Massively Parallel Methods for Engineering and Science Problems

W. J. Camp, S. J. Plimpton, B. A. Hendrickson, and R. W. Leland
Massively Parallel Computing Research Laboratory
MS 1111, Sandia National Laboratories
Albuquerque, NM 87185

Abstract

Massively parallel computers promise unique power for engineering and scientific simulations. Development of parallel software tools and algorithms that take full advantage of this power is a continuing challenge to computational researchers. In this paper we discuss the advantages a message–passing multiple–instruction/multiple–data (MIMD) programming approach has for parallel simulations and two domain–decomposition and load–balancing methods designed to optimize data locality and minimize communication costs on large parallel machines. The methods have proven useful for parallel computations in a broad range of technical fields including structural and fluid dynamics and chemical, biological, and materials science.

This work was supported by the Office of Scientific Computing of the U.S. Department of Energy and was performed at Sandia National Laboratories, operated for the DOE under contract No. DE–AC04–76DP00789.

1 Introduction

Scientific research and engineering development are relying increasingly on computational simulation to augment theoretical analysis, experimentation, and testing. Many of today's problems are far too complex to yield to mathematical analyses. Likewise, large-scale experimental testing is often infeasible for a variety of economic, political, or environmental reasons. At the very least, testing adds to the time and expense of product development. In the worst cases, the long timescales needed to develop and test prototypes prevent new products from being brought to market and adversely affect competitiveness. Thus, "agile" manufacturing, with limited need for prototyping, is the goal in today's fast moving marketplace. Reaching this goal will require the ability to perform larger, faster, and more complex simulations.

Massively parallel (MP) computing is seen by engineers and scientists as a tool useful for reaching this goal. Unfortunately, for those who simply wish to be users of the tool, exploiting parallelism in physical problems brings with it a new set of challenges. Issues that were not important for vector supercomputer implementation can seriously impact a simulation's performance on a MP machine. These include decomposing the physical problem into naturally parallel parts, load–balancing the computation across multiple processors, efficiently communicating data between processors, and fast movement of data in and out of the machine. It is these issues, which the engineer/scientist views as tangential to the task of actually solving the problem at hand, that have created the perception that parallel computers are a specialty breed and have slowed their acceptance by industry.

In our view there is a three–fold solution to this problem. The first part is exposure. As large parallel machines become more generally available to the traditional vector supercomputer user community, and programmers become accustomed to thinking about their problems from a parallel viewpoint, creating new parallel simulations will become commonplace and thus easier. A second part of the solution is auxiliary tools supplied by vendors and third–parties. In recent years, MP computing companies have been investing large fractions of their resources in software development – parallel libraries, compilers, performance monitors, etc. These supplementary tools are taking time to mature, just as they did for vector supercomputers, but are continually becoming more robust and useful. The third part of the solution is perhaps the most important – the development of new algorithms and methods that extract maximum parallelism from a particular computational problem. Often this requires new ways of structuring data, mapping data to processors, and communicating between processors to take full advantage of a parallel machine's capabilities. This is important because a large class of complex simulations, those with irregular domains and/or dynamically changing interactions, will not (in the near future) yield to effective automatic parallelization. It is these complex simulations that industry is accustomed to running in a production mode on their vector supercomputers and is most interested in running on new parallel machines so that larger problems can be solved faster. Examples include DYNA–3D and PRONTO–3D for structural mechanics modeling, GAUSSIAN and DMOL for quantum mechanical simulation of molecules, CHARMM and DISCOVER for molecular dynamics simulation of organic systems, and FIDAP for modeling complex fluid flows.

In this paper we discuss some work that has been done recently at Sandia in this third area of developing new parallel methods and algorithms for irregular problems. In essence these are tools that enable more effective use of MP computing as a tool itself. First, in Section 2 we give a brief overview of the different styles of parallel computing in common use. In particular, the advantages, in our view, of the explicit-message-passing model of parallel computing are discussed. In Section 3 we first contrast two common approaches to parallelizing regular problems and then discuss in some detail two methods developed for different kinds of prototypical irregular problems. The first of these is actually a collection of tools for partitioning irregular domains among a group of processors, widely used at Sandia and elsewhere for simulations of partial differential equations on irregular meshes. The second tool is a force–decomposition method we have developed that is appropriate for simulations of particles where the interactions that must be computed are either random or rapidly changing. We have used the method in a variety of molecular dynamics simulations. In both the partitioning and force–decomposition sections, we will illustrate the utility of the tools with performance results from Sandia application codes. Finally, in Section 4, we draw some brief conclusions.

