program without annotations for a parallel computer.

Short of the first solution, an automatic parallelizing compiler, we can consider a language that allows the explicit expression of parallel algorithms in a machine independent fashion, while permitting (or requiring) the programmer to manage details in distributing the data and iterations (or workload). This is IPfortran's approach.[3]

Beyond getting a program through the compiler and producing correct code, we require that the compiler produce efficient code. Numerous chapters of this thesis are concerned with the expressibility of efficient parallel algorithms that are portable. We are very interested in realizing efficient algorithms for the obvious reason that efficient

[3] HPF, on the other hand, lies somewhere between automatic compilers such as SUIF and the manual style of IPfortran.

algorithms best utilize resources. The replicated algorithm in Chapter 8 and spatial decomposition in Chapter 9 are two extensive case studies where Pfortran can both express the parallel algorithms and produce programs with performance within our theoretical expectations.

1.1.3 The software life cycle

Maintainability is a significant consideration in the life cycle of any production–grade application. A thorough test of a parallel programming methodology will follow the software through the software development cycle. Commonly, incremental changes are made to the software throughout its existence to meet with changing user requirements. Parallel development adds another dimension to software modification. In our molecular dynamics case studies, we apply an incremental development process whereby we address those parts of the program where parallelization yields the greatest benefits. However, Amdahl's law rings true, and once insignificant portions of a program often become the limiting factor to speedup. A good, but benign, example of this is the bonded–interaction calculation in the replicated algorithm. Relative to the nonbonded interaction calculation cost, they are minor. However, after parallelization of the nonbonded force calculation, they are an asymptote which total performance cannot cross (Figure 8.7), thereby requiring parallelization for further performance enhancement.

When in a program's life cycle the parallelization of the current factor limiting speedup will take place depends on the use of the program and the type and number of resources available. For example consider *subdomain redistribution*, the redistribution of atoms in the spatial decomposition. EulerGromos uses a point-to-point algorithm for subdomain redistribution, rather than the more sophisticated *shift* algorithm (Chapter 9). For the acetylcholinesterase simulation on an Intel Paragon, the asymptotically exponential growth of point-to-point does not pose a scaling problem until after 512 processors (Figure 9.5), a number of processors outside what is *currently* available to most researchers. It would be advantageous to replace the point-to-point communication with the more efficient shift algorithm should more than 512 processors become available, or when processor speeds increase.

It follows that parallelization raises the stakes for cost–effective incremental software development. The parallel development methodology described in this section, while underscoring the need for a paradigm receptive to incremental development, also highlights the fact that low–cost program segments, easily swept under the rug by a cursory parallelization, can be critical to scalable performance. To use a specific example, that a paradigm can effectively be used to parallelize the nonbonded force loop for the replicated algorithm (Chapter 12), is one small measure of success. But, can the paradigm be used to load balance the force and pair list calculation, or be used to implement an efficient spatial decomposition?

We have followed the development cycle of the parallel molecular dynamics application threaded through this work alongside computational chemists applying the program to scientifically relevant biochemical systems. This work has been conducted over a period encompassing non-trivial machine changes from the Stellar GS1000 to the Intel Paragon, along with problem sizes ranging from 6,000 to 130,000 atoms, more than a factor of twenty.

1.1.4 Data abstraction

The source of IPfortran's expressibility contributes to its shortcoming: the programmer must write the logic to distribute data and iterations. This is the nature of programming in a local name space. However, IPfortran automates data and iteration distribution for data combined with reduction operations.

Abstract data typing associates an operation with a data type. The IPfortran language extends FORTRAN77 with two operators and associated data types: the @ and the associated process source or destination, and the distributed data type forming an Abelian semi-group with respect to the the binary reduction operation.[4] These data abstractions allow the programmer to clearly express operations common to parallel applications. In the EulerGromos parallelization the occurrence of IPfortran operators is numerous (Chapter 9). The reduction and @ operators aid in developing and reasoning about a parallel program, facilitating prototyping and porting efforts. Furthermore, the compiler can easily identify reduction operations and implement efficient algorithms for them.

1.1.5 Software engineering perspective summary

To test rigorously the software engineering aspects of a language is an extensive undertaking, critical to do in-house. It is difficult to draw broad conclusions based on one

[4] IPfortran could equally be described as an extension of Fortran 90, particularly with its use of Fortran 90 array notation. However, at the time of the IPfortran compiler development, Fortran 90 compilers were not available.

application. However, that the IPfortran language and compiler were proven effective for a production–level, irregular program with numerous users across several platforms, is strong evidence that the concept would apply to other "dusty decks." To undertake two or three more such studies would be prohibitive, although other researchers use IPfortran for equally challenging parallel application development [9, 74, 75, 10].

1.2 I/O

The cost of performing output during a molecular dynamics calculation can be daunting. To output the entire trajectory, that is, the phase space coordinates of the entire system at every time step, is usually not possible. At the outset, I/O was not a concern. As we parallelized the computationally dominant portions of the program, though, outputting costs became significant. It turns out that the spatially–decomposed molecular–dynamics parallelization is at odds with trajectory outputting.

In Chapter 11 we examine the trajectory outputting issues and costs for simulating a currently grand–challenge sized system, the acetylcholinesterase dimer. These costs are framed relative to the other large parallel overhead, communication costs. We discuss a method we developed for reducing the volume of output data by allowing the simulationist to selectively discard atoms. Scalability is examined assuming parallel I/O systems. In Chapter 3 we discuss implementation of several I/O strategies with IPfortran that are portable.

The automatic data decomposition developed by Kremer, et al., for regular problems also addresses I/O [67]. However, the strategy there is as good as the machine performance model: I/O subsystems are extremely difficult to model. I/O was not addressed in HPF-1, but it is considered for inclusion in HPF-2 in order to provide scalable access to secondary storage [40]. In agreement with our studies, several basic requirements are flagged for HPF-2 including: transparent parallel access to files, the ability to checkpoint and restart jobs, and the ability to stripe files over disks.

1.3 Properties of a molecular dynamics simulation

A molecular dynamics simulation is considered a chaotic process. In other words, small perturbations on the order of roundoff error can send the simulation into different parts of phase space. This makes it difficult to determine program correctness. An additional source of complexity stems from the structure of biological systems that are the focus of the algorithms we discuss in this work. Unlike a typical solid state molecular dynamics simulation, a biological system can consist of numerous types of molecules including solvent, solute molecules, and various ions. These features add to the complexity of the program.

Molecular dynamics simulates the motion of a collection of atoms using Newton's classical equations of motion. Atoms are modeled as soft spheres with a uniform electron density, implicitly taking into account the electron wave functions comprising the electron density. At the heart of the molecular dynamics simulation is the force field, or Hamiltonian. This analytic function of the system's coordinates is used to determine the forces acting on each atom. A typical potential energy function has the form

$$\begin{aligned} V = \quad & \tfrac{1}{2}\textstyle\sum_{bonds} & K_b(b - b_o)^2 \; + \\ & \tfrac{1}{2}\textstyle\sum_{angles} & K_\theta(\theta - \theta_o)^2 \; + \\ & \tfrac{1}{2}\textstyle\sum_{dihedrals} & K_\phi(1 + \cos(n\phi - \delta)) + \\ & \textstyle\sum_{nonbondedpairs} & \left[\frac{A}{r^{12}} - \frac{C}{r^6} + \frac{q_1 q_2}{Dr}\right]. \end{aligned}$$

The independent variables are typically derived from the Cartesian coordinates of the atoms. A function can be derived for the force as the analytical derivative of the potential. The force field contains a number of parameters, for example, the Lennard–Jones parameters A and C. These give atoms their characteristic interaction properties and are determined by various experimental and computational techniques. The first three terms sum over the bonded interactions. The computational load of these terms is insignificant relative to the fourth term over *nonbonded pairs*. Furthermore, parallelization of the bonded terms is trivial for both the replicated and spatial decomposition. The extent of the nonbonded interactions is determined by a cutoff radius. The spherical neighborhood containing the nonbonded interactions for each atom provide a spatial locality that is exploited by the spatial decomposition.

Under biological conditions the density of a solvated protein is about $12\text{Å}^3/atom$, with the protein more densely packed than the solvent. An average simulation involves about 10,000 atoms in a roughly $50 \times 50 \times 50\text{Å}^3$ box; a simulation with 100,000 atoms or more is considered very large by today's standards. Typical cutoff radii range from 8Å to 20Å [73]. Typical time scales for a simulation range from about 1 picosecond to 1000 picoseconds performed with time steps of 1 to 5 femtoseconds. Longer simulations provide better statistics for analysis. A 1000 picosecond simulation is considered to be quite long.

Over 90% of the computational workload is spent in the nonbonded interaction calculation. A pair list, which is updated every 10 time steps or so, is typically used to keep track of the nonbonded interactions. The number of floating–point operations per pairwise interaction in a GROMOS nonbonded force routine are: 18 multiplies, 8 divides, 22 adds, 10 subtractions, and 2 exponentials.[5] Additional operations result from periodic boundary conditions. On a single–processor SGI with a MIPS R8000 CPU and R8010 FPU, it takes about 7sec/timestep to simulate an 11,000 atom myoglobin system, with pair list updates at 10 time–step intervals using the molecular dynamics program ARGOS.[6]

A trajectory consists of a series of points in $6N$–dimensioned phase space, where N is the number of atoms in the system. One point consists of the the Cartesian coordinates for position and velocity. The frequency these data are output depends on the simulation goals.

1.4 Measuring the speedup of a molecular dynamics simulation

We take this opportunity to discuss speedup measurements of molecular dynamics. There are many parallelizations of molecular dynamics in the literature (see section 1.5. In addition, molecular dynamics is one of a number of popular applications used to report speedups for evaluating parallel architecture design [88], parallel languages [86, 88].

The usual caveats of slow processors or poor algorithms hiding interprocess data exchange costs, molecular dynamics calculations have other, unique features, that could effect the perceived quality of a molecular dynamics program parallelization, or the performance of a language or architecture that uses molecular dynamics as a benchmark.

The relative workload in a molecular dynamics calculation can depend on

1. pair list update frequency,
2. trajectory output frequency,
3. cutoff radius used,

[5] These counts ignore the cost of the pair list construction and numerous pointer arithmetic.
[6] ARGOS is one of the better molecular dynamics programs for pipelined architectures [100].

4. whether long–range force calculated,
5. whether shake is implemented.

The pair list update frequency can have a time complexity ranging from $O(N \log N)$ to $O(N^2)$. Typically a pair list is used to reduce the frequency of the updates [105]. The total workload will depend on the frequency of the pair list update. With the typical update frequency of ten–step intervals, the $O(N)$ force calculation is usually greater than the pair list calculation.

The nonbonded force work load goes as R_c^3, for cutoff radius, R_c. A typical simulation will use a cutoff radius of ranging from 8 to 15 Angstroms [73]. In some simulations a long range force is calculated beyond R_c. This force is sometimes calculated during the pair list calculation, a common practice to reduce the overhead of finding the pairs; the long–range force is then reused until the next pair list update.[7]

Finally, shake is a method used to constrain bonds to a fixed length thereby permitting a time step up to five times greater than that used without shake (see Chapter 8). Shake is expensive to parallelize for distributed algorithms due to the many interprocess dependencies arising from bonds crossing domains. While the parallel efficiency of a parallelized molecular dynamics program will likely be greater without shake than with it, in practical terms, a non-constrained simulation can produce less work per wall–clock time unit than a constrained one.[8]

These factors are important to keep in mind when considering performance results, particular those that report only *total time*. In addition, because of the variability in the simulation parameters and parallelization approaches, one should take care in comparing different molecular dynamics parallelizations.

1.5 Related work

1.5.1 Languages

ℙfortran was developed contemporaneously with numerous other parallel language efforts including [66, 54, 11, 23, 87]. The goals of these languages include portability and minimizing the parallelization effort. Paralation Lisp formulates "reductions" as combining operations [91] which are not necessarily associative. Viswanathan and Larus

[7] The pair list calculation in Figure 8.7 includes a long range force calculation.

[8] Some simulations remove or include constraints for scientific purposes. We address the cases here where constraints are used for performance enhancement.

describe a recent implementation of user–defined reduction operations in C** [106]. We briefly discuss several other influential languages.

Kali

Kali provided a global-name space for distributed-memory programs with compiler annotations specifying the data distribution and iteration distribution. Kali generates the *node code* with communication using a few basic distributions [66]. Kali was one of the first systems to incorporate both compile-time and run-time message generation in a single system. Kali generates compile-time communication sets for certain computations with only regular data access[9] thereby satisfying interprocessor data dependencies through information gathered from

- data distributions from the *dist* annotation
- loop iteration placement from the *on* clause
- data dependencies from the index function *f*.

Using this information, the Kali compiler writer derives theorems that specify the communication patterns in terms of

- send sets
- receive sets
- local iterations
- nonlocal iterations.

These sets are constructed at compile time with the necessary communication placed in the program. Irregular computations use the inspector and executor paradigm.

Fortran D

Fortran D derives node programs based on data decompositions given by annotations [54]. In Fortran D arrays are *aligned* with virtual *decompositions* which are *distributed* across processors. Fortran D supports both regular and irregular distributions. (Fortran D is discussed in more detail in Chapter 8 and in Chapter 12.)

[9] Operationally, *regular* refers to those data dependences determinable at compile time. Shen, et al., give a definition based on linearity of subscripts [97].

1.5.2 Compiler targets

Parti

A number of paradigms are targets for parallel languages. PARTI (Parallel Automated Runtime Toolkit at ICASE) encompasses a set of primitives that carry out the distribution and retrieval of globally indexed but irregularly distributed data [92, 12]. PARTI implements a runtime preprocessing phase that partitions work, maps data structures and schedules the movement of data between processor memories. PARTI, Fortran D and Kali use the *inspector* and *executor* for handling the irregular distribution.

During the preprocessing, or inspector stage, the interprocessor communications that are carried out allow the determination of all sends and receives that must take place. A hash table is used to keep track of nonlocal fetches and stores. With the *hashed cache*, multiple references to the same datum can be recognized and multiple transmissions avoided [110].

MPI

MPI provides a standard interface for point–to–point communication and communication between groups of processors [41]. The programming model is SPMD. Message tags are used to order messages. Collective communication support includes operations for a number of scenarios, including

1. barrier synchronization
2. one–to–all broadcast
3. one–to–many broadcast

The collective communication routines require that all processes in a group invoke the communication call with matching arguments.

A key concept behind MPI communication is that of *communicators*. a communicator defines a group of processes that may communicate, and contexts within the communicator; a context is a partition of the communication space. A communicator is formed from a given communicator for a group of processes, potentially creating a subgroup of processes. The application must take care that communicator calls are correctly placed to avoid deadlock.

To write MPI message–passing logic by hand can be be a tedious and error prone task. However, MPI's popularity as a message-passing interface along with its rich set of primitives makes it a viable target for compilers.

Global array tools

Nieplocha, et al., have developed the *Global Array* paradigm which allows processes to access distributed data asynchronously, through global arrays [82]. The environment is distributed where the only shared data is that provided by the global array. The authors cite two assumptions motivating their work:

1. Most high performance parallel computers will continue to have physically distributed memories with non-uniform memory access, and will work best with programs that have a high degree of locality.

2. Programming such machines will require extra effort.

The goal of global arrays is to reduce that effort particularly for parallel programs [82]

- requiring task parallelism, in addition to data parallelism,

- accessing relatively small blocks of matrices that are too large to hold in the memory of any single processor,

- with wide variation in task execution time with attendant unpredictable data reference patterns,

- having a large ratio of computation to data movement, making it possible to retain high efficiency while accessing remote data on demand.

Asynchronous access is permitted by creating an array as a *global array*; data coherency is left to the programmer. Processes can then access the array with a set of primitives, e.g., *ga_put* and *ga_get*. The *get* and *put* operations are defined in terms of copying data to and from a local buffer. The application may avoid the copy cost with the *access* function. A synchronization operation is provided, facilitating user devised coherency. In addition, a send and receive message–passing interface is provided.

The communication requirements of molecular dynamics are regular, in spite of the application being irregular. Thus the global array paradigm, while it can effectively cope with the task parallelism of the *Dining Philosophers Problem* (see Chapter 3), it is not required for molecular dynamics.

1.5.3 The mapping problem and collective communication

Several of the molecular–dynamics papers mentioned above touch on the mapping problem for the Intel hypercube. The problem of mapping an algorithm to a topology is in general NP-complete [14]. Some methods for solving this problem involve minimizing a cost function using greedy heuristics [70] or simulated annealing [81]. Other methods include applying the recursive spectral bisection method to partition and map finite element grids [60].

Dikaiakos, et al., apply several clustering heuristics to the N–body problem, formulated as a fast multipole and Barnes Hut algorithm, for message–passing multiprocessors [35]. The N–body problem differs fundamentally from the nearest–neighbor approach since with the N–body problem a particle interacts with the entire system. They conclude that more practical heuristics are needed that take into account communication costs.

Collective communication takes advantage of structured data exchanges among processors in order to minimize contention, the number of data transferred, and the number and rate of messages into and out of a node. Collective communication algorithms in the literature are numerous. Early work in this area includes [61, 90]. Other work includes [43, 103]. The algorithms typically take into account the underlying interconnection network topology. However, some collective communication algorithms can derive benefits over scatter–gather–like point-to-point algorithms irrespective of topological considerations.

1.5.4 Molecular dynamics

Parallel molecular dynamics algorithms

In this section we overview a representative sample of the the numerous reports of molecular dynamics parallelizations in the literature. Fincham gives an early overview of issues in methods in [39]. There the focus is primarily on all–pairs interactions, with a number of discussions on methods to organize the data exchange. Smith presents the replicated algorithm (possibly the term was coined here), the link–cell method and a systolic algorithm in [99]. Smith analyzes the algorithms using a ratio of communication to computation. Actual simulations and parallelizations are not presented. Pinches, et al., give a detailed analysis of the "link–cell" method, a form of the spatial decomposition [83]. Along with Smith, they discuss the so–called "north–south–east–

west" exchange algorithm.[10] They use a master–slave model on the iPSC/2, and also discuss results for a Cray Y-MP. The system simulated is an unspecified Lennard–Jones liquid. Much attention is given to analyzing the saw tooth effect resulting from the cutoff radius not dividing the link–cell processor domains.

Many molecular dynamics parallelizations in the literature up through the early 1990s were for simple systems such as Lennard–Jones (LJ) liquids or gases, or for solid–state systems. Our work for the replicated algorithm and the spatial decomposition are among the earliest parallelizations of molecular dynamics programs designed for biological molecules. These programs are more complex in structure than LJ and solid–state systems. Another work addressing biological simulations is that of Esselink, et al. There a geometric decomposition is used where the subdomain size restriction of earlier work (e.g., [83] has been lifted [38]. Unlike our work, they assume a homogeneous distribution of particles with equally sized processor subdomains. Other parallelizations include the replicated algorithm for the molecular dynamics program CHARMM [18, 19] and AMBER [93, 32]. Plimpton presents a comprehensive study of the replicated algorithm, spatial decomposition and a new force–assignment algorithm in [84]. He presents data from the simulation of a LJ fluid on a wide range of computers including a Cray C90, Cray T3D, Intel hypercube and Intel Paragon. His complexity analysis is presented in terms of numbers of operations for communication and computation as functions of the number of atoms, N, and number of processors, P.

Constraints in molecular dynamics

Constraints are a problem in parallelizing a molecular dynamics program due to the dependencies that need be satisfied during the iteration process (See Chapter 9 for more details about molecular dynamics with constraints.) The general method is often referred to as shake [89]. An early parallelization of shake by Plathe and Brown uses a coloring scheme to assign segment of the molecule(s) to processors such that there are not interprocessor dependencies [80]. DeBolt and Kollman parallelized the SHAKE algorithm for the replicated algorithm [32]; this was possibly the first parallelization of shake reported in the literature where interprocess dependences were allowed to persist between processors. In earlier parallelizations shake was performed redundantly at each processor along with the integration step.

[10] A form of what we call the *shift* algorithm.

Sequential algorithms for molecular dynamics

Bentley developed a method for reducing the $\mathcal{O}(N^2)$ nearest neighbor pairlist calculation to $\mathcal{O}(N \log N) + \mathcal{O}(N)$ from the recursion $T(N) = 2(T(N/2) + \mathcal{O}(N)$. This approach is reminiscent of the spatial decomposition complexities (see Chapter 9). Bentley's analysis relies on an adequate number of divide and conquer steps, partitioning the system into units of the cutoff radius. It turns out that with typical biomolecular systems, the number of recursions would range from one to four, given simulation box and cutoff–radius sizes (Section 1.3).

Boris devised a scheme related to Bentley's work called the Monotonic Lagrangian Grid (MLG) to reduce the complexity of extracting nearest neighbors. The scope of the MLG as presented in [15, 69] is vector–processing architectures. A template is used to extract nearest neighbors from the grid; there is some problem in selecting the correct template to gather the collection of atoms within a fixed cutoff. Boris analyzes in detail the statistical problem associated with extracting nearest neighbors from the MLG.

1.5.5 Other applications ported with IPfortran

This dissertation presents an extended study of the parallelization of the molecular dynamics application GROMOS. In this section we discuss other parallelizations performed using IPfortran.

UHBD

The molecular dynamics method is typically used to simulate molecular motion using time steps on the order of femtoseconds, characterizing the event time scale that can be practically observed. Brownian dynamics is a complementary method with longer time steps and a continuum solvent model [59]. The Brownian dynamics calculation consists of two principal parts: the calculation of the electrostatic field using Poisson's equation and the trajectory calculation for the diffusion of particles (or molecules) in the electrostatic field. The latter is trivially parallel.

Ilin, et al., have parallelized both phases of UHBD using IPfortran. They report supercomputer–level performance on modest numbers of processors, and identify techniques to use hundreds of processors effectively [59]. The most severe constraint in using distributed shared memory computers in their parallelization is memory limitations. In addition to Brownian dynamics simulations, UHBD is used to perform electrostatic calculations, resulting in multiple versions of the application, tailored for specific applications.

QCMD

Classical molecular dynamics methods lack a description of the quantum mechanical phenomena required to model the making and breaking of bonds, and electron and proton transfers. The inclusion of a quantum dynamical description into these models has been deterred in part by the extremely large computational requirements [10]. For biological molecules, a portion of the system is treated quantum dynamically, with the bulk treated classically. In this manner the parallel Quantum-Classical Molecular Dynamics code (QCMD) was developed as a derivative of UHGromos (Chapter 8) in a straight forward fashion that reused the pairlist and load balancing code to manage the various interactions arising in the application.[11] This successful derivative work further illustrates the "real–world" character of the application class targeted by IPfortran.

1.6 Overview

In this work we demonstrate that the complex interplay from high–level details of problem decomposition down to the low–level details in machine mapping are addressable with IPfortran. We discuss IPfortran in detail in Chapter 3. In Chapter 4, the IPfortran programming model is applied to a variety of kernels ranging from simple *baton passing* to a *Gaussian elimination* program. We explore the limitations of IPfortran for expressing task parallelism in a solution to the *dining philosophers problem*, showing how it can be done, albeit in a convoluted way.

The case study threaded through this work is a molecular dynamics program, Gromos. In its complexity, the molecular dynamics program is a better test than several simple programs since the different phases of the program can interact through data use and definition while having different data distribution requirements. Other kernels are introduced in language chapters.

In Chapter 2 we discuss our early parallelization of Gromos for the Stellar GS1000, a four processor shared–memory computer. For this machine we used an

[11] The parallel methodology for QCMD is currently unpublished.

MIMD approach often referred to the replicated algorithm, since the approach replicates the property arrays and the integration step. We used FORTRAN77 along with Stellar *test–and–set* primitives for this effort, making minimal changes to the sequential original. The model we used for the replicated algorithm is standard molecular dynamics with a cutoff radius. Our goal was to improve the turn around time for the average simulation of that period, typically several thousand atoms simulated over about 100 picoseconds. The parallelization was successful, however, since it used machine specific directives, the work was not directly portable to other parallel platforms.

The first application of IPfortran to a "industrial–strength" code was the replicated molecular dynamics application, UHGROMOS. In Chapter 7 we discuss that parallelization of GROMOS. It turns out that the replicated algorithm is quite suitable for distributed memory computers. With minimum, machine–independent modifications we implemented the replicated algorithm for GROMOS. Since we used IPfortran for the parallelization, UHGROMOS could readily run on any platform IPfortran was ported to (Chapter 5.4.3).

IPfortran consists of FORTRAN77 with extensions that permit the programmer to specify the location of "off–processor" data using machine–independent operators. To port applications written in IPfortran reduces machine specific concerns to the FORTRAN77 compiler, not interprocessor data exchange primitives. Those machine–specific details are managed in the IPfortran library. IPfortran was used to perform the development of UHGROMOS presented in Chapter 7 in addition to [74, 75, 10].

Performance of the *replicated algorithm* depends on the communication algorithm coupled with molecular dynamics modeling parameters in an analytical way. In Chapter 8 we derive an upper bound on processor utilization beyond which additional processors are wasted. Looking ahead, if we assume an $\mathcal{O}(N + \log P)$ global exchange algorithm, then the ratio of computation to communication is the ratio of the number of nonbonded pairs to the number of processors. Communication optimization strategies are also discussed in Chapter 8 for the IPfortran and Fortran D compilers.

The replicated algorithm requires that a global summation of the force array be performed at every time step. This global sum limits processor utilization (Chapter 8). Furthermore, since the force and coordinate arrays are replicated, memory usage scales poorly.[12] A spatial decomposition is a natural approach to the problem.

[12] The pair list is distributed in this method, however, for sufficiently large systems the coordinate and forces arrays can be prohibitively large.

In Chapter 9 we discuss our spatial decomposition of GROMOS. A far more complex algorithm than the replicated algorithm, the spatial decomposition practically required a rewrite of the entire *dusty deck* GROMOS. Under memory–constrained scaling [98], the spatial decomposition can theoretically scale indefinitely since communication is confined to a spatial locale determined by the cutoff radius. However, the communication costs can increase with increasing numbers of processors, beyond the relatively slow increase in the latency, from contention for the interconnection network. In Chapter 10 we examine the communication requirements for the spatial decomposition using a cutoff radius. We prove that the *shift algorithm* used by EULERGROMOS has optimal complexity. We present experiments illustrating contention effects on performance that we treat using several mapping heuristics designed to reduce contention for the special case of mapping a three–dimensional spatial decomposition to two–dimensional mesh.

The Fortran D prototype manages many of the low–level details a IPfortran programmer must perform. The Fortran D compiler is far more complex as a result. In addition, the Fortran D compiler requires that a program conform to a set of general guidelines in order that it can manipulate the program efficiently and correctly. In Chapter 12 we discuss a set of general guidelines that evolved from our parallelization of the replicated algorithm for GROMOS using Fortran D. Even though the replicated algorithm is a low–impact approach requiring modification to only a few key program sections, the entire GROMOS program is subject to analysis by the Fortran D compiler making the task a complex one given the structural condition of sequential GROMOS.

In Chapter 13 we overview future work related to this thesis. In the remainder of this chapter we discuss basic properties associated with a molecular dynamics simulation along with background material and related work.

Chapter 2

An early parallelization of a molecular dynamics program

In this chapter we present a method for parallelizing the non-bonded pair list generation and non-bonded force calculation algorithm for molecular dynamics. Using the parallelism inherent to existing molecular dynamics algorithms, it is possible, with minor modifications, to adapt the non-bonded routines to multiple–instruction, multiple–data (MIMD) computer architectures. This methodology has was applied to the molecular dynamics program GROMOS for the Stellar GS1000 Graphics Supercomputer. Aspects of the Stellar GS1000 architecture and programming environment are presented with attention to the performance of the molecular dynamics program. A speed enhancement factor [speedup] of about three was obtained relative to the serial execution of the program, which is close to the theoretical maximum factor of four for this machine. The overall speed enhancement factor increases to about six with vectorization enhancements. With parallelization and vectorization the program executed at about 35% of the speed obtained on a single–processor Cray X-MP.[1]

2.1 Introduction

The molecular dynamics method is widely used to study the dynamic and thermodynamic behavior of molecular assemblies via computer simulation [2, 78, 20]. At the heart of the molecular dynamics simulation is the potential energy function. The gradient of the potential supplies the force acting on each atom, which, in turn, is

[1] This chapter is based on the paper "Parallelization of a Molecular Dynamics Non-Bonded Force Algorithm for MIMD Architecture" [25]. This paper was one of the first published works of the method to be known as the *replicated algorithm*.

used to solve Newton's equations of motion for the system. The potential function consists of several terms that can be categorized either as non-bonded interactions or bonded interactions. Of special interest in terms of computational speed [overhead] are the non-bonded calculations, namely, the pair list generation routine and the the force matrix calculation. In a typical molecular dynamics calculation, these routines can consume well over 75% of the computation time.[2]

In principle, non-bonded calculations can be speeded significantly by using either of two type of parallel architectures, namely multiple-processor architectures where each processor executes an independent stream of instructions; or vector architectures which make use of pipelined or array processor hardware. The key program component behind vectorization, the array data structure, distinguishes this form of parallelism from the multiple–processor, multiple–data-path (MIMD) parallelism [56], where larger granularity parallelism contained in program sections comprised of many instructions can be exploited.

The ongoing effort to exploit parallelism inherent to problems and their algorithms is an area of great interest in computational chemistry [107, 31]. For the parallelization of molecular dynamics simulations, a variety of methodologies have been used [30, 104, 39, 15, 57]. Some approaches take advantage of the fact that for any time step in a molecular dynamics simulation, forces act simultaneously (and computationally independently) on each atom (or particle) of the system. This inherent parallelism has been utilized in producing algorithms adaptable to a variety of multiprocessor computers. Fincham discusses a scheme whereby pair interactions are calculated concurrently by parallel processes using circulating force accumulators and particle coordinates.

A standard *pair list algorithm* (Section 8.4) has a time complexity of $O(N^2)$ for N particles. The explicit tabulation of the pair list can be eliminated by organizing the atoms spatially on processors [57]. A segmentable, logical grid suitable for asynchronous, multi-processor parallelism avoids the need to check the N^2 possible interactions by spatially ordering the N particles [15].[3]

In this chapter we describe a strategy for the effective application of coarse granularity parallelism in non–bonded calculations on MIMD architectures; the modifications to the sequential algorithm were minimal. Results are presented for the GROMOS

[2] This estimate is conservative; 90% or more is more typical.

[3] We are concerned here with molecular dynamics using a cutoff radius where a fraction of the entire set of N^2 interactions are calculated.

```
SUBROUTINE NBPAL

DO I = 1, N
   DO J = I+1, N
      ...
      NJNB = NJNB + 1
      JNB(NJNB) = GLOBAL_ATOM_NUMBER
      ...

   ENDDO
   INB(I) = NJNB
ENDDO
```

```
SUBROUTINE NONBAL

KMIN = 0
DO I = 1, N
      ...
   KMAX = KMIN +INB(I)
   KMIN = KMIN + 1
      ...
   DO J = KMIN, KMAX
      ...
      F(I) = F(I) + NEW_FORCE
      F(J) = F(J) - NEW_FORCE
      ...
   ENDDO
   KMIN = KMAX
ENDDO
```

Figure 2.1: Sequential GROMOS routines for the pair list calculation and the force calculation.

molecular dynamics program [47], on the Stellar GS1000. The further effect of vectorization of the code for the vector processing units of the Stellar are also discussed.

2.2 Sequential algorithm

The calculation of the force array for non-bonded atom pairs in molecular dynamics is often divided into two parts: the pair list generation followed by the force calculation. Within both phases the computation can proceed independently for each atom i of the pair ij. We illustrate this below with the nomenclature of GROMOS.

2.2.1 Pair–list generation

For an N atom molecular assembly, a total of N(N-1)/2 ij-pairs exist, where i and j are sequential atom numbers.[4] These pair interactions are determined by a two–level nested loop where the outer loop ranges over the i of the ij pair; the inner loop ranges over the j component. Figure 2.1 shows the loop structure for the pair list generation routine in GROMOS. Pairs are determined according to a cutoff radius, where interactions outside of an interaction sphere are discarded. The resulting non-bonded interactions are in the arrays INB and JNB. Indexed by the atom sequence number i, INB(i) contains the number of j-partners in JNB for atom i. The first memory cell for atom i in JNB is the cell following atom $i-1$. The last JNB cell for

[4] In later chapters the more descriptive term of *global atom number* is used.

atom i is the last cell for atom $i-1$ plus INB(i). Since the induction variable j of the inner loop of Figure 2.1 is always greater than i, each ij pair will be considered once and only once. Thus, each cycle of the outer loop generates a unique set of ij pairs independently of other iterations of the loop.

With this method to generate the pair list, and assuming a uniform distribution of the system geometry in the list of atoms, there will be more pairs with an i-value of m than pairs with an i-value of $m+k$, for some positive integers m and k. This fact plays a key role in adjusting the segmentation [load balancing] in the parallel version of the pair–list calculation.

2.2.2 Force calculation

The non–bonded force calculation proceeds by calculating the gradient of the potential energy of interactions for atom pairs to come up with a force on each atom. This part of the algorithm also uses a level–two nested loop. The outer loop ranges over the atom sequence number i; the inner loop ranges over the set of js for i thereby unraveling the atoms pairs ij from the INB and JNB arrays (Figure 2.2, right). The time complexity for this calculation will be something less than order of N^2, in contrast to the approximated $\mathcal{O}(\mathcal{N}^{\epsilon})$ pair list generation.[5] The force on each atom, accumulated in an array, can have numerous contributions depending on the number of pairs the atoms form.[6] The separate contributions to the force array are independent of one another.

2.3 Parallel algorithm

The fact that the pair list construction and pairwise force calculation are independent of other interactions, the non-bonded force calculation can be performed concurrently. The strategy we use to parallelize the calculation segments the INB array and assigns a segment of INB to each processor. In the bulk of this chapter (remaining true to the original paper), a four–processor architecture is assumed, however, INB can be segmented into an arbitrary number of segments up to the number of atoms in the system; we present an expression for the general case below.

[5] This fact is exploited by calculating the pair list at k step intervals, reusing it for intervening force calculations [105].

[6] For typical protein and water densities, the average number of pairs for atom is about 100 for a 10 Angstrom cutoff radius.

To implement the scheme, the changes to GROMOS were kept to a minimum. In both phases of the algorithm – pair list generation and force calculation – a setup section preceding the parallel sections calculates pointers into INB and initializes arrays. Results are merged at synchronization points following the parallel sections into the force array.[7]ote that formally, the Stellar is a shared–memory machine. The strategy used here resembles a fork and join, however, at the joined sections the same calculation proceeds redundantly at all processors. This is an inefficient use of the Stellar, but it set the stage for future MIMD development on massively parallel architectures.

2.3.1 Pair–list generation

For pair–list generation each processor calculates the pair list for one segment of atoms, that is, a segment of INB. Each processor implements an identical copy of the pair list generation subroutine differing from the original, sequential version primarily in loop control variables. Figure 2.2 (left) shows the modified loops in the subroutine NBPAL of GROMOS.

The output from each calculation, the JNB segments corresponding to the INB segments, are merged into a JNB at the synchronization point. It is not feasible to share JNB during its construction. Even if it were determined that resulting costs in merging separate JNBs are greater than sharing JNB, the elements in JNB must be contiguous by atom for each atom i, a feat that can be accomplished only after completing the calculation for the INB segments.

2.3.2 Force calculation

In the force calculation segments of INB are passed to each processor so that each processor calculates the force for a partition of the ij–pairs set. As with pair–list generation, an identical copy of the non–bonded force routine used by each processor differs from the sequential version mainly in the loop control variables. Figure 2.2 (right) shows the modified loops in subroutine NONBAL of GROMOS.

An atom number will appear as an i of an ij pair in one and only one segment. An atom number can, however appear in more than one segment as a j partner. Since the force array is indexed by atom sequence numbers, more than one processor can simultaneously access the same cell, corresponding to some atom. Here, treating

[7]N

```
SUBROUTINE NBPAL_Parallel

DO I = A(my_procId), B(my_procId)
   DO J = I+1, N
      ...
      NJNB = NJNB + 1
      JNB(NJNB) = GLOBAL_ATOM_NUMBER
      ...

   ENDDO
   INB(I) = NJNB
ENDDO
```

```
SUBROUTINE NONBAL_Parallel

KMIN = 0
DO I = A(my_procId), B(my_procId)
      ...
   KMAX = KMIN +INB(I)
   KMIN = KMIN + 1
      ...
   DO J = KMIN, KMAX
      ...
      F(I) = F(I) + NEW_FORCE
      F(J) = F(J) - NEW_FORCE
      ...
   ENDDO
   KMIN = KMAX
ENDDO
```

Figure 2.2: Abstracted GROMOS standard routines nonbonded interaction routines modified for parallel execution.

the access of the force array as a critical section [4], or setting up an array for each processor to store results separately, are both viable options. For the implementation reported here, the latter approach was used, requiring that the results be joined after a synchronization point.

2.3.3 Load balancing

It is desirable to keep each processor equally busy since an even distribution of work minimizes the wall clock time that a calculation will take. When a processor finishes its work, an assigned INB segment, it will wait until all segments are processed before the calculation proceeds. This wait can be viewed as a measure of wasted resources, assuming one is maximizing the real time performance of the calculation.[8]

The key to creating an even distribution lies in the segmenting of INB. Given the imbalance of JNB cells corresponding to INB blocks, the equal segmenting of INB will result in load imbalance. Below we develop a set of closed expressions to closely approximate balance workload distributions.

Let a, b, c, and d be pointers into INB for the processors 1,2,3 and 4, respectively. Define segments of INB for the four processors to be $[1, a]$, $[a+1, b]$, $[b+1, c]$, and $[c+1, d]$, where d is the number of atoms, N. Consider the following loop

```
do 100 I = A, B
```

[8] Certainly this is the case in space–shared machines such as the Intel hypercube and mesh series.

	P1	P2	P3	P4
INB	1 2	3 4	5 6	7 8
	1 2	3 4	5 6	7 8
	2 3	4 5	6 7	8 -
	3 4	5 6	7 8	
JNB	4 5	6 7	8 -	
	5 6	7 8		
	6 7	8 -		
	7 8			
	8 -			

Table 2.1: Load imbalance illustrated by laying out the iteration space for $P = 4$ processors and $N = 8$ atoms. Each entry represents a pairwise interaction. For example, $P4$ has pairs $(7,7)$,$(7,8)$ and $(8,8)$. The load imbalance results from the triangular loop in the pair list generation.

```
      do 100 J = I+1,N
100   statement.
```

We are interested in executing *statement* an equal number of times on each processor. The loop bounds A and B are the pointers defining the INB segment $[A, B]$. With the balanced execution of INB, *statement* will be executed 15, 11, 7, and 3 time, for processors 1, 2, 3, and 4, respectively. The sub columns of Table 2.1 each iteration is illustrated with a single entry. On close inspection of Table 2.1, one see that same atom interactions, where $i = j$, are considered; this is done for clarity and does not affect the applicability of the arguments here.

The loads for the four processors can be expressed as

$$S_1 = \sum_{i=z+1}^{N} i, \quad S_2 = \sum_{i=y+1}^{z} i \quad S_3 = \sum_{i=x+1}^{y} i, \quad S_4 = \sum_{i=1}^{x} \tag{2.1}$$

where $S+j$ corresponds to processor j, and i is the number of inner loop iterations for each cycle of the outer loop. For even segmentation, $x = 2$, $y = 4$, and $z = 6$.

The pointers segmenting INB can be written as

$$a = N - z, \quad b = N - y \quad c = N - z, \quad \text{and} \quad d = N. \tag{2.2}$$

By choosing x, y, and z such that $S_1 = S_2 = S_3 = S_4$, the a, b, c and d of Equations 2.2 can be used to segment INB so as to achieve a nearly even distribution of work for the four processors. From the closed forms for Equation 2.1 and Equation 2.2 we have

$$
\begin{aligned}
(z+y+1)(z-y) &= (y+x+1)(y-z) \\
(y+x+1)(y-x) &= (x+1)x \\
(N+z+1)(N-z) &= (z+y+1)(z-y).
\end{aligned}
$$

An exact solution to these equations will not usually be possible with x, y, and z in the natural numbers; assuming x, y, and z are large relative to one, we make the approximation

$$
\begin{aligned}
(z+y)(z-y) &= (y+x+1)(y-z) \\
(y+x)(y-x) &= (x+1)x \\
(N+z)(N-z) &= (z+y)(z-y)
\end{aligned}
$$

arriving at

$$
\begin{aligned}
x &= \sqrt{\frac{N^2+3}{4}} \\
y &= \sqrt{2x^2-1} \\
z &= \sqrt{2y^2-x^2}.
\end{aligned}
$$

The results for a, b, c and d in terms of N and for four processors are the numbers: $a = 0.13397N$, $b = 0.29289N$, $c = 0.50000$, and $d = N$.

General form

A general expression for a load balanced segmentation can be arrived at algebraically or through a geometric approach. For P processors the balanced segmentation of INB for the jth processor is for I from $L(j)+1$ to $L(j+1)$ where

$$
L(j) = N - \left(\frac{(P-j)N^2+j}{P}\right)^{\frac{1}{2}}.
$$

For the pair list calculation the scheme is close to exact since all pairs must be considered. For the force calculation, on the other hand, the success of the method depends on the geometry of the molecular assembly relative to atom positions in the INB array. In Chapter 7 the force load balancing is addressed. In the remainder of this chapter, results using some test simulations are presented.

2.4 Implementation

Until now the discussion was concerned with a four–processor architecture where segments of INB are assigned to individual processors. How this assignment is made is irrelevant to the overall procedure. With the Stellar GS1000, the machine used for the implementation, the Unix operating system will schedule concurrent segments of a process (threads) for execution. In the multi-tasking environment, similar to Cray micro tasking [13], the processor–to–concurrent segment association breaks down since a concurrent segment can at different times during its execution be assigned to more than one processor.

2.4.1 Stellar GS1000

The Stellar GS1000 graphic supercomputer provides a single vector/floating–point unit with a 45 MFLOP pear performance. The MIMD part of the machine consists of four bit slice processors with an aggregate of 20–25 MIPS. The processors share a high–speed, 1 MBYTE cache with a bandwidth of 1.28 GBYTE/second. The machine is (was) configurable with up to 128 MBYTE of main memory with an access rate of 320 MBYTE/second. The Stellar compiler automatically parallelized and vectorized suitable code.[9] A common situation is a level–two loop with data dependencies that permit vectorization of the inner loop and parallelization of the outer loop [22].

The Stellar compiler provided directives for parallelizing programs with inherent parallelism not detectable by the compiler. In Figure 2.3 the calls to the Fortran subroutines Nonbal and Nbpal illustrate an application of the *parallel* directive. The *parallel* directive set up a parallel region for the loop immediately following it such that loop iterations will execute concurrently.