2 Overview

Realizing the full potential of parallel computing will require great hardware and software advances. Today, powerful "production" supercomputers can carry out a few billions of operations per second and can store around a billion or so words of data in fast memory. With traditional semiconductor electronics, there appear to be hard economic and physical limits to how much faster individual processors can be built. Increasingly, the push for faster compute speeds requires greater parallelism. At the individual processor level this currently translates into pipelined vector processors or wide–processor architectures which can process more than one independent instruction stream in the same clock cycle, hierarchical caches, and wider data paths to memory. These fine–grained (or inner–loop) uses of parallelism are combined with coarse–grained (or outer–loop) parallelism in massively parallel (MP) supercomputers, in which many hundreds or thousands of processors cooperate to solve a problem.

Because memory contention limits performance in shared memory computers as the number of processors increases, all current MP computers use distributed or hybrid memory concepts. In distributed memory computers, memory is physically local to an individual processor; data is exchanged between processors by communication on a network. A hybrid architecture replaces the processor in a distributed memory computer by a multi–processor node in which a few processors share memory on a bus or fast–switch architecture. An example of distributed memory MP computers are the nCUBE 2 supercomputers at Sandia, in which 1024 custom processors, each with 4 MB of local memory, are interconnected by a 10–dimensional hypercube network. An example of a hybrid machine is the Intel Paragon with MP nodes. The MP nodes being developed by Intel will each contain two or more compute processors which share a common memory. The nodes on the Paragon are interconnected as a two–dimensional mesh. There are also parallel computers whose physical memory is distributed but which (through software) make it accessible by the user as global shared memory. For example, the Cray T3D has a global address space and provides extensive support for traditional shared–memory applications. Similarly, the Kendall Square Research KSR-1 machine supports a global address space and shared memory model in software and network hardware.

In addition to the distinction of whether physical memory is shared or distributed, parallel architectures may be SIMD (single–instruction, multiple–data) or MIMD (multiple–instruction, multiple–data). In SIMD machines all processors execute the same instruction at each clock cycle. Typically, a "front-end" computer issues each instruction to the array of SIMD processors. SIMD supercomputers are particularly effective in data parallel applications, such as many problems in fluid dynamics, where the computational parallelism arises through the (usually physically) parallel structure of the data. This parallelism can often be succintly expressed in array–oriented languages such as the CM–Fortran implemented by Thinking Machines for its SIMD CM–2 supercomputer. If branches or other conditionals are encountered in the code, processors which do not satisfy the condition are masked and remain idle during the remainder of that computational thread. This leads to loss of efficiency for problems with data irregularities or irregular computation requirements. In general the more realistic or complex the simulation being undertaken, the greater the loss in efficiency encountered in data parallel languages on SIMD machines. That is one of the reasons most current parallel supercomputer designs are based on the MIMD computing paradigm.

In distributed–memory MIMD supercomputers, each processor has a local copy of instruction code which it executes on its local data. Processors cooperate to exchange data in the form of messages sent from one processor to another. Because MIMD computers store code on each processor, they are capable of running with no "front-end". The exchange of data messages in a MIMD machine can either be automatically generated by the system at compile or run time or can be explicitly programmed by the user in the application code. We view the hope of automatic parallelization by a compiler as a long–term goal in distributed memory MIMD computation. One important research thrust in this area has grown out of the dataflow community and involves functional languages and multi-threaded architectures. While promising, much research remains to be done, and these approaches are not yet in common use. Similarly, we do not view the run–time generation of message passing (triggered by a remote memory access) as an efficient way to extract maximum parallelism from an application. While system support for these operations is provided on machines like the CRAY T3D and KSR-1, the user can almost always get much better performance by structuring the data and operations in the code to better reflect the parallelism inherent in the problem being solved.