Concurrent multiple entries to the same subroutine text was facilitated by the Stellar Fortran reentrant flag. This caused the local variables for to be stored on the stack for each invocation, rather than static memory. To use the same copy of a subroutine eliminates the overhead of maintaining multiple, identical copies with the added benefit of improving the program's locality of reference during execution.

[9] In practice, the parallelizing compiler did a poor job. In addition, many customers were disappointed with the Stellar due to its low MIPS rate.

```
C$DIR PARALLEL
      DO I = 1, 4
        CALL NBPAL(parameters,I)
      ENDDO

      ...  join results from NBPAL

C$DIR PARALLEL
      DO I = 1, 4
        CALL NONBAL(parameters,I)
      ENDDO

      ...  join results from NONBAL
```

Figure 2.3: The use of the Stellar parallel directives for executing concurrent invocations of the four calls to the pair–list routine, NBPAL, and the nonbonded force routine, NONBAL. Note the loop index variable is an additional argument to the subroutines.

	Argon	Peptide	DNA
Standard sequential scalar	154	534	711
Convex sequential scalar	165	644	853
Standard parallel scalar	51	181	277
Convex parallel scalar	55	216	302
Standard sequential vectorized	128	420	578
Standard parallel vectorized	49	161	249
Convex sequential vectorized	61	182	276
Convex parallel vectorized	28	93	144

Table 2.2: Test case timings in wall–clock seconds on a dedicated machine, using 32-bit precision (add approximately 14% more time for 64-bit. The Argon and Peptide data are from 50 step runs, with a pair list update every 10 steps. The DNA data are from 10 step runs, with one pair list update per run.

2.5 Results

We modified two pairs of Gromos pairlist and nonbonded routines for concurrent pair–list and force calculations. The standard set of routines do not vectorize well; the second pair was written for the Convex vectorizing compiler. (The standard and Convex–adapted routines are from the 1987 standard release tape of the Gromos software.)

2.5.1 Correctness

Prior to performing timing runs, the correctness of the program was ascertained. We performed molecular dynamics simulations for 1 ps and 20 ps using a step size of 1 fs. The analyses of these simulations were compared with the analyses of identical simulations performed with Gromos runs on a VAX 8650. The 1 ps analyses consisted of a comparison of phase space coordinates and energies [63].

Since molecular dynamics trajectories often diverge due to different roundoff algorithms or order of operations, as is the case with the parallel implementation, analysis longer simulations requires the analysis of properties which are more–or–less independent of the trajectory. For the 20 ps simulations we calculated solvent radial distribution functions and diffusion constants [2]. Among the test systems used: a box of 864 Argon atoms (liquid Argon), a 1037–atom system consisting of a five residue peptide with four ions in 322 waters (peptide system), and 4 DNA fragments with 2234 atoms and 22 sodium ions in a bath of 1220 water molecules (DNA system).

2.5.2 Timings

We performed 10–50 steps of molecular dynamics using the test cases, and versions of the nonbonded force and pairlist described above. The results (Table 2.2) illustrate the comparison of the standard non-bonded routines with the Convex structured ones. The gain from vectorizing the standard routines is modest (compare *standard sequential scalar* with *standard sequential vectorized*). A more substantial speedup with minor code modification (2.6–3 times) was obtained by parallelization (compare *standard sequential scalar* with *standard parallel scalar*). Vectorization of the Convex code is much more effective than vectorization of the standard code, yielding speedups of 2.7 to 3.5.

Parallelization of the vectorized codes increases the overall speedup factor to

	Thread Number			
	1	2	3	4
NBPAL	36/23	31/22	26/23	7/23
NONBAL	40/29	30/24	21/24	9/24

Table 2.3: Percentage of time in threads for NBPAL and NONBAL. Entries indicate the percentage without load balancing and percentage with load balancing.

2.9–3.3 for the standard code, and 5.9–6.9 for the Convex code. The greateer average speedup obtained by parallel execution of the standard, vectorized (2.5 times) comparid to that for the parallel execution of the Convex, vectorized version (2.0 times); the poorer speedups for the vectorized version may reflect contention for the vector processor.

2.5.3 Profiling

We determined the percentage of time spent by thread in the concurrent regions using a Stellar profiling tool. The normalized data are present in Table 2.3. From these data we see that the load balancing methodology is very effective for the pairlist calculation (NBPAL), but inexact for the force calculation (NONBAL), as we mentioned in Section 2.3.3.

2.6 Discussion

Not all existing programs are amenable to significant gains via vector optimizations without undergoing extensive rewriting. Indeed, this applies to parallelizing languages as well. (In Chapter 12 the parallelization of GROMOS using High Performance Fortran is presented.) The standarg NONBAL and NBPAL subroutines are sufficiently convoluted (that is to say vector resistant) that without restructuring the code, speedups from vector optimizations are modest as seen in Table 2.2.[10] While the original Stellar scheme was developed for a four–processor architecture, each atom pair could assigned to separate processors (see Chapter 7).

We found that the load balancing scheme works better for the pair list generation routine, NBPAL, than for the force calculation routine, NONBAL. There is no guar-

[10]The structure also causes performance problems due to poor locality, making for inefficient use of the cache and pipelined processor architectures.

antee that the loop of Figure 2.2 (right) will iterate like the idealized loop discussed above, but the pair list loop of Figure 2.2 will. To achieve a load–balanced distribution of INB for the force loop requires re-analysis after every non–bonded pair–list update; the scheme would have to consider the contents of JNB directly because the results of the inner loop of the pairlist generation are geometry dependent (see Chapter 7).

When combined with effective vectorization, the use of MIMD parallelism yields significant speedups (about six times) on the Stellar GS1000. Comparative benchmarks show that the resulting speed is about five times that of the DEC 8650 and about 0.35 times that of a single–processor Cray X-MP.

Chapter 3

IPfortran

3.1 Introduction

IPfortran is a language designed to facilitate the programming of multi-process, data-parallel applications.[1] Based on a partitioned-memory paradigm, IPfortran extends Fortran with a small set of operations. The duality of the send and receive operations in the message-passing paradigm [55] is encapsulated in an infix operator (Section 3.3) and reducing functions (Section 3.4). The IPfortran translator generates generic calls to a system-dependent library from user-supplied expressions containing communication operators.

Abstracting the send and receive with an operator results in streamlined code and reduced development time. Errors are reduced by leaving the machine–dependent communication generation to the translator.

The partitioned-memory paradigm is equally efficient on shared-memory machines and provides a universal programming model at a higher level of abstraction than message passing. It provides a way to distribute data that is not explicitly available in a shared-memory programming model. Certain types of loops are parallelized in the partitioned-memory paradigm with far less change to a sequential code than in a shared-memory programming model.

IPfortran makes the important assumption that each processor knows the names of the variables in all processors. To this end we require that all processors run the same IPfortran code; our programming model is Single-Program Multiple-Datastream, or SPMD. The programmer, using a *local* view of the data, is responsible for its distribution and access.

[1] The material in this chapter comes from [8] and [5].

$\mathbb{P}$fortran programs are written using the *local* approach, with explicit logic for each computational element (processor or process) with data decompositions propagated by programmer-supplied control structures. This methodology, while crude relative to globally-viewed, automatic decompositions [27], can provide greater flexibility. Moreover, industrial-strength codes have been ported to $\mathbb{P}$fortran in a short amount of time [26].

With the $\mathbb{P}$fortran programming model, the logical processor number pinpoints the data item in an expression through the use of the @ (Section 3.3) where $i@p$ represents a different variable from $i@q$, if $p \neq q$.

Since each Fortran program operates on its own memory the question arises, what can be different at each processor and what can be the same? Prior to the execution of the first instruction, all processes have the same data and instructions, and therefore are in identical states. The key difference between the P processes is the variable *ThisNode*, the logical processor number. Any logic in the program involving the variable *ThisNode* can potentially create a condition whereby two or more processes are in different states, meaning that for at least one variable name in the replicated program (recall the SPMD model), there will be different values. Without the variable *ThisNode* and using the the SPMD model on identical processors, we would have rather uninteresting computations.

As a consequence of the logical processor number *ThisNode*, the values at each processor, corresponding to some variable, can be partitioned into two classes that we call *local* and *global*. *Local* values are (or more accurately, *could be*) different from other corresponding values at other processors. *Global* values are the same at all processors (in general, *local* and *global* can be applied to the set $Q \subseteq$P). With the current version of $\mathbb{P}$fortran, the programmer must keep track of which values are *global* and which are *local*. In any case, it is useful for the programmer to be aware of the distinction as it can affect efficiency. In general, *target* and *source* values must be global for those processors participating in a communication (see Section 5.4).[2]

Finally, one can always incorporate message-passing code with $\mathbb{P}$fortran code where $\mathbb{P}$fortran falls short, provided the message-passing model is available on the target system.

[2]The names *uniform* and *non-uniform* are used in [7], more succinctly describing *global* and *local*. Bagheri also devises formal semantics for the *guarded–memory model* to which $\mathbb{P}$fortran belongs.

3.2 IPfortran Constants

IPfortran defines the following constants for writing the node program. These will be used throughout this manual.

- *ThisNode* is the logical processor number of the processor executing the program. This default name can be modified at compile time.
- *NumNode* is the total number of processors.
- *CubeDim* a architecture-dependent variable defined for hypercube-topologies giving the dimension of the cube, i.e., $\log_2 NumNode$.

Processors are logically numbered $0, \ldots,$ $NumNode-1$.

The individual entity executing a particular copy of the program with a particular value for *ThisNode* will be referred both as a processor and a *node*. Strictly speaking, this should be thought of as an individual *process* rather than *processor*. However, we will ignore this distinction between physical and logical entities in our discussion.

In some systems, there is a host processor, and a host program may even be required. In such systems, it is assumed that at most one process on the host can communicate with the nodes (processors). In most applications, however, a host program is not used; specialized tasks such as I/O (see Section 3.6) are performed from a designated node, usually logical node 0.

3.3 The Operator @

Accessing variables at other processors can be done in IPfortran using a new binary operator, @, which indicates that a variable is *at* some processor. For example,

$$x = y@0$$

assigns the value of variable y of processor 0 to x at every processor (*broadcast*). Similarly

$$x = y@(n+1)$$

broadcasts the value of variable y of processor $n+1$ to x (at every processor). Mixed expressions such as

$$x = y@p + z$$

assigns to x (at every processor) the sum of z at that same processor and y from processor p.

Multiple uses of @ in a statement are permitted, but only in assignments (i.e. expressions involving =) or in reduction operations (Section 3.4). (@, in principle, makes sense in any expression, e.g. predicates, but has not been implemented there.)

We can now give a simple program segment to compute a sum of values at each processor, say in the variable y. First, all processors broadcast their own value and then each sums the values independently.

```
      DO 10 n=0, numnode-1
10       x(n) = y @ n
      sum=0.0
      DO 20 n=0, numnode-1
20       sum = sum + x(n)
```

3.3.1 Local vs. global values

This technique for summing values over all processors, while not efficient, serves to illustrate an important point concerning our assumption about communication in IPfortran. In the first loop in the code above, the processor from which we receive data is a variable. However, at each step in the iteration the value of this variable is the same at each processor. Thus n denotes a value universally agreed upon at each iteration, a value *global* to all processors, P. If the value of n were different in different processors, then a *local* value results and a more complicated implementation would be needed.

If values referring to processor numbers are *global*, we can predict the communication pattern required for the data exchange. Otherwise, for processor numbers given by *local* values, the communication pattern must be determined at run time. This is not difficult to do, but it requires a significant amount of overhead.

In a future version of IPfortran we plan to do compile-time analysis to determine which variables referring to processor numbers are *local* or *global*. At the moment, the programmer should limit the use of variable processor numbers to *global* values (see Section 5.4). Our experience has shown that this is not a significant hindrance.

An example of converting local variables to global ones is as follows. Consider the code fragment

```
i=f(...)
x=y@i
```

In this example, we assume that i will not have the same values at all processors. The correct behavior will be obtained with the current compiler if this is replaced by the (equivalent) code

```
i=f(...)
do j=0, numnode-1
   ithere(j) = i@j
enddo
do j=0, numnode-1
   x@j = y@ithere(j)
enddo
```

The set of the values `ithere` can be collected in a much more efficient way using various algorithms, to be discussed subsequently.

3.3.2 An example: sorting

To illustrate the power of the operator @, we consider the algorithm for a bitonic sort citeknuth. We consider the simplest case in which each processor begins with only one value, `n`, and the intent is for each processor to have only one value at the end, but in sorted order with the largest value at processor 0. In the following code we generate `n` with a simple modulo function, but anything could be used. We have hard-coded this for a number of processors equal to a power of two.

```
integer i,j,d,locdim,n,x,iswch
n = mod(17*THISNODE , 63)
locdim = 1
ipadr=THISNODE
do 1000 j=0, cubedim -1
  d = locdim
  locdim = locdim*2
  do 1000 i=0, j
    notd=MINOT(d)
    ieven=MIAND(ipadr,notd)
```

```
          iodd=MIOR(ipadr,d)
          x@ieven = n@iodd
          x@iodd = n@ieven
          iswch= mod((ibit(THISNODE,locdim)+ibit(THISNODE,d)),2)
          if( iswch .EQ. 1 ) then
            n = min(x,n)
          else
            n = max(x,n)
          endif
          d = d/2
100     continue
1000  continue
```

In this, we have used the bit-manipulation routines. `MIOR`, `MINOT` and `MIAND` for the bitwise logical "or", "not" and "and" functions, respectively. Similarly, `bit` is a bit-manipulation routine that returns the bit value in a given bit location. These can be defined in C as follows, and linked with Fortran on most systems.

```
int minot_(mask)
   int *mask;
{return(~*mask);
}
int miand_(mask,nask)
   int *mask,*nask;
{return(*mask&*nask);
}
int mior_(mask,nask)
   int *mask,*nask;
{return(*mask|*nask);
}
ibit_(m,d)
int *m,*d;
{
   if( *m & *d ) { return( 1 );}
   else { return( 0 );}
}
```

3.3.3 Ranges of variables

It is often necessary to exchange collections of variables among different processors, not just singletons. This is allowed in IPfortran using Fortran90 array syntax, such as `X(7:11)` to denote the range of variables `X(7)`, `X(8)`, `X(9)`, `X(10)`, `X(11)`. For example, if `numnode = 2**CUBEDIM` then the array `ithere` can be collected via the loop

```
      do j=0, numnode-1
         ithere(j) = -thisnode
      enddo
      ithere(thisnode)=thisnode
      muchmo = 1
      mask = 1
      mmask = numnode - 1
      ipadr=THISNODE
      DO 123 idim = 1, CUBEDIM
        notmask=MINOT(mask)
        ieven=MIAND(ipadr,notmask)
        iodd=MIOR(ipadr,mask)
        ievbot=MIAND(mmask,ieven)
        ievtop=ievbot+muchmo-1
        iodbot=MIAND(mmask,iodd)
        iodtop=iodbot+muchmo-1
        ithere(iodbot:iodtop)@ieven = ithere(iodbot:iodtop)@iodd
        ithere(ievbot:ievtop)@iodd = ithere(ievbot:ievtop)@ieven
        mask = mask*2
        mmask = mmask*2
 123  muchmo = muchmo*2
```

Here, `OR`, `NOT` and `AND` are the logical "or", "not" and "and" functions, respectively (see Section 3.3.2). At each step, the data exchanged is twice that in the previous step, with the neighbor in the *idim*-th (hypercube) dimension. After $\log_2(numnode)$ steps the entire array has been exchanged. We note that this algorithm would execute correctly on any architecture but there could be potential communication contention on ones that are not based on a hypercube network.

Ranges of variables are discussed in more detail in Section 3.5.

3.3.4 Ranges of processors

There are many instances in which sets of processors or arrays of values are useful objects. We have implemented one class of such sets, namely *ranges*. The construct

$$\text{<expr>@ (}<mexp>_1 \;:\; <mexp>_2\text{)}$$

refers to the value of <expr>in the range of processors from the value of $<mexp>_1$ to the value of $<mexp>_2$. It can appear on the left-hand side of an assignment statement. For example,

$$x@(1:5) = y@0$$

means: "Assign the value of variable y at processor 0 to variable x at processors 1 through 5." On the right-hand side, the range logically returns multiple values. For expression

$$x@(i:j) = y@(r:s),$$

the range $(r:s)$ will map 1:1 onto the range $(i:j)$ on the left-hand side; if $j-i \neq s-r$, then the statement is not defined with unpredictable results. In addition to a single range or scalar processor identifier, lists of scalars and ranges are allowed, but also with a 1:1 correspondence if more than one processor is specified on the right-hand side. Some examples follow:

$$a@(i,j,k) = b@(m,n,o)$$
$$a@(1,2,3,4) = b@0$$
$$a@(k+m:n) = b@0$$

3.3.5 Functions of *thisnode*

It is convenient to write $x@thisnode = y@(thisnode - 1)$, however, this is incorrect in the current version of IPfortran and will result in deadlock. It is not unreasonable to expect the translator to detect the presence of *thisnode* and generate correct code. The remainder of this section talks about the first case, detecting the direct presence of *thisnode* in a communication statement. This has been implemented in IPfortran's cousin, PC [6].

Using standard compiler algorithms, analysis can detect the dependence of a variable's value on *thisnode*. The expression

$$x = y@mod((thisnode + 1), numnode)$$

means: assign the value of variable y of processor *(thisnode+1)* (modulo *numnode*) to x in *thisnode*. We can have any expression which references *thisnode* explicitly, including functions which return integers, arithmetic operators, and so on.

An example using functions of thisnode can easily be constructed to give a slightly more efficient summation algorithm. Think of our nodes as making up a simple ring. Then we can sum by accumulating and passing partial sums around this ring:

```
c Ring Sum WITH functions of thisnode
        program ringSum1
        integer n,sumIn,sumOut
        sumOut = thisnode
        do n = 1, numnode-1
            sumIn@thisnode = sumOut@MOD(thisnode+1,numnode) + thisnode
            sumOut = sumIn
        enddo
        print*,thisnode, sumOut
        end
```

This has the effect of computing the n-th partial

$$(\ldots(y@me + y@(me+1)) + \cdots + y@(me+n+1))$$

at the n-th iteration, where node indices in the latter representation must be identified modulo *numnode*.

It is possible to implement the ring summation algorithm without functions of thisnode by adding an inner loop over logical processor numbers:

```
c Ring Sum WITHOUT functions of thisnode
        program ringSum2
        integer n,p,sumIn,sumOut
        sumOut = thisnode
        do n = 1, numnode-1
          do p = 0, numnode-1
```

```
    sumIn@p = sumOut@MOD(p+1,numnode) + p
  enddo
  sumOut = sumIn
enddo
print*,thisnode, sumOut
end
```

3.4 Reduction Operations

Having introduced a formalism for distributed variables via the @ operator, we can now think of them as a new data type. It is natural to consider appropriate operators on this new data type. In this section we introduce one class of such operations based naturally on corresponding operations on each component of the distributed data structure. Such operations may, moreover, be defined on a restricted set of distributed data. (As mentioned above, an expression such as x@(i:j) may refer to many variables. However, the current implementation of IPfortran reduction operations only supports the case with the entire set of the distributed values.)

The concept of a *reduction operation* was introduced to accomplish a specialized operation on a set of data; the operation is a reduction in the sense that a set of data is reduced to a single datum of the same type through the repeated application of a binary operator on pairs of the data. Furthermore, the operation is defined such that the result will itself be distributed in the same way the operands were distributed.

3.4.1 Reduction operations defined

To allow reduction operations, IPfortran introduces the following constructs:

$$<id> \ = <id> \ \{ \ <\text{expr}>[\text{,opt args}]\}$$
$$\text{call} \ <id> \ \{ \ <\text{expr}>, \ <id> \ [\text{,opt args}]\}$$

where <expr> can reduce to a variable or constant. Functions and subroutines used along with {} are called *reducing functions* and *reducing subroutines*, respectively. Some examples of reduction operations that one may reduce include: SUM, MIN, MAX, OR, and AND. IPfortran provides the reduction operation + described in Section 3.4.6. Other binary operators can be cast as reduction operations as described below.

The distributed value is represented by the symbolic name closest to the left curly brace {. The *set notation* emphasizes the notion of a collection of data objects, distributed in some manner across a collection of processors. *function_name* (respectively, *subroutine_name*) must be a function (respectively, subroutine) of at least two (respectively, three) parameters. Functions must return a value of the same type as <expr>. The target variable is given by the left-hand side when using the function syntax, or as the second parameter, with procedure syntax (following the familiar mnemonic of *in,out* for parameter order). The designated operator, encapsulated in a function or subroutine, is then applied to the set of distributed values. As with other ℙfortran statements, Fortran90 array syntax may be used (see Section 3.5).

We symbolically write the reduction operation for a p-process model

$$A = \oplus\{B\}$$

or

$$\text{call} \quad \oplus\,\{B, A\}$$

as

$$A = B_1 \quad \oplus \quad B_2 \quad \oplus \quad \cdots \quad \oplus \quad B_p \ .$$

Since the order of application of $\oplus$ to the set of values is not specified, $\oplus$ must be associative and commutative. (More precisely, we are assuming that the set of values of the type of B forms an Abelian semi-group with respect to the binary operator $\oplus$.) Although the order of evaluation may depend on the particular architecture or even the number of processors, we intend that will be deterministic in the sense that it will always be done in the same order on a given machine with a given number of nodes.

These concepts are illustrated with a concrete example. Consider a user defined reduction operation implementing a summation of a set of scalars distributed across p processors into a single scalar. Expressed mathematically as

$$\forall j, \quad result@j = \sum_{i=0}^{p-1} scalar@i.$$

By defining a function, `mysum`,

```
REAL*8 FUNCTION mysum(in1,in2)
mysum = in1 + in2
RETURN
END
```

now we can write

```
result = mysum{scalar}.
```

3.4.2 Reduction operation procedural interface and parameter typing

A reduction operation can be associated with either intrinsic functions or user defined functions; we consider examples of each. Using the intrinsic Fortran functions, `AMIN1`, the statement

```
y = AMIN1 { x }
```

computes the minimum of all values of `x` in all processors and assigns this value to the variable `y` (at all processors).

The general form of the reduction procedure invocations are

function: `out = f1{in}`

subroutine: `call s1{in,out}`

where `in` is the distributed operand. `f1` and `s1` implement some binary operation with the results assigned to the distributed variable `out`. The corresponding function and subroutine definitions are of the form

function: `"type" FUNCTION f1(in1,in2)`

subroutine: `SUBROUTINE s1(in1,in2,out)`

where `"type"` denotes the type (e.g. `REAL` or `INTEGER`) of the function `f1`. With the function, the returned value is passed through the function return mechanism whereas in using the subroutine, the optional parameter `out` is provided. In both cases, the parameters `in1` and `in2` are not used to return results.

The user-defined reduction operations may include optional parameters. These parameters are assumed to be global parameters, the same at all processors, and they are not communicated, but are included in the calling sequence as shown above. These parameters can carry type information for the parameters `in` and `out`, or can be used in generating side-effects with the reduction operation.

Optional parameters are often useful. For example in FORTRAN77, without user-defined types, these parameters can carry type information. The contrived statement

```
CALL opuratur { in,out,param1,param2 }
```

illustrates the use of optional parameters. The corresponding subroutine definition is

```
SUBROUTINE opuratur(in1,in2,out,param1,param2)
```

where `param1` and `param2` are optional parameters and `out` is the output parameter. We notice that the parameters supplied to the function or procedure implementing the binary operator may be used nondestructively, that is, read-only.

The reduction of a function, e.g. `out=opratr{in,param1,param2}`, requires a function with the definition

```
REAL FUNCTION opratr{in1,in2,param1,param2}.
```

Recall the array `ithere` from Section 3.3. Its accumulation may be accomplished as follows.

```
izero=0
do j=0, numnode-1
   ithere(j)=0
enddo
ithere(thisnode)=i
call intplus{ithere,ithere,izero,numnode}
```

where the subroutine `intplus` is defined by

```
subroutine intplus(ithere,jthere,kthere,istart,n)
dimension ithere(1), jthere(1), kthere(1)
do j=istart,istart+n-1
   kthere(j)=ithere(j)+jthere(j)
enddo
end
```

3.4.3 Parameter conformance

The actual parameters `in` and `out` must conform in shape, and if `out` is an array, `in` must also be an array. The formal parameters `in1` and `in2` correspond to some pair of operands for an operation $B_i \oplus B_j$, using the notation of Section 3.4.

The memory for at least one of the two input parameters, perhaps both, comes from a IPfortran scratch array; if possible, the output parameter `out` is used as working memory through the duration of the reduction operation. This implementation detail is of practical concern only in assuring that the IPfortran temporary array is of sufficient size to accommodate intermediate results (see Appendix 5.4.3.)

The array-section actual parameter will be reshaped to *rank 1* arrays, where the reshaping will construct the *rank 1* array in array order from the array subsection `in`, provided `in` is not already *rank 1*. The array section syntax denotes the description of a subarray with a shape indicated by the subscripts; Section 3.5 discusses IPfortran array usage in detail.

3.4.4 The nondestructive subroutine interface

A slightly more efficient reduction operation with a subroutine interface can be defined for arrays by using IPfortran function syntax for the global operation. IPfortran does not support anything but scalar-typed functions for global operations using the syntax $a = \oplus\{b\}$ since FORTRAN77 only supports this type of function. However, it is more convenient to write

$$\mathtt{A} = \oplus \{\mathtt{A}\}$$

than

$$\mathtt{call}\ \oplus\{A_{in}, A_{out}\}\qquad.$$

Therefore, using the nondestructive procedural interface, the user is forced to supply the additional array A_{in} even though the destructive use (i.e., read–write), would have been acceptable.

To allow a destructive subroutine interface, with the spinoff of a streamlined syntax, IPfortran supports the statement such as `In = Globul{In}` where the user has defined the subroutine `Globul`:

```
SUBROUTINE Globul(In,InOut)
Dimension In(1),InOut(1)

   InOut = In ⊕ InOut

RETURN
END
```

Recall the array `ithere` from Section 3.3. Its accumulation may be accomplished as follows.

```
izero=0
```

```
do j=0, numnode-1
   ithere(j)=0
enddo
ithere(thisnode)=i
ithere = intplusnd{ithere}
```

where the subroutine `intplusnd` is defined by

```
subroutine intplusnd(ithere,jthere)
dimension ithere(0:numnode-1), jthere(0:numnode-1)
do j=0, numnode-1
   jthere(j)=ithere(j)+jthere(j)
enddo
end
```

3.4.5 Length one arrays

The IPfortran translator generates a Fortran function interface and a subroutine interface for scalar or array parameters, respectively. Some care must be taken with regard to the interpretation of arrays of length one. In particular, note that `in(1)` denotes a scalar, whereas `in(1:1)` denotes an array (of length one). Similarly, if there is a statement such as `Dimension In(1)` then the parameter `In` (no parentheses) denotes an array. Thus it would not be correct usage to write `mysum{In}` for the function *mysum* defined above.

3.4.6 Intrinsic reduction operation

The accumulation of the values of the elements of an array into a target array of the same shape is a common operation worthy of an intrinsic function. We wish to write code for the following sum:

$$\forall i, j, \quad C(i)@j = \sum_{k=0}^{p-1} A(i)@k.$$

Using IPfortran we can write

```
do k = 0, P-1
  B = A@k
```

```
    do i = 1, n
      C(i) = B(i) + C(i)
    enddo
  enddo
```

Alternatively, we write the more concise expression using the IPfortran intrinsic reduction operator, +{ }, for example

```
      A = + { A }
```

Note that the result of the user-defined reduction

```
      result = mysum{scalar}
```

described earlier can be achieved using the intrinsic `result = +{scalar}` just as easily.

3.4.7 Reduction operation applications

The reduction of a set of distributed data by a binary operator is a common occurrence in scientific and engineering computations. Having a concise and intuitive notation streamlines programs, resulting in enhanced readability. Furthermore, by flagging the operation in a single statement, efficient algorithms can be implemented by the translator: these operations can be efficiently implemented in at most $log_2 p$ steps for p processors. We conclude this section with a some examples.

Consider a user defined reduction operation implementing the global `minmod` function. In solving discretized hyperbolic partial-differential equations, the flux of a quantity at a point is characterized by the smaller of finite-differences of that quantity in some vicinity of the point because near a shock the larger values may be unreliable. The binary function `minmod` is used to find the smaller value. It is defined as follows. Given two input values of the same sign, it returns the value with the minimum modulus (absolute value). When there is a sign disagreement between two values, zero is returned. (This corresponds to the case of a variable that cannot decide whether it is coming or going.) We define the scalar function `minmod` as

```
      REAL FUNCTION minmod(s1,s2)
      if (s1*s2 .gt. 0.0 .and. s1.lt.0.0) then
C       s1 and s2 are < 0
```

```
          minmod =  AMAX1(s1,s2)
        else if (s1*s2 .gt. 0.0)  then
C         s1 and s2 are > 0
          minmod = AMIN1(s1,s2)
        else
C        If s1 and s2 are not of the same sign or one is 0
          minmod = 0.0
        endif
        return
        end
```

To compute the `minmod` of a set of values `myslope` over all processors, one simply writes `slope = minmod{myslope}`.

A slightly more complicated application of a reduction operation arises in Gaussian elimination with row pivoting. Suppose an $m \times n$ matrix is stored by rows across P processors where row_i is assigned to processor $q = mod(i-1, P)$ and that processors are numbered $0, \ldots, P-1$. Define the space $\mathcal{X} \equiv \mathcal{Z}^+ \times \mathcal{R}^+$, where $\mathcal{Z}^+ \equiv$ *positive integers* and $\mathcal{R}^+ \equiv$ *non-negative real numbers*. This space represents the Cartesian product of *row numbers* with absolute values of *pivot entries* from the matrix. We now define a function $\mathcal{C} : \mathcal{X} \times \mathcal{X} \to \mathcal{X}$ where

$$\mathcal{C}((i,x),(j,y)) = \begin{cases} \text{(i,x)}, & \text{if } x > y \\ \text{(j,y)}, & \text{if } y > x \\ \text{(i,x)}, & \text{if } i < j,\ y = x \\ \text{(j,y)}, & \text{if } j < i,\ y = x \\ \text{(i,x)} = \text{(j,y)}, & \text{if } i = j,\ y = x. \end{cases}$$

Thus, $\mathcal{C}$ is commutative and associative and can be utilized in a reduction operation. The following subroutine illustrates the use and definition of $\mathcal{C}$ to calculate a row-pivot element among P processors.

```
       SUBROUTINE pivot(B,m,n)
       IMPLICIT NONE
       PARAMETER (size=64,minp=1)
c the rows for this processor
       REAL*8 B(size/minp,size)
```

```
c matrix dimensions
       INTEGER m,n
       INTEGER row,i
c cycle in Gaussian Elimination
       INTEGER k
    c  pivot(1) has the value, pivot(2) the row number as a real
       REAL*8 pivot(2),pivot2(2)

    c   LOOP OVER COLUMNS OF MATRIX
        DO 500 k = 1, n
    c  basis: determine pivot row from local rows
          pivot(1) = B(1,k)
          pivot(2) = thisnode+1
          do 20 irow = 2, m/numnode
            row = thisnode + 1 + (irow-1)*numnode
            pivot2(1) = B(irow,k)
            pivot2(2) = row
            call Cpivot(pivot2,pivot)
     20   continue

     c determine pivot row globally
          call Cpivot {pivot(1:2)}

     c output results
          print* ,thisnode, k, INT(pivot(2)), pivot(1)
   500 continue
       END
```

Note that we could have equivalently used the statement

```
call Cpivot {pivot}
```

above, as `pivot` has been dimensioned to be of length 2. The definition of `Cpivot` is as follows.

```
SUBROUTINE Cpivot(p1,p2)
REAL* 8 p1(2),p2(2)
```

```
c  calculate a pivot where p1 and p2 contain
c  row(i,k) element and  value k
c  output tuple goes to p2
        INTEGER* 4 i,j
        REAL* 4 v,w
        v = p1(1)
        i = p1(2)
        w = p2(1)
        j = p2(2)
        if ( v .gt. w ) then
          p2(1) = p1(1)
          p2(2) = p1(2)
        else if ( v .eq. w ) then
          if ( i.eq.j ) then
            print*,'Error p1=p2'
          else if ( i .lt. j ) then
            p2(1) = p1(1)
            P2(2) = p1(2)
          endif
        endif
        RETURN
        END
```

Quite complex commutative operators can be devised. Consider the following approach to computing a matrix times a vector, `matvec`, that uses the reducing version of `addvec`, a subroutine that adds vectors. For this new data type, we define *addition* by introducing a subroutine that adds two vectors and returns their sum:

```
        SUBROUTINE addvec(a,b,c,D)
        dimension a(1),b(1),c(1)
        DO 1 i=1,D
  1             c(i) = a(i) + b(i)
        RETURN
        END
```

We now define matrix multiplication using the reducing version of `addvec`. In the following, note that D is the dimension of the matrix, and $N << D$ is the number of columns stored in each node. The index $global(i)$, `i`= $1, \ldots,$`N`, indicates how the columns are distributed across processors, but the details of a particular choice of such a distribution is left unspecified. One such choice is defined in the following:

$$\text{global(j)=THISNODE} + \text{j} * \text{NN}$$

where $NN = D/N$. The matrix is assumed to be distributed in a similar way.

```
          SUBROUTINE matvec(res, mat, vec, D, N)
          dimension res(N), mat(D,N),vec(N), s(1024),t(N)
          DO 1 i=1, D
              s(i) = 0.0
              DO 1 j=1, N
                  s(i) = s(i) + mat(i,j) * vec(j)
1           continue
           call addvec{s,t,D}
           DO 2 i=1,N
2            res(i) = t(global(i))
```

Note that the function `addvec` is clearly associative and commutative; its global version adds all the components among all processors.

3.4.8 Compilation of reduction operations

The optimal implementation of a particular reduction operation will depend, in general, on the size of the data being exchanged. For example, the addition of large vectors

```
A = + { B(1:N) }
```

on a hypercube message-passing system can be done more efficiently by recursively subdiving the vector and adding parts of the vector at each step when `N` is sufficiently large [43]. (However, for small `N` this approach is suboptimal [103].)

For a user-defined operation such as

```
A = myvecsum{ B(1:N) }
```

it is beyond the scope of current compilation techniques to attempt to infer whether `B(1:N)` could be segmented and still give a correct result. Thus the IPfortran compiler takes a conservative approach when compiling user-defined reduction operations.

However, for operations such as + which are defined in the language, a more efficient implementation can be used.

3.5 Arrays

In a distributed computing model where parallelism is expressed explicitly, it is essential to have some way of expressing operations on entire arrays or sections of arrays. IPfortran suppplements FORTRAN77 array notation with a subset of Fortran 90 syntax. Array names may be used in three settings to reference

1. a single scalar $A(i,j)$
2. a section $A(:,j)$
3. the entire array A.

In IPfortran, any of these three references can be used in expressions where @ is used or as actual parameters to reducing procedure invocations; the array extensions are not recognized outside of expressions involving @ or reducing function references.

3.5.1 Arrays in assignments

IPfortran permits arrays to be referenced in assignments of the type $A(m:n) = B(m:n)@p$. In the assignment $A(2:4)@q = A(1:3)@p$, if $p = q$, then A refers to the same array, and the subarrays, therefore, share storage. The assignment to the left-hand side is made using the values of $A(1:3)$ prior to the assignment. If $A(1:4) \equiv (1,2,3,4)$ then after the assignment $A(1:4) \equiv (1,1,2,3)$ and not $(1,1,1,1)$. These semantics come to bear in, for example, broadcasts where $A(2:4) = A(1:3)@p$ results in the assignment $A(2:4) = A(1:3)$ at processor p.

3.5.2 Arrays as parameters

A common programming style in Fortran uses procedure interfaces to rename arrays or to change the offset into an array. With IPfortran global functions and procedures, the subscripted references to the actual parameters $A(i)$ and $B(j)$ are treated as **scalar** references: there is no guarantee that the actual parameters generated by IPfortran will be arrays.

The following subroutine definition and use, where A and B are rank-1 arrays, will result in incorrect code: `call myglob{A(ioff),B(ioff),n}` where the definition is

```
SUBROUTINE myglob(in1,in2,out,n)
REAL in1(1), in2(1), out(1)
INTEGER n
DO i = 1, n
 out(i) = in1(i) + in2(i)
RETURN
END
```

The functionality that the preceding global definition incorrectly attempts, can be accomplished with

```
SUBROUTINE myglob(in1,in2,out,n,off)
REAL in1(1), in2(1), out(1)
INTEGER n,off
DO i = off, n+off
 out(i) = in1(i) + in2(i)
RETURN
END
```

with the reduction operation invocation `call myglob{A,B,n,ioff}`. (With future versions of IPfortran, these semantics may be replaced with those standard to Fortran.)

While IPfortran does not enforce conformance across procedural interfaces, the formal parameter at the very least will take the shape into account: Conformance across the reducing procedure interface is managed through copying array sections. Shape information is not available in the called routine in the present version of IPfortran. As a parameter in a reducing function or procedure, the array section will be copied into a rank 1 array in array-element order corresponding to the shape of the array section. In effect, this rank 1 array will have the characteristics of the array section referenced as the actual parameter.

Consider, for example, the array declared as DIMENSION $A(3,3)$. The reference A, which is equivalent to $A(:,:)$ and $A(1:3,1:3)$, is **not** equivalent to $A(1:2,1:3)$; if $A(3,3) = /1,2,3,4,5,6,7,8,9/$ then the elements of $A(1:2,1:3)$ in array element order are $/1,2,4,5,7,8/$. IPfortran assumes that the procedures being reduced via {} are independent of such index shifts.

3.5.3 Performance consideration: Array sections and the communication system

The use of array sections may add a copy overhead to the compiled IPfortran program. To transmit a block of data to another processor, the communication subsystem is given an address of a data buffer, a byte count and a destination address. The communication subsystem, expecting to find the data contiguous in memory, will either copy this to a system buffer, or send the data to the destination processor directly from the user's address space.

Since the array section is a portion of some parent array, it may not be contiguous in memory. For sections derived from $rank \geq 2$ arrays, the array section may be copied into a buffer by the application in order to achieve contiguous storage, although this is done automatically by the IPfortran translator.

3.6 I/O

In this section, parallel I/O in the IPfortran programming environment is introduced. Multiple processors, when accessing the same file, introduce complications beyond the usual language and operating system dependent features of single processor applications. With multiprocessor systems, the options for distributing the data to processors are numerous. Often, answers to the I/O question are not clear, a priori.

A significant complicating factor is that I/O system of a given computer model can vary dramatically depending on the intended use. One system may have only one disk subsystem attached to a single processor, whereas the same model from the same company could have the same disk subsystem attached to every processor at another site. Thus the I/O system cannot be modeled in a simple way that is universal for all computers of a given type.

In order to find a suitable I/O strategy, both the application and the computer architecture must be taken into account. Numerous questions can be asked. Will the application tolerate serial access to the file, or does that result in an unreasonably high IO-to-computation ratio? A 128-node computation, effectively parallelized and with a large data set, might require a more sophisticated I/O strategy. If so, how will the disk subsystem perform with simultaneous accesses to the same file? Should one perform concurrent I/O from the same file, or should a single processor access files, using the communication network to distribute the data to other processors? The use of the communication network to distribute the data from a single process might be the preferred method. For outputting, is post sorting of data feasible, or can the application tolerate seeks to the same file by multiple processors in order to write the data in place, or should a single processor collect the data and write to the file?

In the remainder of this section, three parallel I/O strategies will be discussed within the IPfortran program model. Recall, IPfortran assumes a separate namespace and a separate address space for each process, requiring that data be explicitly assigned to all nodes. We will examine, with examples, the parallel I/O methods described above:

- a single node reading the data from disk and communicating that data to the other nodes,
- all nodes directly reading data from disk,
- all nodes reading data concurrently.

In each case, the IPfortran program model, very close in form to "node code," requires that data be distributed explicitly as a function of the logical processor identifier, *Thisnode*. The examples illustrate P processors, reading from the file, *coord*, into the array, X, with the N data accessed, later in the calculation, by a blockwise distribution. The time for I/O to be performed from a single process is T_s.

3.6.1 Serial I/O from Disk

This method serializes multiprocess access to the file. Processes can use direct access methods to read their assigned data, or they can read through the entire file, discarding unnecessary data. This method can be unacceptably slow for many process applications. In the sample code section that follows, the serial I/O requires time, $\mathcal{O}(P \times T_s)$, and space, $\mathcal{O}(N/P)$.

```
      do p = 0, numnode-1
        if (p.EQ.thisnode) then
          OpenFile(coord,unit)
c
c skip data
          do b = 0, thisnode-1
            READ(unit,*) (X(i),i=1,N/P)
          enddo
c
c get data block for calculation
          READ(unit,*) (X(i),i=1,N/P)
c
c Each file closes its unit, conserving file
c descriptors, a finite system resource.
          CLOSE(unit)
        endif
c
c This serializes the disk file access. See example below.
        call synchronize
      enddo
```

The synchronize subroutine may be a call to a machine dependent synchronization routine, for example, the Intel iPSC subroutine, gsync(). Alternatively, a variation of the baton passing algorithm from Section 4.1 can be used. A IPfortran synchronization algorithm is given below. This algorithm guarantees that processes will not proceed beyond the call until all other processes have entered this point.

```
      subroutine mysync()
      integer p,src,dst,baton
      do src=0,2*(numnode-1)
         dst = mod(src+1,numnode)
         baton@dst = baton@src
      enddo
      return
      end
```

A more efficient algorithm uses a dimensional exchange.

```
subroutine mysync2()
integer baton(1)
baton = +{baton}
return
end
```

Note that a user defined *noop* function could be used in place of the global sum.

3.6.2 Single Process I/O from Disk

This method assigns the I/O task to a single process. The process used should be either process 0, or numnode-1, in order that the number of processors can vary. The single process will read the data from disk and distribute it, using the computer's communication network. Ideally, data will be distributed in blocks, to reduce latency, and those blocks will be distributed as they are read from disk, to reduce the memory requirements of the I/O process.