By contrast, explicit message passing is a well–established programming technique for MIMD supercomputers. All of the distributed memory machines discussed above support it, as do many machines which can be programmed in a shared memory mode, such as multi–processor Cray Y–MPs. It is also, of course, an increasingly popular way to parallelize applications for networks of workstations through standardized protocols such as PVM. The explicit message passing model we consider here is one where each processor executes a code written in a standard high level language such as C, C++, Fortran 77, or Fortran 90. Data stored on other processors is obtained by explicit calls to read/write functions that pass a buffer of information between processors. This may be done in either a synchronous or asynchronous fashion. In this model single–processor code can be straightforwardly tuned for either superscalar or vector processors and more sophisticated communication constructs can be easily built on top of the basic read/write primitives.

While this approach "burdens" the programmer with the task of specifying what data is exchanged when, it has great generality and flexibility. It makes few limiting assumptions about the nature of the problem to be solved. For example, it allows MIMD computers to run SIMD applications by replicating the same code on each processor and synchronizing their activities only as needed. On MIMD machines these message–passing codes often run as effectively as their data parallel counterparts do on SIMD machines (see the next section). Within this model, MIMD computers can also run more loosely synchronized data parallel applications such as in a SPMD (single–program, multiple–data) model. While data parallel compilers like for CM–Fortran on Thinking Machine's CM–5 supercomputer extract some additional parallelism from conditionals by assigning processors to work on both cases of the conditional, load imbalance still arises as threads end at different times. Explicit message passing can often overcome this bottleneck and extract additional parallelism within the SPMD model at a coarser level of granularity. This can be achieved through a variety of methods: reducing load imbalance through context switching, reducing communication costs through innovative message routing, or masking communication costs by overlapping them with computation.

Finally, the explicit message passing model can also deal with several types of parallelism for which the data parallel model breaks down. One example is problems whose computational tasks are unequal parcels of independent work which can be assigned to whichever processor is available. Such problems arise in exploration geophysics and in the modeling of complex radar images, for instance. Several powerful hierarchical master–slave techniques developed for these problems are described in [9]. These techniques can achieve both high performance and high efficiency by maintaining near perfect load balance among processors even as they perform radically different tasks. Another example of applications that do not fit the data parallel model, but for which message passing on MIMD machines is well–suited, is problems with irregular data structures and/or dynamic communication patterns. For example, in molecular dynamics, as particles move the data structures that describe their interactions are constantly changing and cannot be efficiently recreated and processed in a data parallel mode. Similarly, problems where particles move through grids have changing communication and load–balance requirements which can be detected and adapted for most efficiently in a MIMD code.

3 MP Computational Tools

Most computational problems in science and engineering exhibit a significant degree of parallelism [3, 8]. If the problems are "regular" in that identical operations can be performed in parallel on each computational element (e.g. grid cell or particle) and "static" so that the topology of interactions between neighboring elements does not change as the simulation progresses, then both SIMD and MIMD implementations can be successful at achieving high parallel efficiencies. However, the two methodologies differ in their approach to the problem. It is instructive to consider a typical regular problem solved in both the MIMD and SIMD styles as a starting point, which we do in the next section. We then discuss two tools designed for irregular problems in the following sections.

3.1 Regular Problems

Consider a partial differential equation discretized on a topologically regular 3–D Cartesian grid. This is the underlying mathematical description for a variety of flow and wave–propagation simulations. The basic idea for parallelizing such a model is to perform a physically–based decomposition of the domain. This involves dividing up the spatial domain into P sub–domains, where P is the number of processors working on the problem. Each processor needs information from its adjacent processors to periodically update what are, essentially, the boundary conditions of its sub–domain. This decomposition is illustrated in schematic form in Figure 1 for a 2–D example.

Examples of large fluids dynamics codes parallelized using this decomposition include the ZEPHYR–3D code developed by Cline and Schutt [6] in which the full, incompressible, three–dimensional Navier-Stokes equations are solved semi–implicitly and the PCTH explicit, three–dimensional, multi–material shock–hydrodynamics codes, successfully parallelized by Robinson et. al. [15]. The largest problems run to date with ZEPHYR–3D were on Sandia's 1024–processor nCUBE 2 on meshes of around 8 million cells. Each processor was assigned a sub–domain of size 16x16x32 (8192) grid cells. Such a sub–domain has 2432 "surface" cells that need data from other processors to complete their computational updates. Two points regarding this decomposition are important. First, nearly 30% of the cells in a processor's sub–domain are surface cells – even for this very large problem. Second, for a typical iterative solution method such as the pre–conditioned conjugate gradient solver in ZEPHYR–3D in which several dozen iterations might be required to achieve adequate convergence, each processor will need to receive several hundred thousand variable values from neighboring processors in the course of a timestep. If this communications is not carefully structured it can seriously affect problem solution time.