The following code segment performs I/O in space, N/P, and time

$$T_s + \alpha(N - N/P) + \beta(P - 1),$$

where α and β are machine-dependent constants for throughput and latency, respectively.

```
if (thisnode.EQ.numnode-1) OpenFile(coord,unit)
do p = 0, numnode-1
  if (thisnode.EQ.numnode-1) READ(unit,*) (X(i),i=1,N/P)
  X(1:N/P)@p = X(1:N/P)@(numnode-1)
enddo
if (p.EQ.thisnode) CLOSE(unit)
```

3.6.3 Concurrent I/O

With concurrent I/O, processes read data simultaneously from disk. Intel provides support for this via their Concurrent File System. At a first glance, this technique appears to be the preferable method, if there are multiple disks and controllers. However, disk contention could serialize I/O at the disk if data is not carefully laid out. The following code fragment illustrating concurrent I/O requires space, N/P, and time, AT_s, where A is some machine and application dependent parameter, and file positioning is assumed to be free.

```
call OpenFile(unit,coord)
call PositionFile(unit,(N/P)*thisnode-1)
READ(unit,*) (X(i), i=1,N/P)
CLOSE(unit)
```

3.6.4 Intermediate I/O

By intermediate I/O, we mean the bits of I/O from occasional print statements, usually of a non-critical, diagnostic flavor. This is I/O that takes place after program initialization, but before termination where large amounts of I/O may be required. It could be be the outputting of a single energy value or an entire configuration, in the case of particle dynamics. Intermediate I/O should be blocked so that I/O buffers are well utilized and latency effects are reduced.

Chapter 4

Applying IPfortran to Kernels

4.1 Basic examples

In this section, some fundamental parallel algorithms cast in IPfortran are presented.

4.1.1 Baton passing

Baton passing derives its name from a relay where a team of runners run in some sequence chosen by their coach, with the current runner in possession of the single baton. Each runner, except for the first, will wait their turn for the pass of the baton, which signals a runner's turn to snap into action.

Baton passing is a way to order processes, so they can, for example, perform some activity in some order. The following program segment results in the processes outputting "hello world" according to their logical ordering.

```
  SUBROUTINE Baton
  INTEGER baton
  if (ThisNode.eq.0)
+      print*, ´hello from´,ThisNode
  baton = 1
  do i=1, NumNode-1
     baton@i = baton@(i-1)
     if (baton.eq.ThisNode)
       print*, ´hello from´,ThisNode
       baton = baton +1
     endif
```

```
enddo
END
```

4.1.2 Column oriented LU decomposition

This example illustrates prudent application of cyclic assignment for an LU decomposition algorithm. Geist and Romine [44] describe the following algorithm for column-to-processor assignment of a matrix using *LU decomposition* on an $n \times n$ matrix.

```
for k = 0, n-1
 if (I own column k) then
  for i = k+1, n-1
     l(i,k) = a(i,k) /  a(k,k)
  end for
  broadcast l
 else
  receive l
  for (all columns j>k that I own)
   for i = k+1 to n-1
    a(i,j) = a(i,j) - l(i,k) * a(k,j)
   end for
  end for
 end if
end for
```

The natural unit of decomposition, columns of an $m \times n$ matrix, leads to an efficient implementation of this algorithm using *cyclic assignment*; $column_j$ belongs to processor, p, where $p = mod(column_j, numnode)$ and $0 \leq column_j < n$. An alternative decomposition, *contiguous assignment*, leads to increased processor idling since the lower indexed columns are eliminated first.

In the IPfortran *LU decomposition* example that follows, both the data and iteration distribution are illustrated. The two are intimately related, and depend of course on the arrangement of data in memory.

```
  REAL*8 a(64), m(64)
C distribute columns cyclically
  do  i = 1, 64
    if (thisnode.eq.mod(i,numnode)
 .      call getcolumn(i,a)
  enddo
  do k = 1, N-1
C owner computes column of multipliers
    if (thisnode.eq.mod(k,numnode)) then
     do i = k+1, n-1
       m(i,k) = a(i,k)/a(k,k)
     enddo
    endif
C broadcast column of multipliers
    m(k+1:N) = m(k+1:N)@mod(k, numnode)
C reduce columns
    do i = k+1,N
      if (mod(i,numnode).eq.thisnode) then
        do j = k+1, N
         a(j,i) = a(i,j) - m(j)*a(k,j)
         enddo
      endif
      a(i,j) = a(i,j) - m(i)*a(k,j)
    enddo
  enddo
```

4.1.3 Grid problems

Grid problems arise frequently in approximating the solution of differential equations via finite difference, finite element or integral equation methods. The simplest example results in the simple iteration

```
 REAL*8 newsoln(1000000), solution(1000000), data(1000000)
 INTEGER  bignum
 bignum = 1000000
```

```
delta_x = 1.0/float(bignum - 1)
dxsq = delta_x**2
solution(1) = 0.0
solution(bignum) = 0.0
do i=2, bignum - 1
  newsoln(i) = (solution(i-1)+solution(i+1)-dxsq*data(i))*0.5
enddo
do i=2, bignum - 1
  solution(i) =  newsoln(i)
enddo
```

This is the Jacobi iterative method applied to a standard finite difference method for the two-point boundary value problem

$$\begin{aligned} -u'' = f \quad on \quad [0,1] \\ u(0) = u(1) = 0. \end{aligned}$$

It is a very inefficient method, but it serves to illustrate the necessary communication statements.

We distribute by blocks both the work and the data for the loop.[1] Let us assume that $bignum - 1$ is divisible by $numnode$ and let $k := (bignum - 1)/numnode$. Let x denote either *solution* or *newsoln*. We let z denote a local copy of either *solution* or *newsoln*, with $z(1)$ and $z(k+1)$ reserved to hold values that will be needed from other processors. The above loop becomes

```
REAL*8 newz(10000), oldz(10000), locdata(10000)
INTEGER  bignum
bignum = 1000000
k = (bignum-1)/numnode
delta_x = 1.0/float(bignum - 1)
dxsq = delta_x**2
do j= numnode-1,1,-1
   oldz(1)@j = oldz(k)@(j-1)
enddo
do j=0, numnode-2
```

[1]Block–wise distributions are described in Chapter 6.

```
    oldz(k+1)@j = oldz(2)@(j+1)
enddo
do i=2, k
    newz(i) = ( oldz(i-1) + oldz(i+1) - dxsq*data(i) )*0.5
enddo
do i=2, k
    oldz(i) =  newz(i)
enddo
```

Note the reversed order of traversal in the loop

```
do j= numnode-1,1,-1
    oldz(1)@j = oldz(k)@(j-1)
enddo
```

This is because the IPfortran compiler implements communication using the *golden rule* on message-passing systems: first a processor sends data to other processors that need it, then it receives the data that it needs. The loop in this order allows the necessary sends and receives to occur simultaneously. In the reverse order, viz.

```
do j= 1, numnode-1
    oldz(1)@j = oldz(k)@(j-1)
enddo
```

all send-receives will occur sequentially. P1 (processor 1) will not send data to P2 until it receives data from P0, and P2 will not send data to P3 until it receives data from P1, and so on.

If functions of *thisnode* are correctly implemented, an alternate implementation is as follows.

```
REAL*8 newz(10000), oldz(10000), locdata(10000)
INTEGER  bignum
bignum = 1000000
k = (bignum-1)/numnode
delta_x = 1.0/float(bignum - 1)
dxsq = delta_x**2
```

```
oldz(1) = newz(k)@max((thisnode-1),0)
oldz(k+1) = newz(1)@min((thisnode+1),numnode-1)
do i=2, k
   newz(i) = ( oldz(i-1) + oldz(i+1) - dxsq*data(i) )*0.5
enddo
do i=2, k
   oldz(i) =  newz(i)
enddo
```

4.1.4 Dimensional exchange

In this section we take a look at the hypercube dimensional exchange implemented by the ℙfortran translator. For a hypercube, it generates the following code for $A = op\{B\}$:

```
      real*4 o10000
      real*4 o10001
      integer*4 o10002
      integer*4 o10003
Cpf      a = op{b}
      a=b
      o10002=1
      do 10 o10003 = 1, cubedim
         call pf_inctag()
         call pf_snd(xor(thisnode,o10002),a,4)
         call pf_rcv(o10000,4)
         a=op(o10000,a)
         o10002=o10002 * 2
 10   continue
```

where $cubedim = log_2 P$. Here, *pf_inctag*, *pf_snd* and *pf_rcv* are routines that are system dependent and implement tag-incrementing, message-sending and message-receiving, respectively. This is probably isomorphic to what one would write by hand. Without the reduction operations and using the @ one could write

```
      k = 1
      do 60 d = 1, cubedim
        do 50 p = 0,numnode-1
          dst = xor(p,k)
          tmp@dst = a@p
          if (dst.eq.thisnode) then
            a = op(tmp,a)
          endif
50      continue
        k = k * 2
60    continue
```

Another loop is introduced in this algorithm adding an $\mathcal{O}(P)$ amount of work. We see that @ alone is not enough to write efficient algorithms, but coupled with reduction operations implemented via dimensional exchange (or other algorithm well suited to a given network topology, such as a mesh) it is usually enough.

A *fan-out* is another common operation well suited to hypercube topology. Here a single processor, say 0, sends a datum to processor 1. In the next step, processor 0 sends to processor 2, and processor 1 sends to processor 3, and so on. In $log_2 P$ steps all processors will have the datum initially at processor 0. In other words, processor 0 broadcasts its datum to all other processors. IPfortran uses the broadcast mechanism supplied by the communication system (when available) for $a = b@p$.

The inverse of fan-out, *fan-in*, reduces data from a set of processes to a datum at a single processor. The reduction operation in IPfortran accomplishes this, but with all processors participating in the exchange. While we intend eventually to produce code for syntax such as `a=op{b@(i:j)}` in IPfortran, to communicate efficiently among an arbitrary subset of processes, for now only `a=op{b}` is supported.

Using IPfortran *as it stands*, one can design algorithms using the standard reduction operation simply by assigning null data to the non-participating processors' variable. We consider an algorithm that accumulates values from a set of processors. In the following loop, let $A(0 : P-1)$ indicate a processor's participation in a fan-in or dimensional exchange; if $A(i) = 1$ then processor i will contribute its data.

```
      if (A(thisnode).ne.1) then
        contribute = 0.E0
```

```
else
  contribute = value
endif
accumulator = +{contribute}
```

For a *fan-in* to a single processor, p, one can write a@p = +{value}. For a hypercube topology, a slightly more efficient algorithm exists for a fan-in than the general algorithm (currently dimensional exchange) for a = +{value}. However, it is not used at the moment; a@p = +{value} is computed simply by computing a = +{value} and then specializing the assignment only to processor p.

If the IPfortran mechanisms cannot effectively express what one wishes to do, then the low level *sends* and *receives* can be used directly, and if appropriate, may be interspersed with IPfortran statements.

4.1.5 Gaussian elimination revisited

We now consider a more serious application code to solve linear equations by Gaussian elimination (no pivoting) using an algorithm for distributed memory computers which allows the overlapping of communication and computation. This algorithm is quite similar to one described in [44]. We defer to that article for motivation and explanation of the algorithm. In particular, we present the "pipelined" algorithm for column storage. Included in the code is a test of the accuracy of algorithm using a matrix `a` with a known factorization (given by `b` in the code). Some care must be taken for large matrix sizes as the generation of `a` and `b` involves exponentially small quantities and underflow can result.

```
      real*8 a(1024,64), b(1024,64), m(1024), nextm(1024)
      real*8 ainv, above, mflops, err

      integer*4 t

      integer*4 i,j,k,l

      ipasign(j,NN) = mod(j-1, NN)
C    processor assigned to store global column no. j  of the matrix

      iglobal(TN,j,NN) = TN + 1  + (j-1)*NN
C     global column address of the j-th local column held by thisnode

      local(j,NN) = ((j-1) / NN) + 1
```

```
C     local column address (in proc.  ipasign(.,j,.)) of global
C             column no. j  in the matrix

        matdim = 64
        nocols = matdim/numnode

 c      write(6,101)  thisnode
101     format(´ processor ´, I4, ´ beginning initialization ´)
        do 20 i=1,matdim
             do 10 j=1,nocols
                  a(i,j) = 2**(-IABS(i-iglobal(thisnode,j,numnode)))
                  if( i .eq. 1  .or.
     1                 iglobal(thisnode,j,numnode) .le. i ) then
                       b(i,j) = a(i,j)
                  else
                       b(i,j) = 0.75*a(i,j)
                  end if
10           continue
20      continue

        t = mclock()

        if ( thisnode .eq. ipasign(1,numnode) ) then
C   this processor computes first set of "multipliers" for Gauss elimination
        write(6,103)  thisnode
103     format(´ processor ´, I4, ´ computing multipliers at step 0´)
            l = local(1,numnode)
            ainv = 1.0/a(1,l)

            do 30 i=2,matdim
                  a(i,l) = a(i,l) * ainv
30                m(i) = a(i,l)

        end if

        m(2:matdim) = m(2:matdim)@(ipasign(1,numnode))

        do 80 k=1,matdim-2
             if (thisnode .eq. ipasign(k+1,numnode) ) then
C   this processor computes next set of "multipliers" for Gauss elimination
        write(6,104)  thisnode,k
104     format(´ processor ´, I4, ´ computing multipliers at step´,I4)
                  l = local(k+1,numnode)
                  above = a(k,l)
                  do 40 i=k+1,matdim
40                     a(i,l) = a(i,l) - m(i)*above

                  ainv = 1.0/a(k+1,l)

                  do 50 i=k+2,matdim
                       a(i,l) = a(i,l) * ainv
50                     nextm(i) = a(i,l)
```

```
              end if

             nextm(k+2:matdim) = nextm(k+2:matdim)@(ipasign(k+1,numnode))

             jstart = local(k+2,numnode)
      if( ipasign(k+2,numnode) .gt. thisnode) jstart = jstart + 1
      if( jstart .le. nocols) then
               do 60 j = jstart,nocols
                  above = a(k,j)
                  do 60 i=k+1,matdim
                      a(i,j) = a(i,j) - m(i)*above
60             continue
             end if
             do 70 i=k+2,matdim
70                m(i) = nextm(i)

80       continue
         if(thisnode .eq. ipasign(matdim,numnode)) then
       a(matdim,nocols) = a(matdim,nocols) - m(matdim)*a(matdim-1,nocols)
         end if

c        write(6,111)  thisnode
111      format(´ processor ´, I4, ´ finished factoring ´)
         t = mclock() - t

         mflops = float(matdim*matdim*matdim)/(3000*t)

C   Check that the answer is correct
         err = 0.0
         do 90 i=1,matdim
              do 90 j=1,nocols
90                 err = dmax1( err , dabs(b(i,j) - a(i,j)) )


         write(6,100)  thisnode, mflops, err
 100     format(´ processor ´, I4, ´ achieved ´, E13.3,
     1             ´ Mflops with an error of ´, E13.3)
         END
```

The overlapping of communication and computation occurs because the variable `nextm` (which holds the precomputation of the multipliers for the next, not the current, step of elimination) is available long before it is needed (nearly one full step of elimination in advance). This can be seen from the text of the code, as there are several lines of code after the definition of `nextm` in statement 50 before its use in statement 70.

A further transformation can be applied to this code whereby the send corresponding to *nextm* is done as soon as it is possible (after statement 50 in this case) and the receive is postponed until as late as possible (before statement 70 in this case). At the moment, the send/receive pair appears adjacently in the code generated by the translator for message-passing systems. The code generated by the IPfortran translator for

```
nextm(k+2:matdim) = nextm(k+2:matdim)@(ipasign(k+1,numnode))
```

is similar to what appears in section 4.1.4 on dimensional exchange and will not be repeated here.

4.2 Dining philosophers problem

With this example we veer from our numerical-applications trajectory and apply IPfortran to a classic problem in concurrency control. Our objective is not to advocate IPfortran for this type of parallel programming problem. Instead we include this example to show how the inner workings of IPfortran can be used to achieve things beyond the obvious. We do so in the spirit that Fortran is used by some as a macro-processor. More precisely, we are referring to situations when one uses not only the formal properties of the language but also knowledge of how its compiler works. Thus we will show how the IPfortran translator works to some degree by showing how it can be used to implement a procedural-parallelism solution to the dining philosophers problem (*a complete program text is included at the end of this section*).

The dining philosophers are a group of five philosophers who from time-to-time compete for chopsticks [33]. The philosophers spend their time alternating between eating and thinking. When, during thinking, hunger strikes, the philosopher will enter the dining room and take a position at a round table. On this table surrounded by five chairs, are five plates and five chopsticks. A plate is located at each of the five positions with a single chopstick placed in the spaces between plates. In order to eat, a philosopher must have two chops in hand, taken from either side of the plate. It is easy to see that problems could arise if all philosophers sat down simultaneously and picked up the chopstick to their right; this is just one of many potential *deadlock* scenarios. If the philosophers are not going to starve, then at least one must relent, put down a chopstick, and try again later.

Solutions to the dining philosophers problem pose clever ways to manage the shared and finite resource of chopsticks such that deadlock will not occur and such that no philosopher will starve to death (bounded waiting). There are analogous cases with operating systems where processes may compete for a finite resource, such as a cpu or a printer. Counterparts also occur with numerical programming where processes may be dedicated to performing some specialized task such as sorting output. Strictly speaking, the SPMD model does not encompass this procedural-parallelism programming model. With the SPMD model, all processors execute the same code, with variations in branches of `if` statements or number of iterations for a loop, for instance. In our implementation of IPfortran we take the SPMD model to heart and require the symmetric execution of communication statements in order to produce correct message passing code.

Various concurrency constructs and their ability to deal with the subtleties of deadlock and progress have been tested against the dining philosophers problem – monitors, semaphores, Hoare's CSP, to name a few. We present a IPfortran solution whereby ten processes implement the five chopsticks and five philosophers.

Staying within the SPMD model, we use a single program. Chopstick processes will execute one portion of the program, philosopher processes the other. In their respective code sections, each philosopher will move between thinking, becoming hungry and trying to acquire the two chopsticks alongside their plate, and upon acquiring two chopsticks, eating. The chopstick processes will take requests from the philosophers, but will only allow a single philosopher to use them at any time. Two problems challenge an SPMD solution: symmetry is violated and philosophers are hungry at arbitrary intervals.

Chopsticks and philosophers communicate, but from separate program sections. So how can we guarantee that messages will be read by the respective recipient? Correct communication in IPfortran requires that each *send* be matched with a corresponding *receive*, greatly simplifying the programmer's task. The sends and receives originate from the @, which are placed both where data is needed and where it is available. Since messages may be buffered, we need some way of labeling or tagging them. Herein lies an important ramification of the asynchronous SPMD model.

It should not be a secret that processes can astutely exchange data on a message passing machine with complete confidence about message origins in the program and at what incarnation of a cycle or cycles, through *message tagging*. With each send/receive pair generated by IPfortran, there is an accompanying tag incrementing operation. The tag is global to the program and is incremented once at each communication statement. The tag becomes part of a message sent and will be used by a receiving process to request the correct message for its point in execution. This allows processes to proceed asynchronously. For the scheme to work in the current version of IPfortran, however, *all* processes must execute *all* tag-increment operations.

If a process should miss a tag increment, then all bets are off. This is why symmetry is so important and why the SPMD model is so appealing. We're fortunate that scientific applications fit so well into this model. The solution to the dining philosophers problem that we've outlined violates this symmetry, but since we know about message-tags, we can construct corresponding code sections for chopsticks and philosophers where the tags are incremented, but in different statements; chopsticks increment in their subroutines, and philosophers increment in their subroutines.

The program segment that follows uses the pair of communication statements

$$\text{message@bamboo} = \text{activity@human}$$
$$\text{message@human} = \text{reply@bamboo}$$

for philosopher to chopstick and chopstick to philosopher communication in the philosopher subroutine.

The left-hand side of the first statement and the right-hand side of the second statement are sleights of hand crafted for the IPfortran translator. The values of *message* and *reply* at *bamboo* in this subroutine (i.e. at this place in the code) are undefined – the chopstick processes never enter this subroutine. This is fine because in the chopstick subroutines, another pair of IPfortran statements have equally bogus right-hand and left-hand sides ostensibly making data assignments from variables within the chopstick subroutine.

Using the SPMD, data-parallel model to simulate a style of programming involving procedural parallelism has resulted in some obscure code. In addition, we are using a deterministic model for a fundamentally non-deterministic problem. Our dining philosopher solution works, however, because the programmer built in the necessary symmetry with an understanding of message tags. The key is that the message-number (tag) identifies the message needed for a correct algorithm. And what about the sends and receives that are generated with the undefined data? IPfortran generates the following code for the two statements in the philosopher subroutine:

```
Cpf message@bamboo = activity@human
 call pf_inctag()
 if (TN.eq.bamboo.and.TN.eq.human) then
    message=activity
 else if (human.eq.TN) then
    call pf_snd(bamboo,activity,4)
 else if (bamboo.eq.TN) then
    call pf_rcv(message,4)
 endif
Cpf message@human = reply@bamboo
 call pf_inctag()
 if (TN.eq.human.and.TN.eq.bamboo) then
     message=reply
 else if (bamboo.eq.TN) then
     call pf_snd(human,reply,4)
 else if (human.eq.TN) then
     call pf_rcv(message,4)
 endif
```

In the first translation, the receive by bamboo (chopstick) will never be executed, here. And likewise, the send in the second translation will always be skipped. However, the other send and receive will be executed as we want.

If one takes the straight and narrow with IPfortran, these gymnastics are not of concern. This shows, however, why one should not write the statement

```
if(ThisNode.eq.p) a@p=b@q
```

without good reason. We have explored a limitation of this programming model and its implementation; while a programming style using procedural parallelism can be expressed, it is unnatural. Moreover, our solution to the dining philosophers problem falls short of the mark. There is progress without starvation and deadlock is avoided, but not without a price. In order to keep message tags in sync with the chopsticks, the philosophers are required to participate in table activities even while thinking, a violation of the original problem as formulated by Dijkstra.

The IPfortran listing for the dining philosophers problem follows.

```
      PROGRAM Philosophize
c     Dining Philosophers Problem using {\Plang}
c
      if (thisnode .lt. 5) call philosopher(thisnode)
      if (thisnode .gt. 4.and.thisnode.lt.10) then
          call chopstick(thisnode)
      endif
      stop
      END

      SUBROUTINE go_eat(activity,iseed)
      IMPLICIT NONE
      REAL fate, enough
      INTEGER activity,iseed
      PARAMETER(enough=0.50)
      INTEGER orate,null_message,busy,not_busy,
     . hungry,putdown,thinks,eats,true,false
      PARAMETER(orate=1, null_message=0, busy=1,
     . not_busy=2, hungry=3, putdown=6, thinks=7,
     . eats=8, true=1, false= 0)
       if (fate(iseed) .gt. ENOUGH) then
           activity = putdown
       else
           activity = eats
       endif
       if (orate.eq.1.and.activity.eq.putdown) then
          print*,´ I am nourished and ready to THINK´,
      .         ´ Philosopher ´,thisnode
       else if (orate.eq.1.and.activity.eq.eats) then
         print,´ I continue to EAT´,
      .  ´ Philosopher ´,thisnode
      endif
      RETURN
      END

      SUBROUTINE go_think(activity,iseed)
      IMPLICIT NONE
      REAL fate,enough
      INTEGER activity, iseed
      INTEGER orate,null_message,busy,not_busy,
      .  hungry, putdown, thinks, eats,true,false
      PARAMETER(ENOUGH=0.50)
      PARAMETER(orate=1, null_message=0, busy=1,
     . not_busy=2, hungry=3, putdown=6, thinks=7,
     . eats=8, true=1, false= 0)
      if (fate(iseed).gt.enough) then
          activity = hungry
      else
          activity = thinks
      endif
```

```
      if (orate.eq.1.and.activity.eq.hungry) then
        print*,´by virtue of wisdom I perceive HUNGER´,
     .      ´  philosopher ´,thisnode
      endif
      RETURN
      END

      REAL FUNCTION fate(itn)
      IMPLICIT NONE
      REAL a, ran
      INTEGER itn
      fate = ran(itn)
      RETURN
      END


      SUBROUTINE philosopher(human)
      IMPLICIT NONE
C     node number of this philosopher
      INTEGER human
C     left and right chopsticks
      INTEGER bamboo1,bamboo0
C     current chopstick index
      INTEGER bamboo
C     chopstick count, must be 2 to eat
      INTEGER chopsticks
      DATA chopsticks/0/
      INTEGER message,reply
      INTEGER activity
      INTEGER orate,null_message,busy,not_busy,
     . hungry,putdown,thinks,eats,true,false
      PARAMETER(orate=1, null_message=0, busy=1,
     . not_busy=2, hungry=3, putdown=6, thinks=7,
     . eats=8, true=1, false= 0)
      INTEGER debug,iseed
      data debug/.false./
      data activity/thinks/
      iseed = human
      bamboo = human + 5
      bamboo0 = human + 5
      bamboo1 = MOD( ( human + 1 ),5) + 5

c forever a philosopher
100 continue
       if ( bamboo .eq. bamboo0 ) then
             bamboo = bamboo1
       else
             bamboo = bamboo0
       endif

       message@bamboo = activity@human
       message@human = reply@bamboo
```

```
    if ( activity.eq.thinks) then
       call go_think( activity, iseed )
    else if (activity.eq.hungry) then
       if (message.eq.not_busy) then
           chopsticks = chopsticks + 1
       else
           chopsticks = 0
       endif
       if (orate.eq.1 .and.chopsticks.eq.2) then
         print,'Philosopher ',human,' eats now'
       endif
       if (chopsticks .eq. 2 ) activity = eats
       if (chopsticks .eq. 1 ) activity = hungry
       if (chopsticks .eq. 0 ) activity = putdown
    else if (activity.eq.eats) then
       call go_eat( activity, iseed )
    else if (activity.eq.putdown) then
       if (chopsticks.eq.2) then
            chopsticks = chopsticks - 1
       else
            chopsticks = 0
       endif
       if (chopsticks .eq. 1 ) activity = putdown
       if (chopsticks .eq. 0 ) activity = thinks
    endif
    goto 100
C   for philosophers lifetime
    RETURN
    END


   SUBROUTINE chopstick(bamboo)
   IMPLICIT NONE
   INTEGER bamboo
   INTEGER message,activity,reply
C  philosopher adjacent to this chopstick
   INTEGER human
   INTEGER human0,human1
   INTEGER orate,null_message,busy,not_busy,
 .         hungry,putdown,thinks,eats,true,false
   PARAMETER(orate=1, null_message=0, busy=1,
 . not_busy=2, hungry=3, putdown=6, thinks=7,
 . eats=8, true=1, false= 0)
   INTEGER ibusy
   DATA ibusy/0/
   human =  bamboo-5
   human0 = bamboo-5
   human1 = MOD( human0+4, 5)

C always a chopstick
100  continue
```

```
if (human.eq.human0) then
    human = human1
else
    human = human0
endif

message@bamboo = activity@human

if (message.eq.hungry) then
  if (ibusy.eq.1) then
      reply = busy
  else
      reply = not_busy
  endif
  ibusy = 1
else if (message.eq.putdown) then
  reply = null_message
  ibusy = false
else if (message.eq.eats) then
  reply = null_message
else if (message.eq.thinks) then
  reply = null_message
endif
message@human = reply@bamboo
goto 100
RETURN
END
```

4.3 An extended example: particle dynamics

We now consider an extended application arising in the simulation of many-particle systems governed by Newtonian mechanics. We assume that there is a function $f(x, y)$ that determines the force between two particles at positions x and y. For simplicity of notation, we will ignore the implementation details regarding the fact that the positions are themselves multidimensional. Let x_i denote the position of the i-th particle in the system, with $i = 1, \ldots, n$. We will also assume that the total force on the i-th particle is given by the sum of pairwise forces, namely,

$$F(x_i) := \sum_{j=1}^{n} f(x_i, x_j). \tag{4.1}$$

The motion of the system of particles is then governed by the system of ordinary differential equations

$$F(x_i) = m_i \frac{d^2 x_i}{dt^2} \tag{4.2}$$

where m_i denotes the mass of the i-th particle. To simplify notation in the code, the pairwise force function will be denoted by `ourforce`.

The computational problem associated with approximating the solutions of this system of equations is quite simple if n is small. However, for large n the computational effort becomes prohibitive and several simplifications are utilized to reduce the cost. One such technique is to use a *cut-off radius* R. The force $F(x_i)$ is split into two parts as follows:

$$F(x_i) := \sum_{|x_i - x_j| < R} f(x_i, x_j) + \sum_{|x_i - x_j| \geq R} f(x_i, x_j). \tag{4.3}$$

The part of the force arising from particles outside the cut-off radius is updated less frequently, reducing the computational cost per time-step to $O(nR^3 + \epsilon n^2)$ from $O(n^2)$.

Another simplifying technique is to constrain certain particles to move as a group, rather than separately. Such an example might be the three atoms comprising a water molecule. This turns the system into an *algebraic o.d.e.* and complicates the code significantly, but it allows much larger time steps to be utilized in many cases. In molecular dynamics, this technique is known as the *shake* algorithm (see Chapter 8). We will consider the effect both of these techniques has on the development of a parallel code for particle dynamics.

4.3.1 Pair generation

The use of a cut-off radius requires the generation of a "pair list" `JNB` which keeps track of which particles are within the cut-off radius. `JNB` can be determined via the following pseudo code. The array `INB` records the numbers of particles within the cut-off radius for each particle.

```
C         Pairlist computation:  original version
       J0 = 0
       JJ = 0
       for I = 1, N
         for J = I+1, N
           if( | X(I) - X(J) | < R ) THEN
             JJ = JJ + 1
             JNB(JJ) = J
           endif
         endfor J
```

```
    INB(I) = JJ - J0
    J0 = JJ
  endfor I
```

The `JNB` array lists all neighbors in a linear data structure that is very efficient to use. Additional arrays, `start` and `end`, are needed in order to access it, as follows.

```
start(1)=1
end(1)= INB(1)
for I = 2, N
   start(I)= end(I-1) + 1
   end(I) = end(I-1) + INB(I)
endfor I
```

Thus the neighbors of particle i are to be found in positions `start(I)` to `end(I)` in the pair list, i.e., `JNB(start(I):end(I))`.

The critical difficulty with parallelizing this algorithm is the dependence in the `JNB` array on previous iterations of the `for I` loop. We note that this sort of dependency could not be dealt with by existing compiler techniques. However, the dependency does not occur in the *values* of `JNB` but rather in the *locations* of the values. The values of `JNB` are independent of each other, but the locations of the values are dependent on other locations, which are computed at the same time as the values in the code above. However we may compute each iteration separately and figure out later where to put them. By introducing the notion of a "scratch space" `JNBL` we can easily remove the dependencies as follows.

```
C     Pairlist computation: transformed version
      for I = 1, N
            JL = 0
            for J = I+1, N
              if( | X(I) - X(J) | < R ) THEN
                JL = JL + 1
                JNBL(JL,I) = J
              endif
            endfor J
            INB(I) = JL
      endfor I
```

```
          .
          .
          .
      for IALL = 1, N
        JNB( start(IALL) : end(IALL) ) = JNBL( 1:INB(IALL) , IALL)
      endfor IALL
```

Now each iteration of the `I` loop is independent of the others and can be computed in parallel. Note that this parallelism has been created at the expense of subsequent communication and at the expense possibly of increased storage. One difficulty in doing this automatically is that there is no *a priori* bound on the size of the scratch copies.

The above approach translates directly into the model needed in IPfortran. Each processor will compute a collection of the iterations, `ISTART@p` $\leq$ `I` $\leq$ `IEND@p`. (In the following code, we will assume this information is in the arrays `ISTART(p)` and `IEND(p)`, known at all processors.) The number may depend on `p` due to the triangular shape of the double loop [26]. Each processor will have a (local) copy of part of the the `JNB` array. In fact, the original code can be used, the only exception being that it may be preferable to keep the local `INB` array distinct.

```
C         Pairlist computation:  |Pfortran version
      J0 = 0
      JJ = 0
      for I = ISTART(thisnode), IEND(thisnode)
        for J = I+1, N
          if( | X(I) - X(J) | < R ) THEN
            JJ = JJ + 1
            JNB(JJ) = J
          endif
        endfor J
        INB(I) = JJ - J0
        J0 = JJ
      endfor I
```

Since `JNB` is the largest data structure in the problem, it is likely that it would not ever be collected in one place, but rather it would be left distributed. However, it would be essential for load balancing to know the values of INB, which could be collected in IPfortran as follows.

```
      DO 55512 p = 0, NPROCS - 1
55512        INB(ISTART(p):IEND(p)) = INB(ISTART(p):IEND(p)) @ p
```

This exchange of information is much less than optimal in terms of communication. An approach that is much better is described in Section 8 on the "shake" algorithm. Also see the code in Section 3.3.3 for concatenating `ithere`.

4.3.2 Particle force calculation

The sequential form of the particle force calculation can be written as

```
for I=1,N
  F(I)=0.0
  for J=start(I), end(I)
    tmpforce = ourforce( X(I) , X(JNB(J)) )
    F(I)=F(I) + tmpforce
    F(JNB(J))=F(JNB(J)) - tmpforce
  endfor J
endfor I
```

where we are assuming that the JNB array points only to neighbors with larger particle index, in order to avoid duplicate calculations of `ourforce`. Note also that the force F and the positions X are three dimensional vectors. It is easy to see how to load-balance the above. It can be written

for I=1,N
    do $C(end(I) - start(I) + 1)$ units of work
endfor I

for some constant C. Thus once we know `INB(I)=end(I)-start(I)+1` (which may change in an outer loop), we can schedule the separate tasks appropriately. In the IPfortran implementation of `UHgromos` [26], all of the INB array is kept at each processor. Each processor then participates in the load balancing step, determining a contiguous set of indices `Istart(thisnode):Iend(thisnode)` for which to compute the above. The most time consuming step in the load balancing is the exchange of the JNB array values for these I. This will be discussed in a subsequent section. However, once the processor has the relevant data, it simply computes

```
      DO 1 I=1,N
1        F(I)=0.0
      DO 2 I=Istart(thisnode),Iend(thisnode)
        DO 2 J=startL(I), endL(I)
          tmpforce = ourforce( X(I) , X(JNB(J)) )
          F(I)=F(I) + tmpforce
2         F(JNB(J))=F(JNB(J)) - tmpforce
```

Note that new names (`JNBL, startL, endL`) are introduced only to clarify the fact that we do not want to store the entire `JNB` array locally and thus these represent pointers.

Due to the nature of the above algorithm, at the end of the loop, no processor has fully computed `F`. Thus it is necessary to accumulate all the values, which is easily done in IPfortran via the reduction operation

```
      F = +{F} .
```

Note that the sequential semantics of the original code have not been kept, since the order of summation may be different. At this point the positions `X` can be updated via a time-stepping scheme together with an algorithm designed to preserve certain constraints (see Section 8).

Note that this implementation is somewhat communication intensive in that the forces are not computed locally. However, this avoids duplicate force computations and is in keeping with the spirit of the sequential algorithm. The experience with the `UHgromos` code parallelized in this way [26] indicates that the communication overhead is tolerable for modest numbers of processors. However, for large numbers of processors, the cost of collecting `F` exceeds the cost of computing it in a distributed fashion.

In Chapter 8 we developed the expression

$$INB_{ave} := \frac{1}{N}\sum_{i=1}^{N} INB(i).$$

Then [27] both the computation of `F` and the collection of `F` scale linearly with N, and the ratio of computation cost to communication cost is proportional to

$$INB_{ave}/P.$$

Thus for fixed INB_{ave}, the cost of communication will dominate for large P, independent of N. In the experiments with `gromos` parallelized in this way, for INB_{ave} about one hundred, the cross-over point occurs near 100 processors on the Intel *i*PSC/860. However, note that INB_{ave} increases cubicly in the cut-off radius, R, so that the break-even number of processors, given by $P \propto INB_{ave}$, increases cubicly with the cut-off radius. In the limit of having an infinite cut-off radius (that is, no cut-off at all), we have $INB_{ave} = N$ and the communication costs are swamped by the computational cost: the algorithm is fully scalable as N increases.

One way to achieve a scalable code for a fixed cut-off radius appears to be to allow each processor to compute all contributions to part of `F`, to eliminate the global reduction operation `F=+{F}`. This is done by having `JNB` include *all* neighbors, not just those with larger particle index. This eliminates the assignment

```
F(JNB(J))=F(JNB(J)) - ourforce( X(I) , X(JNB(J)) )
```

as well as the communication cost of `F=+{F}`, but at the expense of *doubling* the amount of computation.

Note that the IPfortran implementation mixes the ownership of the force array `F`. Any part of the array may be computed by a large number of processors.

4.3.3 Exchange of the `JNB` array

The exchange of the `JNB` array is described in detail in one case here. The basic algorithm is to line up all the work in a row, divide the length of the row roughly equally, and then push jobs in either direction as necessary to achieve a balance.

```
      SUBROUTINE BalJnb(inb,jnb,inbx,jn,nr,segb,myi,myj,njnb)
c for use with NTN=1
        implicit real*8 (a-h,o-z)
        integer*4 inb(1),jnb(1),inbx(1),jn(1)
        integer*4 segb(2,*),nr
        integer*4 myi,myj,njnb
c
        integer*4 lj,loweri,upperi,pseg,jindex
        integer*4 lis,uis,jta,js,allocate
c
      call inb_exchange(segb,inb,nr,jn)

      lj = 0
      do 796 i=1,nr
           lj = lj + inb(i)
           inbx(i) = inb(i)
```

```
796   continue
      do 797 ii = 1,njnb
            jn(ii) = jnb(ii)
c see here
            jnb(ii) = 0
797   continue

      lis = 1
      upperi = 0
      pseg = 1
      js = 1
      inbsave = 0
      allocate = lj/numnode
      do 878 i = 1, numnode
            if (inbsave.eq.0) then
                  loweri = upperi + 1
            else
                  loweri = upperi
                  inbx(loweri) = iover
                  if (thisnode.eq.i-1) inb(loweri)=inbx(loweri)
            endif
            icount = 0
            upperi = loweri - 1
            do while (((icount.lt.allocate).or.
     .           (i.eq.numnode.and.upperi.ne.nr)).and.upperi.lt.nr)
                  upperi = upperi + 1
                  icount = icount + inbx(upperi)
            end do
             if (i.eq.numnode) upperi = nr

c count until i´s allocation filled for thisnode,if thisnode
c pseg initialized to 1
            jta = 1
            uis = 0
            lis = loweri
            do while (uis.lt.upperi)
               if (segb(2,pseg).lt.lis) then
                  pseg = pseg + 1
                     js = 1
               endif
c how much of its jnb will processor pseg send to i
                 if (upperi.lt.segb(2,pseg)) then
                  uis = upperi
               else
                  uis = segb(2,pseg)
               endif
c count jnb items to be communicated
               jnbc = 0
               do 876 ii=lis,uis
                  jnbc = jnbc + inbx(ii)
876               continue
               if (pseg.eq.i-1) then
```

```
                  do 877 ii = js, js+jnbc-1
c see here
                  if (thisnode.eq.i-1) jnb(jta) = jn(ii)
                  jta = jta + 1
877                  continue
                  js = js + jnbc
               else
           jnb(jta:jta+jnbc-1)@(i-1) = jn(js:js+jnbc-1)@(pseg-1)
                  jta = jta + jnbc
                  js = js + jnbc
               endif
               lis = uis + 1
            end do

            if ((i-1).eq.thisnode) then
                  myi = loweri
                  myj = upperi
            endif
878   continue
      RETURN
      END

      subroutine inb_exchange(segb,inb,nr,tmp)
        implicit real*8 (a-h,o-z)
      include 'gromos.h'
c
c segb: indices into inb for p of P
c inb: what is being exchange and updated
c nr: number of atoms --> size of inb in words
c tmp: scratch
c
c logP exchange of inb.
c
      integer*4 segb(2,*),inb(1),nr,tmp(1)
      integer*4 i,j,k,p,dst,d
      integer*4 map(MAXNODES),imap(MAXNODES)

      do 10 i = 1,numnode
10    map(i) = 0
      map(thisnode+1) = 1
      k = 1
      do 60 d = 1, cubedim
      do 50 p = 0,numnode-1
            dst = xor(p,k)
            tmp(1:nr)@dst = inb(1:nr)@p
            imap(1:numnode)@dst = map(1:numnode)@p
            if (dst.eq.thisnode) then
                do 40 j = 1,numnode
                    if (imap(j).eq.1 .and. map(j).eq.0 ) then
                    map(j) = 1
                    do 30 i = segb(1,j),segb(2,j)
                        inb(i) = tmp(i)
```

```
30                  continue
                  endif
40                continue
              endif
50    continue
        k = k*2
60    continue

      return
      end
```

Chapter 5

IPfortran Implementation and Usage

5.1 IPfortran Implementation

The IPfortran translator consists of a lexical analyzer, parser and code generator. We used the lexical analyzer and parser from *f2c* program developed by AT&T Bell Laboratories and Bellcore. The parser actions consisting of the creation of the internal representation and generating the final FORTRAN77 with message passing consists of about 9000 lines of C.

IPfortran source is processed a single line at a time. If the line contains a IPfortran construct, namely: , then a syntax tree is constructed. A traversal of the syntax tree produces the native Fortran77 with calls to wrapper routines that invoke the machine–specific communication routines in the IPfortran libraries. Code without IPfortran constructs, are passed through the code generator and buffered *as is* in an internal buffer with the generated code.

Specialized functions implement global reductions, generating a global exchange algorithm based on binomial spanning trees optimized for hypercubes. Collective communication routines are performed for all processors or none. If a subset of processors are involved in a data exchange, point–to–point communication is used.

Message tags are generated automatically by code generated by the IPfortran compiler. It is possible to escape from the communication algorithms used by IPfortran, however, one must integrate the tag space with the IPfortran tag space to avoid duplicate messages with the same tag, or processes with tags out of phase. The IPfortran library provides a tag *get and increment* routine for this purpose. The routine *pf_getinctag* creates a tag space of a specified size and returns the first tag value in the space. The next automatically generated tag will be at the end of this space.

The shared memory version of IPfortran uses a distributed–memory message passing approach. Each process attaches a set of queues, one per process, that resides in System V shared memory. Access to the shared–memory queues is synchronized via machine–dependent semaphores. Blocks of shared memory comprising a message are taken out of the sending processors queue and inserted into the receiving process queue to effect message passing. Broadcasts are optimized by copying only once, then passing the message header among the processes involved in the broadcast.

5.2 IPfortran extensions to FORTRAN77 grammar

The operator precedence in IPfortran, from highest to lowest precedence, is

$$@ \quad \star\star \quad (\star, /) \quad (+, -)$$

So for example, $a = b@p + 1$ is not equivalent to $a = b@(p + 1)$ and $a@p + 1 = b@q$ is meaningless where $a@(p + 1) = b@q$ is not.

@ is non-associative, therefore, $x = y@i@p$ is illegal and $x = y@(i@p)$ or $x = (y@i)@p$ are legal.

The following productions summarize the IPfortran extensions to the FORTRAN77 grammar:

expr $\longrightarrow$ expr @ expr | expr @ subs_list | expr @ indx_list

expr $\longrightarrow$ name (subs_list)

subs_list $\longrightarrow$ expr : expr | subs_list , expr : expr

expr $\longrightarrow$ name { arglist }

statement $\longrightarrow$ *call* name { arglist }

where $\{\}@, : ()$ and *call* are literals.
The second rule corresponds to subarrays; the third rule covers both subarrays and processor ranges. The fourth and fifth rules represents global functions and procedures, respectively.

5.3 Usage

IPfortran has been ported to a number of parallel machines for porting applications, classroom use, and research. This section describes the usage of IPfortran. Much of the development has resulted from ports that employeed IPfortran. In addition, IPfortran has been used regularly for classroom instruction at the University of Houston, and by researchers developing parallel algorithms.

5.3.1 Installation

The IPfortran default target machine can be hard–wired at compiler time with the Makefile.pf variable `TARGET` where valid machine types may be taken from: IPSC, NCUBE, CM5, PVM3, KSR, and SGI. This default machine can be overriden with compiler switches (see below).

5.3.2 Invoking

IPfortran compiler **pfc** can be invoked with

pfc [*switches*] f_1.pf [f_2.pf [...]]

or alternatively

pfc [*switches*] $\star$.pf

where:

f_i.pf: the name of the IPfortran (or FORTRAN77) source file; this filename *must* have *.pf* as a suffix.

switches: Optional command line switches listed below.

5.3.3 Include files

Optional include file search directories can be used with the syntax:

$pfc -Id1 -Id2 ... -Idn myprog.pf

for directories d1 ... dn.

Up to 10 directories can be specified with one directory per I flag. The directories are searched in order of occurrence; the current working directory is always searched and it is searched last unless -I. is the first include search directory. The maximum nesting depth of include files is 10.

5.3.4 Switch summary

The following switches select the indicated target system for code generation:

-ipsc	Intel
-ksr	KSR-1; uses library SHMSG or TCGMSG.
-sgi	SGI; uses SHMSG.
-cm	CM5.
-pvm	PVM.
-ncube	NCUBE.

The remaining switches are

-I	Search this directory for include files.
-scratch <*n*>	Number of bytes for IPfortran buffer (default 100000).
-T string	Use *string* as variable name for THISNODE (default).
-i<*n*>	Default integer size of n bytes.
-r<*n*>	Default real size of n bytes.
-d<*n*>	Default double precision size of n bytes.

5.3.5 Variable sizes

The default variable sizes, in bytes, used by IPfortran are given below. These sizes can be overridden with the compiler switches described above. (See Section 5.3.11 for KSR-1 details.)

machine	integers	single precision	double precision	complex
KSR	8	8	16	8
Others	4	4	8	4

5.3.6 IPfortran Library

System-dependent sends and receives are made from the IPfortran library, called *pflib*. There are several libraries supported by this version:

pflibintel	Intel hypercube, simulator, Touchstone Delta, and Paragon.
pflibshm4,8	KSR-1 shared-memory message passing.
pflibksr	KSR-1 TCGMSG.
pflibsgi	SGI multiprocessor shared-memory message passing.
pflibpvm3	PVM based for networks of workstations or Cray multiprocessors

Consult your local *pfc* man page for details on the naming conventions and library locations.

The IPfortran library, or its equivalent, must be linked to (loaded with) a program translated by the IPfortran compiler.

5.3.7 Scratch memory and variables

Temporary memory for arrays uses a block of storage declared with the array *pf_ib*. The default size of this storage can be modified using the **-scratch** switch. This storage is re-typed as needed through the use of equivalence statements.

IPfortran generates scalar variables for buffering or naming quantities. These variables are named from the base string *o10000* with successive variables incrementing the suffix *10000*. Clashes can occur if these variable names are used as *implicitly* typed variables in your program.

Labels are also generated by IPfortran in generating FORTRAN77 loops. These are guaranteed to be unique to the program section.

5.3.8 Intrinsic names

In addition to the intrinsic functions used by the target processor, IPfortran has a reserved variable-name set.

THISNODE Processor number of the processor running the code.

THISPROC Process number of the process running the code.

NUMNODE Total number of processors.

The current implementation of the compiler also uses the following names:

pflib IPfortran library module.

pf_ib Integer buffer.

pf_sb Integer2 buffer.

pf_rb Real buffer.

pf_db Real8 buffer.

pf_cb Character buffer.

pf_xb Complex buffer.

pf_inctag Tag increment subroutine (pflib).

pf_snd Send subroutine (pflib).

pf_rcv Receive subroutine (pflib).

iqdim Cube dimension subroutine interface.

5.3.9 Using IPfortran

IPfortran translates the input *.pf*, directing output into a *.f* file. Code not involving IPfortran extensions is output "as is," including comments and blank lines. Code generated from translating statements involving {} or @ adheres to the rules of FORTRAN77.

There are three levels of advisory messages with IPfortran: fatal, error and warning. *Fatals* result in immediate termination of the compiler with a diagnostic message and `.f` file removal. *Errors* also result in `.f` file removal, but the compiler will continue processing the input. *Warnings* result in advisorys alone with output written as usual to the `.f` file.

We emphasize that the intrinsic variable set for IPfortran includes variables such as *thisnode*. These can be used freely in any subroutine or function that has been processed by IPfortran – redeclaration of these intrinsic variables will result in error.

IPfortran can be used on one, or all, of the modules composing a program with the exception of the entry point to the Fortran program: *The main program (entry point to a program) must be processed by the IPfortran processor.*

5.3.10 Sample Makefile for the *i*PSC/860

Having added another file-type suffix slightly complicates the compilation process. The *make* program, however, can be used to automatically invoke the correct sequence of compilations. With the following Makefile, the *make* program will automatically check dates for the file sequence *planet.pf*, *planet.f*, *planet.o* and the executable *planet*, and recompile as necessary.

sample makefile:

```
 PFC = pfc
 F77 = if77
 PFLAGS = -scratch 1000000
 FFLAGS = -O2 -node
 LFLAGS = -node
 LIBS = -lpf
 .SUFFIXES: .pf .f .s .o
 .f.o:; $(F77) $(FFLAGS) -c $*.f
 .pf.f:;$(PFC) $(PFFLAGS) $*.pf
 .pf.o:;$(PFC) $(PFFLAGS) $*.pf
$(F77) $(FFLAGS) -c $*.f

 planet:$$@.o
$(F77) $(LFLAGS) -o $@ $@.o $(LIBS)
```

5.3.11 IPfortran Programs and the KSR

KSR encourages the use of a thread–based parallelism within a shared–address space, for example, KSR loop tiling [58]. Efficient message passing and distinct address spaces characteristic of MIMD architectures and SPMD programming styles were not provided by KSR. The separate address space programming methodology with messages was crucial for a direct port of applications developed for IPfortran, currently without a shared memory implementation. To this end we developed the SHMSG message–passing libary (see below). Other researchers developed a similar system, TCGMSG, to accomodate their codes [51]. Both TCGMSG and SHMSG implement the separate address space, memory–passing model with a multiple processes communicating through shared-memory. IPfortran targets both TCGMSG and SHMSG.

IPfortran and TCGMSG

To run a compiled program on the KSR, the program is passed as an argument to the TCGMSG program *parallel*. Suppose we want to run the program *baton* from the current working directory. In this directory, the file *baton.p* must be present, or created, prior to issuing

$ parallel baton

The file *baton.p* contains:

```
# This template is piped through sed to replace
# a with the name of the program.
# Add other hosts and processes as desired.
clark bubba 2 baton /home/clark/cwd
```

giving the user name, the host name, the number of processors, the executable name and the directory from which to run the program. Be sure to check the local system policies for processor set allocation.

A checklist for making the KSR executable with IPfortran are:

- Use the **-ksr** flag with IPfortran compiler,
- Use the *KSR* version of *pflib.f* (or libpf.a),
- Compile with the **-para** and **-i4** flags,
- ld with the libraries *-ltcgmsg* and *-lrpc*.

A sample Makefile, for the IPfortran program *baton*, follows.

```
LIBDIR = /usr/local/lib
TCGMSG = $(LIBDIR)/libtcgmsg.a
LIBS = -lrpc -lpthreads
FFLAGS = -i4
LFLAGS = -para
PFFLAGS = -ksr
F77 = f77
PFC = /usr/local/bin/pfc

.SUFFIXES:      .pf .f .pc .c .s .o
.pf.o:; $(PFC) $(PFFLAGS) $*.pf
        $(F77) $(FFLAGS) -c $*.f
.f.o:; $(F77) $(FFLAGS) -c $*.f
.pf.f:; $(PFC) $(FFLAGS) $*.pf

baton:   baton.o pflib.o
        $(F77) $(LFLAGS) -o baton baton.o pflib.o $(LIBS) $(TCGMSG)
```

Note the **-i4** switch is required; 8–byte integers are not supported under TCGMSG, but are for the KSR SHMSG implementation described in Section 5.3.11.
There are working examples on the TCAMC KSR at /usr/local/src/pf1.x/suite. A common mistake in using the KSR IPfortran is to execute the application without the *parallel* command described above. A IPfortran program will execute apart from parallel, but not correctly.

The KSR-1 and IPfortran variable sizes

IPfortran interprets variable sizes according to the specific architecture. The size of integers, reals, double precision can be controlled with compiler flags, overriding the IPfortran defaults. We recommended that variable sizes are consistent throughout a program, that is, all reals the same size, all double precision the same size, and so on. With experience, however, it's possible to use almost any combination of sizes permitted by the native compiler.

The following table is useful for determining how to size variables in your program. It shows the default IPfortran variable sizes for the KSR-1, and the effect of

IPfortran switches. The 3rd and 4th columns respectively. Note that a dash indicates no effect.

IPfortran variable sizes for the KSR-1			
type	default	-i4	-r8
integer	8	4	–
real	8	–	–
double	16	–	8
complex	8	–	–

IPfortran and SHMSG

The SHMSG system is a stand-alone library of message-passing routines for shared-memory architectures. Internally, it is similar to TCGMSG, however, there are differences in the user interface. In addtion, the SHMSG implementation supports 4–byte or 8–byte integers (see Section 5.3.5 for defaults). The usage described here applies to both KSR-1 and SGI multiprocessors.

Two libraries are required for running a SHMSG–based program on the KSR: the standard IPfortran library and the message–passing library. The standard library comes in 4–byte and 8–byte versions, corresponding to 4–byte or 8–byte integer native Fortran. These libraries are

1. *pflibshm4.f*
2. *pflibshm8.f*

The message passing library is called *shmsg.a*.

Compiling programs for SHMSG

A sample Makefile follows, however you are encouraged to check your local machine's man page for the location of test programs and Makefiles.

```
F77 = f77
CC = cc
INC = .
PFC     =       /home1/tclark/bin/pfc
SHMSG   =       /home1/tclark/x/shmsg.a
```

```
LIBS    =       $(SHMSG) -para

# 4-byte integers
PFFLAGS      = -ksr -I$(INC) -scratch 10000 -i4
FFLAGS       =       -i4 -noopt
PFL     =       pflibshm4.o

# suffix rules
#
.SUFFIXES:      .pf .f .pc .c .s .o

.c.o:; $(CC) $(CFLAGS) -c $*.c
.f.o:; $(F77) -I$(INC) $(FFLAGS) -c $*.f
.pf.f:; $(PFC) $(PFFLAGS) $*.pf
.pc.c:; $(PC) $(CFLAGS) $*.c
.pf.o:; $(PFC) $(PFFLAGS) $*.pf
        $(F77) -I$(INC) $(FFLAGS) -c $*.f

test4:  test4.o $(PFL)
        $(F77) -o test4 test4.o $(PFL) $(LIBS)
```

Specifying processor configurations for SHMSG based programs

A configuration consists of the number of processes (P) and the number of cells (C) on which the processes will execute. Processes will be bound to some cell. If P > C, processes will be cell mapped many-to-one.

A third parameter here is the number of cells in your pset (actual_C), or the "machine" that your application sees. If C > actual_C, SHMSG will adjust and inform you that intra-application pset timesharing will occur.

Applications can have problems with intra-application pset timesharing, especially when P > 2C. We have found that either it works, or it does not, apparently because of a problems with the operating system. We discourage this usage.

The configuration, then, conists of two integers contained in a file called Config (upper case 'C'). This file must be in the directory where program executes. The values of P and C are integers on the first line of this file, e.g., a first record of "8 8" specifies 8 processes to be executed on 8 processors.

Processor configurations should be powers of two for Pfortran compatiblity. That limitation will disappear with an expanded Pfortran communication library. Non-power of two processor operations will work with SHMSG in any case, and with Pfortran provided global exchanges are not used there.

Running SHMSG based programs

Type the program name in the directory where the Config file resides. Input and output behave as usual.

Stopping SHMSG based programs

So that your application releases system resources, a cleanup routine must be invoked prior to normal termination. The Pfortran compiler will translate a Fortran **stop** to a *pfstop* which in turn invokes the routines to release resources. A Fortran program should terminate with a Fortran **stop**.

For interactive processes, use ctl-c. If the process is not attached to a terminal, send it the SIGINT signal, i.e. kill -2 <pid>. If all else fails, use kill -9, with the disadvantage that the processes cannot release system resources when hit with the SIGKILL signal.

Things to look out for

Hung program: If your program hangs, or abnormally terminates, take a look at system ipc resources using the command "ipcs." If this seems like too much trouble, just run bubba:: `~clark/bin/rmem`. This will remove the shared-memory segments that the message-passing system used, and did not release, due to abrupt termination, e.g. kill -9.

Intra-application pset timesharing: When an application's processes timeshare cells, a less efficient algorithm is used for coordinating access to shared data in the message-passing system. This stems from a bug in a hardware mechanism used for enforcing mutual exclusion.

5.3.12 KSR specific limitations

Using IPfortran on the KSR results in efficent, and portable programs. There are, however, some limitations that you should be aware of. If any of these limitations impede your work, then again, submit comments to the system administrator; software solutions are available for all of the limitations listed below.

- Using *IMPLICIT NONE* will result in compiler warnings as the KSR switch does not produce declarations for some intrinsic KSR functions. Workaround: put the declarations in the IPfortran source.
- Global sum algorithms require power of two processors, e.g. 1,2,4, ... Using other than powers of two processors will result in erroneous results for IPfortran global sum operations. Workaround: use powers of two total number of processors, or do not use the IPfortran intrinsic global sum operations.
- The KSR debugger is not operational for IPfortran programs. Workaround: Use print statements or debug on another platform with a process–based parallel debugger, e.g., the Intel hypercube.

5.3.13 IPfortran Programs and PVM

The PVM version of IPfortran is layered on the Oak Ridge version of PVM (version 3 or later) **[ref]** and requires the supplied library *pflibpvm3* as well as the PVM libraries [1].

The PVM version of IPfortran has been used on networks of workstations, including HPs and SGIs, and on an eight–processor Cray EL.

Note that vendor-supplied versions of PVM may not be compatible with the version used in IPfortran; refer to vendor documentation for details. If you experience problems, you may wish to obtain and install PVM from the Netlib archive.

Compiling and linking programs for PVM

To produce the correct calls to PVM libraries, supply the switch **-pvm3** to *pfc*. Compilation of IPfortran programs for PVM is as usual. The following additional libraries are required at link time:

[1] PVM software can be obtained electronically via anonymous ftp to `ftp://netlib.att.com/netlib/pvm3` or via the World-Wide Web at URL `http://www.netlib.org/pvm3/index.html`.

- The IPfortran library for PVM, *libpfpvm3.a* (supplied with the distribution).
- The Fortran PVM3 interface library, *libfpvm3.a*.
- The PVM3 group services library, *libgpvm3.a*.
- The base PVM3 library, *libpvm3.a*.

Supply the libraries to the linker in the order presented above.

Sample Makefile for PVM

Here is a sample Makefile for PVM (locations and names of object files and libraries may differ on your system):

sample makefile:

```
EXE         = PFNODE
PFC         = pfc
F77         = f77
LD          = $(f77)
PFFLAGS     = -pvm3 -scratch 1000000
FFLAGS      = -O2
LIBDIRS     = -L/usr/local/Pfortran/lib -L/usr/local/pvm3/lib
LFLAGS      = $(LIBDIRS)
LIBS        = -lpfpvm3 -lfpvm3 -lgpvm3 -lpvm3
.SUFFIXES: .pf .f .s .o
.f.o:; $(F77) $(FFLAGS) -c $*.f
.pf.f:;$(PFC) $(PFFLAGS) $*.pf
.pf.o:;$(PFC) $(PFFLAGS) $*.pf
$(F77) $(FFLAGS) -c $*.f

$(EXE):$$@.o
       $(LD) $(LFLAGS) -o $@ $@.o $(LIBS)
```

Please note the following:

1. IPfortran and PVM libraries should be specified in the order shown. You can indicate alternate directories for library locations with the **-L** switch.

2. The PVM mechanism for spawning node programs (PVM3 routine **pvm_spawn**) requires a character string which names the executable to run on nodes of the virtual machine. This is "hardwired" as the name **PFNODE** in the IPfortran initialization routine **pf_com** and is therefore used in the link step above (*make* macro **EXE**). You can specify an alternate name by editing *pflibpvm3* and rebuilding the library.

Running PVM IPfortran programs

IPfortran programs for PVM follow the same steps for execution as other PVM programs.

1. Initiate the PVM daemon, either directly or using the PVM console.
2. Define the virtual machine, using PVM console commands or by supplying a host configuration file to the PVM daemon.
3. Execute **PFNODE** (or the executable name specified in **pf_com**) to run your IPfortran program.

Additional reserved names for PVM

The PVM version of *pflib* defines two additional COMMON blocks, containing a number of PVM-related variables. Normally, there will be no conflict with user variables, as the IPfortran compiler does not insert these COMMON blocks into user program modules. They are provided below for completeness.

pf_pvm INTEGER data used for PVM task management.

pf_pvmc CHARACTER data used for PVM task management.

pf_mytid PVM task ID for this node process.

pf_info PVM status variable.

pf_numt Number of tasks spawned in this virtual machine.

pf_tids Array of PVM task IDs.

pf_speeds Relative speed of hosts in the virtual machine.

pf_hosts Host identifiers.

pf_archs Architecture identifiers.

These data are used when the virtual machine is created and tasks are spawned. Internally, PVM uses "task IDs" to identify the node processes; IPfortran for PVM maps between PVM's task IDs and IPfortran's node ID's by storing the task IDs in the *pf_pvm* COMMON block.

Restrictions on IPfortran for PVM

1. IPfortran for PVM does not generate data-type-dependent communications; all messages are sent as a simple sequence of bytes. As a result, heterogeneous virtual machines are not supported and will likely produce incorrect results due to variations in number representation. At present, you should execute IPfortran PVM programs on homogeneous virtual machines (e.g. workstation clusters or symmetric multiprocessors).

2. As mentioned in the section on KSR-specific limitations, IPfortran intrinsic global sum operations will not work correctly with non power of two processors.

3. There is an internal limit on the maximum number of PVM node programs which may be spawned. This is specified in *pf.h* and *pflibpvm3* as the constant **PVM_MAXTASKS** (currently 64).

5.3.14 IPfortran Programs and the CM-5

The CM-5 version of IPfortran is layered on Thinking Machines Corporation's CMMD message-passing library and requires *pflibcm5*, supplied with the IPfortran distribution. CMMD supports both "hostless" and "host/node" programming models; IPfortran uses the hostless model.

Compiling and linking programs for the CM-5

The process compilation for CM-5 IPfortran programs is similar to other supported systems. Supply the switch **-cm5** to the *pfc* compiler and use a supported Fortran77 compiler such as *f77* or *af77*. To produce a CM-5 executable, you should use the linker *cmmd-ld*, standard with CM-5 system software.

Sample Makefile for the CM-5

A sample Makefile was presented for the iPSC/860 previously. Here is the equivalent Makefile for the CM-5 (locations and names of object files and libraries may differ on your system):

sample makefile:

```
PFC         = pfc
F77         = f77
LD          = cmmd-ld
PFFLAGS     = -scratch 1000000
FFLAGS      = -O2
LFLAGS      = -comp $(F77)
LIBS        = -lpfcm5
.SUFFIXES: .pf .f .s .o
.f.o:; $(F77) $(FFLAGS) -c $*.f
.pf.f:;$(PFC) $(PFFLAGS) $*.pf
.pf.o:;$(PFC) $(PFFLAGS) $*.pf
$(F77) $(FFLAGS) -c $*.f

planet:$$@.o
       $(LD) $(LFLAGS) -o $@ $@.o $(LIBS)
```

Note that *cmmd-ld* requires a language specification (**-comp**); failure to supply this switch will produce an error at link time.

Additional reserved names for the CM-5

The CM-5 uses separate networks, called the *data* and *control* networks, for point-to-point message-passing and cooperative communications such as reduction operations, scans, and broadcasts. Accordingly, CMMD provides separate library routines for point-to-point and cooperative communication. IPfortran for the CM-5 generates calls to the appropriate routines; this is implemented by additional routines in the CM-5 version of *pflib*. Other utility routines are provided for convenience; they are summarized below:

pf_bsnd Broadcast routine (send).

pf_brcv Broadcast routine (receive).

pf_sync Synchronization of all nodes.

pf_clock Timing function.

CM-5 I/O considerations

Hostless CMMD programs may perform I/O (through an I/O server running either on the partition manager or I/O control processor). Several different I/O modes are supported: *local independent*, *global independent*, *global synchronous broadcast*, and *global synchronous sequential*. CMMD opens *stderr*, *stdin*, and *stdout* (Fortran units 0, 5, and 6, respectively) in global synchronous broadcast mode. By default, subsequent I/O streams are opened in local independent mode. During initialization, IPfortran resets the I/O mode of *stdout* to global independent so that node programs may perform output at any time without global node synchronization.

IPfortran programs may reset the I/O mode by directly calling the appropriate CMMD routines (**CMMD_set_open_mode** and **CMMD_set_io_mode**). If this is done, be sure to include the CMMD Fortran include file (`/usr/include/cm/cmmd_fort.h`).

Be aware that the pool of available file descriptors may be depleted by opening even a small number of files in local independent mode.

Use of CMMD node timers

IPfortran uses the standard CMMD timing facility, which provides up to 64 independent node timers measuring busy, idle, and elapsed time. These timers are numbered 0 through 63; IPfortran initializes node timer 63 at startup – calls to the routine *pf_clock* return the busy time (in double precision seconds) measured by timer 63 since the program started. Manipulation of this timer by user programs may result in unexpected return values.

5.4 Typical Programming Errors in IPfortran

To insure correct communication using a IPfortran statement, the communicating processes must both execute the communication statement, agreeing on the source and destination values. Furthermore, any processes participating in any communication must execute **all** communication statements in order to maintain consistency with message tags.

5.4.1 Conditional guards

One source of difficulty with any parallel programming system concerns communication statements that occur inside blocks that are "guarded" by conditional expressions. For example, the statment

$$\text{IF(thisnode .EQ. 7) x = y@11}$$

is not the same as

$$\text{x@7 = y@11}$$

in IPfortran. The latter correctly assigns the value of the variable `y` in processor 11 to the variable `x` in processor 7, whereas the former will simply cause processor 7 to deadlock waiting for processor 11 to execute the same statement, which it will never do. Variants of the former statement can be correct, e.g.,

$$\text{IF(thisnode .EQ. 7 .OR. thisnode .EQ. 11) x = y@11}$$

will set `x` at both processors 7 and 11 to the value of `y` at 11. However, the latter programming style in IPfortran is discouraged. Rather, the expression `x@7 = y@11` is preferred.

5.4.2 Inconsistent source and destination process identifiers

In the current compiler implementation, a faulty communication expression results for `a@p= b@q` if the values of `p` and `q` do not agree at processors `p` and `q`, that is,

$$p@p \neq p@q \quad \text{or} \quad q@p \neq q@q.$$

See Section 3.3.1 for an alternative method.

5.4.3 Processor number out of range

A statement such as

```
x = y@p
```

is incorrect (with unpredictable results) unless $0 \leq \texttt{p} < \texttt{numnode} - \texttt{1}$. See Section 4.1.3 for examples of how to limit processor values appropriately.

Chapter 6

High Performance Fortran Overview

High Performance Fortran (HPF) is an extension of Fortran 90 designed to facilitate the task of writing machine–independent data–parallel programs using a global name space [53, 65, 40]. The language specification, an outgrowth of data–parallel languages such as Fortran D [54, 102] and Vienna Fortran [11], was formulated by the High Performance Fortran Forum (HPFF), a coalition of academic and industrial groups. This section gives a brief overview of programming with HPF.

6.1 Programming with HPF

The HPF language provides constructs for specifying *data distribution* and *iteration distribution*. With the exception of the *extrinsic directive*, the compiler automatically transforms the global name–space program into the appropriate parallel executable, typically a local name space.

The basic goal in data mapping is to describe a decomposition of data onto processors to maximize locality. HPF uses a two–level mapping in assigning data to processor memories which involves: 1) the optional alignment of arrays; 2) the distribution of arrays or templates onto a logical processor array. (A template is an index domain without associated storage.)

A common distribution is a BLOCK distribution which we define below. Since the array is two dimensional, it is convenient for mapping purposes to consider a two dimensional logical grid of processors. The goal is to distribute the data onto these processors, and then to distribute the calculation.

To distribute the data blockwise, a function is used to map array positions to processors: $f : i_k \rightarrow p_k$, where i_k and p_k are the array index and processor indices in the kth dimension, respectively. For this example k is zero or one. For a block-wise decomposition in two dimensions there are $P_1 \times P_2$ processor coordinates in each dimension. For the array, $A(N_1, N_2)$, there are N_1 and N_2 coordinates in dimensions one and two, respectively.

With a block-wise decomposition, contiguous blocks of array elements are assigned to processors. In this example we will distribute the second dimension and replicate the first. (In other words, contiguous blocks of columns will be assigned to processors.) The number of elements, m, in the distributed dimension is $\frac{N_2}{P_2}$; in the replicated dimension there are N_1 elements at each processor. The processor coordinate for each element is determined by

$$f_k = \lceil \frac{i_k}{m} \rceil = \lceil \frac{i_k P_k}{N_k} \rceil \tag{6.1}$$

for $k = 1, 2$. A CYCLIC distribution has a similar function that maps array indices to processor coordinates as follows:

$$f_k = 1 + \lfloor \frac{i_k}{m} \rfloor \mathrm{MOD}(P_k) \tag{6.2}$$

where the default value for m is 1.

Suppose we are interested in block distributing a rank–2 array across P processors such that data is stored only at the processor where it is write accessed. The function f_k (equation 6.1) defines a blockwise decomposition mapping used by HPF. The distribution itself is performed by the code generated by the HPF compiler for the following source:

```
          REAL A(N1,N2)
!HPF$     DISTRIBUTE A(*,BLOCK)
```

where !HPF$ is the HPF directive marker [53, 65, 40].

A "$*$" in a distribute format list can be thought of as a compression or replication along that dimension. In the example above, we can visualize compressing the columns of A into a single column of rows, then block distributing the column.

In addition to distributing the data, we want to distribute the iterations. A heuristic used by HPF compilers to accomplish this is the *owner–computes rule* [22]. The owner–computes rule preferably results in write accesses of data stored in local memory; read accesses can be performed on local or non-local data and will depend on the program design. For variations to the owner–computes rule see Chapter 8 and [40].

The following loop initializes A to contain zeroes:

```
DO I2 = 1, N2
  DO I1 = 1, N1
    A(I1,I2) = 0
  ENDDO
ENDDO
```

The iterations of this loop are independent of one another (there are not any loop carried dependencies) making it possible to schedule the iterations in any order on any number of processors. The owner–computes rule is a good choice here. With P processors and the blockwise distribution above, each processor could execute

```
DO I2 = 1, N/P
  DO I1 = 1, N
    A(I1,I2) = 0
  ENDDO
ENDDO
```

Notice that the loop bounds and storage have been reduced by a factor of P^{-1}, the perfectly scalable case. Loop bounds reduction is not always possible.[1]

The data–parallel initialization of A is extremely simple so that even a unsophisticated HPF compiler could detect the parallelism and distribute the iterations of the *DO* loop along with the data. For more complex codes where the parallelism is not obvious or is impossible to detect, even a sophisticated HPF compiler could benefit from assistance. We list, without further discussion, HPF language elements that can assist the compiler in expressing the parallelism in a program:

- the FORALL statement and construct
- the INDEPENDENT statement
- a Fortran 90 array expression.

PROCESSOR directive

The PROCESSOR directive declares one or more rectilinear (logical) processor arrangements onto which data is mapped. The 2D array of processors at the beginning of this chapter is an example.

[1] See Chapter 12 for a discussion about loop bounds reduction.

The following three examples show how to specify one, two and three dimensional processor arrays. The HPF intrinsic function *NUMBER_OF_PROCESSORS()* returns at runtime the number of processes (or processors) the application is to execute on.[2]

```
!HPF$       PROCESSORS ::    ONED(NUMBER_OF_PROCESSORS())
!HPF$       PROCESSORS ::    TWOD(NUMBER_OF_PROCESSORS()**1/2,
                                  NUMBER_OF_PROCESSORS()**1/2)
!HPF$       PROCESSORS ::    THREED(NUMBER_OF_PROCESSORS()**1/3,
                                    NUMBER_OF_PROCESSORS()**1/3,
                                    NUMBER_OF_PROCESSORS()**1/3)
```

It is the compiler's job to map the logical processor array to the physical processors, although the HPF specification allows for implementation dependent directives to specify this. In addition, the processor directive is an optional part of the distribute statement. HPF-2 gives further flexibility in mapping data to the physical processors [40].

ALIGN and DISTRIBUTE statements

The *align* statement maps an array onto a template, or another array. Arrays mapped to the same template are aligned with each other. After alignment, the *distribute* statement is used to distribute arrays onto an abstract machine defined by the *processors* statement; BLOCK and CYCLIC distributions are supported. The following program aligns the arrays A, B, C and D with the TEMPLATE T, which is then distributed.

```
          REAL A(N),B(N),C(N),D(N)
!HPF$     TEMPLATE T(N)
!HPF$     ALIGN WITH T :: A,B,C,D
!HPF$     DISTRIBUTE T ONTO ONED
          DO I = 1, N
              A(I) = B(I)*C(I) + D(I)
          ENDDO
```

Alternatively, one can align perform the alignment with an array as follows.

[2] The method used to specify the number of processors for a program will be compiler implementation specific.

```
!HPF$       ALIGN WITH A :: B,C,D
!HPF$       DISTRIBUTE A ONTO ONED
```

Section 6.3 illustrates the use of the distribute statement for a three dimensional array. The number of processors in Figure 6.1 can be inferred by the number of volume elements in the diagram, a result of equation 6.1.

6.2 Other HPF features

An HPF program can use the intrinsic reduction functions provided by Fortran 90. HPF-2 suggests the use of user–defined reduction functions. Input and output is not addressed in HPF-1, however, HPF-2 suggests some features for I/O. Other features suggested by HPF-2 include: enhanced mapping functionality, communication pattern reuse, and irregular mappings. A kernel HPF has been described in [40], limiting the HPF directives to a core subset toward better performance. Finally, HPF provides an intrinsic function where one can escape from HPF into node code or another parallel paradigm.

6.3 Spatial decompositions with HPF

In Chapter 9 we describe an algorithm for the spatial decompositon of GROMOS. In this section we illustrate how one could implement a spatial dcomposition using HPF. Figure 6.1 illustrates four blockwise decompositions distributed onto processor arrays of varying dimensions. Data dependencies are present between the neighboring processor domains, and possibly more, depending on the penetration of the cutoff radius.[3] another. Choosing the best decomposition for a 3D spatial problem depends on numerous factors. To minimize communication, the general rule is to minimize the surface-to-volume ratio of the processor domains.

Without a value–based decomposition [49, 40], the programmer must determine an atoms logical spatial domain, then assign the atoms to the regularly distributed domain, such as those in Figure 6.1. The HPF compiler will move data to the appropriate processed according to the assignment of atom data to a region of space. Alternatively, the programmer can use an irregular decomposition to assign a processor number to a map array dimensioned to the number of atoms. The spatial decomposition is then

[3] Processor domains refer to the portion of the array assigned to a processor.

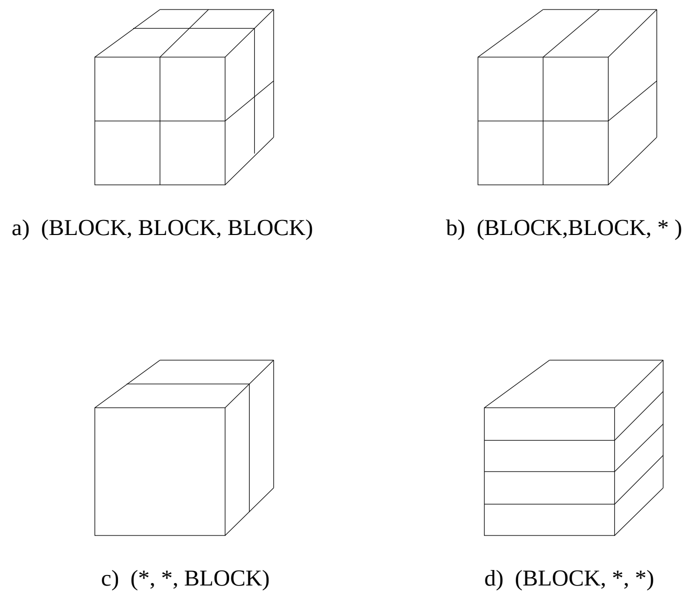

Figure 6.1: Decompositions and distributions for the array $A(N, N, N)$ illustrating a) *blockwise* for P=8 b) *columnwise* for P=4 c) *slicewise* for P=2 d) *slicewise* for P=4.

implemented by aligning the map array with coordinate and property arrays, and distributing the map array with an irregular decomposition. Figure 6.2 uses an irregular decomposition [42] as illustrated in Figure 6.1 (a) to spatially distribute atoms. The irregular map is calculated in lines 9 through 13.

In either case, in order to implement a spatial decomposition, the programmer must implement low–level mapping details analogous to the BLOCK or CYCLIC functions described above. At the same time, the program must conform to HPF constructs. The HPFF suggests value–based and irregular decompositions for HPF-2 to address the spatial decomposition and related programming problems [40].

```
SUBROUTINE Irreg_Linear_to_Block(blkX,blkAtoms,BlkCnt)
REAL blkAtoms(MatmBlk,Mblk(1),Mblk(2),Mblk(3))
REAL blkX(MatmBlk,Mblk(1),Mblk(2),Mblk(3))
REAL blkCnt(Mblk(1),Mblk(2),Mblk(3))
DISTRIBUTE blkCnt(BLOCK)
DISTRIBUTE lan2gan(BLOCK)
REAL X(1:Matms)
INTEGER blkCnt(1:MBlk3)
ALIGN blkAtoms WITH BlkX

DO gan = 1, Natoms
   blkInd(1:3) = Coord_to_Index(X(1:3),Mblock(1:3))
   blkCnt(blkInd(1),blkInd(2),blkInd(3)) ++
   Map(gan,1:3) = blkInd(1:3)
ENDDO

REDISTRIBUTE blkX(*,:,:,:)  WITH Map

RETURN
```

Figure 6.2: Pseudo code for a spatial decomposition using an irregular distribution directive.

Chapter 7

UHGromos: the replicated algorithm revisited

This chapter describes the results of our port of a molecular dynamics (MD) program, Gromos [47], to the iPSC/860.[1] The objectives of this study were three fold. First, we wanted to assess the suitability of our software tools for porting existing Fortran codes (dusty decks) [8]. This involved developing Fortran versions of previously developed parallel extensions to the C programming language. Secondly, we wanted to be able to quantify various components of an MD simulation with respect to communication and computation costs. Here the objective was to have timing data that would aid in the design of a scalable code that could execute efficiently on a large number of processors. Finally, we wanted to see if execution speeds comparable to a Cray could be achieved easily with a modest number of current–generation nodes. Such performance seemed feasible based on a rough assessment of current processor performance and communication speed. One of our principal conclusions is that we were able to achieve performance on a standard benchmark problem comparable to reported performance of a Cray–2 using between 8 and 16 processors on an iPSC/860. Moreover, this was done with a minimal amount of coding and debugging, due primarily to the use of our software tools.

7.1 Description of MD algorithms

The basic model of molecular dynamics involves only classical mechanics, although the forces, F, between atoms are more complicated than the gravitational or electrostatic

[1] Material in this chapter comes from [26].

forces encountered in basic mechanics [2, 77]. The model reduces to a system of ordinary differential equations for the positions, x, of the atoms which express essentially that

$$F = m\frac{d^2x}{dt^2}.$$

The most computationally intensive part of the simulation is the evaluation of the forces between atoms (on the face of it, a computation involving $\mathcal{O}(N^2)$ operations for N atoms). This consists of two parts, the *bonded* forces among atoms linked by covalent bonds, and the *non–bonded* forces among atoms not so linked. The latter involve the greater amount of work, since the bonded forces require only $\mathcal{O}(N)$ operations for N atoms.

Efficient approximation algorithms for evaluating the non–bonded forces among atoms have been introduced recently using diverse techniques. An approximation algorithm for evaluating the Coulombic forces has been considered by Greengard and Rokhlin using multipole expansions [46]. A related algorithm for computing simultaneously the Coulombic and short–range forces (e.g. Lennard–Jones potentials) has been introduced by Draghicescu in her thesis [36]. Both of these techniques (as well as techniques that involve solving a Laplace equation for the electrostatic potential yield a close approximation to the actual interaction among all particles /citeDavMcCam. At the moment, techniques of this type are not in wide–spread use in existing molecular dynamics codes. However, we intend to examine such algorithms for possible use in the next version of our port of GROMOS.

Existing codes employ, typically, some sort of *cut–off radius* outside of which a very simple approximation to the forces among atoms is employed. Interactions between an atom and its neighbors inside that atom's cut–off radius are computed exactly. The key to efficiency in this approximation is the way lists of neighboring atoms are handled. The approach taken by GROMOS will be described in detail in the next section. Another effective implementation divides space into boxes and keeps track of particles inside each box via a *linked list* [83]. The spatial subdivision simplifies updating the neighbor lists. (Atoms may move from one box to another during the computation.) Only neighboring boxes can have interactions if the cut-off radius is smaller than the dimension of the boxes. Boris *et al.* use specialized data structures geared toward vector-processing architectures for this problem [15, 69, 16].

7.2 The Gromos program and its parallelization

The Gromos suite consists of 39 separate programs in 127 files, with over 74K lines of Fortran [47]. The MD module comprises 27 of these files with over 14K lines of Fortran. Our efforts at parallelization in UHGromos focused only on the MD module beginning with a version of the code that had been developed previously for shared–memory MIMD machines [25].

The main parts of the computation are updating the pair list, computing the non–bonded forces and the time-stepping. The pair list is treated as a pair of arrays, `INB` and `JNB`. For each atom, `I`, `INB(I)` is the number of neighboring atoms, `J>I`, within the cut–off radius, and `JNB(Start(I)+1:Start(I)+INB(I))` is the corresponding set of atom indexes (`Start(I)`$= \sum_{k=1}^{I-1}$ `INB(k)`).

To compute these arrays, Gromos loops over pairs of atoms, involving $\mathcal{O}(N^2)$ work. This, however, is done only every few (ten, in the benchmark runs) time–steps, so that for typical problems, the amount of work is minimal compared with the computation of the non–bonded forces. But, we wish to improve this in later modifications to Gromos.

Data parallelism can be exploited simply by subdividing both the `INB` and `JNB` arrays appropriately. Firstly, the arrays themselves can be computed in parallel. The amount of work to be done for each `I` is proportional to `N-I` (compute the distance from atom `I` to the remaining atoms). Thus the work load can be viewed as a triangle that must be divided into trapezoids of roughly equal area. For P processors, one can laboriously verify (see Chapter 2) that the balanced workload for the j-th processor is to compute `INB(I)`, and the corresponding part of `JNB`, for `I` from `L(j)+1` to `L(j+1)` where

$$L(j) = N - \left(\frac{(P-j)N^2 + j}{P} \right)^{\frac{1}{2}}.$$

Once `INB` and `JNB` are computed, the work load to compute the non–bonded forces is likely not well balanced using the indices just described in the scheme for pairlist generation. Thus it is essential to distribute the load in such a way that the cumulative lengths of the parts `JNB` assigned to each processor are roughly balanced. First of all, a global accumulation of `INB` is done (in $\log P$ steps on a hypercube) which provides each processor a complete picture of the total work load. The balanced work load can be visualized as an atom sequence `JNB` that needs to be divided into groups of whole subunits (`JNB` segments) consisting of the neighbors of blocks of atoms, with the

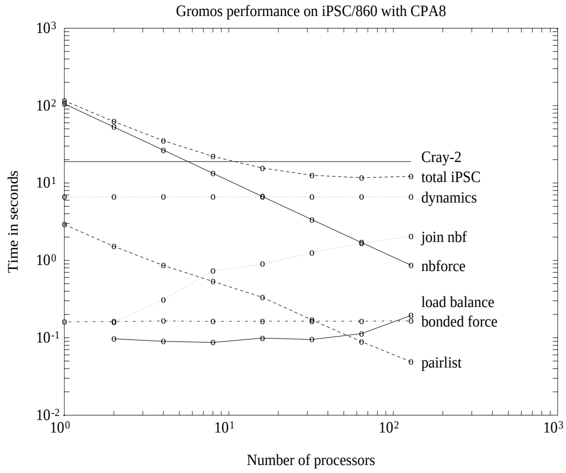

Figure 7.1: UHGromos performance on the iPSC/860.

lengths of the subunits approximately the same for each processor. In the benchmark described subsequently, each atom has typically at most on the order of 100 neighbors. However, it is hard to predict how imbalanced the load could be before balancing is done (in contrast to the balancing for the pairlist generation).

Once the load is balanced, the non–bonded forces are computed in parallel and independently accumulated at each processor. After calculating the non–bonded forces for their assigned pair interactions, the processors participate in a global–sum operation resulting in an identical copy of the force array at each node. The time–step, which involves solving a constrained ordinary differential equation, is then executed redundantly at each processor. This rather crude approach will be refined dramatically in the next version of the code.

7.3 UHGromos Results

We used as a test problem a standard benchmark for the Gromos code, *CPA8* [47]. The problem involves ten time steps with 2382 atoms, with only one pairlist generation (and corresponding load–balancing). The results of the computation on the iPSC/860 are summarized in figure 7.1, which indicates the times of the various components of the simulation as well as the total time as a function of the number of processors used. For reference, a horizontal line is drawn to indicate the time for the benchmark problem that has been reported for the Cray–2. Note that the performance of a sixteen–processor iPSC/860 exceeds the Cray–2 on this benchmark.