For MIMD implementations such as those listed above, a useful book–keeping device that is often used [6] is to augment the variable set on each processor by a shell of "ghost" variables which are obtained from the neighboring sub–domains. How the message–passing reads and writes are structured to update these ghost cells depends upon the communications parameters of the particular architecture. In general, if the cost of starting up a message is large, if there is no effective limit on the length of a message, and if the communications bandwidth is high, it is cost-effective to bundle up as much information as possible in a single message in order to amortize the high cost of setting up the message. This is the case on most MIMD parallel machines such as the nCUBE 2 and Intel Paragon. By contrast, if the message start–up time is low or messages are severely restricted in length, there is no advantage to trying to "coarse–grain" the communications process and ghost cell values can be acquired one–at–a–time.

For the ZEPHYR–3D code discussed above the execution of one iteration of an update requires that ghost cell values be updated once though their values are referenced many times. The stencil operations are complex enough that for the large 8192 cell/processor problem the update requires about one second of CPU time on the nCUBE 2. The communications with the 6 six neighbors (in 3–D) are structured as six read/write pairs, one for each face of the sub–domain. It takes about 11 milliseconds to communicate all necessary data (6 messages @ 150 microsec startup time per message = 1 millisec; 2400 words or 20 Kbytes data transmitted @ 2Megabytes/sec communications bandwidth = 10 millisec). Thus the communication cost for the MIMD implementation is a negligible one or two percent of the total computation time.

Now contrast this with a SIMD–like implementation of the same problem using standard data parallel techniques, e.g. in Fortran 90. No ghost cells are used and the entire 3–D domain is treated as a single global array by the parallel program. Stencil operations are carried out in the usual way and references to cells that are actually owned by a neighbor processor trigger communication of a single data value. This is the way the problem would typically be implemented on SIMD machines such as the Thinking Machines CM–2 and the Maspar machines, as well as in data parallel Fortran on the CM–5 or Cray T3D. To understand the import of this result, we assume that a distant read (with communication) takes no more than 10 times the cost of a local read. We also assume that 10% of the one second local computation time arises from local data fetches. However 30% (2432/8192) of the fetches for this decomposition will be to surface cells owned by other processors. This is 3% of the total time (30% of 10%) but with the factor–of–10 penalty for distant reads, becomes a 30% communications overhead factor in the SIMD–like implementation. The key point is that in a MIMD implementation all surface cell data can be acquired once, stored in ghost cells, and then accessed cheaply numerous times. In the SIMD–like implementation communication takes place every time a memory access to a distant variable occurs. The only work–around is to make a local–memory copy of the distant variables. However, in a SIMD–like implementation the arrays are all global and the data–decomposition and processor boundaries are invisible to the programmer. Thus using ghost cells in a SIMD–like implementation requires making (numerous) copies of the entire data set, not just surface cells, a usually unacceptable memory cost.

Cline and Schutt [6] have recently used the MIMD techniques outlined above to solve a large Navier-Stokes problem, a 3–D Kelvin–Helmholtz instability resolved using 8.4 million grid cells on the 1024 processor nCUBE-2 at Sandia. A snapshot after 300 timesteps from that simulation is shown in Figure 2. The three-dimensional complexity of the unstable flow is evident, while the x–y cut through the flow shows typical vortex structures. This code ran on the nCUBE at about 1.5 Gigaflops; by comparison a highly vectorized version of ZEPHYR–3D runs at about 220 Megaflops on a single Cray Y–MP processor.

3.2 Graph Partitioning

One of the most important concerns in parallel computing is the proper distribution of workload across processors. For most scientific applications on massively parallel machines, the best approach to this distribution is to employ *data parallelism*; that is, to break the data structures supporting a computation into pieces and then to assign those pieces to different processors. Collectively, these partitioning and assignment tasks comprise the *domain mapping* problem.

Extensive practical experience has shown that in order to achieve high performance, a domain mapping must be found in which each processor has about the same workload and in which the overhead due to interprocessor communication is kept small. Unfortunately, discovering such a mapping can be difficult, particularly for the large, unstructured domains that typically appear in the solution of partial differential equations for scientific and engineering calculations.