Figure 7.1 shows the nearly $\mathcal{O}(N^2)$ part of the computation; the pair list computation also has nearly ideal performance. No other parts of the code were parallelized, and these clearly limit efficiency using more than, say, 32 processors on this benchmark problem. Calculations were also done with the same atomic data, but with a cut–off radius of twice the size. This increases by a factor of eight the amount of computation in the nonbonded–force calculation, and the code remains efficient for a much larger number of processors. If no cut–off radius were used, this code would have a very high scaled speed–up as a function of the number of atoms (see chapter 8).

Chapter 8

Communication with the replicated algorithm

The purpose of this section is to examine the practicalities of parallelizing the basic algorithms of molecular dynamics for distributed-memory multiprocessors using annotations to sequential Fortran programs.[1] This set of algorithms represents an important class of unstructured problems in scientific computation. In general, however, unstructured computations are difficult to map onto parallel systems using either automatic or hand-written techniques. Parallelizing molecular dynamics is therefore a problem of great intellectual and practical importance.

The greatest potential for parallelism in many scientific codes is due to data parallelism, meaning operations applied to all elements of a large data structure. As we will see later, this is also the case with molecular dynamics. Exploiting data parallelism on distributed-memory MIMD machines also requires careful partitioning of the data and computation to exploit locality. This locality often proves beneficial for shared-memory machines [72].

A number of language extensions have been studied which allow the programmer to provide such distribution information. In this section, we examine the facilities available in two such extended languages, IPfortran [8] and Fortran D [42]. We consider the communication costs and interprocessor accesses to arrive at an intuitive understanding of distribution and blocked transfers. We also prove that the replicated algorithm has a fundamental limit to the application of processors based on properties inherent to the algorithm: algorithmic approaches are needed to overcome the built–in limit. We also consider the tradeoffs of parallelizing existing sequential code ("dusty

[1] Material in this section comes from [27].

deck") and writing a parallel program from scratch. Our examples will illustrate the need for modification of algorithms to achieve scalability. Finally, this section gives a detailed description of the basic molecular dynamics algorithm, useful for this and later sections.

The remainder of this section is organized as follows. Section 8.1 describes the GROMOS code [47], a standard molecular dynamics program that we have parallelized [25, 26]. Section 8.2 gives a short description of the parallel languages used, IPfortran and Fortran D. Sections 8.3 and 8.4 describe the parallelization of key phases of GROMOS. Section 8.5 gives some performance results, and we discuss in Section 8.6 some lessons learned from those projects.

8.1 Molecular dynamics

First developed for simulating atomic motion in simple liquids, molecular dynamics is used routinely to simulate biomolecular systems [77]. Using the compute-intensive data obtained from a molecular dynamics simulation, various kinetic, thermodynamic, mechanistic, and structural properties can be obtained [78]. In molecular dynamics, the motion of each atom, represented as a point mass, is determined by the forces exerted on it by other atoms.

Molecular dynamics algorithms commonly iterate over the sequence:

1. Calculate bonded and nonbonded forces on each atom as the analytical gradient of a potential-energy function of the atom positions. The pairwise, nonbonded interactions dominate the computation with $\mathcal{O}(N^2)$ time complexity [25] and therefore are key considerations in both the model and its implementation [26], see Sections 8.3 and 8.4.

2. Integrate Newton's equations of motion to determine the new atomic momenta and positions. By removing uninteresting, high-frequency motions, larger timesteps can be taken resulting in a more efficient computer utilization [80, 89]. This usually involves the constraining of some molecular motions [89].

3. Save data to disk as appropriate for post analysis.

The molecular dynamics program used in this study is from the GROMOS (GROningen MOlecular Simulation) suite designed for the dynamic modeling of biomolecules [47]. GROMOS provides programs for the simulation of biological molecules (and arbitrary molecules) using *molecular dynamics* or *stochastic dynamics*. In addition, *energy minimization* and *analysis* programs are provided. The approximately 127 files comprising GROMOS consist of about 74,000 lines of FORTRAN77, comments included.

8.2 Parallel languages

Both ℙfortran and Fortran D aim at providing a more convenient interface for programming parallel machines. Users rightfully expect compilers to handle the low-level, machine-dependent details of programming. A familiar example of this is register allocation on sequential machines. On parallel architectures, an equivalent challenge is generating synchronization and interprocessor communication operations. Both languages provide support for this, in contrast to other languages which required those operations to be programmed explicitly. The languages differ, however, in that one uses a local memory model and the other uses a global–memory model as well as in other respects.

ℙfortran utilizes a local memory model [5]; the key concept of ℙfortran is to provide a better abstraction for interprocessor communication than simple message-passing [55]. ℙfortran programs use an SPMD (Single-Program Multiple-Data) style of programming. Variables are implicitly local to each processor; thus, X on processor 1 may have a different value from X on processor 2. Nonlocal accesses are denoted by the @ operator, so $A(i)@j$ means the i-th element of array A on processor j. Note that only the processor *using* a nonlocal value must reference it, and that the reference may be made within a larger expression. This is in contrast to message-passing languages, which require matching but separate "send" and "receive" operations. Global reductions are also supported. For example, $+\{X\}$ denotes the sum of the set of all values of X on all processors. This collection of nonlocal access and reduction operations supports programming at a convenient level of abstraction, while still allowing a relatively simple compiler to produce excellent code.

Fortran D utilizes a global memory model, providing a modified shared name-space for array elements [42]. Arrays are declared to be their full, global size and are *aligned* with virtual *decompositions* which are *distributed* across processors. Statements are executed sequentially (except for the **FORALL** loop, for which the iterations conceptually execute simultaneously). The compiler must detect and exploit opportunities for parallel execution, in addition to inserting any necessary communic-

```
DO I = 1, N
   DO J = firstJ(I), lastJ(I)
      force = nbf(X(I) - X(JNB(J)))
      F(I) = F(I) + force
      F(JNB(J)) = F(JNB(J)) - force
      ENDDO
ENDDO
```

Figure 8.1: Sequential form of the nonbonded force calculation.

ation.

8.3 Nonbonded force calculation

The GROMOS code approximates nonbonded forces by calculating them only for atom pairs which are within a certain *cutoff* radius of each other. As described in Section 8.4, these pairs are stored in a pair list which is updated in regular intervals. In the original code, this pair list is represented by two arrays, INB and JNB. $INB(I)$ gives the number of partners of atom I, and JNB can be thought of as a concatenation of lists of partners, one list for each atom. We also introduce the arrays $firstJ$ and $lastJ$, so that the array section $JNB(firstJ(I) : lastJ(I))$ gives the list of partners of atom I. Obviously, $INB(I) = lastJ(I) - firstJ(I) + 1$.

An important optimization is due to the fact that for each force exerted by an atom A on an atom B, atom B exerts an equal but opposite force on atom A. We therefore can cut the number of force calculations in half by storing each atom pair only once, for example by storing only partners with a higher atom index. The resulting sequential version for N atoms is shown in Figure 8.1, where we assume that the force array F is initialized to 0 (similarly in the following code samples). Note also that in practice the forces F and the positions X are vectors in $I\!R^3$.

In the IPfortran implementation [26], each processor executes the outer loop of the sequential version for a range $(firstI(me) : lastI(me))$ of atom indices. (Here, as throughout the text and the code samples, me stands for the local processor id, and P stands for the total number of processors.) This range is determined in an initial call to a load balancing routine, of which gathering the corresponding portion of JNB, namely $myJNB$, is the most time consuming part. The code finishes with accumulating forces across processors, see Figure 8.2. An important point to keep in

```
DO I = firstI(me), lastI(me)
   DO J = firstJ(I), lastJ(I)
      force = nbf(X(I) - X(myJNB(J)))
      F(I) = F(I) + force
      F(myJNB(J)) = F(myJNB(J)) - force
   ENDDO
ENDDO
F = +{F}
```

Figure 8.2: Force calculation, IPfortran version. The JNB array is now partitioned into local portions $myJNB$.

mind is that the atom numbering and the resulting pair list do not inherently have a good locality, *i.e.*, atoms close together in space do not necessarily have similar numbers. This is inherited from sequential GROMOS. Another parallel version we are currently developing overcomes this limitation using a *hierarchical decomposition* (see chapter 9).

Note that this implementation *replicates* the force array F. A Fortran D version of this kind of algorithm can be written by expanding each array by one dimension (the *processor dimension*), the introduced index being the processor number, and then distributing that dimension blockwise. The resulting code is shown in Figure 8.3.

However, this approach does not really take advantage of the Fortran D data distributions, and it ultimately limits our scalability. Instead of replicating the force array F, we should distribute it to distribute the workload. To allow load balancing, we distribute the data irregularly using a mapping array Map [42], for which

$$firstI(p) \leq I \leq lastI(p) \Leftrightarrow Map(I) = p.$$

must hold.

We should also rethink how to distribute the neighbor list *JNB*, which represents the largest data structure of the problem. In the IPfortran version this was done by replacing the global pair list *JNB* with just portions thereof, *myJNB*. We then modeled that with our first Fortran D version by adding an extra processor dimension. In our second Fortran D version, we also use a two dimensional structure, *JNBL*, but now the second dimension does not represent a processor number, but a partner number instead. So, *JNBL*$(I,:)$ represents all of the partners of atom I whose atom

```
DECOMPOSITION AtomD(N, P), PairD(MP, P)
DISTRIBUTE AtomD(*, BLOCK), PairD(*, BLOCK)
ALIGN FF with AtomD, myJNB with PairD

FORALL me = 0, P-1
   DO I = firstI(me), lastI(me)
      DO J = firstLocJ(I), lastLocJ(I)
         force = nbf(X(I,me) - X(myJNB(J,me)))
         FF(I,me) = FF(I,me) + force
         FF(myJNB(J,me),me) = FF(myJNB(J,me),me) - force
      ENDDO
   ENDDO
ENDFORALL

FORALL me = 0, P-1
   FORALL I = 1, N
      REDUCE(SUM, F(I), FF(I,me))
   ENDFORALL
ENDFORALL
```

Figure 8.3: Force calculation, Fortran D version 1. MP is the maximal number of pairs per processor.

number is greater than I. We have

$$JNBL(I, 1 : INB(I)) = JNB(firstJ(I) : lastJ(I)).$$

We guide the compiler by using a **FORALL** loop to indicate the fact that the operations are independent. For enhanced locality, we combine that with an **ON** clause. The resulting code is shown in Figure 8.4.

It is worthwhile to aid our intuition about actual costs of computation and communication and about locality in general with some analysis. We start with defining a predicate which indicates whether an atom pair (I, K) is stored in the pair list:

$$isPair(I, K) = \begin{cases} 1 & \text{if } \exists J,\ JNBL(I, J) = K, \\ 0 & \text{otherwise.} \end{cases}$$

We are also interested in how many force calculations involving atom K we have to perform on processor p:

$$Partners(p, K) = \sum_{I=firstI(p)}^{lastI(p)} isPair(I, K).$$

```
DECOMPOSITION atomD(N), PartnD(N, MaxINB)
DISTRIBUTE atomD(Map), PartnD(Map, *)
ALIGN F,X WITH atomD, JNBL WITH PartnD

FORALL I = 1, N ON HOME F(I)
   DO J = 1, INB(I)
      force = nbf(X(I) - X(JNBL(J))
      REDUCE(SUM,F(I),force)
      REDUCE(SUM,F(JNBL(I,J)),-force)
   ENDDO
ENDDO
```

Figure 8.4: Force calculation, Fortran D version 2.

For each processor p, we sum this up over all atoms:

$$Pairs(p) = \sum_{K=1}^{N} Partners(p, K) =$$

$$\sum_{I=firstI(p)}^{lastI(p)} \sum_{K=1}^{N} isPair(I, K) = \sum_{I=firstI(p)}^{lastI(p)} INB(I).$$

In terms of averages (which are denoted by a subscript ave), we have

$$Pairs_{ave} = \frac{1}{P} \sum_{I=1}^{N} INB(I) = \frac{N \times INB_{ave}}{P}.$$

We notice that $Pairs(p)$ is proportional to the computational load of processor p. Therefore, the overall computational cost is given by

$$T_{comp} \propto \max_{p=1}^{P} Pairs(p) = \max_{p=1}^{P} \sum_{I=firstI(p)}^{lastI(p)} INB(I).$$

This results in a typical load balancing problem [50], where the goal is to lower T_{comp} down to $T_{ideal} \propto Pairs_{ave}$. This means that we have to choose $firstI$ and $lastI$ such that we have for all p:

$$\sum_{I=firstI(p)}^{lastI(p)} INB(I) \approx \frac{N \times INB_{ave}}{P}$$

After considering the computational costs, we now take a closer look at communication costs and, closely related to that, scalability. Most of the overall communication is associated with the potentially nonlocal assignment to the force array (statement 9 in Figure 8.4). Compiling this naïvely, without message blocking, would result in roughly $Pairs_{ave}$ (short) messages per processor. For systems whose communication time to send m units of data is well modeled by $\lambda + \beta m$ with $\lambda \gg \beta$ [17], this would increase the communication cost by a factor λ/β. This would make it unacceptable to send individual messages for each nonlocal access, instead of combining them at the end of the loop.

Another issue besides raw message blocking is how much we can gain by combining non-local reductions. For example, if processor p has to make several contributions to an $F(I)$ owned by processor q, $p \neq q$, then it would be profitable to combine these contributions on p which then sends only the sum to q. We should estimate how often this may occur. While the assignments may be skewed for various reasons (for example, because a pair (I, K) is stored only if $I < K$), the average number of assignments which involve a particular atom, per processor, is simply $Pairs_{ave}$ divided by N:

$$Partners_{ave} = \frac{INB_{ave}}{P}.$$

When $INB_{ave} \leq P$, $Partners_{ave}$ can be interpreted as the probability of writing any given F(I).

Thus for $P \ll INB_{ave}$, we can expect every processor to contribute to most elements of F; for $P \gg INB_{ave}$, each processor contributes to very few elements of F. Therefore, the ratio $Partners_{ave}$ is crucial for how we should distribute F and perform the reduction operation, which we will discuss subsequently.

In the IPfortran version and the corresponding first Fortran D version, F is replicated and all local contributions to an element of F are summed up locally. After computing the global sum $\{F\}$, all forces are everywhere available. Using a simple dimensional exchange, this summation can be done in $\mathcal{O}(N \log P)$, with $\log P$ messages per processor (this works best for hypercubes, but can also be done on other parallel architectures). A more sophisticated divide-and-conquer approach works in $\mathcal{O}(N)$, with $2 \times \log P$ messages per processor [43]. If we use the latter approach and have a balanced workload, then we have communication cost $T_{comm} \propto N$ and computation costs $T_{comp} \propto Pairs_{ave}$. Their ratio then is

$$R_{comp/comm} = Pairs_{ave}/N = INB_{ave}/P,$$

which is again $Partners_{ave}$. Thus for fixed INB_{ave}, the cost of communication will dominate for large P, independent of N. In the experiments with GROMOS parallelized in this way, for a system with $INB_{ave} \approx 80$, the cross-over point occurs near 128 processors on the Intel iPSC/860, see Figure 8.7. However, INB_{ave} increases as the cube of the cutoff radius, so that the break-even number of processors, for which $P \propto INB_{ave}$, increases cubicly with the cutoff radius as well. In the limit of having an infinite cutoff radius (that is, no cutoff at all), we have $INB_{ave} = N$ and the communication costs are swamped by the computational cost for any $N \gg P$. In that case the algorithm is fully scalable as N increases.

In our second Fortran D version we do not replicate F, but instead distribute it across processors. However, to allow message blocking, the compiler should still provide buffer space for all elements of F which are accessed nonlocally. The global combination would then be performed via a reduction operation like *scatter_add* [12]. A typical implementation of *scatter_add* would again sum all local contributions to a particular array element up before combining them globally with other contributions. However, it would probably not use a dimensional exchange, but instead send point-to-point messages. The size of each of these messages would be $\mathcal{O}(N/P \times \min(1, Partners_{ave}))$, where the limiting 1 stems from combining reductions locally. Whether the overall cost of a *scatter_add* would be less than a global exchange of the full array depends on $Partners_{ave}$.

To summarize, it appears that distributing F is advantageous when $P \geq INB_{ave}$ roughly holds, otherwise replicating F is appropriate.

8.4 Pair list generation

Figure 8.5 shows the original version of the generation of the pair list used in the nonbonded force calculation. The critical obstacle for parallelizing this algorithm is the loop carried dependence on JJ, which causes a dependence for the assignment to JNB. However, the dependency does not occur in the *values* of JNB, but rather in the *locations* of the values. Thus we may compute each iteration separately and figure out later where to put them. This amounts to computing the $JNBL$ array which appeared already in Figure 8.4. $JNBL$ serves again to remove dependences as shown in Figure 8.6, similarly to scalar privatization. Here $JNBL$ can also be viewed as a "scratch space" to hold the values which are then combined into JNB. One difficulty in doing this automatically is that the second dimension of $JNBL$ has to be large enough to fit $max_{I=1}^{N} INB(I)$ elements, for which there is no *a priori* bound that could

```
J0 = 0
JJ = 0
DO I = 1, N
   firstJ(I) = JJ + 1
   DO J = I+1, N
      IF | X(I) - X(J) |< R THEN
         JJ = JJ + 1
         JNB(JJ) = J
      ENDIF
   ENDDO
   INB(I) = JJ - J0
   lastJ(I) = JJ
   J0 = JJ
ENDDO
```

Figure 8.5: Pairlist computation, sequential version.

```
DO I = 1, N
JL = 0
DO J = I+1, N
   IF | X(I) - X(J) |< R THEN
      JL = JL + 1
         JNBL(JL,I) = J
      ENDIF
   ENDDO
   INB(I) = JL
   J0 = JJ
ENDDO
⋮

DO I = 1, N
   JNB(firstJ(I):lastJ(I)) = JNBL(I, 1:INB(I))
ENDDO
```

Figure 8.6: Pairlist computation, parallel–vector version.

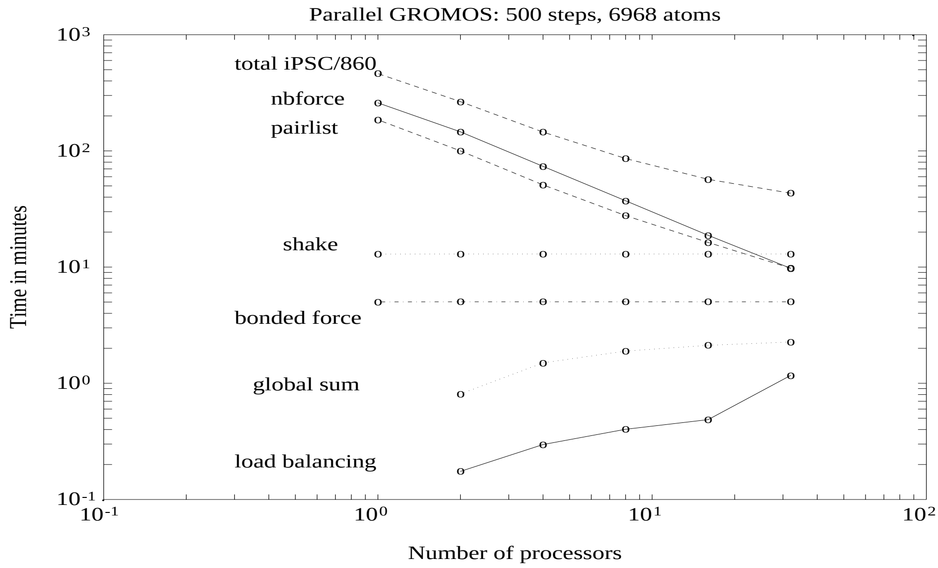

Figure 8.7: UHGROMOS performance results on the iPSC/860.

be easily derived by a compiler.

Now we can parallelize the outer loop by letting each processor p compute a range $(firstI(p) : lastI(p))$ of the iterations. In ℙfortran we use processor specific loop bounds; the Fortran D version distributes $JNBL$ and uses the owner computes rule. The range sizes vary across processors due to the triangular shape of the double loop [26]. Each processor will have a local copy of only *part* of the JNB array. However, for performing load balancing we need to know all of INB, where the $\mathcal{O}(N)$ data can be collected in $\mathcal{O}(N + \log P)$ time dimensional exchange [103].

8.5 Performance results

Figure 8.7 gives performance results for the parallel implementation of GROMOS using ℙfortran. The calculation on an iPSC/860 uses a model for the enzyme Superoxide Dismutase [96], with a total of 6968 atoms, for 500 timesteps. Both the overall execution times and the breakdowns into the principal sections of the calculation are given.

The dominating parts of the sequential algorithm, the nonbonded forces and pair list, have been parallelized with nearly perfect speedup. Time-stepping and the bonded force calculation, performed redundantly at every processor, remain constant. These, together with the load balancing and global sum of the nonbonded force, form an asymptote which the total time approaches as the number of processors is increased. As the problem size, N, is increased, the pair list construction (and the long-range force calculation which is computed along with the pair list) grows as $\mathcal{O}(N^2)$; all other parts increase by $\mathcal{O}(N)$, with the exception of load balancing which depends on initial data and, relative to this discussion, is difficult to characterize. Thus we can assert that this simple port of the GROMOS code is an efficient parallelization. Increasing the cutoff radius while keeping N fixed also increases the number of processors that can be used effectively, as discussed in Section 8.3, thereby shifting the nonbonded force curve up by some δ.

8.6 Conclusions

Because this study was limited to a single example program and two languages, we cannot make broad generalizations regarding parallel programming. However, we believe that for a range of scientific codes whose complexity is comparable to molecular dynamics the following observations will apply.

- Scalability in molecular dynamics is an achievable goal, but it requires careful algorithm design where the choice of the right algorithm may also depend on the input characteristics.

- Both local and global models are feasible interfaces for programming distributed-memory parallel machines [94]. In addition, both models can provide the user with a higher-level programming interface than current message-passing languages on distributed-memory machines.

- As to be expected, regardless of the implementation model chosen, the key to a good parallel program is the choice of an appropriate algorithm. This implies that programs will have to be rewritten to some extent for parallelization, rather than relying on compiler optimization of "dusty deck" code. Resembling the "vectorizable style" associated with vector architectures [109], an equivalent challenge in the parallel arena is the development of a "machine-independent parallel programming style," a long-term research target of Fortran D [54].

- IPfortran improves on the usual message-passing environment by eliminating the need for explicit "send" operations. This simplifies code significantly, allowing the programmer to concentrate on higher-level problems such as load balancing. The local model of computation is useful if we want to apply distinct "intra-processor" and "inter-processor" algorithms.

- Fortran D abstracts communication statements out of the program text by providing a modified shared-memory model of computation. The key feature of a Fortran D program is the data distribution, which the compiler uses to generate the low-level communications operations.

Chapter 9

EulerGromos: a spatial decomposition of GROMOS

The replicated algorithm described in Chapter 8 limits the scalability by requiring a global sum of the force array, and storing O(N) data at each processor. A natural, but more complex, algorithm for nearest–neighbor molecular dynamics using a spatial decomposition in order to restrict communication to a local neighborhood of processes. In addition, space can scale as $\frac{N}{P}$.

In this chapter we describe our implementation of the spatial decomposition using IPfortran. The program, EulerGromos, used roughly 196 @ operators, primarily in distributing input data; there are 65 reduction operations, consisting of one user defined function, six FORTRAN77 intrinsics, and 58 sums.[1]

9.1 Introduction

An MD simulation applies Newton's equations of motion to a molecular system to determine a new set of positions and velocities for the atoms at each timestep.[2] The calculation of a single timestep involves an iteration over several major phases in the MD program. EulerGromos parallelized all of those major phases, including numerical integration with constraints (or SHAKE [27]), pairlist construction, and the computation of non-bonded forces (NBF). Since most MD runs perform the bulk of the work (around 90%) in the NBF routine, this phase is of particular interest. A common technique to accelerate the NBF calculation is to ignore all NBF interactions

[1] The global sum is often used in EulerGromos to assign process values into an array by defining off processor elements to be zero. A global sum then distributes the values among processors.

[2] Material in this section comes from [24].

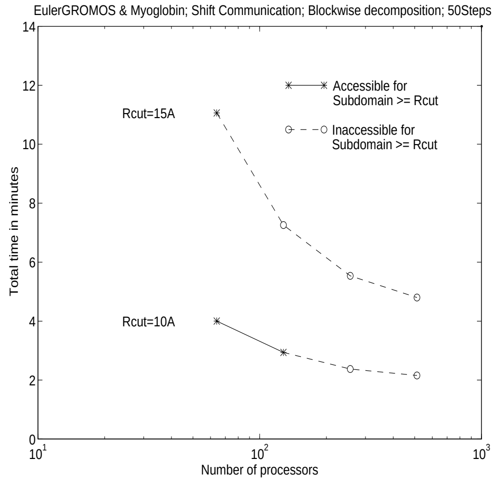

Figure 9.1: Processor accessibility limitation for the standard parallel link-cell method due to subdomain size demonstrated for myoglobin in a 50Å^3 box.

beyond a certain *cutoff radius*, R_{cut} [85]. This in turn provides an access locality that makes Eulerian, application space oriented data distributions desirable. An Eulerian, or geometric, decomposition assigns application space to processors, as compared to Lagrangian mappings where particles are assigned to processors.

A typical limitation of spatial-decomposition parallelizations of codes based on a cutoff-radius approximation is to restrict communication to nearest neighbors. Consequently, each processor subdomain has to be greater than or equal to the cutoff radius size (see Section 9.5). For a fixed problem size, this restriction imposes an upper bound on the number of processors that can be used. However, as shown in Figure 9.1, EulerGromos can make effective use of more processors than would be allowed under this restriction. EulerGromos, with the Gromos molecular dynamics program for its MD kernel, uses a spatial decomposition, but does not limit communication to nearest neighbors. Each processor's overlap region (of thickness R_{cut}) is allowed to penetrate several processor subdomain layers. Figure 9.1 shows that substantial performance gains would not be available otherwise.

EulerGromos provides a choice between two different communication algorithms, one of which (based on shifts) becomes faster as the number of processors increases. Efficient communication, together with low parallelization and distribution overheads, allowed speedup gains with as few as approximately 20 atoms to a processor, including all phases of the molecular dynamics with constraints and I/O. For comparison, typical values for the density ρ of 0.1 $atoms/\text{Å}^3$ and R_{cut} of 15Å would result in a minimum of about 300 atoms per processor subdomain if only nearest neighbor communication were allowed. Another characteristic of EulerGromos is the use of dynamic load balancing, which adjusts individual subdomain sizes to accommodate inhomogeneous atom densities.

The implementation of EulerGromos is described in more detail in Section 9.2. Section 9.3 evaluates the characteristics of the resulting program, presenting performance results for three biomolecular systems. One of these systems, the solvated acetylcholinesterase dimer (AChE) shown in Figure 9.2, contains over 100,000 atoms, about an order of magnitude more than typically simulated. Section 9.5 discusses related work.

9.2 EulerGromos

Imposing a granularity on the problem domain

Using an Eulerian decomposition for distributing atoms and the computations associated with them improves inter-processor locality, which in turn increases scalability and reduces communication costs. However, to make good use of the intra-processor memory hierarchy as well (*i.e.*, to reduce cache misses), increasing access locality is also desirable within a processor. We therefore conceptually divide our overall *problem domain*, which here is the physical space occupied by the set of atoms that we want to simulate, into small rectilinear regions of fixed size, henceforth called *subboxes*. There are n_d subboxes along dimension d, resulting in a total on $n_1 * n_2 * n_3$ subboxes. Each subbox contains a list representation of the atoms resident within its spatial extent [37]. To amortize some of the data access overhead, the linked lists are packed densely and in subbox order for linear traversal and better cache locality.

For each physical dimension d, the number of subboxes, n_d, and their size, box_d, depend on several parameters including the number of processors P $(= p_1 p_2 p_3)$, the mapping strategy, the number of atoms N, the cutoff radius R_{cut}, and a user supplied granularity parameter. There are several tradeoffs and constraints to be observed. We distribute our problem domain across processors with *subbox granularity*, *i.e.*, a certain subbox is treated as indivisible as far as ownership goes, and we assume only one owner per subbox. Therefore, if n_d becomes smaller, load balancing may become less accurate, since the number of different decompositions becomes smaller.

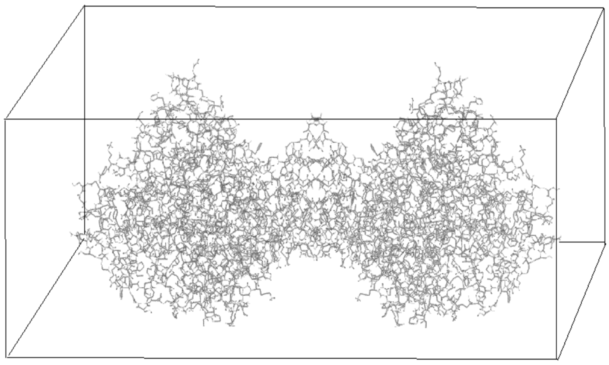

Figure 9.2: The 10,406 atom acetylcholinesterase dimer (AChE) is simulated in a $91 \times 97 \times 160\text{Å}^3$ box. The 121,257 solvent atoms are removed for clarity, with a corresponding box size reduction in the picture.

However, if n_d becomes larger, the overhead associated with a traversal of the subboxes to locate the atoms increases. We also use our subbox structure to limit our search for nonbonded interaction partners of a given atom, which allows us to avoid the naïve $\mathcal{O}(N^2)$ pairlist generation algorithm [26, 99]. For that purpose it is advantageous if box_d is an integral fraction of R_{cut} [83].

The hierarchical decomposition should also be able to balance the workload for the trivial case of a system with constant density. Therefore it must be possible to create subdomains of equal size; for all d, n_d should be a multiple of p_d.

Subdomains, each consisting of a connected set of subboxes, are assigned to processors according to some space-to-processor mapping strategy. Each subdomain s is associated with a certain *overlap area*, which is the set of subboxes that are not in s but reach into the cutoff radius of some subbox in s. Figure 9.3 shows an example configuration of subboxes and subcubes for $P = 64$ processors.

Overview of GROMOS modifications

A library of routines that perform and support the Eulerian decomposition and load-balancing interface with GROMOS. The modifications to GROMOS itself were minimal,

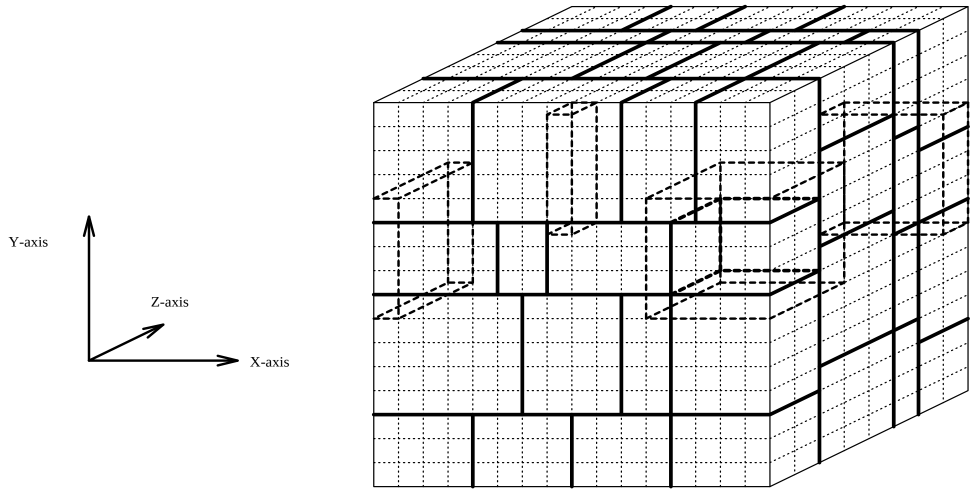

Figure 9.3: Subbox division of the problem domain; we use $P = p_1 p_2 p_3 = 4 * 4 * 4$ processors and $n_{tot} = n_1 n_2 n_3 = 16 * 16 * 8$ subboxes. Dotted lines indicate subboxes, heavy lines delineate processor subdomains. Hashed-lined regions show the overlap area of the processor with logical coordinate $(4, 3, 1)$, which has a subdomain consisting of $4 * 3 * 2$ subboxes located at the center of the edge closest to the reader; note the wrap-around of the overlap due to periodic boundary conditions.

permitting reuse of at least 90% of the approximately 11,000 lines of original code. From the user's perspective, the GROMOS touch and feel are retained; I/O formats are close to identical.

With EULERGROMOS, a processor calculates all interactions involving its local atoms. The nonbonded interactions are determined geometrically, while the bonded interactions are obtained from the molecular topology. The scalability depends on the cost to determine the interactions a processor should calculate and the calculation of those interactions. To determine the nonbonded interactions, the subbox data structure is traversed to create a standard GROMOS pairlist containing the interactions. As a result, the pairlist routine required extensive modification and was rewritten. The nonbonded force routine, which uses the pairlist, was slightly modified.

Based on Newton's Third Law, GROMOS and other MD programs calculate each NBF interaction only once per pair, instead of twice [38, 99, 52]. Our present implementation uses Newton's Third Law within subdomains, but not across subdomains. However, we are currently implementing a version that exploits Newton's Third Law across subdomains; our preliminary analysis indicates a potential for significant savings.

Each processor extracts its local set of bonded interactions by scanning the $\mathcal{O}(N)$

global topology information. While this could limit scalability, such asymptotic limitations are outside of our practical range for N and P (see Section 9.3). Scalability of the calculation of the potential energy (i.e., the interactions) requires a reasonable load balancing, as provided by our hierarchical load-balancing scheme (Section 9.3).

Load balancing

To allow for both inhomogeneous systems and systems that change shape over the course of a simulation, dynamic load balancing is applied to the distribution of data and computation. At the beginning of a simulation, the problem domain is first divided into p_3 slices along the z-axis, then each slice is divided along the y-axis into p_2 columns, and finally each column is divided into p_1 subdomains. This initial distribution is equivalent to a blockwise decomposition into $p_1 p_2 p_3$ subdomains of equal size. Every f_{bal} timesteps, each processor locally computes its own workload. (The current heuristic measures workload as the sum of the local number of atoms and the size of the local pairlist.) Given each processor's workload, a global balancing step adjusts the subdomain boundaries, assuming a homogeneous workload density within each subdomain. This is done hierarchically: first, subdomain boundaries are shifted between slices, followed by column shifts within slices and finally shifts within columns. Figure 9.3 shows a possible hierarchical decomposition. A smoothing factor σ is applied to avoid boundary oscillations [50].

Communication

EulerGromos communicates off-processor data accesses in one of two possible ways: *Point-to-Point* or *Shift*. In Point-to-Point communication, messages are sent directly between processor pairs that share data; in Shift communication, each processor communicates exclusively with its six immediate logical neighbors, relying on those to forward data to other processors it has to exchange data with. This results in a coordinated use of the interconnection network.

The Shift algorithm shifts data in three *phases*: first along the x axis, then along the y axis, and finally along the z axis. For dimension d, phase d consists of k_d subphases, $k_d = \lceil RcutU_d / procU_d \rceil$, where $procU_d$ and $rcutU_d$ are the subdomain extent and the cutoff-radius, respectively, for dimension d in subbox units. A subphase consists of an exchange with the two neighbor processors along the axis of the current phase. Within each phase, the first subphase will shift all data received in all previous phases; within phase one, this will be the set of local atoms, the base case. For all subphases after the first subphase, a processor will communicate to each of its two neighbors the data that were received in the immediately preceding subphase from the other neighbor. Let $k_1 = k_2 = k_3 = k$; *i.e.*, let the cutoff radius penetration be uniform along the three Cartesian axes. Assuming that N atoms are uniformly distributed across P processors, the amount of data received by each processor to exchange overlap atoms is $(8k^3 + 12k^2 + 6k)N/P$. For $k = 1$, the preceding expression evaluates to $26N/P$, the case where each processor subdomain is greater than or equal to R_{cut}. Note that for fixed N and large P, k is asymptotically proportional to $P^{1/3}$, and increasing P decreases the amount of data exchanged.

The scalability of a communication algorithm generally depends on communication volume, bandwidth, latency, and buffering costs. However, for the problems and processor configurations considered here, the communication volume and the number of messages drive the performance. Again assuming a uniform k_d in each dimension, each processor sends $6k$ messages using Shift, compared to approximately $(2k+1)^3$ messages sent with Point-to-Point communication.

Shift communication can be expected to be most efficient at large P, where the data volume communicated per processor decreases. However, the current implementation of the Shift communication requires subdomains of equal size, therefore it cannot be used in conjunction with the load balanced, hierarchical decomposition. Point-to-Point communication can reduce the overall communication volume to a processor, since subbox granularity is used to buffer overlap data.

9.3 Evaluation and Experiments

The applications

We base our evaluation on two molecular simulations that are of current interest in the molecular dynamics community: myoglobin with a 10Å cutoff radius in a $50 \times 50 \times 50\text{Å}^3$ box; and the 131,660 atom AChE system with a 10Å cutoff radius in a $91 \times 97 \times 160\text{Å}^3$ box. (Myoglobin and AChE were simulated using the lowest number of processors that memory constraints would allow.) In addition, a toy dipeptide system with 337 solute and solvent atoms in a $14 \times 15 \times 19\text{Å}^3$ box with a cutoff radius of 7Å was used. All performance data are reported for 100 molecular dynamics timesteps unless otherwise noted. For $P \leq 8$, we used a local iPSC/860 hypercube; runs with $P > 8$

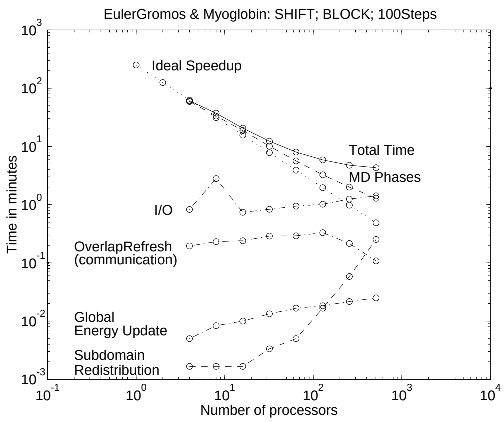

Figure 9.4: Performance of EulerGromos on the 10,914 atom myoglobin system.

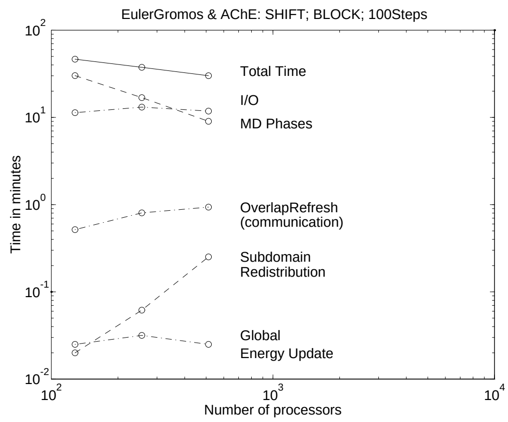

Figure 9.5: Performance of EulerGromos on the 131,663 atom acetylcholinesterase system.

were performed on the 512-node Touchstone Delta mesh at Caltech. Both machines use i860 processors configured with 16 megabytes per node, but the amount of memory available to the application varies: the Delta provides about 12 Megabytes per node, the iPSC/860 provides about 15 Megabytes.

Overall performance

Figure 9.4 and 9.5 shows the raw performance of EulerGromos on myoglobin and AChE, with I/O required for production simulations included in the total time.[3] Note the log-log scale and the processor ranges. MD phases in the figures include: integration, pairlist construction, shake, bonded and nonbonded force calculations. Subdomain redistribution uses point-to-point communication; overlap refresh uses Shift communication. A line with negative unit slope allows comparison to ideal performance.

Due to memory limitations, AChE required a minimum of 128 processors, myoglobin required four. These plots demonstrate a continuous performance improvement up to $P = 512$ for both systems, the largest machine configurations available for the experiments. The curve labeled *MD Phases* for myoglobin in Figure 9.4 deviates from ideal speedup due to redundant interaction calculations and load imbalances (note that those data are from a simulation without load-balancing). However, that deviation is greater for the myoglobin system, where the smaller subdomains lead to greater load

[3] I/O includes coordinates and velocities output every 10 steps and the output of the final configuration.

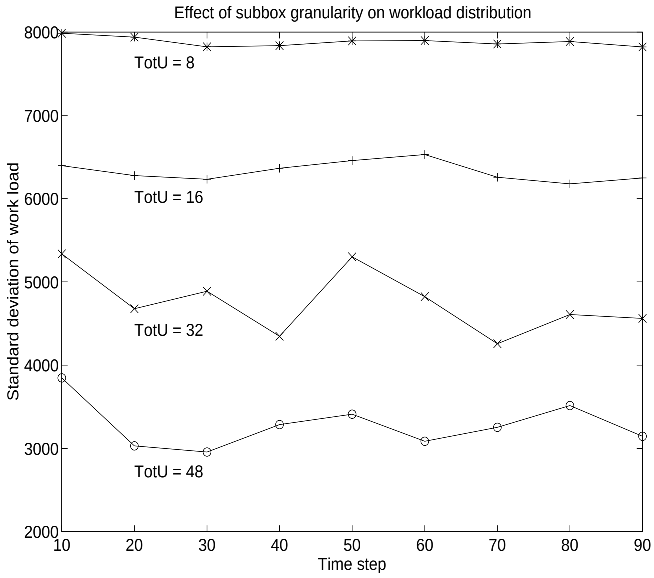

Figure 9.6: Effect of n_d (= TotU) on load balancedness for 64 processors. The mean of the work load is about $5.4 * 10^4$ nonbonded interactions computed per processor per MD timestep.

imbalances and more redundant interaction calculations; the MD phases for AChE demonstrate close to ideal speedup.

Load balancing

Subboxes are viewed as indivisible with respect to subdomain boundaries. In other words, each subbox is owned by just one processor, and boundaries can be shifted only in increments of the subbox sizes. This simplifies and accelerates the mapping between atom coordinates and subdomains, but it also limits the accuracy of our load balancing: the larger the subboxes, the coarser the border shifts will be. To study this effect, we simulated myoglobin ($N = 10,914$) on 64 processors for 100 time steps with varying n_d.

The accumulated absolute border movements along each dimension increase monotonically with n_d. Borders start moving for $n_d \geq 16$. As to be expected, most activity takes place within columns, followed by movements within slices and finally across slices. Figure 9.6 shows the standard deviation of the workload across processors for the different n_d values. It turns out that the standard deviation decreases for higher n_d, so load balancing does achieve the desired effect.