Many approaches to this problem have been tried over the years, but the results have often been disappointing. Approximate global optimization strategies such as simulated annealing and genetic algorithms are capable of finding excellent mappings, but are often too expensive in practice. Simple heuristics based on ordering of coordinate or topological information are quick but they perform poorly on many complicated grids. More sophisticated heuristics typically show erratic behavior or are very problem specific.

We have recently developed several effective and economical domain mapping methods that are appropriate for finite difference, finite element, particle-in-cell and similar types of scientific computations. Our methods employ a graph model of the calculation in which vertices represent computation and edges represent communication. This model reduces the decomposition problem to one of graph partitioning, that is dividing the graph into sets with equal numbers of vertices such that a minimal number of edges cross between sets (an NP-hard problem). The assignment problem then requires that we find a mapping of these sets to processors that avoids messages between architecturally distant processors.

One of our new methods is a generalization of a recently proposed graph partitioning approach which has generated considerable interest because it seems to offer a good balance between generality, quality and efficiency. This *spectral* method partitions a graph by considering an eigenvector of an associated matrix to gain an understanding of global properties of the graph [14]. The graph is bisected using this eigenvector and the process repeated recursively on each of the halves until the desired number of sets is obtained.

We have improved this spectral technique in several important ways. First, we have made realistic modeling of scientific computations possible by allowing for unbalanced computation and communication loads in our graph model. Second, we have reformulated the objective function so that we try to minimize not only the amount of communication between processors, but also the number of messages traveling between distant processors in a hypercube or mesh architecture. We have therefore addressed, for these important architectures, both the decomposition and assignment problems. (Most previous methods, including the original spectral method, have considered only the decomposition problem.) Third, we have developed a method of using two or three eigenvectors to partition the graph into four or eight pieces at each stage of recursion rather than simply bisecting based on one eigenvector. This results in better mappings at a lower net cost than that incurred by other spectral methods. Fourth, we have paired the spectral method with a generalization of a well known graph partitioning technique due to Kernighan and Lin (KL) [11]. The

KL algorithm is a local optimization which improves an initial partitioning. The spectral method is quite good at coarsely dividing a graph, but often does poorly in the fine details, whereas KL exhibits precisely the opposite behavior. When paired, the combined method is substantially more robust and effective than either alone. This work has been reported in [10].

While spectral methods are much more economical than simulated annealing and other methods capable of finding very high quality partitions, they are still costly in comparison with the actual run time of many applications. Consequently, there is a strong interest in developing comparably powerful mapping methods with substantially lower cost. One approach is to accelerate the eigenvector computation in the spectral method by employing a multi-level iterative eigensolver. The graph is coarsened down several levels, an eigenvector is computed on the coarsest grid where it is relatively cheap to do so, and the eigenvector is then projected back up through the grid hierarchy to the finest grid, undergoing a refinement process as it is projected. This method was proposed and implemented by Barnard and Simon [2], who found that it did substantially reduce the run time of spectral bisection.

We have developed an alternative approach in which, rather than propagating an eigenvector of the coarsest grid back through the hierarchy, we instead propagate a partitioning. We first define a hierarchy of grids in which each grid is obtained from the next finer grid by contracting edges and lumping vertices rather than the more common method of removing vertices and inserting new edges. The edges to be contracted are chosen to constitute a maximal independent set, (i.e. selected edges must not share a vertex), and a quantitative history of the coarsening is preserved by adding the weights of edges and vertices which are superimposed by the contraction. Next we apply a spectral method to partition the weighted coarsest grid. Finally we expand the partition back through the grid hierarchy by uncontracting edges and splitting vertices, using the Kernighan-Lin algorithm described earlier to periodically refine the partition. This approach avoids various numerical difficulties associated with the computation and refinement of eigenvectors of very large matrices, and produces comparable partitions for much less effort than the standard spectral methods. Furthermore, this new multilevel method retains the ability to operate as a quadrisection or octasection algorithm, and can be applied to weighted graphs transparently.