While load balancing can be profitable for typical biomolecular systems, such as

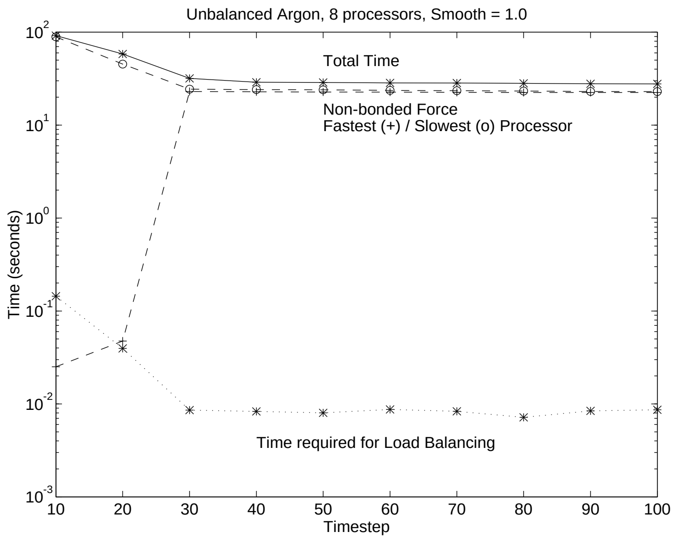

Figure 9.7: Effect of load balancing for Argon in an oversized box. The $N = 10,169$ Argon atoms inhabit one 8 Å^3 corner of a 16 Å^3 periodic box. Load balancing reduces the total time per 10-step interval from 92 seconds to less than 30 seconds.

myoglobin, the advantages tend to be less pronounced there due to the density homogeneity and characteristically long time scales for large-scale motion. Consequently, the real strengths of load balancing are to be expected for systems that are inhomogeneous and change their shape within short time scales, and for varying problem domain sizes (for example, constant-pressure systems).

To study the effectiveness of load balancing for a highly inhomogeneous system, we simulated Argon with an enlarged problem domain. As shown in Figure 9.7, load balancing does adapt very quickly to the inhomogeneity, and after three rebalancing steps we have an even workload on each processor.

Efficiency

We examine the efficiency of EulerGromos with the three systems described in Section 9.3. In Figure 9.8 we have plotted a quantity proportional to the number of atoms divided by the number of processors and the total execution time. Intuitively this corresponds to the number of atoms simulated per processor per unit of time. The principal work in the uniprocessor case is proportional to the number of atoms, N, since the nonbonded force calculation dominates. Thus an approximation for the

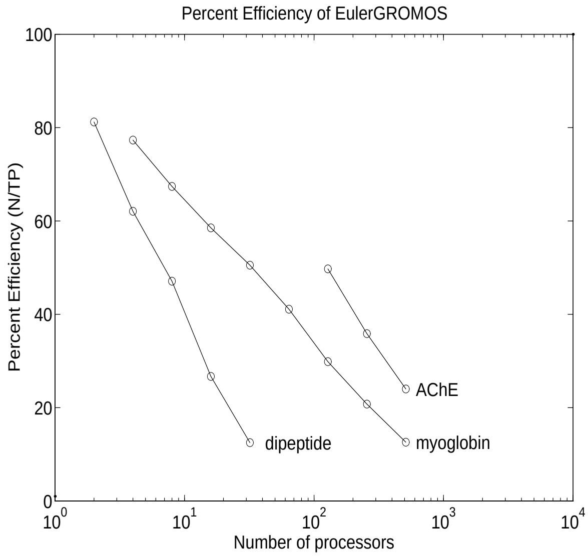

Figure 9.8: Efficiency of EulerGromos for three systems in terms of N/T_pP. The dipeptide nonbonded force time has been scaled by $(10\text{Å}/7\text{Å})^3$, compensating for the smaller cutoff radius.

efficiency of the code is

$$\frac{T_1}{T_pP} \approx \frac{cN}{T_pP},$$

where T_p is the time on P processors and c is a constant.

The ordinate in Figure 9.8 was scaled so that the dipeptide and myoglobin curves intersect P=1 at about 100% efficiency. (We use this strategy in lieu of having data for the single processor case.) For myoglobin, each doubling of the number of processors results in increasing the performance by roughly 90%,

A comparison of the myoglobin and AChE curves in Figure 9.8 illustrate the scalability when varying N for a fixed P. There the myoglobin system size is only 8.4% that of the AChE system, resulting in an efficiency loss of about 20% with 128 processors. A similar drop in efficiency can be observed when comparing myoglobin to the dipeptide. The performance drop for fixed N can be attributed to an increase in interprocessor dependencies relative to a decreasing workload. For a fixed P, decreasing N also results in a performance loss for the same reason: the smaller system takes a larger overhead penalty due to an increase in the communication to computation ratio. Load imbalance is another effect detracting from ideal speedup when decreasing the atom to processor ratio. We did not use load balancing in these runs.

9.4 EulerGromos vs. UHGromos

We are also interested in how EulerGromos performs relative to its cousin, UHGromos [26]. UHGromos is a parallelization of Gromos using the replicated algorithm [26, 99]. The replicated algorithm replicates the full force and coordinate array at each processor. A global sum of the forces is required at every timestep due to a lack of locality [27]. In Figure 9.9, the total time for UHGromos simulating myoglobin is less than the EulerGromos time for $P < 64$, but greater than EulerGromos time for $P > 64$. Note that one should view the *total time* data in Figure 9.9 as a comparison of the two implementations, rather than a comparison of the replicated algorithm and spatial decomposition.

Because of the spatial decomposition, the pairlist calculation is more efficient in EulerGromos for large N. The pairlist calculation grows as $\frac{N^2}{P}$ with UHGromos and as $(\frac{N}{P})^2$ with EulerGromos. For the myoglobin system the pairlist calculation for both codes is about the same (on eight processors its 82 seconds for EulerGromos and 72 seconds for UHGromos). The number of processors at which EulerGromos becomes more efficient than UHGromos will decrease as N increases. Thus we expect AChE to be much less efficient with UHGromos.

A comparison of relative scalabilities of spatial and replicated algorithms using EulerGromos and UHGromos is complicated by implementation details. However, we illustrate an EulerGromos locality benefit by focusing on the nonbonded force calculation and associated overheads. EulerGromos requires two communication phases to manage the atom buffering and ownership required for the nonbonded force calculation. The performance of those phases are shown in Figure 9.4 as *OverlapRefresh* and *Subdomain Redistribution*. For UHGromos, the global sum constitutes the intrinsic overhead required for supporting the nonbonded force calculation and increases with P (Figure 9.9, UHGromos *global communication*). The two EulerGromos communication phases require less time than the UHGromos global communication (Figure 9.9). More importantly, while we use the point-to-point algorithm for subdomain redistribution in our prototype, the better shift algorithm could be used there, with resulting communication costs decreasing further with large P.

Other factors that characterize the scalability of the replicated and spatial approaches, for example, the shake algorithm, are beyond the scope of our discussion.

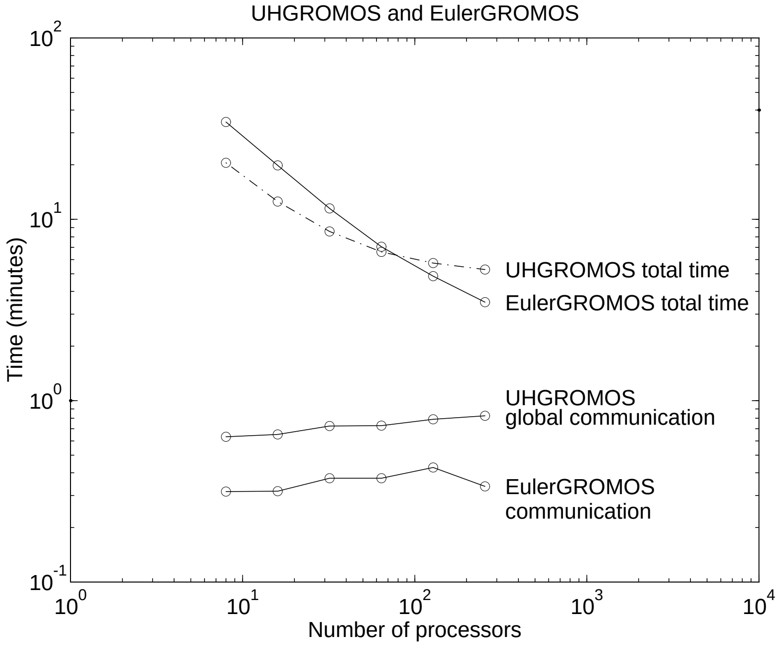

Figure 9.9: Performance of EulerGromos and UHGromos on myoglobin.

9.5 Related Work

A common feature of spatial-mapping approaches is to restrict communication to nearest neighbors; consequently, each processor subdomain has to be greater than or equal to the cutoff radius size [52, 57, 83, 99], which in turn limits scalability. Esselink, et al., report a geometric decomposition where the subdomain size restriction has been lifted [38]. However, they assume a homogeneous distribution of particles with equally sized processor subdomains. Morales and Nuevo decrease the processor subdomain size such that it is less than R_{cut}, but they continue to restrict interactions to neighboring subdomains only, thereby effectively reducing R_{cut}; they evaluate the effect on thermodynamic properties [79]. (See [73] for a detailed study on cutoff radius effects.) Plimpton also allows for subdomains smaller than R_{cut} in an implementation reported for Lennard-Jones particles that is similar to our blockwise decomposition with the Shift algorithm [84].

Chapter 10

Communication and mapping with the spatial decomposition

The mapping of an algorithm to the machine can have a significant effect on performance. In this section we investigate the effect of several mappings on shift communication performance in EulerGromos. We motivate the mappings from simplified experiments that indicate contention as a detractor from realized bandwidth. The remappings are implemented by reassigning each process logical identifier with a wrapper that is a function of the system assigned process identifier. The *new* logical identifier is used in the EulerGromos algorithm. However, the process identifier used in the IPfortran communication library is mapped back to the original system known identifier in order that messages are sent to the correct destination. Mappings to the physical (Paragon) mesh use the convention discussed below.

10.1 Introduction

For spatially–decomposed many–body problems the exchange of boundary data between logical processor domains is the principal, non-computational overhead (apart from I/O). A spatial decomposition preserves the locality of the simulation permitting application of efficient communication algorithms that benefit from the locality. In this section we will examine one such algorithm in detail, the *shift* algorithm, to understand the basis for its performance.[1]

The spatial decomposition with a cutoff radius results in overlapping groups of communicating processes. For this communication shift has been found to be super-

[1] The shift algorithm discussed in this work was developed for the case of uniformly sized processor domains.

ior to a point–to–point exchange [24]. A typical molecular dynamics simulation of the roughly 11,000 atom myoglobin system in Figure 10.1 illustrates the relative performances of shift and a point–to–point exchange. The principal parallel overheads plotted with the computation costs for a typical molecular dynamics simulation. The asymptotic performance of the communication algorithms can be critical to effective scalability. The curve labeled *overlap refresh* represents the cost of exchanging boundary atoms using the shift algorithm. The general trend is a decrease in communication cost because of a decrease in data volume with an increasing number of processors. This feature of shift is developed in detail below. The curve labeled *subdomain redistribution* redistributes atoms that have changed processor domains. This algorithm uses a point–to–point exchange. The exponential cost of the subdomain redistribution is apparent. Moreover, the data volume for the subdomain redistribution is two orders of magnitude less than that for boundary exchange.[2]

Figure 10.2 allows a direct comparison of the point-to-point algorithm and the shift algorithm. The relative performance is not surprising. We conducted this communication experiment using EulerGromos. We added the option in EulerGromos to map logical processor domains to physical domains randomly or with some coherence (the default coherent algorithm is described below as *greedy x*). Logical spatial domains are mapped to processors in a somewhat coherent manner and randomly. Note that the shift algorithm is relatively insensitive to the two mapping strategies, whereas point to point has clear benefit from physically retaining some logical locality.

Experimental methods

The experiments in this section were performed on the Intel Paragon unless stated otherwise. The Paragon has a two–dimensional mesh network with links that can support up to 175 Mbytes/sec, bi-directionally. The mesh routing chips implement oblivious, dimension ordered, wormhole routing. The mesh routing latencies are 40 nanosecond if there is not a dimension exchange, and 70 nanosecond if there is a change from the X to the Y dimension. Synchronous sends and receives were used, meaning that a send blocks until the outgoing message is copied into system space; receives block until the incoming message arrives in the user buffer. Results are presented

[2] The use of the asymptotically exponential point-to-point algorithm for subdomain refresh was suitable in domains of practical interest and has nothing intrinsic to the subdomain redistribution requiring one algorithm over the other.

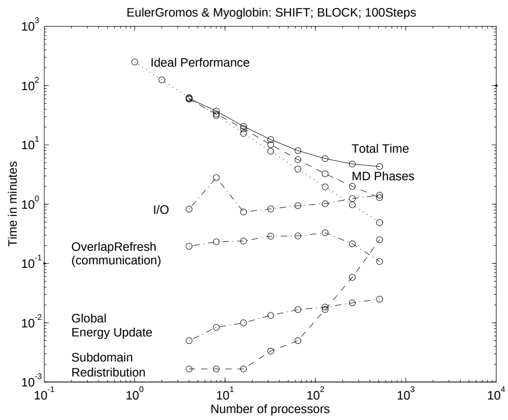

Figure 10.1: Cost decomposition for molecular dynamics using the spatial decomposition.

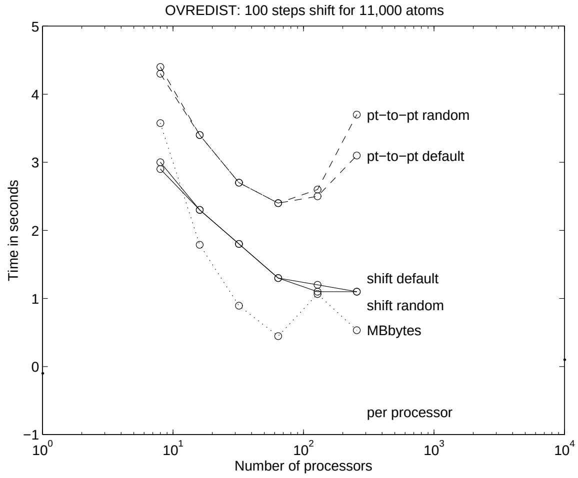

Figure 10.2: Overlap redistribution performed in EulerGromos using point-to-point and shift.

1. As raw times obtained from the double precision function *dclock* which has a 100 nanosecond resolution.

2. As effective bandwidths, that is, the number of bytes received per second at a processor.[3]

10.2 Shift description

Shift is a collective communication algorithm satisfying data exchange between processes in groups. A group consists of processes assigned to adjacent regions of space that are within some cutoff radius, R_{cut}. Groups are not disjoint thereby overlapping communication between processes in them.

The Shift algorithm shifts data in three *phases*: first along the x axis, then along the y axis, and finally along the z axis. For dimension d, phase d consists of k_d subphases, $k_d = \lceil RcutU_d / procU_d \rceil$, where $procU_d$ and $rcutU_d$ are the subdomain extent and the cutoff-radius, respectively, for dimension d in sub box units. A sub phase consists of an exchange with the two neighbor processors along the axis of the current phase. Within each phase, the first sub phase will shift all data received in all previous phases; within phase one, this will be the set of local atoms. For all sub phases after the first sub phase, a processor will communicate to each of its two

[3] Since we assume one port communication with synchronous sends and receives, the effective bandwidth is per link.

neighbors the data that were received in the immediately preceding sub phase from the other neighbor.

Figure 10.3 diagrams a two–dimension version of shift where $k_1 = k_2 = 2$. The central process communicates first (in phase 1) with its two nearest neighbors in the same row; after the second sub phase the central process has nearest neighbor data required from all processes in its row. The procedure is repeated in the column to complete the transfer.

The coupled communication between groups introduces a complexity in designing algorithms that exploit the locality. The problem is to schedule communication on links a way that avoids contention. The communication within groups are properly all-to-all communication. If the process members (and the associated network links) are involved in only one subgroup, then fundamental results can be applied directly (see Johnsson and Ho [61]). However, groups are disjoint for the spatial decomposition only if any process belongs to only one group, which is true only for the case of an all pairs interaction where R_{cut} = boxsize/2.[4] Still, fundamental results in collective communication scheduling for all–to–all broadcasts apply to the overlapping group communication as we show below.

10.3 Complexity analysis

In this section we prove that for nearest neighbor communication in three dimensions the minimum number of communication exchanges is $6k$ and minimum bandwidth requirements are $(8k_d^3 + 12k_d^2 + 6k_d)N/P$ in data.

A process has a single domain (i, j) of the application domain. The set of (i, j) are determined by the spatial decomposition. The domain (process) to processor mapping is unspecified.

Definition 1 *For a spatial decomposition and domain (i,j) the set of* k–adjacent *domains consists of the set of domains*

$$\mathcal{A}_k(i,j) = \{(r,s) \mid\ i-k \le r \le i+k \ \underline{and}\ j-k \le s \le j+k\}$$

Definition 2 *For a process to acquire the data in k–adjacent domains is to satisfy k–adjacency.*

[4]Unless otherwise stated, we assume periodic boundary conditions in this section. See [2] and [24] for details.

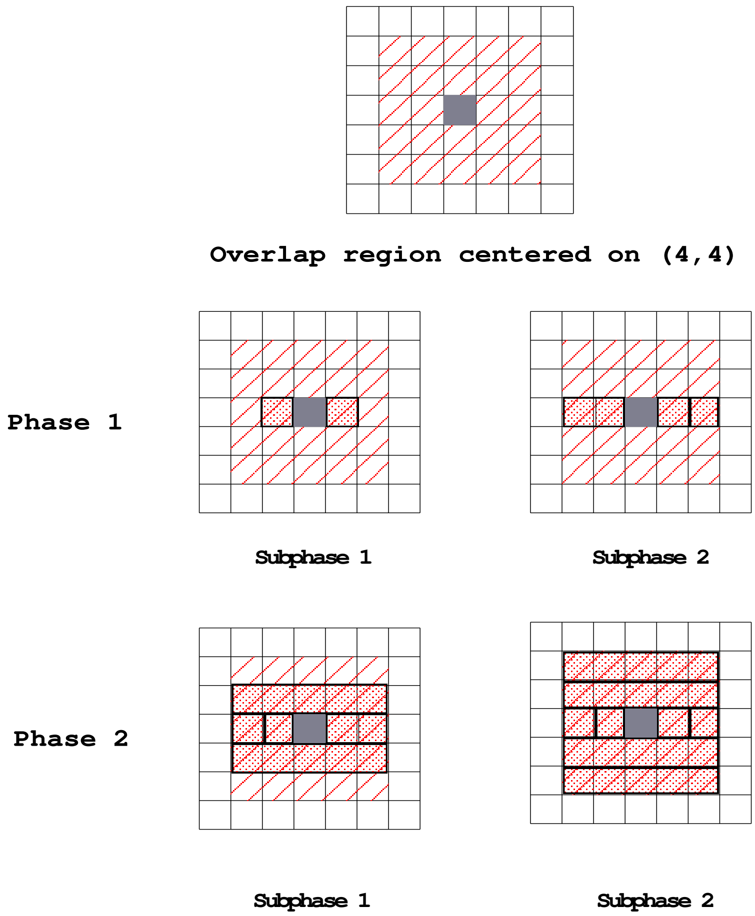

Figure 10.3: Two–dimensional shift communication viewed from the region centered on domain (4,4) with $k_1 = k_2 = 2$. The diagonal hashed regions in the topmost Figure show the domains required by (4,4) to perform its calculation. As the algorithm proceeds (see text for details), those regions buffered by (4,4) by communicating with *adjacent* neighbors are shaded.

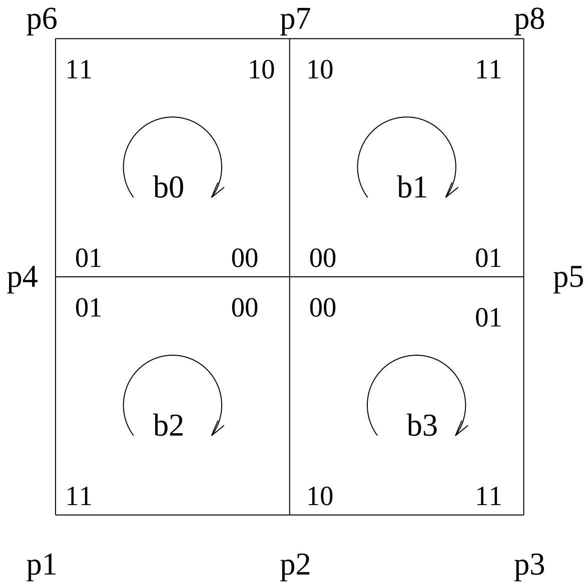

Figure 10.4: Data dependences for the two–dimensional, 1-adjacency problem. Processes (domains) are at intersections, where the lines between processes represent a logical connection defined by the application (here a two dimensional simulation of some sort). The diagram focuses on the central process (domain), p0, binary labeled *00*. Domain binary labels range from $00 \ldots 11$ to emphasize the view that the all–to–all broadcasts are separate. There are four all–to–all broadcasts involved in satisfying the 1-adjacent data requirements for p0 involving the nine nodes $p0 \ldots p8$; these broadcasts are labeled $b0 \ldots b3$. Note that this view centered on p0 is a limited view of a more extensive situation where each process is a central process surrounding by a set of AABGs.

The set of adjacent domains are required to satisfy the boundary conditions for limited range interactions determined by a cutoff radius. The one–adjacent set for the three dimensional case consists of 26 neighboring domains. For the two dimensional case and given domain (i,j), the one–adjacent domains are: (i-1,j-1), (i,j-1), (i+1,j-1), (i-1,j), (i+1,j), (i-1,j+1), (i,j+1) and (i+1,j+1). Note that $\mathcal{A}_0(r, s)$ is just the data assigned to each domain by the spatial decomposition.

Definition 3 *An* all–to–all broadcast group *(AABG) consists of a set of four (eight) processes assigned to the corners of a logical two–dimensional (three–dimensional) domain determined by the spatial decomposition. In Figure 10.4, four AABGs are {p0,p4,p6,p7}, {p0,p7,p8,p5}, {p0,p4,p1,p2}, and {p0,p5,p2,p3}. With periodic boundary conditions there are additional AABGs, for example, {p1,p4,p5,p3}.*

Definition 4 *A* multiple all–to–all broadcast *consists of the all–to–all broadcasts among all AABGs in the spatial decomposition; "all" refers to an AABG, specifically, four processes in the two–dimensional case and eight processes in the three–dimensional case. The execution of a multiple all–to–all broadcast is called a* broadcast phase.

The following complexity analysis for the shift algorithm subdivides the application domain into square groups for the two–dimensional case, and into cubic groups for the three–dimensional case.[5] For the two–dimension case, a set of four groups contains the one-adjacent domains for each processor given by definition 1. (A set of eight groups contain the one-adjacent domains for the three–dimensional case.) Figure 10.4 illustrates the four domains forming the one-adjacent set for the domain at *p0*.

We state the following two lemmas for the two–dimensional case; the extension to three–dimensions is straight forward.

Lemma 1 *At the completion of all–to–all broadcasts carried on independently in the four-domain groups (AABGs), a process will have the data*

$$\bigcup_{-1 \le m,n \le 1} \{(i+m, j+n)\},$$

which is the one–adjacent set of domains. Furthermore, the order in which the all–to–all broadcasts complete is irrelevant.

[5] We will often discuss the two–dimensional case for clarity; the extension from two–dimensions to three–dimensions is straightforward.

Proof: The data set acquired follows from geometric considerations: the process (i,j) participates in the four broadcasts which together contain the complete set of one–adjacent data. The broadcasts completion order is irrelevant since the data supplied by each process for (i,j)'s one-adjacency was present before any broadcast begins; by definition of broadcast, (i,j) will acquire all data in the four independent broadcasts it participates in.□

The following lemma states that the k–adjacency domain set for any domain (i,j) is contained in the (k-1)–adjacency domain set of the neighboring eight domains.

Lemma 2

$$\mathcal{A}_k(i,j) = \bigcup_{-1 \le m,n \le 1} \mathcal{A}_{k-1}(i+m, j+n) \;\forall\, k.$$

Proof: We proceed by induction on k. The basis case asserts

$$\mathcal{A}_1(i,j) = \bigcup_{-1 \le m,n \le 1} \mathcal{A}_0(i+m, j+n).$$

This is obvious by enumerating the one–adjacent set for (i,j):

$$\mathcal{A}_1(i,j) = \{(i-1,j-1),(i,j-1),(i+1,j-1),$$

$$(i-1,j),(i+1,j),(i-1,j+1),(i,j+1),(i+1,j+1)\}.$$

We assume by the inductive hypothesis that

$$\mathcal{A}_{k-1}(i,j) = \bigcup_{-1 \le m,n \le 1} \mathcal{A}_{k-2}(i+m, j+n). \tag{10.1}$$

By definition

$$\mathcal{A}_k(i,j) = \mathcal{A}_{k-1}(i,j) \bigcup a_k(i,j), \tag{10.2}$$

where

$$\begin{aligned} a_k(i,j) \quad &= \{(r,s) \mid r = i \pm k \text{ and } j-k \le s \le j+k, \ \underline{\text{and}} \\ & \qquad i-k \le r \le i+k \text{ and } s = j \pm k\}. \end{aligned}$$

By the inductive hypothesis we rewrite equation 10.2 as

$$\mathcal{A}_k(i,j) = \left\{ \bigcup_{-1 \le m,n \le 1} \mathcal{A}_{k-2}(i+m, j+n) \right\} \bigcup a_k(i,j). \tag{10.3}$$

We substitute the following expanded expressions into equation 10.3:

$$\bigcup_{-1\leq m,n\leq 1} \mathcal{A}_{k-2}(i+m, j+n) \;=\; \{(r,s) \mid i+m-(k-2) \leq r \leq i+m+(k-2) \;\underline{\text{and}}\; j+n-(k-2) \leq s \leq j+n+(k-2)\},$$

for $-1 \leq m, n \leq 1$, and

$$a_k(i,j) \;= \{(r,s) \mid r = i \pm (k-1) + m \text{ and } j-(k-1)+n \leq s \leq j+(k-1)+n, \;\underline{\text{and}}\; i-(k-1)+m \leq r \leq i+(k-1)+m \text{ and } s = j \pm (k-1)+m\},$$

for $-1 \leq m, n \leq 1$. Thus, the right-hand side of equation 10.3 is the set

$$\{(r,s) \;\mid\; i+m-(k-1) \leq r \leq i+m+(k-1) \;\underline{\text{and}}\; j+n-(k-1) \leq s \leq j+n+(k-1)\}$$

which is the result

$$\bigcup_{-1\leq m,n\leq 1} \mathcal{A}_{k-1}(i+m, j+n).$$

□

Theorem 1 *At the completion of* k *multiple all–to–all broadcasts (see definition 3) each process will have the set of k–adjacent domains provided that cycle $k-j$ completes prior to any group starting cycle $k-j+1$, $k+1 \geq j \geq 1$.*

Proof: By Lemma 1, each process gathers data from its one–adjacent neighbors at the completion of each broadcast phase. From Lemma 2, the data from the previous cycle of broadcasts, e.g., cycle $k-j$, will satisfy the data necessary for cycle $k-j+1$, $k+1 \geq j \geq 1$. The completion of cycle $k-j$ prior to starting cycle $k-j+1$ is a sufficient condition for the application of Lemma 2.[6] The three–dimensional case follows directly from the two–dimensional case. □

10.3.1 Lower bounds

In this section we prove that the lower bound for the number of operations to complete k–adjacency data exchange is two times that of the optimum complexity for all–to–all broadcasts involving one group of processes; we assume one–port communication. We build on results from Johnsson and Ho [61], and Saad and Schultz [90], for spanning binomial trees (SBT), some of which we state here for completion.

[6] It is conceivable that synchronization can be loosened system wide; the strict synchronization requirements are

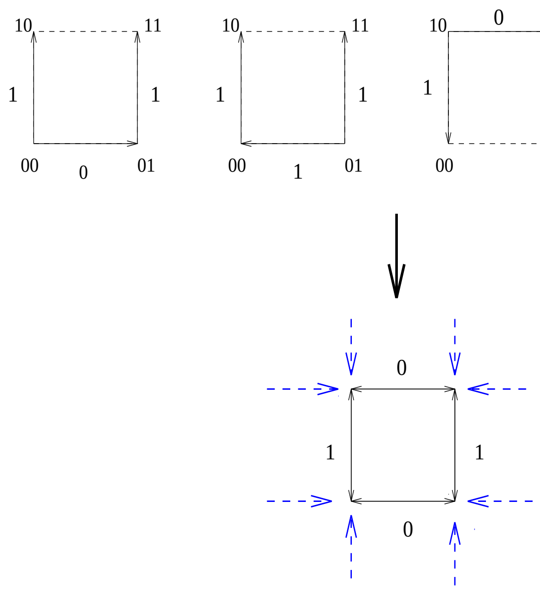

Figure 10.5: Spanning binomial trees (SBT) for a four processor all–to–all broadcast. Trees are rooted at nodes 0, 1, 2 and 3, respectively. The edge labels indicate the step number in the communication schedule.

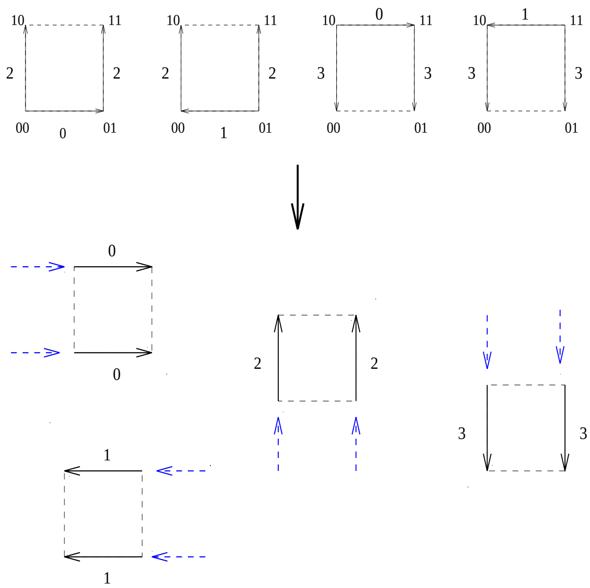

Figure 10.6: Spanning binomial trees for four–process all–to–all broadcasts (two dimensional case) applied to arrive at a one–port schedule. Nodes (domains) are at vertices; edges represent logical connections in two–dimensional space. Edge labels indicate the communication cycle.

Lemma 3 *There are a minimum of* $\log P$ *steps involved in a one–to–all broadcast assuming one–port communication.*

Lemma 4 *A node must send and receive* $\log P$ *messages for an all–to–all broadcast (under the assumption of one–port communication).*

The proof for Lemma 3 follows from observing that the minimum height of the binomial spanning tree for P nodes has a height, $d = \log P$; in order to propagate data from the root to all nodes requires d steps. Lemma 4 results from arranging the nodes in a spanning binomial tree with a single exchange per dimension [90]. Note for the bound of Lemma 4 messages of size $(P-1)M$ result.

In Figure 10.5 the four (SBT) o-graphs and the resulting a-graph are diagrammed. The edge labels indicate the communication step for the exchange between the nodes. (See [61] for detailed definitions and properties of o-graphs and a-graphs.) Notice that the a-graph has additional incoming data due to the two–dimensional mesh overlap; the three–dimensional version of the problem is illustrated in Figure 10.7 which shows a slice of a three-dimensional problem domain.

To resolve the conflict of the additional edge incoming from adjacent all–to–all broadcasts, we relabel the edges of the SBT (recall this is the same SBT at each four– and eight–process group within which an independent all–to–all broadcast takes place for during each phase of broadcasts). The relabeling scheme is diagrammed in Figure 10.6. The relabeling separates the bidirectional, simultaneous shifts along each dimension by ordering the bidirectional exchange, and assigning a separate step to it; the exchange by adjacent groups (the hashed arrows) makes up for the input lost due to making the bidirectional exchange unidirectional.

Definition 5 *The* unidirectional exchange label *is* $2d + (s \oplus d) >> d$*, for source node* s *and exchange dimension* d*.*

Lemma 5 *SBTs with the* unidirectional exchange labeling *is optimal in the number of exchanges for satisfying one–adjacency.*

Proof: The number of exchanges is optimal since

1. Each process receives a message on each broadcast step.

beyond the scope of this work.

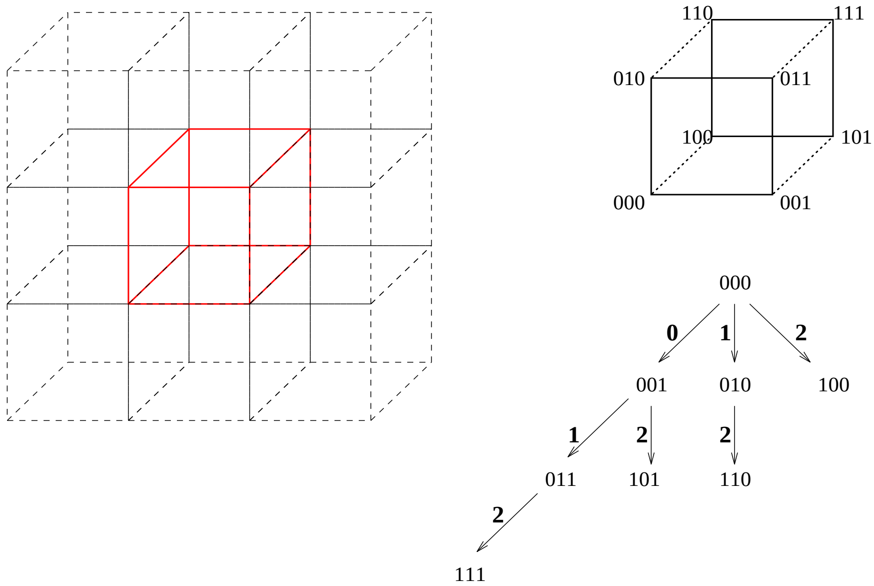

Figure 10.7: A slab from a three dimensional spatial decomposition. With periodic boundary conditions, each box has the situation presented at the central box where the SBT schedule given by the o-graph (tree at right), causes more than one input on each communication cycle. The labeling must be modified as explained in the text.

2. Each group broadcasts using an SBT with a schedule modified to adhere to the one–port criterion. □

Lemma 6 *Satisfying k–adjacency with SBTs and the unidirectional exchange labeling is optimal in the number of exchange steps.*

Proof: By Theorem 1, k SBT exchanges are sufficient to satisfy k–adjacency. Now, suppose the k exchanges are not optimal in the number of exchanges. Then either there exists an algorithm that satisfies one or more one–adjacency exchanges in a number of steps less than that using an SBT with the *unidirectional exchange labeling*, or less than k one–adjacency SBTs will suffice. The former contradicts Lemma 5. To see that k SBT exchanges are necessary, consider the domain $(i+k, j+k)$ in the k–adjacency domain of (i, j). Clearly it will take k steps for $(i+k, j+k)$ to reach (i, j). □

Theorem 2 *A minimum of $6k$ steps are required to satisfy k–adjacency for all domains in the three–dimensional spatial decomposition.*

Consider a single four–process group as described in Lemma 1 for the two dimensional case. Perform each broadcast in the four–process group using a spanning binomial tree. This results in two steps per all–to–all broadcast. However, since each node simultaneously participates in four all–to–all broadcasts (Figure 10.4), there is an additional link (logical) during each step in the broadcast. In other words, there are exactly two messages from different processes simultaneously incoming to a process with an SBT schedule. The factor of k results results from Lemma 6. The lower bound of $6k$ steps for the three dimensional case immediately follows. □

Corollary 1 *The shift algorithm is optimal for satisfying adjacency requirements for near–neighbor interactions.*

Proof: The shift algorithm completes an exchange in $6k$ steps, optimal by Theorem 2. The data transmitted to each processor is exactly the required amount

$$2k + 2k(2k+1) + 2k(2k+1)^2 = (8k_d^3 + 12k_d^2 + 6k_d)N/P,$$

where we assume without loss of generality that $k_1 = k_2 = k_3$. □

One communication step in a broadcast phase using SBTs with unidirectional exchange labeling performs an exchange along all edges in the same dimension in the spatial decomposition; one broadcast phase completes an exchange in each direction of each dimension. The shift algorithm is a variant of this exchange using SBTs with the unidirectional exchange labeling. All exchanges are performed in one dimension, k_d times, before proceeding to the next dimension. The unidirectional exchange labeling indicates the phase order for exchanges with neighbors. Each exchange phase has k_d subphases.

10.4 Mapping to a two–dimensional mesh

10.4.1 Shift contention

The experiments in this section were designed to characterize the contention effects of the shift exchange. These experiments examine shift communication cost as a function of the number of processors, P, and number of data, N, with the following conditions:

1. The message size is a function of only N, not of P.

2. The number of messages per processor is constant for all N and all P.

3. The mapping used is the *default mapping* described below.

Thus, for a given message size, N, the volume of data received and sent by a processor is constant for all P. Consequently, for any two processor configurations, P_i and P_j, $P_i \neq P_j$, and for a given fixed message size, N, only the number of messages on the network varies; the flow of data into and out of a processor is constant. We therefore attribute changes in performance for a fixed N to contention as a function of P.

It is conceivable that increases in throughput can alter the arrival rate to a processor saturating its bandwidth to the network. We consider this a contention effect since our purpose here is to explain the nearest neighbor communication in terms of number of messages on the network and message size; our goal is not to determine directly the effects of interconnection network design parameters.

Note that with the standard shift algorithm the number of messages for a fixed simulation domain varies with respect to the number of processors in accordance with interaction criteria. In molecular dynamics with a cutoff radius, the interaction neighborhood is given by the cutoff radius.

To perform these experiments we developed a spatial decomposition communication driver with a variety of communication options. With the driver we can carry out experiments that violate the particle exchanges needed by the actual application, with the advantage of isolating effects. In this experiment we enforced the constant number of messages condition by setting $k_1 = k_2 = k_3 = 1 \quad \forall\ P$.

Experimental results

We performed two control experiments involving point–to–point communication between two processes using a bidirectional exchange where both processes send then receive, and using the shift algorithm. The effective bandwidths per link from these experiments are presented in Table 10.1. As with the contention experiments, 100 exchanges were performed for each data point and the highest time was used. These experiments show that we are getting about 18 to 25 Megabytes of the peak advertised 175 Megabyte exchange rate, roughly 12%. We speculate that the actual performance is due to buffering overhead and the cost for a processor to inject (and remove) messages from the network.

The raw times using shift algorithm with the *default* mapping and a *random* mapping are plotted as a function of N and P in Figures 10.8 and 10.9. In both cases increasing the number of processors is more detrimental than increasing the message size. For example consider the effective bandwidths in Table 10.2 calculated from the

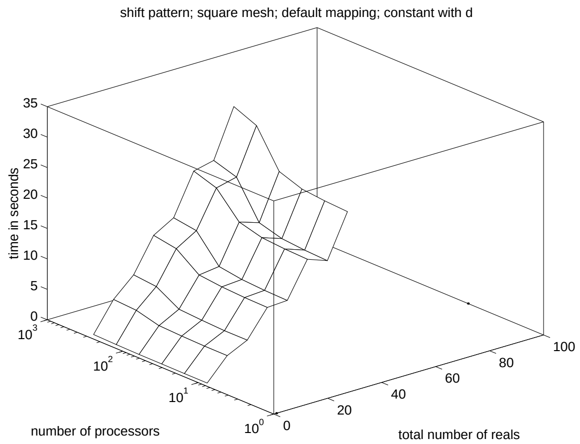

Figure 10.8: Communication time as a function of number of processors and message size. The message size is held fixed in each of the three x, y, and z subphases.

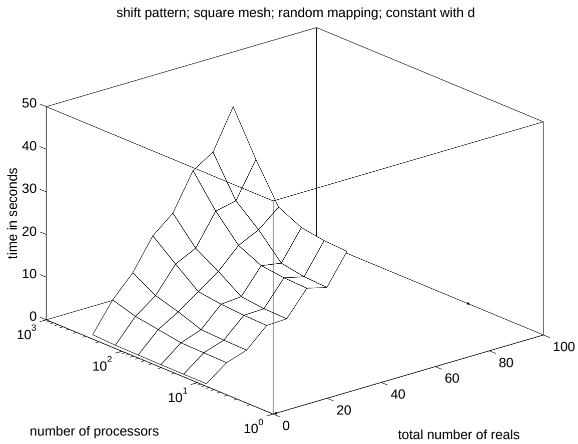

Figure 10.9: Communication time as a function of number of processors and message size with random mapping. The message size is held fixed in each of the three x, y, and z subphases.

data in Figure 10.8 as a function of N and P. We see by keeping the quantity of data on the network fixed that bandwidth$(0.107, 256) <$ bandwidth$(0.206, 128)$. This trend

$$\text{bandwidth}(\text{msgSize}/j, P * j) < \text{bandwidth}(\text{msgSize}, P)$$

is prevalent throughout the data in both Tables 10.2 and 10.3. Going across any row in these tables (fixed P, varying message size) the effective bandwidth is relatively stable in contrast to going down the column (varying P, fixed message size). In other words, for a fixed quantity of data, performance deteriorates more from increasing P than from increasing the message size. Yet, both message size and number of processors have an impact on the effective bandwidth.

We see that up to about 32 processors the shift effective bandwidth is close to the control experiment effective bandwidth. Our goal is to find a mapping that approaches the control effective bandwidth by minimizing contention.

10.5 Mappings

In this section we discuss four heuristics used to map the three–dimensional application domain to the two–dimensional Paragon mesh. The heuristics were driven by the observation that with wormhole routing the greatest opportunity for contention will occur with large messages, all other things equal. We found that minimizing the

Effective Bandwidth per Link (mbytes/second) for two processes								
	Message Size (megabytes)							
	0.008	**0.107**	**0.206**	**0.305**	**0.404**	**0.503**	**0.602**	**0.701**
pt-to-pt	17.1	17.6	22.7	18.2	23.1	18.3	23.2	18.5
shift	13.9	15.7	21.8	18.2	23.6	18.9	24.3	19.3

Table 10.1: Effective bandwidths measured on the Intel Paragon for 100 steps of communication for a constant message size. The pt-to-pt data and shift data serve as a control for later experiments. We consider these times the baseline single-port communication achievable using synchronous sends and receives on the Paragon. Furthermore, we see that the two-processor shift algorithm performs roughly the same as the simple point-to-point algorithm.

Effective Bandwidth per Link (megabytes/second)								
	Message Size (megabytes)							
nProcs	**0.008**	**0.107**	**0.206**	**0.305**	**0.404**	**0.503**	**0.602**	**0.701**
2 (control)	13.9	15.7	21.8	18.2	23.6	18.9	24.3	19.3
8	13.7	16.5	22.6	18.5	24.1	19.0	24.6	19.3
16	13.5	16.5	22.6	18.4	23.8	18.9	24.3	19.1
32	12.7	16.1	21.9	18.0	23.3	18.5	23.8	18.8
64	12.1	15.5	21.0	17.3	22.1	17.4	22.2	17.8
128	10.4	12.5	15.3	13.7	15.8	14.1	16.3	14.2
256	8.5	12.0	14.7	13.1	15.3	13.4	15.5	14.2

Table 10.2: Effective bandwidths measured on the Intel Paragon for a fixed message size per phase, each phase communicating with the six shift–defined nearest neighbors.

Effective Bandwidth per Link (megabytes/second) using random mapping of processes to mesh								
	Message Size (megabytes)							
nProcs	**0.008**	**0.107**	**0.206**	**0.305**	**0.404**	**0.503**	**0.602**	**0.701**
2 (control)	13.9	15.7	21.8	18.2	23.6	18.9	24.3	19.3
8	13.7	16.5	22.6	18.5	24.1	19.0	25.0	19.3
16	12.8	16.0	22.2	18.2	23.7	18.8	24.1	19.1
32	12.5	15.6	21.1	17.5	22.4	18.0	22.7	18.4
64	11.0	12.8	15.8	16.4	16.7	15.7	16.8	16.5
128	9.4	10.9	13.2	11.9	13.7	12.0	13.8	12.2
256	6.4	8.5	9.9	9.2	10.2	9.3	10.2	9.4

Table 10.3: Effective bandwidths measured on the Intel Paragon or a fixed message size per phase, each phase communicating with the six shift–defined nearest neighbors. Logical processes are mapped using a random mapping function.

nProcs	x	y	z
2	2	1	1
4	2	2	1
8	2	2	2
16	4	2	2
32	4	4	2
64	4	4	4
128	8	4	4
256	8	8	4
512	8	8	8

Table 10.4: Spatial decomposition domains per axis. Since each process is assigned one domain, the domain can be thought of as a process. With a fixed cutoff radius, increasing the number of domains on an axis can increase the data exchanged along that axis. (Since increasing the number of domains decreases the size.)

---

contention in the slab– and row–transfer dimensions improved overall performance.

10.5.1 The spatial decomposition

The spatial decomposition used by EulerGromos divides the rectilinear problem space (in molecular dynamics, the simulation "box") into indivisible cubic boxes. In turn, rectilinear volumes of the indivisible cubic boxes are assigned to logical process domains. In Table 10.4 we show the available process coordinates in units of subboxes given some total number of processors. Notice that the x dimension increases first. For example, going from $P = 8$ to $P = 16$, the number of processes (domains) in the x dimension doubles, while the number in the y and z dimensions stays fixed. An increase in processes on an axis increases the number of messages (i.e., the depth of dependencies, or k) along that axis.

10.5.2 Default mapping

The default mapping used by EulerGromos traverses the spatial decomposition element–wise across rows, assigning processor numbers in ascending order. The mapping is shown in Figure 10.10 where it is referred to as the *greedy* x mapping.

The contention results of this default mapping are shown in Figure 10.8 where for any fixed N, communication costs increase with an increase in the number of processors. This increase is likely due to contention for the interconnection network. With an increase in communicating processors, the probability that a circuit between any two processors will be occupied increases. Furthermore, this trend also increases with larger message sizes.

From the data in Figure 10.8 we see that for increasing the message size (recall it is fixed for each subphase in this experiment) the cost also increases. However, given a fixed number of processors P while varying N incurs a data movement cost. We therefore calculate a per–link effective bandwidth, presented in table 10.2. There we see that the effective bandwidth increases with an increase in message size up to 0.404 megabytes, and then begins to drop off. The increase in effective bandwidth is likely due to amortization of message setup costs with larger messages; the drop off after 0.404 megabytes is likely caused by contention offsetting the throughput gains.

Random mapping

The random mapping shows trends similar to the standard mapping, but with a more significant decline in bandwidth (Figure 10.9 and table 10.3). That randomization reduces the effective bandwidth can be attributed in part to the Paragon's oblivious routing coupled with the reduction in contention with mappings that preserve locality.

Greedy mappings

We used two greedy algorithms. The first assigns the sequence of logical process numbers by proceeding from front–to–back in the spatial decomposition. This makes adjacent those nodes which are involved in two–dimensional slab transfers. The other algorithm also assigns logical processor numbers to make most slab exchanges adjacent, but it also attempts to make row exchange nodes adjacent. The extent to which these algorithms achieve this will depend on the spatial decomposition (task graph) and the physical processor mesh dimensions. The algorithms are given in Figure 10.11 and Figure 10.12.

Shift performance results

The performance of *shift* using the four mapping heuristics described above are presented in Figures 10.13 and 10.14. We can at best make general comments without carefully examining the mappings with respect to the geometry of the mesh and the spatial decomposition; keep in mind that the quality of the heuristics in reducing contention depend on the mesh dimensions and spatial decomposition, factors that are not considered in the heuristic.

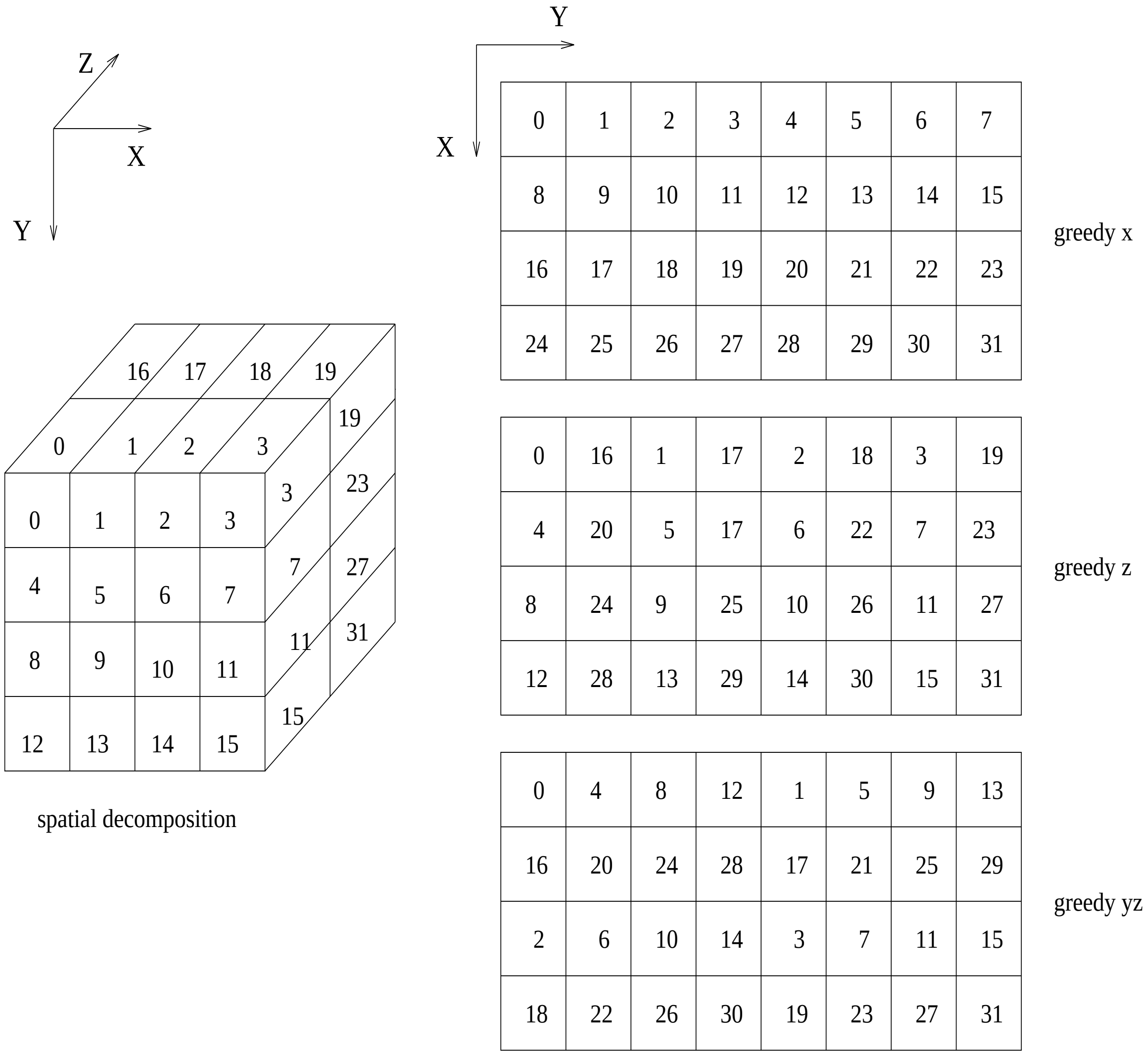

Figure 10.10: Illustration of the mapping of the $4 \times 4 \times 2$ spatial decomposition (left) onto the 4×8 processor mesh using the *greedy z* and *greedy yz* heuristics. Note the rotated orientation with respect to x and y of the mesh and spatial decomposition. The number on the mesh corresponds to the logical domain number from the spatial decomposition; the physical processor and associated identification number is associated with each square of the processor mesh where numbering is from left to right, top to bottom.

```
Procedure GreedyZ
newp = -1
do i = 0, n0d_prc(1)
   do j = 0, n0d_prc(2)
      do k = 0, n0d_prc(3)
         newp = newp + 1
         pdom(1) = i
         pdom(2) = j
         pdom(3) = k
         p = FDom2P(pdom)
         newId(p) = newp
         oldId(newp) = p
      enddo
   enddo
enddo
```

Figure 10.11: Greedy–z heuristic which assigns process identifiers for a meshX $\times$ meshY mesh to n0d_prc(1) $\times$ n0d_prc(2) $\times$ n0d_prc(3) three–dimensional spatial domain by traversing front to back to make slab transfers adjacent. (The *greedy z* mapping in Figure 10.10.) The effectiveness of meeting the heuristic goal will depend on the spatial decomposition and the processor mesh dimensions.

```
Procedure GreedyYZ
offx = 0
ix = 0
iy = 1
do i = 0, n0d_prc(1)
  do j = 0, n0d_prc(2)
    do k = 0, n0d_prc(3)
      ix = ix + 1
      nx = MIN(meshX-offx,nd_prc(3))
      if (ix.GT.nx+offx.AND.iy.LT.meshY) then
        ix = offx + 1
        iy = iy + 1
      else if (ix.GT.nx+offx.AND.iy.EQ.meshY) then
        offx = offx + nx
        nx = MIN(meshX-offx,nd_prc(3))
        ix = offx + 1
        iy = 1
      else if (iy.GT.meshY) then
        offx = offx + nx
        nx = MIN(meshX-offx,nd_prc(3))
        ix = offx + 1
        iy = 1
      endif
      newp = (ix-1)*meshY + iy-1
      pdom(1) = i
      pdom(2) = j
      pdom(3) = k
      p = FDom2P(pdom)
      newId(newp) = p
      oldId(p) = newp
    enddo
  enddo
enddo
```

Figure 10.12: Greedy–yz heuristic which assigns process identifiers for a meshX $\times$ meshY mesh to a n0d_prc(1) $\times$ n0d_prc(2) $\times$ n0d_prc(3) three–dimensional spatial domain in such a way as to make slab– and row–transfers adjacent. (The *greedy yz* mapping in Figure 10.10.) The effectiveness of meeting the heuristic goal will depend on the spatial decomposition and the processor mesh dimensions. Note that one–based counting is used for the processor mesh; zero–based counting is used for the spatial decomposition.

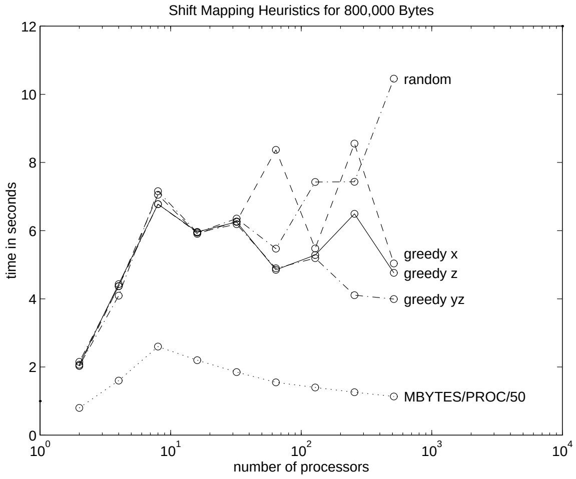

Figure 10.13: Communication time using the shift algorithm with the several mapping heuristics for data corresponding to roughly 22,000 atoms in a 50Å^3 box with a 20Å cutoff radius.

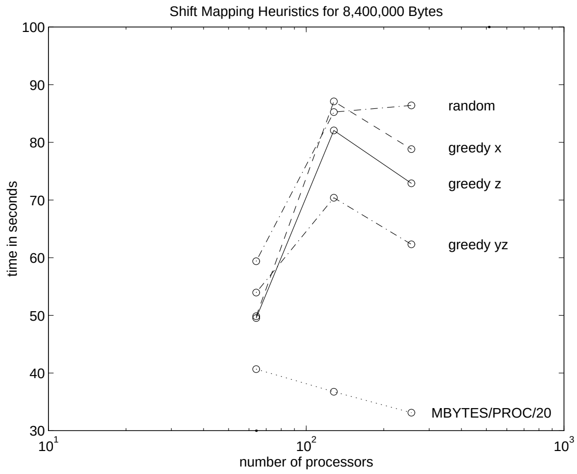

Figure 10.14: Communication time using the shift algorithm with the several mapping heuristics for data corresponding to roughly 250,000 atoms in a 130Å^3 box with a 20Å cutoff radius.

The *greedy yz* heuristic clearly outperforms the other heuristics as P increases. Recall that the contention experiments above show that effective bandwidth decreases with increasing in P. That *greedy yz* increases the effective bandwidth corroborates our interpretation that the bandwidth decrease is caused by contention. The data in Figure 10.14 shows a sharp increase in communication costs going from 64 to 128 processors; in going from 64 to 128 processors there is an increase in processors along the x dimension, that dimension least favored by *greedy yz*. Going from 8 to 16 processors shows a similar trend. In general, it appears that there is a correlation between transmission cost and an increase along a dimension ill–suited for a given heuristic. However, this rule of thumb is sometimes violated, as for example in going from 64 to 128 processors using *greedy x*.

10.6 Discussion

We have shown that the shift algorithm experiences a performance decline due to two related factors related to increasing the number processors for a fixed problem:

1. increase in message setup costs (latency) due to an increase in the number of messages;
2. increase in contention.

The nearest neighbor communication problem was formulated as a collection of all–to–all broadcasts, with the "all" restricted to three dimensional cubes of eight processors. From complexities involving spanning binomial trees, we arrived at the optimal complexity of $6k$ messages to satisfy the collective and overlapping all–to–all broadcasts. We gave a proof showing that the all–to–all broadcasts satisfied the k–adjacency problem. We then showed that the shift algorithm is optimal in the number of data exchanges and in bandwidth requirements.

The performance of the shift algorithm was analyzed from the standpoint of contention. We found that contention was a serious detractor in performance, especially with large numbers of processors. Random mappings of the shift algorithm processes to a two dimensional mesh performed worse than a mapping which preserved locality along rows, first most. We investigated four heuristics for the specialized case of mapping the three-dimensional shift algorithm to the two–dimensional Paragon mesh:

1. greedy–x mapping,
2. greedy–z mapping,
3. greedy–yz mapping,
4. and random mapping.

The greedy z-mapping attempts to preserve locality for the z–dimension exchange, that exchange dimension involving the large data transfer of two–dimensional slabs. The greedy–yz exchange favors locality for the two highest transfer dimensions, y and z, for rows and slabs, respectively. The heuristics do not take into account the spatial decomposition dimensions and the processor mesh dimensions, so as a consequence the effectiveness varies. However, the mappings that preserve locality in those dimensions involving the largest message sizes tend to outperform other mappings.

Message sizes in the shift algorithm increase with the phase. That the greedy–z mapping is generally superior to the greedy-x mapping reflects this. Yet, the k in the x–dimension in our spatial decomposition increases faster than in the other two dimensions. A potential optimization is to invert the decomposition such that the z dimension increases faster, subsequently mapping with the greedy–yz mapping. In this way, both message size and quantity would be better favored by the mapping.

Effective Bandwidth per Link (megabytes/second) using mapping heuristics with the shift algorithm simulating roughly 22,000 atoms				
nProcs	**random**	**greedy x**	**greedy z**	**greedy yz**
2	18.6	19.7	19.3	19.7
4	18.3	18.3	18.1	19.6
8	19.2	18.4	19.2	18.2
16	18.5	18.6	18.5	18.5
32	14.6	14.8	14.8	14.5
64	14.2	9.3	16.0	15.8
128	9.4	12.8	13.2	13.5
256	8.5	7.4	9.7	15.3
512	5.4	11.3	11.9	14.2

Table 10.5: Effective bandwidths measured on the Intel Paragon with a total of 105,000 real numbers which corresponds to the data exchanged for about 22,000 atoms during overlap refresh; a box size of 50Å^3 with a cutoff radius of 20Å was used. These bandwidths were derived from the raw data presented in Figure 10.13.

Effective Bandwidth per Link (megabytes/second) using mapping heuristics with the shift algorithm				
nProcs	**random**	**greedy x**	**greedy z**	**greedy yz**
64	13.7	16.3	16.4	15.1
128	8.6	8.4	9.0	10.4
256	7.7	8.4	9.1	10.6

Table 10.6: Effective bandwidths measured on the Intel Paragon with a total of 1,050,000 real numbers which corresponds to the data exchanged for about 250,000 atoms during overlap refresh; a box size of 140 Å^3 with a cutoff radius of 20 Å was used. These bandwidths were derived from the raw data presented in Figure 10.14.

Chapter 11

Trajectory outputting

Data redistribution can occur implicitly within the context of I/O.[1] It sometimes happens that the data layout in disk is different from the data layout for the calculation. In this section we present output costs in the context of a molecular dynamics simulation. The overhead of the output costs are analyzed with respect to the overall algorithm, and importantly, with respect to a representative data–analysis rate. We show that the calculation can be I/O bound for some numbers of processors. Moreover, to remap the data from the spatial distribution to a standard order for the trajectory file requires a redistribution of data.

11.1 Introduction

Already an invaluable tool in pharmacological applications, the role of molecular modeling makes steady advances with the computational advantages provided by high performance computing. An increase in cpu cycles fosters simulations based on more detailed models which can more accurately simulate biochemical reactions. An improvement in predictive capacity results. By testing new molecules on the computer quickly, "computer alchemy" reduces trial and error in the laboratory where it could take a team of Ph.D.s months and millions of dollars to design a new drug.

The speed provided by the digital computer is absolutely essential to modeling biochemical systems, so one might naturally ask what data production rates are needed and can they be achieved?

The resources required for a molecular dynamics simulation vary in accordance not only with the system size, but with the goals of the simulations and the analysis per-

[1] Material in this section comes from [28].

formed on it. A usual concern is to sample sufficiently a vast phase space (coordinate and momentum space) in order to draw reliable conclusions about biochemical properties of interest. The components key to any parallel simulation's rate of progress are the computation and communication involved, the I/O involved, the length of the simulation, and the hardware. With a good algorithm where the computation scales linearly, stepping up the number of processors will eventually reduce the cost of the computation to a negligible level.[2] What typically remains are communication and I/O costs.

To complete a representative calculation for the acetylcholinesterase (AChE) system described in this paper can require a month or more of a 512-node Paragon, far short of a desirable turn–around time. Beyond this state-of-the-art machine, one of the largest typically in use, a straight–forward extrapolatation of our experimental performance data reveals that the output rates eventually dominate the total cost of the simulation. In this paper we address the question, what resources are required to meet modern computational biochemistry needs, and why, by comparing the analysis consumption rates against the potential bottlenecks to data production, namely, I/O and communication.

The remainder of the paper is structured as follows. section 11.2 provides background on the medicinal relevance of AChE, with motivation and goals for our simulations. In section 11.3 we describe the simulation whose computational characteristics and performance are used throughout the paper. Also in section 11.3 we overview the analysis phase of the study to develop a feel for a scientist's ability to process data. The output models and communication algorithm are outlined in section 11.4. In section 11.6 we describe and extrapolate performance results to address the question regarding the data production rates that can be attained with the Intel Paragon and EulerGromos.

11.2 The AChE and tacrine study

The enzyme acetylcholinesterase is responsible for degrading the neurotransmitter acetylcholine in species from man on down to insects. AChE is a target for many commonly used drugs and toxins. For example, clinical studies suggest that acetylcholinesterase inhibitors such as tacrine (tetrahydro-9-aminoacridine) may be useful

[2]For a review of parallel molecular dynamics algorithms see [84]. The machine can of course be made to perform more efficiently by manipulating machine parameters such as cache size, instruction set, and so forth [88].

in enhancing memory in patients with Alzheimer's disease [68]. However, tacrine induces a high incidence of liver toxicity [76], thereby motivating a search for alternative inhibitors. The crystal structure of AChE obtained from an electric ray (Torpedo californica), and solved by Joel Sussman's group at the Weizmann Institute in Israel [101], provides the starting point for our simulations.

The rational design of inhibitors for AChE requires a detailed understanding of the active site where binding occurs, and on the mechanism of product entry and substrate release. Toward this end we are performing two simulations of the AChE dimer, one with the tacrine inhibitor, the other unliganded. With the dimer simulation, a closer representation of nature, we can study intermonomer effects on the motion of key amino acids comprising the active site. Recent simulations of the unliganded monomer by Gilson, et al. [45] suggest a "back door" which may facilitate movement of reactants to, and products from, the active site. Our unliganded dimer simulation is a first step to enhance the model used in [45] so to develop an improved picture of the active site.

We are simulating the tacrine–bound AChE dimer to gather data for inhibitor perturbation analyses designed to suggest strategies for developing stronger inhibitors. Both the unliganded and liganded dimer simulations will allow us to compare the average structure and fluctuations, especially around the active site, while investigating intermonomer motion effects there. In addition, the dimer simulation can try to relax large-scale effects of crystal packing forces to generate improved models for Poisson Boltzmann and Brownian dynamics simulations.

We will frame the data–production–rate analyses in this paper with two related studies of fluctuations we are performing. One fluctuation, the relative motion of the monomers' centers of mass, involves coordinated group motions of globular regions that occur with characteristic times from 10^{-10} to 10^{-7} seconds, or 100 to 10,000 picoseconds (ps). To observe such events usually requires a simulation of at least that length. We are also investigating the magnitude of fluctuations of individual amino acids in the active–site region. These higher frequency motions, with characteristic times ranging from 10 to 100 ps, can reveal details about the functionality of the AChE active site. Several periods of such fluctuations must be observed. Thus we take 1000 ps (simulation time) as a standard objective for a typical computational study.

11.3 The simulation

Basics

We used the program EulerGromos [24], an aggressive spatial decomposition of the molecular dynamics program GROMOS [47], to simulate the solvated AChE dimer, both with and without the tacrine inhibitor. These AChE systems, with about 131,000 atoms, are simulated using a 10Å cutoff radius in a box roughly $90 \times 98 \times 160\text{Å}^3$. The simulations were performed on the Intel Paragon with one i860-XP compute processor and one communication processor per node.

The solvated acetylcholinesterase dimer requires special consideration at almost every stage of the simulation process. To begin with, the file formats used by molecular modeling programs cannot accommodate the system size. Next, even with the aggressive data distribution by EulerGromos, the AChE system requires 128 Paragon nodes with 32 Megabytes of memory per node to satisfy the system's memory requirements without using virtual memory. Finally, large quantities of output data, particularly the ongoing trajectory output, introduces scaling challenges.

What data–production rate is useful?

The analysis of the trajectory, requiring considerable human intervention and guidance, determines what data production rate will be useful. An attempt to quantify that creative process would not be unlike an attempt to quantify the production of a painting. In addition, the types of studies and their approaches vary widely. We can, however, loosely bracket a desirable turnaround time for the simulation with an *estimate* of the time to perform the analysis for a typical study.

The complete modeling process involves producing data, moving the data, processing the data with analysis tools, and finally, synthesis and organization of the results. In some cases, instantaneous turn around time of the modeling could be useful. For example, if it were possible to determine the free energy for a compound at the push of a button (at this time such automation is not feasible) the chemist could follow the free energy of binding across a set of compounds to arrive at the best one, a potentially very useful process in drug design. In practice, though, simulations are carefully monitored with considerable creative engineering going into the analysis.

The economics of this research, to analyze and synthesize results from a trajectory such as the fluctuations described in section 11.2, in the context of other projects and tasks common to scientific research, suggest roughly a 1-week time frame for some logical unit of study. On average, we can expect to observe the high frequency displacement of the active–site amino acids and the low–frequency motion involving inter-monomer motion on time scales on the order of 100's and 1000's of picoseconds, respectively. To perform a 100 ps simulation of AChE requires about 10 days of a 128 node Paragon; a 1000 picosecond simulation requires about 2.5 months (section 11.6). Even setting aside the more demanding, futuristic "instantaneous" free–energy calculation described in this section, we see that a factor of five or more improvement over our current turn–around time would be immediately gainful.

A 1000 ps AChE run produces about 150 GBytes of raw data for analysis (section 11.4). This would require a week or so of even cursory analysis, but it would therefore be useful to get such results computed in a day (or less) of cpu time. To transfer this data from a supercomputing center to a home disk would take 24 hours with an ftp rate of about 1.7 MBytes/sec, an unlikely ftp–session bandwidth and connect time with today's internet. It is interesting that sending the taped data via Federal Express could be a preferred method to move the data, provided adequate tape facilities were available. The data produced from a 1-day 1000-ps run stretches to the limit data transfer technology.

11.4 Bottlenecks to data production

The MD computation

The memory and computational requirements of an EulerGROMOS simulation depend in the first order on the number of atoms in the simulation and the cutoff radius size, with the simulation "box" size, which usually grows linearly with the number of atoms, playing a secondary role. With the load–balanced spatial decomposition implemented by EulerGromos, the MD computation scales close to linearly with as few as 20 atoms per processor [24].

———————————————————————————

Trajectory outputting

Trajectory outputting, the principal output of the molecular dynamics simulation, can be the single most costly component, limiting speedup and determining asymptotic

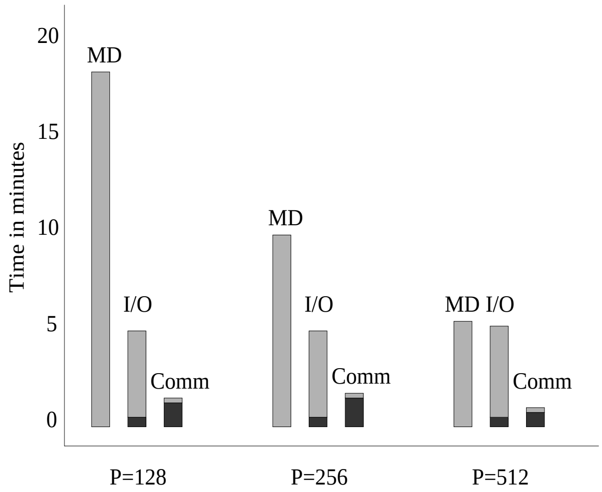

Figure 11.1: Solvated AChE dimer simulation times for 100 steps on the Paragon. The principle components of the total time are: the molecular dynamics computation (MD); one-time initial and final I/O (light shaded region of the I/O bar; trajectory outputting water and protein every 10 steps (the dark shaded region of the I/O bar); shift communication costs (dark region of Comm rectangle); and coordinate gathering to processor zero (light region of Comm rectangle).

performance. It follows that trajectory outputting must be addressed in order to improve the simulation turn–around time.

Performance enhancements to outputting can be made in several categories

- science–based reductions in output quantity,
- efficient output models and algorithms,
- and compression techniques.

In this subsection we discuss the output models used by EulerGromos, an intrinsic sorting problem and application–based output reduction. Compression techniques are also of potential value, however, standard compress utilities are ineffective for unformatted data.

11.5 Output models and sorting

A GROMOS trajectory file consists of a sequence of configurations, with each configuration composed of a sequence of atom positions (or velocities) associated with atoms positionally. During the course of an EulerGromos simulation, a processor can own a varying set of atoms since atoms can move across processor boundaries.

The spatial decomposition disregards the *global atom number* order understood by the analysis programs for these atoms. Consequently, the trajectory must be sorted into global atom number order at some point.

Sorting the spatial decomposition into global atom number order complicates output development. Where and how the sorting is performed leads to different approaches for trajectory outputting with different tradeoffs. Possible methods include

- sort during the simulation before outputting,
- sort during outputting by writing to correct (sorted) positions in the file,
- sort after the simulation in a post processing step.

EulerGromos provides two methods for outputting the trajectory: *in–place sorting* and *post sorting*. With in–place sorting, a sort and gather to the logical output processor precedes that processor outputting the configuration to a single file. In contrast, with post sorting, processors simultaneously output their local atoms (i.e., atoms resident at that processor) along with the global atom numbers using one file per processor. A post–processing utility sorts and merges those files into the usual GROMOS trajectory file.

The trajectory outputting methods used by EulerGromos assume sequential semantics with file access, thereby achieving machine independence. Simultaneous output of sorted data to a direct access file could improve performance, but the enhancement has been unnecessary for the processor ranges typically available to us for production runs. (Section 11.6 presents scalability issues relative to output costs.)

Science–based output reduction

We can reduce output rates without changing the output model by decreasing either the frequency or the number of atoms output, or both. The approximately 120,000 solvent atoms far outnumber the 10,000 or so AChE solute atoms. Many of these solvent atoms, while critical for establishing a sound environment for protein dynamics, are without a role in many analyses and can be ignored after the simulation. EulerGromos takes advantage of this fact by providing an atom selection facility which allows the user to specify a subset of the system for trajectory output. EulerGromos permits solute and solvent selections based on combinations of (1) atoms lists, (2) geometric volumes, and (3) pattern matching [21]. These selection criteria have proven beneficial for reducing trajectory output to files and to real–time graphics displays attached to EulerGromos. The potential reduction is significant: one configuration of AChE, consisting of double–precision Cartesian coordinates with 24 bytes per atom, amounts to about 3 Mbytes for the entire system and 0.25 Mbytes for the protein only.

Communication

Communication overhead can be another significant detractor from parallel speedup. The communication phases performed by EulerGromos in order of decreasing cost are:

1. update *overlap regions*[3] between processors at each MD step and during *shake*,[4]
2. change atom ownership at periodic intervals,
3. perform temperature update, a global scalar exchange, and
4. miscellaneous global scalar exchanges.

The dominating communication phase is the overlap region update [24]. EulerGromos uses the shift algorithm for updating the shadow region (chapter 9). In principle, the shift communication costs depend on the number of atoms per processor and the cutoff radius. The shift algorithm cost decreases with increasing numbers of processors, P, since the amount of data per message decreases at a rate fast enough to counteract the increase in the number of messages. EulerGromos uses an arbitrary mapping of the logical application space to the physical topology, which could increase contention and dilation effects with an increase in P. Yet, even though EulerGromos ignores the Paragon topology, the communication cost still tends to decrease with increasing P.

In spite of the aggressive data distribution applied by EulerGromos, the program performs some global communication. These are scalar values for maintaining constant temperature, workload metrics for performing load balancing, and one-time broadcasts for initializing data structures. Note that the overhead to sort the trajectory discussed in Section 11.5 can further contribute to communication costs.

[3] Each processor subdomain, s, is associated with an *overlap region*, which is the space not in s, but within a cutoff radius distance of s.

[4] We use shake to constrain bond lengths, thereby eliminating the highest frequency motion in the system, which is usually uninteresting. This allows us to increase the step size by about a factor of 5. Consequently, the use of shake communication overhead is an important component of the analysis here.

Trajectory output time in seconds for P=128				
	unformatted to UFS		unformatted to PFS	
data output	*post sort*	*sort on fly*	*post sort*	*sort on fly*
solute	no data	6	no data	7.3
solute+solvent	79	28	no data	42

Table 11.1: Times to output AChE trajectory using EulerGromos for a 100 time step simulation, with an output frequency of 10-steps. For *post sort*ing each processor outputs its atoms to a private file, one file per processor. For *sort on fly*, data are sorted to processor zero which then outputs to a single file. The UFS is the CalTech Paragon filesystem; the PFS is their parallel RAID filesystem with 16 RAID disks and a 64 KByte stripe size. The *no data* entries for the *post sorting* approach resulted from hangs caused by multiple processors writing to the file system from the same application.

11.6 Performance results

Overall performance

The histogram in Figure 11.1 decomposes the total time for 100 step runs for 128, 256 and 512 processor runs into the dominating program sections. The molecular–dynamics core calculation (MD) scales almost linearly from 128 through 512 processors at about 90% efficiency.

Communication

To sort and gather the trajectory data to processor zero, we used a simple $O(N + P)$ algorithm.[5] At the processor ranges we experimented with, the resulting sort costs were small relative to the computation (Figure 11.1, light–shaded portion of *Comm* rectangle). We recall that this overhead is counterpart to outputting a sorted trajectory during the calculation. The small cost for the global communication of scalars at about 1 second per 100 steps is not plotted in Figure 11.1.

The shift algorithm's updates of overlap regions after coordinate updating and during *shake* comprise the bulk of communication costs (Figure 11.1 dark area of *Comm* rectangle). As P increases, these costs tend to decrease overall, but with periodic increases resulting from ceiling functions used to calculate the number of interacting processors, and consequently, the amount of data transmitted.

To extrapolate the shift algorithm's performance to larger machine sizes, we employ a theoretical communication model, $\alpha\mathrm{m} + \beta\mathrm{n}$, where: the latency $\alpha = 40$ μsec;

[5] Where N represents the number of atoms and P the number of processors.

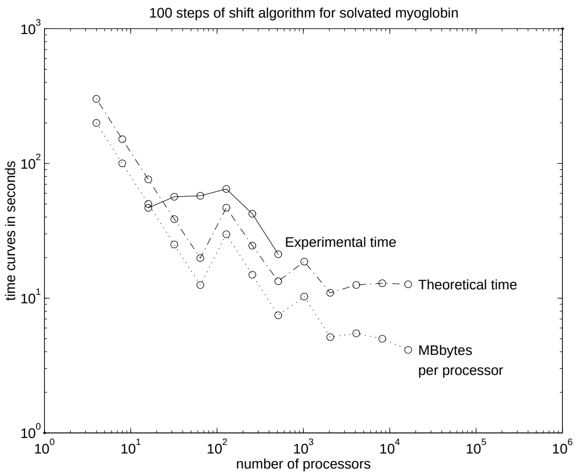

Figure 11.2: Extrapolated performance (beyond P=512) of the shift algorithm for 11,000 atoms.

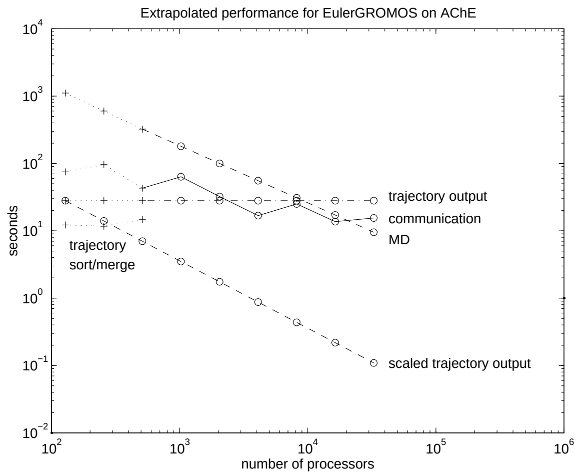

Figure 11.3: Observed performance for EULER-GROMOS simulating 100 steps of AChE for a total of 0.2 ps on the Paragon.

the transmission cost per byte $\beta = 0.080$ μsec; the total number of messages $m = 6\lceil r_f P^{\frac{1}{3}} \rceil$, where r_f is just $\frac{Rcut}{totA}$, the fraction of the application domain edge ($totA$) covered by the cutoff radius ($Rcut$); and the total number of bytes $n = (8k^3 + 12k^2 + 6k)N/P$, where k is the number of interacting processor subdomains in each dimension.[6] This model, which ignores contention, internode distances, and buffering overheads, is intended to give a picture of communication cost trends.

Figure 11.2 plots the theoretical time for performing the shift algorithm, invoked 100 times, using the model described above for the 10,914 atom myoglobin system described in [24]. The latency and bandwidth parameters used in the model are 300μsec and 0.50×10^{-6}sec/byte; experimental data are from the Touchstone Delta. The experimental curve deviation from the theoretical curve results in part from buffering costs in the application. Nonetheless, the theoretical curve predicts the decrease in communication costs from 128 to 512 processors, increasing our confidence to apply it to the extrapolated performance analysis for AChE developed in section 11.6.

I/O

The I/O costs consist of costs for data initialization (one–time costs), trajectory outputting and the output of a restart file at the end of the simulation.

[6] k depends on N and P in a complex way, where, for example, increasing N and fixing P with homogeneous density decreases k; increasing P and fixing N causes k to increase. See chapter 9 for more details.

We experimented with several methods for outputting the trajectory by: (1) changing the I/O subsystem, (2) varying the amount of data by outputting either solute only, or solute+solvent, (3) sorting during and after the calculation. Trajectory outputting costs for those experiments using 128 processors are presented in Table 11.1. Our results show that sorting during the calculation (table columns labeled *sort on the fly*) is the more efficient method for handling the trajectory. Also note the significant savings gained by not outputting solvent.

Figure 11.1 reports unformatted trajectory outputting costs to a filesystem located on a Paragon disk: writing the entire AChE system unformatted to the UFS takes about 28 seconds per 10 writes for a total bandwidth of about 1.1 Mbytes/sec. The costs for trajectory outputting are roughly the same going from 128 to 512 processors, as we expect, since the same number of data are output and a single processor does the outputting.

We will further consider output costs relative to performance in the next section to develop a rough estimate about how this overhead affects the progress of the EulerGromos AChE simulation.

What data production rates can be achieved?

We estimate achievable simulation rates by extrapolating experimental data to larger processor configurations. The methods and rationale for the extrapolations follow:

- Molecular dynamics costs, labeled *MD*, are successively scaled according to the approximately 90% efficiency observed experimentally.
- Shift-based communication, labeled *communication*, is extrapolated using the model described in section 11.6.
- Two versions of trajectory outputting costs are presented:
  - the performance data for unformatted, sorted output of all atoms to the UFS at 10-step intervals (Figure 11.1),
  - the scalable model based on current algorithms.
- We assume sufficiently long runs such that the one–time I/O costs, at about 5 minutes per run, are ignored.
- Other I/O costs and miscellaneous overheads average about 5 seconds per 100 steps, and we therefore ignore them.

1000 ps AChE Simulation Times (days)									
# Processors	128	256	512	1024	2048	4096	8192	16384	32768
time with actual output	**74**	**48**	**32**	16	10	6	5	4	4
time with scalable output	74	47	30	15	8	5	4	2	2
number of disks	8	16	32	64	128	256	512	1024	2048

Table 11.2: Row *time with actual output* gives the total time in days using the *sort on fly* approach (see text), with actual times boldfaced; all other times are extrapolated as described in text. Row *time with scalable output* subtracts the reduction in output costs determined by doubling the number of disks from row *time with actual output*. The better than expected ideal speedup going from 512 to 1024 processors reflects simplifying assumptions in the model (see section 5.2).

- The costs for sorting coordinates and velocities to processor zero are extrapolated from P=512 as a conservative estimate.

In Figure 11.3, the experimental curves for 128 through 512 processors are appended with performance data estimated for up to 32,768 processors. The actual performance is represented by the dotted line (from 128 to 512 processors). Performance is extrapolated up through 32,768 processors (dot-dash line). Our model predicts diminishing performance gains somewhere between 8,192 (about 16 atoms/processor) and 16,384 processors (about 8 atoms/processor), where communication and trajectory output costs approach computation costs.[7]

A total time consists of a term each for the molecular dynamics costs, shift–algorithm–based communication, sorting of coordinates for trajectory outputting, and trajectory outputting to disk; total times based on these components, and using the idealized and extrapolated experimental trajectory output performance, are summarized in Table 11.2. The components scale linearly as a function of the number of time steps, so, for example, the cost for 1 ps is derived by multiplying the costs shown in Figure 11.3 by five.

At 512 processors, the total experimental time to simulate AChE for 1000 ps is about 1 month. At about 4096 processors the time reaches our 1-week goal, however, with MD costs rapidly dropping off with increasing P, the degrading effect of trajectory outputting diminishes gains, where at 8192 processors it equals MD costs. Still, the extrapolations show that applying 16,384 processors reduces the total time for performing the 1000 ps simulation to roughly 4 days, even though trajectory outputting

[7] To illustrate using the experimentally determined speedup of an N=337 atom system: there is speedup from about 40 atoms/processor to 20 atoms/processor; speedup flattens from 20 to 10 atoms/processor with diminishing returns beyond that.

and communication limit performance gains there.

Furthermore, because the algorithm used for sorting the trajectory is simple and not specialized, it disproportionately dampens trends in the extrapolation. This algorithm could be improved for larger P if needed.

We are achieving a roughly 1.1 Mbyte/sec bandwidth during our simulations using an approach that routes trajectory data to processor zero for sorted outputting. To develop a theoretical upper bound on performance, which we have seen to be restricted by output costs, we scale our current trajectory output costs linearly using a scalable output model. To achieve this we assume full disk utilization at current experimental output rates, with a 16-to-1 compute-node-to-I/O-node ratio. The resulting modeled performance shown in Figure 11.3, *scaled trajectory output*, effectively reduces trajectory overhead far below the total performance, now driven by communication costs. Intuitively, it appears that the shadow region update is the intrinsic limit to the spatial decomposition's asymptotic performance, provided output costs scale suitably. Table 11.2 gives the cost to perform the 1000 ps simulation with the scalable output in units of days. The improved output costs reduce the total times, however, communication and the conservative estimate for trajectory sorting counter gains.

11.7 Discussion

A large scale biomolecular calculation such as the AChE simulation strains massively parallel computers to deliver robust turn around times. We described a 1000 picosecond simulation, where given the economics of this research, the data can be comfortably assimilated by a scientist in about a 1 week period. For our parallel molecular dynamics simulation to meet this modest schedule requires that the EulerGromos core computation cost be reduced by applying between 2048 and 4096 processors. To arrive at that performance estimate we extrapolated actual performance data using simple models.

Due to the close to linear speedup of the computation, and localized communication, the performance picture for increasing P depends largely on the trajectory output method used. Outputting is complicated by a sort of atoms from the spatial decomposition to a standard ordering. In our experiments using up to 512 processors, the best performance was obtained from sorting and gathering the coordinates to processor zero, which then outputs to a single file. The gathering phase can be avoided altogether by post sorting the trajectory on a workstation with cheaper machine cycles. But, apart from a bug with the filesystem encountered by this method, the time and effort involved to post sort a large number of files make sorting during the computation the more attractive approach.