We have tested these new methods on a number of unstructured grids from finite element, finite difference and particle-in-cell application codes. Several representative problems are presented here. The first is the dual graph of a finite element airfoil mesh generated by Barth at NASA Ames with 8034 vertices and 11813 edges. A decomposition of this mesh into eight sets using the new spectral octasection method combined with KL is shown in Figure 3.

It is clearly difficult to judge the merit of the partition visually, even for this relatively small two-dimensional problem. This reflects the general difficulty of the domain mapping problem. We can, however, calculate objective measures of success with respect to our graph model, and these are given in Table I. In addition to entries for the methods discussed, we have included an entry for the *inertial* method [16] which treats each mesh in the recursion as a rigid structure which is cut by a plane orthogonal to its principal axis. This is a very fast method which typically generates partitions of better quality than several other simpler heuristics in common use (e.g. coordinate bisection and level set partitioning). The *cuts* values are the number of edges that connect vertices assigned to different sets, which corresponds to the total volume of interprocessor communication. The values in the *hops* columns weight each cut edge by the number of wires between the corresponding processors in a hypercube network. In most scientific applications, many messages are being simultaneously routed, and this second metric accounts for message congestion in a mesh or hypercube multiprocessor.

The second example mesh is a three-dimensional finite element mesh of a complex manufacturing component generated using advanced meshing software at Sandia. This graph has 6673 vertices and 55664 edges. Results of a partition into eight sets are given in Table II, and the mesh decomposition produced by spectral octasection plus KL is shown in Figure 4.

For both these problems spectral octasection plus KL was clearly the best method at minimizing hops. For minimizing cuts, the multi-level bisection algorithm and spectral bisection plus KL were about equivalent, but the multilevel algorithm ran significantly faster. The multi-level octasection method, while still competitive, did not perform as well as expected. We believe this is because a simple local strategy like Kernighan-Lin has difficulty refining partitions into more than two sets.

To give an idea of the real impact of this technology, we partitioned a large, 3D finite element graph taken from a chemically reacting flow application (170k vertices, 204k edges) into 64 sets for execution on a portion of Sandia's Intel Paragon computer. The partitioning using the inertial method was completed in about 50 seconds on a Sparc 2 workstation. The combined KL/spectral multilevel bisection algorithm required about 12 minutes on the workstation, and reduced the run time on the parallel machine by about 30% per time step compared to the inertial decomposition. Over the course of a long simulation this represents a very considerable savings in computing resources.

3.3 Decompositions for Many–Body Problems

Particle or many–body simulations are another type of irregular and dynamic computational problem where load–balancing and domain–decomposition issues are important. They are computationally challenging because large numbers of particles are simulated for many thousands or even millions of timesteps. Even if the force interactions between particles are limited in range, the neighbors of each particle can change rapidly as they diffuse through the system. Molecular dynamics (MD) is a prototypical particle simulation in this respect. It shares common load–balancing and domain–decomposition problems with other kinds of particle simulations such as smooth particle hydrodynamics, vortex methods in fluid dynamics, and particle–in–cell models for plasma physics. MD is a commonly used simulation tool in many of the physical sciences [1]. It models the motion of atoms in materials and molecules by integrating the classical equations of motion, namely Newton's laws. Typical applications of MD in physics, chemistry, and material science include studies of the phase diagrams of novel materials, the properties of defects in solids, transport mechanisms in liquids and solids, and structural properties of pores and surfaces for catalysis applications. Applications of MD in biological systems include modeling of steric effects in complex molecules and conformational studies of protein folding [4].

To simulate N atoms on P processors, two parallel MD approaches have been widely used. The first is a particle decomposition, where each processor is assigned N/P atoms for the duration of the simulation. Each processor computes forces on only its atoms and updates their positions. To do this it must know the positions of potentially all other atoms in the simulation. At each timestep this information is communicated between processors by some kind of all–to–all exchange. The key point is that this communication operation scales as N, independent of P. Thus, while this algorithm has the advantage of simplicity, it does not scale well to large numbers of processors.

The second parallel method exploits the locality of short–range forces by assigning each processor a small region of the simulation domain – a spatial decomposition of the workload. Each processor computes forces and updates the positions of only the atoms in its region. Communication is performed every timestep to exchange positions with the processors owning neighboring regions. If the regions are larger than the range of the force interaction, the communication cost in this algorithm scales as N/P.