We can readily reduce output costs by an order of magnitude or more, independent of the output model, by decreasing the output frequency and reducing the amount of data output. The AChE study described here can tolerate an output frequency of once every 10 steps. Yet, such parameters will depend on the study, and are not always possible. We also experimented with the Paragon parallel I/O subsystem, PFS, and found that the non-parallel filesystem, UFS, better for our I/O models, which are coded without machine dependencies.

Without a machine–tuned I/O model, we estimate meeting the 1-week turn-around-time goal for the 1000 ps simulation with continuous use of about 4096 processors. Beyond that simulation, a solid example scenario, far more rigorous scenarios exist, including those with more frequent output, those that could benefit from a faster turn around time, or those with longer simulation lengths. Moreover, increases in computation rates, either with faster processors or more of them, when applied to scalable programs on scalable architectures increase the stakes for good I/O support at all levels, from linguistics to hardware.

Chapter 12

HPF Language Issues

To complete our study we applied the HPF prototype, Fortran D, to GROMOS to implement the replicated algorithm. In principle the basic constructs provided by HPF are capable of implementing the replicated algorithm, although, irregular decompositions are useful for load balancing the pairlist and non-bonded force calculations. However, the unstructured nature of GROMOS was a quagmire of obfuscation for the Fortran D compiler, making the implementation of an otherwise simple algorithm, difficult.

Clark, et al., have parallelized a FORTRAN 77 version of GROMOS, a popular dusty-deck program for molecular dynamics, using FORTRAN D, a data-parallel dialect of FORTRAN [29]. During this transformation we encountered a number of difficulties that probably are neither limited to that particular application nor do they seem likely to be addressed by improved compiler technology in the near future. Our experience with GROMOS suggests a number of points to keep in mind when developing software that may at some time in its life cycle be parallelized with a data-parallel compiler. This chapter presents some guidelines for engineering data-parallel applications that are compatible with FORTRAN D or HIGH PERFORMANCE FORTRAN compilers. Moreover, these incompatibilities obstructing the direct application of HPF and Fortran D motivate continued development of low–level programming constructs.

It is *not* our intention in this chapter to compare the merits of HPF and IPfortran. The ease or difficulty of an implementation in fact seems to reflect the program structure relative to the language model. Instead we highlight the need for constructs from both programming paradigms using an example from an important class of programs.

12.1 Structured programming guidelines

One concern often not foremost in a scientific programmer's mind at the outset of software development is parallelization.[1] Yet, even for scientific applications developed for sequential execution, it is not unlikely that someone at sometime will parallelize the software. As it turns out, some programming styles are easier to parallelize than others. Moreover, for programs to yield to the data-parallel approach of compilers for Fortran D [42] or High Performance Fortran (HPF) [65], certain structural properties must be present in the software. Elements of a program style congruous with an HPF compiler include, for example, consistent distribution and use of data arrays and structured flow of control. It appears that writing such programs from the outset largely embraces good software engineering techniques (such as the structured-programming approach advocated by Dijkstra [34]). In this chapter, we discuss some of these requirements and provide guidelines for engineering data-parallel applications to be compatible with compilers for data-parallel languages. Alternatively, the observations presented here could also serve as guidelines for making an existing program suitable for data-parallel compilation. In the following, we will first outline some general guidelines, then illustrate them with short code examples, and finally give some additional suggestions to facilitate effective data-parallelism.

One underlying idea of data-parallel languages, such as Fortran D or HPF, is that the user does not explicitly specify the parallelism inherent in a program, but instead annotates the program with directives on how to distribute the data, and lets the compiler work from there. Performance and investment-preserving independence from environment-specific details are two key objectives. The art of data-parallel programming might be defined as achieving the former without compromising the latter. This also means that while the programmer should understand certain characteristics of data-parallelism, he or she should not have to develop a style which will have an adverse effect on code readability, maintainability or efficiency in non-data-parallel environments.

Performance depends on the degree of parallelism and on overheads, such as synchronization and communication costs.[2] Compilers for distributed memory architectures typically try to achieve parallelism by distributing loop iterations across

[1] Material in this chapter comes from [29].

[2] Dynamic data distribution (re-distribution at runtime) can result in considerable overhead. In this correspondence, however, we address only those data distributions that may be interpreted statically.

```
PROGRAM Good1_FortD
REAL x(100)
DISTRIBUTE x(BLOCK)

do [i] = 1, 100
   x([i]) = ...
ENDDO
```

```
PROGRAM Good1_Node
REAL x(25)

do i = 1, 25
   x(i) = ...
ENDDO
```

Figure 12.1: Simple example loop with matching array access; the FORTRAN D program (left) can be compiled into a node program (right) with reduced loop bounds (assuming $N_{proc} = 4$). Distributed array and loop indices are framed.

processors. The first guideline for designing data-parallel programs should therefore be:

> *The flow of control should be structured; e.g.,* `DO`*-loops are preferable to* `GOTO`*s.*

Distributing loop iterations is commonly driven by some heuristic for minimizing communication, such as the owner-computes rule [22] or variations thereof. This heuristic may fail even for "embarrassingly parallel" algorithms if the program expressing the algorithm obscures the parallelism by some (perhaps apparently unrelated) means. Even though there are many other issues crucial for achieving successful parallelization, realizing these points and designing program and data structures accordingly should in themselves go a long way towards good performance for many applications.

The perhaps most important guideline that should guide design decisions is:

> *Loops and arrays should match.*

That is, in computationally intensive code regions, array subscripts and loop indices should be related to each other in a simple manner, allowing the compiler to derive a loop parallelization directly from an array distribution. To justify this guideline, let us briefly digress into the workings of a data-parallel compiler for distributed memory machines, such as the FORTRAN D compiler prototype at Rice University.

Assume the compiler is given a simple code segment as shown in Figure 12.1 on the left, $\texttt{Good1}_{\texttt{FortD}}$.[3] The FORTRAN D compiler will generate a node program $\texttt{Good1}_{\texttt{Node}}$, shown in Figure 12.1 on the right, which in turn will be compiled by the

[3] We use pseudo code in all code fragments in this correspondence. The Fortran D syntax to distribute an array requires the array to be aligned with a *decomposition* which in turn is distributed [22]; whereas with HPF the *align* directive is optional [65].

```
PROGRAM BAD1_FortD
REAL x(100)
DISTRIBUTE x(BLOCK)

do i = 1, 100
   j = ...
   x(j) = ...
ENDDO
```

```
PROGRAM BAD1_Node
REAL x(25)

do i = 1, 100
   j = ...
   IF ((j-1)/25 .EQ. my$p) THEN
      x(mod(j-1,25)+1) = ...
   ENDIF
ENDDO
```

Figure 12.2: Example loop not matching the array access; the FORTRAN D program (left) will be compiled into a node program (right) with full loops and a guard (assuming $N_{proc} = 4$). The id of an individual processor is denoted by **my$p**, which ranges from 0 to $N_{proc} - 1$.

native compiler of the target machine.[4] This program will be written in local name space, as opposed to the single, global name space of the FORTRAN D program. It will contain the instructions for an individual processor, and it may contain communication statements. Here we will focus on the distribution of computation; for a discussion of communication generation and other compilation issues we refer the reader to other publications [48, 102]. The compiler tries to parallelize the `i`-loop in `Good1`$_{\texttt{FortD}}$ by applying the owner-computes rule to the distributed array reference, `x(i)`. Assuming no sequentializing dependences on the rhs of the assignment, the owner-computes rule works fine here since induction variable `i` and array subscript `i` are in a simple relationship – they are identical. The loop and the array *match*. The compiler can fully apply the owner-computes rule at compile-time and perform loop-bounds reduction: assuming that there are $N_{proc} = 4$ processors, each processor will perform only a quarter of the total number of iterations.

Now consider the program `BAD1`$_{\texttt{FortD}}$ in Figure 12.2. Similar to `Good1`$_{\texttt{FortD}}$, the array and the loop match in size and we can parallelize the loop, assuming again no dependences. However, the loop index `i` and the arrays subscript `j` do not match; the compiler cannot apply loop-bounds reduction, but instead has to apply the owner-computes rule at run time with a guard. The core of the computation, which we assume to be the assignment to `x(j)`, will still be executed in parallel, but scalability is likely to be limited due to the fully replicated loop iteration set.

In some cases, it will be difficult to establish a simple relationship between loops and subscripts, for example in irregular access patterns of distributed arrays. However, subscript-analysis complications are often avoidable artifacts of program-

[4] We have also assumed that the loop bounds and array shapes are known to the compiler. This turns out not to be true in many cases of practical interest, further complicating efficient generation of node code.

```
PROGRAM Bad2a
REAL x(300)
DISTRIBUTE x(BLOCK)

do i = 1, 100
   do d = 1, 3
      j = 3*i + d - 2
      x( j ) = ...
   ENDDO
ENDDO
```

```
PROGRAM Bad2b
REAL x(300)
DISTRIBUTE x(BLOCK)

j = 0
do i = 1, 100
   do d = 1, 3
      j = j + 1
      x( j ) = ...
   ENDDO
ENDDO
```

```
PROGRAM Good2
REAL x(3, 100)
DISTRIBUTE x(*, BLOCK)

do  i  = 1, 100
   do d = 1, 3
      x(d,  i ) = ...
   ENDDO
ENDDO
```

Figure 12.3: FORTRAN D programs with linearized (left) and delinearized (right) array accesses.

ming styles that obscure compiler analysis in general, not only in the data-parallel context. For example, consider the FORTRAN D fragments in Figure 12.3 (from now on we will omit the `FortD` subscript in the examples). `Bad2a` and `Bad2b` illustrate the popular practice of linearizing arrays, for example by storing a set of coordinate triplets into a 1-D array. Such linearizations are used for example to eliminate a loop nest, facilitating vectorization of the resulting single loop [71]. For data-parallel programming, linearization often results in blurring the distinction between distributed and replicated subscript components. For example, in `Bad2a` or `Bad2b` one would typically want to distribute `x` along the triplets, but keep each individual triplet on a single processor; *i.e.*, the `i` loop should be parallelized, while the `d` loop should be replicated. This, however, is obscured to the compiler by the way the array is indexed, and, in `Bad2b`, by the artificial self-dependence in incrementing the counter `j`.

In the fairly clean and simple case of `Bad2a` and `Bad2b` one might still be able to teach a compiler to apply correctly loop-bounds reduction to the outer loop; in the general case, however, it is likely that the compiler will resort to replicating both loops and inserting guards similar to $\texttt{Bad1}_{\texttt{Node}}$. It is much more desirable to clearly reflect the programmer's intent by splitting the subscript of `x` into the distributed index `i` and the replicated dimension index `d`, as shown in `Good2`. In general, a guideline for making the compiler's life easier is:

Arrays should not be linearized.

12.1.1 Guidelines applied to a molecular dynamics program

Before turning to our experience with parallelizing GROMOS [47] using the FORTRAN D compiler, we first give a brief introduction into the underlying application for the examples presented here; for more details, we refer the reader to the literature [24, 77]. Molecular dynamics (MD) is a classical mechanics approach typically used to determine the motion of large molecular systems. At the core of the simulation, a force is calculated for each atom from the analytic derivative of a potential energy function. This force displaces the atom from its position in the previous time step. The MD program iterates over some number of time steps in the course of calculating a molecular dynamics trajectory. Since each atom interacts with other atoms in some spatial neighborhood, dependences arise between atoms in so far as the potential energy function for each atom is evaluated with the positions of surrounding atoms from the previous time step. A pair list indicating which atoms interact with each other is computed every t steps, where t usually ranges from 10 to 50. Since there are typically tens to hundreds of interaction partners for each atom, the data structures representing the pair list tend to be the most space consuming in the program. Within a time step, the computation for each atom is independent from the computation for all other atoms and therefore is inherently parallel.[5] We base this report on the replicated approach [26], where we distribute the pairlist data structures, `inb` and `jnb`, while replicating the other principal arrays, which includes the coordinate and velocity arrays, `x` and `v`, and the forces, `f`. For studies on more aggressive distributions we refer the interested reader to the literature [24].

Delinearizing the non-bonded force calculation

The non-bonded force (NBF) constitutes the main component of the molecular dynamics computation performed by GROMOS and other MD programs. Since a force is computed for each atom, one natural implementation of the NBF algorithm loops over atoms and their partners, computes the force between them, and accumulates it according to Newton's Third Law, as shown in the (highly abstracted) GROMOS subroutine `NBF_Lin_At()` in Figure 12.4. The underlying computation is inherently parallel; however, the compiler will fail to parallelize this loop nest. The reason is the loop-carried dependence on `cnt`, which is similar to the dependence on `j` in `Bad2b` (Figure 12.3). Here, however, even advanced compiler analysis cannot identify a simple relationship between the array subscript `cnt` and the loop indices `i` and `p`. The problem is that in order to retrieve the list of partners for some atom `i`, one has to calculate the correct offset into `jnb`, which in turn depends on `inb(i´)` for $1 \leq$ `i´`

[5] Inter-atom dependences can arise for the integration step in simulations with constraints [89], but, in any case, the energy calculations are independent.

```
SUBROUTINE NBF_Lin_At()
INTEGER inb(Natom)
INTEGER jnb(MaxAllP)
DISTRIBUTE inb(BLOCK)
DISTRIBUTE jnb(BLOCK)

cnt = 0
DO i = 1, Natom
   DO p = 1, inb(i)
      cnt = cnt + 1
      j = jnb(cnt)
      ff = nbf_func(i,j)
      f(i) = f(i) + ff
      f(j) = f(j) - ff
   ENDDO
ENDDO
```

```
SUBROUTINE NBF_Delin_At()
INTEGER inb(Natom)
INTEGER jnb(MaxAtomP,Natom)
DISTRIBUTE inb(BLOCK)
DISTRIBUTE jnb(*,BLOCK)

DO i = 1, Natom
   DO p = 1, inb(i)
      j = jnb(p,i)
      ff = nbf_func(i,j)
      f(i) = f(i) + ff
      f(j) = f(j) - ff
   ENDDO
ENDDO
```

```
SUBROUTINE NBF_Delin_Chg()
INTEGER inb(Nchg)
INTEGER jnb(MaxChgP,Nchg)
DISTRIBUTE inb(BLOCK)
DISTRIBUTE jnb(*,BLOCK)

DO ii = 1, Nchg
   DO i = firstAt(ii), lastAt(ii)
      DO p = 1, inb(ii)
         jj = jnb(p,ii)
         DO j = firstAt(jj), lastAt(jj)
            ff = nbf_func(i,j)
            f(i) = f(i) + ff
            f(j) = f(j) - ff
         ENDDO
      ENDDO
   ENDDO
ENDDO
```

Figure 12.4: NBF kernel with linearized (left) and delinearized (middle) atom-based pair list, and with delinearized charge-group-based pair list (right). `Natom` is the number of atoms, `MaxAllP` the maximum total number of partners, and `MaxAtomP` the maximum number of partners per atom. `Nchg` is the number of charge groups, and `MaxChgP` the maximum number of partners per charge group. `firstAt` and `lastAt` give the range of atoms for a charge group.

$<$ `i`; *i.e.*, to determine the range of `j` for some iteration `i`, one has to iterate through all previous `jnb` segments. The advantage of linearizing `jnb` this way is space conservation; instead of having to reserve an equal amount of storage for each atom's pairlist, we only need to reserve enough storage to accommodate the sum of partners. However, we can expect the storage savings in distributing `jnb` across processors to more than compensate for this increased space requirement, and to do so requires a delinearization as shown in `NBF_Delin_At()` in Figure 12.4. This will also allow parallelization, since now the distributed and replicated array dimensions are separated, and they directly correspond to the surrounding parallel and sequential loops.

Delinearizing the pairlist construction

The force calculation in `NBF_Delin_At()` now corresponds to the pattern in `Good2`, so, in itself, it can easily be parallelized. However, we must also consider the construction of the pair list; `Pairs_Delin_At()` in Figure 12.5 shows a simplified version of the corresponding GROMOS routine, with `jnb` delinearized according to `NBF_Delin_At()`. It turns out that the criterion for including pairs of atoms in the pairlist actually depends on which charge group each atom belongs to, where a charge group is a

```
SUBROUTINE Pairs_Delin_At()
INTEGER inb(Natom), jnb(MaxAtomP,Natom)
DISTRIBUTE inb(BLOCK), jnb(*,BLOCK)

DO ii = 1, Nchg
   DO jj = ii + 1, Nchg
      isPair(jj) = isPair_func(ii, jj)
   ENDDO

   DO i = firstAt(ii), lastAt(ii)
      cnt = 0
      DO jj = ii + 1, Nchg
         IF (isPair(jj)) THEN
            DO j = firstAt(jj), lastAt(jj)
               cnt = cnt + 1
               jnb(cnt, i ) = j
            ENDDO
         ENDIF
      ENDDO
      inb( i ) = cnt
   ENDDO
ENDDO
```

```
SUBROUTINE Pairs_Delin_Chg()
INTEGER inb(Nchg), jnb(MaxChgP,Nchg)
DISTRIBUTE inb(BLOCK), jnb(*,BLOCK)

DO  ii  = 1, Nchg
   cnt = 0
   DO jj = ii + 1, Nchg
      IF (isPair_func(ii, jj)) THEN
         cnt = cnt + 1
         jnb(cnt, ii ) = jj
      ENDIF
   ENDDO
   inb( ii ) = cnt
ENDDO
```

Figure 12.5: Pair-list generation with delinearized atom-based pair list (left) and charge-group-based pair list (right).

collection of atoms treated collectively by the MD model. (Two atoms are considered "close" if their respective charge groups are "close.") `Pairs_Delin_At()` determines interacting pairs by looping over charge groups `ii`, determining for each charge group which other charge groups `jj` are close to it, storing the charge-group partners of `ii` in an array `isPair`. Then looping over the atoms `i` of charge group `ii`, `inb(i)` and `jnb(1:inb(i),i)` are constructed accordingly. Distributed and replicated array dimensions are cleanly separated; however, we again have unmatched loop and data structures. The distributed dimensions of `inb` and `jnb` are both indexed by atom index, whereas the enclosing loops iterate over charge groups (`ii`) and atoms within each charge group (`i`). The problem is that the granularity of the pair list computation is not the atom, but the charge group. We therefore switch to a charge-group-based representation, as in `Pairs_Delin_Chg()`; this not only allows parallelization by loop-bounds reduction, but also preserves memory.[6] To finalize the data-parallelization (at the level presented here), we now must also modify the NBF calculation to use the charge-group-based pair list. The result is `NBF_Delin_Chg()` shown in Figure 12.4.

[6] GROMOS already provides two versions of the pair-list construction and corresponding NBF calculation, an atom-based version and a charge-group-based one; however, both versions use linearized pair-list representations.

12.1.2 Discussion

We have stressed the importance of matching array and loop structures for data-parallel programming. However, there are many other issues influencing the quality of a compiler's analysis and the performance of the resulting code. We list, without further elaboration, other guidelines which may be of particular significance to dusty-decks.

Do not use distributed arrays as work space for other, non-distributed (or differently distributed) data.

Keep unrelated computations separate.

Keep array uses consistent across procedure boundaries.

As a general rule of thumb, one may say that programs that are hard to parallelize by hand will be even harder to parallelize for the compiler; data-parallel compilers provide only limited help with the high-level task of extracting exploitable parallelism from an application. However, that is not to say that a compiler can be of little use for parallelization. High-level data-parallel languages such as Fortran D and HPF remove from the programming task the tedious, error-prone, machine-specific, low-level work that has traditionally accompanied parallel computing. This note intends to help programmers harness that power to its fullest potential.

We have seen, however, the need for considerable restructuring to permit the effective use of Fortran D directives. Qualitatively, the IPfortran programming effort was simplier, and more tolerant of the awkward structure of the GROMOS dusty deck. With well–structured Fortran, the potential of HPF–like languages can be better harnessed. But, until the necessary structured programming style becomes the rule, a need for low–level paradigms will persist.

Chapter 13

Summary and Directions for Future Work

13.1 Languages

IPfortran extends the semantics of FORTRAN77 with the @ operator and a reduction–operator syntax. These operators permit the concise expression of frequent operations performed by parallel scientific algorithms. While IPfortran has separate–address–space semantics, reduction operators could be used to express bulk data transfers for shared memory data structures. The IPfortran implementation [8] is the first MIMD implementation of user–defined reduction operations reported in the literature that we know of.

IPfortran has been seen to be useful for the design and implementation of parallel algorithms [9, 74, 75, 10] and has been ported to a number of parallel platforms (Chapter 5.4.3). IPfortran has the advantage of operating at a low level, thereby minimizing restrictions in the expression of an algorithm for a given architecture. Indeed, many of the constructs proposed for HPF-2 are aimed at situations successfully addressed by IPfortran. Some of these include [40]:

- Irregular array mapping.
- Computation mapping to processors.
- Reduction operations in an INDEPENDENT DO loop.
- User defined reduction operations.
- Asserting reuse of communication patterns.

ℙfortran achieves portability through the use of machine–dependent libraries. ℙfortran's advantages are also the source of disadvantage. A low–level paradigm, ℙfortran requires that the programmer distribute data structures and loop iterations. This task is automated in some other languages, notably, Fortran D and HPF. However, while Fortran D and HPF automate much of the tedious work in parallel algorithm development, they require well–structured programs, a key issue in parallelizing "dusty decks."

Further ℙfortran work includes:

1. Accommodating collective communication between subgroups of processes.

2. Accommodating task parallelism for irregular systems.

3. A more sophisticated intermediated representation facilitating communication hoisting.

4. Compiling directly for shared memory computers, in addition to the current method utilizing a message passing system.

13.2 Parallel molecular dynamics

Parallel molecular dynamics with a cutoff radius is well suited for a spatial decomposition. A three–dimensional mesh would seem to be the connection network of choice, cost arguments aside. Load balancing was seen to be effective. The need for load balancing of a typical biological system will be a function of the number of atoms per processor since these systems tend to have uniform density. For I/O, to accommodate storage of the entire trajectory requires the use of parallel I/O systems. Otherwise, a reduction in the output volume is necessary. For trajectory outputting, a merging problem was identified that could further restrict the quantity of data output per time step. The decomposition is closely coupled to the merging problem for trajectory outputting. The use of shared memory or direct access I/O could help to reduce the overhead of this phase.

Collective communication for updating nearest–neighbors data benefits from our greedy-yz mapping heuristic using the shift algorithm, which we showed is itself optimal in complexity. Further work in this area includes

1. Research into mapping methods for the blockwise decomposition for meshes and other networks.

2. Determining lower bounds for communication in performing nearest neighbor communication for a 3D system on a 2D mesh.
3. Development of collective communication algorithms and mapping strategies for the hierarchical decomposition.

Molecular dynamics modeling is undergoing a trend to improve the electrostatics modeling. The improved methods tend to take into account all particle interactions, thereby requiring significant global communication. This alters the communication character from that discussed in this work for the replicated and spatial decomposition algorithms. It is not clear that the spatial decomposition will be the decomposition of choice given the global communication requirements.

Bibliography

[1] A.V. Aho and J.D. Ullman. *Principles of Compiler Design.* Addison-Wesley, Reading, MA, 1977.

[2] M. J. Allen and D. J. Tildesley. *Computer Simulation of Liquids.* Oxford University Press, 1987.

[3] Jennifer M. Anderson, Saman P. Amaraasinghe, and Monica S. Lam. Data and computation tranformations for multiprocessors. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 166–178, July 1995.

[4] G.R. Andrews and F.B. Schneider. *Computer Surveys*, 15:3, 1983.

[5] B. Bagheri, T. W. Clark, and L. R. Scott. IPfortran (a parallel extension of Fortran) Reference Manual. Research Report UH/MD–119, Dept. of Mathematics, University of Houston, 1991. Available via anonymous ftp from `hpc.uh.edu` as `pub/pf/pfmanual`.

[6] B. Bagheri, B. Raghavachari, and L. R. Scott. Sharing variables in parallel programs. Research Report UH/MD–79, Dept. of Mathematics, University of Houston, 1990.

[7] Babak Bagheri. *Parallel Programming with Guarded Objects.* PhD thesis, Pennsylvania State University, 1994.

[8] Babak Bagheri, Terry Clark, and L. Ridgway Scott. Pfortran: a parallel dialect of Fortran. *ACM Fortran Forum*, 11(3):20–31, 1992. Available via anonymous ftp from `hpc.uh.edu` as `pub/articles/fforum92`.

[9] Babak Bagheri, Andrew Ilin, and L. Ridgway Scott. Parallelizing UHBD. Technical report, Texas Center for Advanced Molecular Computation, University of Houston, Houston, TX 77204-5502, 1993.

[10] P. Bała, P. Grochowski, B. Lesyng, and J. A. McCammon. Quantum–classical molecular dynamics and its computer implementation. *Computers & Chemistry*, 1995.

[11] S. Benkner, B. Chapman, and H. Zima. Vienna Fortran 90. In *Scalable High Performance Computing Conference*, Williamsburg, VA, April 1992.

[12] H. Berryman, J. Saltz, and J. Scroggs. Execution time support for adaptive scientific algorithms on distributed memory machines. *Concurrency: Practice and Experience*, 3(3):159–178, June 1991.

[13] M. Bieterman. *Journal of Supercomputing*, 2:381, 1988.

[14] S. H. Bokhari. On the mapping problem. *IEEE Transactions on Computers*, C-30(3):207–214, 1981.

[15] J.P. Boris. A vectorized "near neighbors" algorithm of order n using a monotonic logical grid. *J. Comp. Phys.*, 66:1–20, 1986.

[16] J.P. Boris. New directions in computational fluid dynamics. *Ann. Rev. Fluid Dynamics*, 21:345–385, 1989.

[17] D. K. Bradley. First and second generation hypercube performance. Technical Report UIUCDCS–R–88–1455, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1988.

[18] Bernard R. Brooks, Robert E. Bruccoleri, Barry D. Olafson, David J. States, S. Swaminathan, and Martin Karplus. CHARMM: A program for macromolecular energy, minimization and dynamics calculations. *Journal of Computational Chemistry*, 4(2):187–217, 1983.

[19] Bernard R. Brooks and Milan Hodoscek. Parallelization of CHARMM for MIMD machines. *Chemical Design Automation News*, 7(12):16–22, 1992.

[20] M. Karplus C. L. Brooks III and B. M. Pettitt. *Proteins: A Theoretical Perspective of Dynamics, Structure and Thermodynamics*. Wiley, New York, 1988.

[21] Yuan Cai and T. W. Clark. Users' Guide: Trajectory outputting and post–merging with filter. Technical Report 199, Dept. of Mathematics, University of Houston, March 1995.

[22] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *Journal of Supercomputing*, 2:151–169, October 1988.

[23] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[24] T. W. Clark, R. v. Hanxleden, J. A. McCammon, and L. R. Scott. Parallelizing molecular dynamics using spatial decomposition. In *Scalable High Performance Computing Conference*, pages 95–102, Knoxville, TN, May 1994. IEEE Computer Society. Available via anonymous ftp from `softlib.rice.edu` as `pub/CRPC-TRs/reports/CRPC-TR93356-S`.

[25] T. W. Clark and J. A. McCammon. Parallelization of a molecular dynamics non-bonded force algorithm for MIMD architectures. *Computers & Chemistry*, 14(3):219–224, 1990.

[26] T. W. Clark, J. A. McCammon, and L. R. Scott. Parallel molecular dynamics. In *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing*, pages 338–344, Houston, March 1991. SIAM. Available via anonymous ftp from `hpc.uh.edu` as `pub/articles/siam91`.

[27] T. W. Clark, R. v. Hanxleden, K. Kennedy, C. Koelbel, and L. R. Scott. Evaluating parallel languages for molecular dynamics computations. In *Scalable High Performance Computing Conference*, Williamsburg, VA, 1992. IEEE. Available via anonymous ftp from `softlib.rice.edu` as `pub/CRPC-TRs/reports/CRPC-TR992202-S`.

[28] Terry W. Clark, J. Andrew McCammon, L. Ridgway Scott, and Stanislaw Wlodek. I/O limitations in parallel molecular dynamics. In *Proceedings of the 1995 ACM/IEEE Supercomputing Conference*, San Diego, CA, December 1995. IEEE Computer Society Press. Frank Baker and Janus Wehmer, production editors. Available on CD-ROM and at http://www.supercomp.org/sc95/proceedings.

[29] Terry W. Clark, Reinhard v. Hanxleden, and Ken Kennedy. Experiences in data-parallel programming. Technical Report CRPC-TR94495-S, Center for Research on Parallel Computation, December 1995. To appear in Scientific programming, 1996. Also available via anonymous ftp from `softlib.rice.edu` as `pub/CRPC-TRs/reports/CRPC-TR94495-S`.

[30] E. Clementi, G. Corongiu, J. Detrich, J. Kahnmohammadbaigi, S. Chin, L. Domingo, A. Laaksonsen, and N.L. Nguyen. *Physica*, 131B:74, 1988.

[31] D.L. Cochrane and D.G. Truhlar. *Parallel Comput.*, 6:63, 1988.

[32] Stephen E. DeBolt and Peter A. Kollman. Ambercube md, parallelization of amber's molecular dynamics module for distributed–memory hypercube computers. *Journal of Computational Chemistry*, 14(3):312–329, 1993.

[33] E. W. Dijkstra. Communicating sequential processes. Technical Report EWD-123, Technological University, Eindhoven, The Netherlands, 1965.

[34] E. W. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, March 1968.

[35] Marios D. Dikaiakos, Anne Rogers, and Ken Steiglitz. A comparison study of heuristics for mapping parallel algorithms to message–passing multiprocessors. Technical Report TR-446-94, Princeton, 1994.

[36] C. I. Draghicescu. *Efficient Algorithms for Particle Methods*. PhD thesis, Dept. of Computer Science, The Pennsylvania State University, 1991.

[37] J. Eastwood, R. Hockney, and D. Lawrence. PM3DP – the three dimensional periodic particle-particle/particle-mesh program. *Computer Physics Communications*, 19:215–261, 1977.

[38] K. Esselink, B. Smit, and P.A.J. Hilbers. Efficient parallel implementation of molecular dynamics on a toroidal network. Part II. Multi-particle potentials. *Journal of Computational Physics*, 106:101–107, 1993.

[39] David Fincham. Parallel computers and molecular simulation. *Molecular Simulation*, 1:1–45, 1987.

[40] High Peformance Fortran Forum. Hpf-2 scope of activities and motivating applications. Technical Report Version 0.8, November 1994. Available at ftp://hpsl.cs.umd.edu/pub/hpf_bench/hpf2.ps.

[41] Message Passing Interface Forum. MPI: a message–passing interface standard. Technical report, Work supported by ARPA and NSF under grant ASC-9310330, the NSF and Technology Center Coooperative Agreement no. CCR-8809616, and by the commission of the European comunity through Esprit project P6643(PPPE), May 1994.

[42] G. C. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Technical Report TR90-141, Dept. of Computer Science, Rice University, December 1990. Revised April, 1991.

[43] G. C. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Multiprocessors*. Prentice-Hall, 1988.

[44] Geist and Romine. Lu factorization algorithms on distributed-memory multiprocessor architectures. *SIAM J. Sci. Stat. Comput.*, 9:639–649, 1988.

[45] M.K. Gilson, T.P. Straatsma, J.A. McCammon, D.R. Ripoll, C.H. Faerman, P.H. Axelsen, I. Silman, and J.L. Sussman. Open “back door” in a molecular dynamics simulation of acetylcholinesterase. *Science*, 263:1193–1344, March 1994.

[46] L. Greengard and V. Rokhlin. On the evaluation of electrostatic interactions in molecular modeling. *Chemica Scripta*, 29A:139–144, 1989.

[47] W. F. van Gunsteren and H. J. C. Berendsen. GROMOS: GROningen MOlecular Simulation software. Technical report, Laboratory of Physical Chemistry, University of Groningen, Nijenborgh, The Netherlands, 1988.

[48] R. v. Hanxleden. *Compiler Support for Machine-Independent Parallelization of Irregular Problems.* PhD thesis, Rice University, December 1994. Available via anonymous ftp from `softlib.rice.edu` as `pub/CRPC-TRs/reports/CRPC-TR94495-S`.

[49] R. v. Hanxleden, K. Kennedy, and J. Saltz. Value-based distributions in fortran d — a preliminary report. Technical Report CRPC-TR93365-S, Center for

Research on Parallel Computation, December 1993. Available via anonymous ftp from `softlib.rice.edu` as `pub/CRPC-TRs/reports/CRPC-TR93365-S`.

[50] R. v. Hanxleden and L. R. Scott. Load balancing on message passing architectures. *Journal of Parallel and Distributed Computing*, 13:312–324, 1991.

[51] Robert J. Harrison. harrison@tcg.anl.gov, 1992.

[52] Fredrik Hedman and Aatto Laaksonen. Data parallel large-scale molecular dynamics for liquids. *International Journal of Quantum Chemistry*, 46:27–38, 1993.

[53] *Proceedings of the High Performance Fortran Forum*, Houston, TX, Jan. 27–28, 1992.

[54] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng. An overview of the Fortran D programming system. In *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, CA, August 1991.

[55] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, NJ, 1985.

[56] R.W. Hockney and C.R. Jesshope. *Parallel Computers*. Adam Hilger Ltd., Bristol, 1981.

[57] Tsutomu Hoshino and Kiyo Takenouchi. Processing of the molecular dynamics model by the parallel computer PAX. *Computer Physics Communications*, 31(4), 1984.

[58] Henry Burkhardt III. Overview of the KSR1 computer system. Technical Report KSR-TR-9202001, Kendall Square Research, Boston, February 1992.

[59] Andrew Ilin, Babak Bagheri, L.Ridgway Scott, James M. Briggs, and J.Andrew McCammon. Parallelization of Poisson-Boltzmann and brownian dynamics calculations. 1995.

[60] Zdenek Johan, Kapil K. Mathur, S. Lennart Johnsson, and Thomas J.R. Hughes. Mesh decomposition and communication procedures for finite element applications on the connection machine cm-5 system. Technical Report TR-08-94, Harvard University Center for Research in Computing Technology, 1994.

[61] S. Lennart Johnsson and Ching Tien Ho. Data motion and high performance computing. *IEEE Transactions on Computers*, 38(9):1249–1268, 1989.

[62] S.L. Johnsson. Communication efficient basic linear algebra computations on hypercube architectures. Technical Report YALEU/CSD/RR-361, Yale University, September 1985.

[63] M. Karplus and J. A. McCammon. Dynamics of proteins: Elements and function. *Ann. Rev. Biochem.*, 53:263–300, 1983.

[64] Ken Kennedy and Ulrich Kremer. Automatic data layout for high performance fortran. Technical Report CRPC-TR94498-S, December 1994. Available at ftp://softlib.rice.edu.

[65] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr., and M. Zosel. *The High Performance Fortran Handbook.* The MIT Press, Cambridge University Press, 1994.

[66] Charles Koelbel and Piyush Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, 1991.

[67] Ulrich Kremer. *Automatic data layout for distributed memory machines.* PhD thesis, Rice University, October 1995.

[68] V. Kumar and M. Calache. Treatment of Alzheimer's disease with cholinergic drugs. *Int. J. Clin. Pharmacol., Ther. Toxicol.*, 29:23, 1991.

[69] S. G. Lambrakos and J. P. Boris. Molecular dynamics simulation of $(N_2)_2$ formation using the monotonic lagrangian grid. *J. Chem. Phys.*, 90:4473–4481, 1989.

[70] S.Y. Lee and J. Aggarwal. A mapping strategy for parallel processing. *IEEE Transactions on Computers*, C-36(11):1384–1397, Nov 1987.

[71] L.M. Liebrock and K. Kennedy. Parallelization of linearized application in Fortran D. In *Proceedings of the 8th International Parallel Processing Symposium*, New York, NY, April 1994.

[72] C. Lin and L. Snyder. A comparison of programming models for shared memory multiprocessors. In *Proceedings of the 1991 International Conference on Parallel Processing, Vol. 2*, pages 163–170, 1991.

[73] Richard J. Loncharich and Bernard R. Brooks. The effects of truncating long-range forces on protein dynamics. *PROTEINS: Structure, Function and Genetics*, 6:32–45, 1989.

[74] J. A. Lupo. Modeling NLO materials using parallel computers. In *Contributive Research and Development Volume 144, Final Report Task 167*, 1994. Wright Laboratory, Wright-Patterson AFB OH, Contract F33615-90-C-5944.

[75] J. A. Lupo. Benchmarking UHGROMOS. In *Proceedings of the Twenty-Eighth Annual Hawaii International Conference on System Sciences*. IEEE, January 1995. Maui, HI.

[76] J.T. Marx. Alzheimer's drug trial put on hold. *Science*, 238:1041, 1989.

[77] J. A. McCammon. Computer-aided molecular design. *Science*, 238:486–491, October 1987.

[78] J. A. McCammon and Stephen C. Harvey. *Dynamics of proteins and nucleic acids*. Cambridge University Press, Cambridge, 1987.

[79] Juan J. Morales and Maris J. Nuevo. A technique for improving the link-cell method. *Computer Physics Communications*, 60:195–199, 1990.

[80] Florian Müller-Plathe and David Brown. Multi-colour algorithms in molecular simulation: Vectorisation and parallelisation of internal forces and constraints. *Computer Physics Communications*, 64:7–14, 1991.

[81] A.K. Nanda and L.N. Bhyuyan. Mapping applications onto a cache coherent multiprocessor. *Proceedings of Supercomputing 1992*, pages 368–377, Nov 1992.

[82] Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global arrays: A portable "shared-memory" programming model for distributed memory computers. In *Proceedings of Supercomputing 1994*, pages 340–349. IEEE Computer Society Press, 1994.

[83] M. R. S. Pinches and D. J. Tildesley. Large scale molecular dynamics on parallel computers using the link-cell algorithm. *Molecular Simulation*, 6:51–87, 1991.

[84] Steve Plimpton. Fast parallel algorithms for short–range molecular dynamics. *Journal of Computational Physics*, 117:1–19, March 1995.

[85] A. Rahman. Correlations in the motion of atoms in liquid argon. *Physical Review*, 136(2A):405–411, 1964.

[86] M.C. Rinard, D.J. Scales, and M.S. Lam. Jade: a high-level, machine-independent language for parallel programming. *IEEE Computer*, 26(6):28–38, June 1993.

[87] M. Rosing, R. Schnabel, and R. Weaver. The DINO parallel programming language. *Journal of Parallel and Distributed Computing*, 13(1):30–42, September 1991.

[88] Edward Rothberg, Jaswinder Pal Singh, and Anoop Gupta. Working sets, cache sizes and node granularity issues for large–scale multiprocessors. *Comp. Arch. News*, 21(2):14–25, 1993.

[89] Jean-Paul Rycaert, Giovanni Ciccotti, and Herman J.C. Berendsen. Numerical integration of the cartesian equations of motion of a system with constraints: Molecular dynamics of n-alkanes. *Journal of Computational Physics*, 23:327–341, 1977.

[90] Youcef Saad and Martin H. Schultz. Data communication in hypercubes. *Journal of Parallel and Distributed Computing*, 6:115–135, 1989.

[91] Gary W. Sabot. *The Paralation Model*. Massachusetts Institute of Technology, 1988.

[92] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8(2):303–312, 1990.

[93] Hiroyuki Sato, Yasumasa Tanaka, Hiroshi Iwama, Shigetsugu Kawakida, Minoru Saito, Kenji Morikami, Toru Yao, and Shigenobu Tsutsumi. Parallelization of AMBER molecular dynamics program for the AP1000 highly parallel computer. In *Scalable High Performance Computing Conference*, Williamsburg, VA, 1992.

[94] L. R. Scott, J. M. Boyle, and B. Bagheri. Distributed data structures for scientific computation. In M. T. Heath, editor, *Proceedings of the 3rd Hypercube Multiprocessors Conference*, pages 55–66, Philadelphia, PA, 1987. SIAM.

[95] C.L. Seitz. The cosmic cube. *Communications of the ACM*, 28:22–33, 1985.

[96] Jian Shen and J. A. McCammon. Molecular dynamics simulation of Superoxide interacting with Superoxide Dismutase. *Chemical Physics*, 158:191–198, 1991.

[97] Zhiyu Shen, Zhiyuan Li, and Pen-Chung Yew. An empirical study of fortran programs for parellizing compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):356–364, 1990.

[98] Jaswinder Pal Singh, John L. Hennessey, and Anoop Gupta. Scaling parallel programs for multiprocessors: Methodology and examples. *IEEE Computer*, pages 42–50, July 1993.

[99] W. Smith. Molecular dynamics on hypercube parallel computers. *Computer Physics Communications*, 62:229–248, 1991.

[100] T. P. Straatsma and J. Andrew McCammon. ARGOS, a vectorized general molecular dynamics program. *Journal of Computational Chemistry*, II(8):943–951, 1990.

[101] Joel L. Sussman, Michal Harel, Felix Frolow, Christian Oefner, Adrian Goldman, Lilly Toker, and Israel Silman. Atomic structure of acetylcholinesterase from torpedo californica: A prototypic acetylcholine-binding protein. *Science*, 253:872, 1991.

[102] C. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines*. PhD thesis, Rice University, January 1993.

[103] Robert A. van de Geijn. On global combine operations. LAPACK Working Note 29.

[104] W. F. van Gunsteren, H. J. C. Berendsen, J. Hermans, and J. P. A. Postma. *Proc. Natl. Acad. Sci. USA*, 80:4315, 1983.

[105] Loup Verlet. Computer experiments on classical fluids. I. thermodynamical properties of lennard-jones molecules. *Physical Review*, 159(1):98, 1967.

[106] Guhan Viswanathan and James R. Larus. User-defined reductions for communication in data-parallel languages. Technical Report CS-TR-96-1293, University of Wisconsin–Madison, Madison, WI, January 1996.

[107] K.J. Wallace. *Phil. Trans. R. Soc. Lond., Ser. A*, 326:481, 1988.

[108] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and Jonn L. Hennessey. The SUIF compiler system: a parallelizing and optimizing research compiler. Technical Report CSL-TR-94-620, Computer Systems Laboratory, Stanford University, 1994.

[109] Michael J. Wolfe. Semi-automatic domain decomposition. In *Proceedings of the 4th Conference on Hypercube Concurrent Computers and Applications*, Monterey, CA, March 1989.

[110] Janet Wu, Joel Saltz, Seema Hiranandani, and Harry Berryman. Runtime compilation methods for multicomputers. In *1991 International Conference on Parallel Processing*, pages II–26–II–30, 1991.