Clearly, the second algorithm is the best choice for large MD simulations in regular domains. However, simulations of organic materials create special problems for spatial–decomposition approaches. First, organic molecules are often simulated in a vacuum or with a surrounding solvent that does not fill a 3–D box. Thus it is not easy to divide the simulation domain so that every processor has an equal number of atoms in it. This is particularly true in a dynamically changing system. Load–imbalance is the result. Second, because of the $1/r$ dependence of Coulombic energies, long force cutoffs are often used in organic simulations. This means each processor must communicate with many surrounding processors to acquire needed information. The extra communication results in lower parallel efficiencies. For these reasons, particle–decomposition methods have been the method of choice in organic MD simulation codes that have been parallelized to date [5, 7, 12]. They have the additional advantage that the extra 2–, 3–, and 4–body forces that must be computed in organic simulations within the topology of the molecules are easily divided among the processors in a load–balanced fashion because each processor knows the positions of all atoms.

Recently, we have developed a new domain–decomposition strategy for many–body problems that is particularly appropriate for these irregular organic system simulations. Details of the method are given in [13]. It improves on the $O(N)$ scaling of the communication in a particle–decomposition method while retaining its simplicity and geometry independence. By this we mean that the decomposition ignores the topology and spatial locations of the molecules (unlike the spatial–decomposition methods), so that irregular and dynamically changing interactions in the MD model are automatically load–balanced. The algorithm is a good example of how changing the assignment of data and computations to processors can reduce communication overhead in a parallel implementation. This algorithm is also closely related to parallel techniques for performing matrix–vector multiplication.

Consider the $N \times N$ matrix representing the pairwise interactions between all atoms. Element (i, j) of the matrix is the force on atom i due to atom j. In general, for molecular problems, the (i, j) term will have a non–bonded component if atoms i and j are close to each other and a bonded component if atoms i and j are coupled together by 2–, 3–, or 4–body force. Thus the matrix is sparse (short–range forces) and skew–symmetric (Newton's 3rd law). If the atoms are randomly ordered, the sparsity pattern will be roughly uniform. Each processor is assigned a block of the matrix as in Figure 5 for the 16–processor case. We call this a force–decomposition of the workload.

Each processor also owns two pieces of the N–vectors x and f representing position and force respectively. These pieces are of length $N/\sqrt{P}$ and are the sub–pieces of x and f corresponding to the row and column of the matrix the processor occupies, as for processor 6 in the figure. The processors in a row also further sub–divide their common $N/\sqrt{P}$ atoms so that each owns N/P atoms for the purpose of updating their positions. With these assignments, the force–decomposition MD algorithm proceeds as follows, assuming each processor has current, updated atom positions at the beginning of the timestep. The processor numbers refer to the figure.

(1) Each processor computes the non–bonded (and bonded) forces in its block.

(2) The forces are summed across each row (6 communicates with 4,5,7) so that each processor acquires the forces on its N/P atoms.

(3) Each processor updates the positions of its N/P atoms.

(4) The new positions are shared within each row (6 communicates with 4,5,7).

(5) The new positions are also shared within each column (6 communicates with 2,10,12).

While this simple explanation ignores many details, there is one key point to understand. The communication steps 2, 4, and 5 involve small groups of $N/\sqrt{P}$ processors exchanging data of size $N/\sqrt{P}$. This is a direct consequence of the block–wise decomposition of the matrix (as contrasted with a row– or column–wise decomposition). Since the computational work (steps 1 and 3) is uniformly distributed among the processors it scales as N/P; thus the overall scaling of the algorithm is $N/\sqrt{P}$. This is not the optimal N/P scaling that a spatial–decomposition algorithm might achieve on best–case problems, but in practice it allows many more processors to be used effectively than does the particle–decomposition whose communication cost scales as N. And the force–decomposition approach does not suffer from the load–imbalance problems discussed above that seriously affect a spatial–decomposition algorithm's usefulness for organic system simulation.

We have tested the force–decomposition idea in an MD simulation program that computes bonded and non-bonded force fields compatible with the commercial CHARMM program. Figure 6 shows timing results for a simulation of 27 liquid crystal molecules (6750 atoms) using the force–decomposition algorithm as a function of the number of processors on the nCUBE 2 and Intel Paragon machines at Sandia. Timings on the same problem for a particle–decomposition algorithm are also shown. The particle–decomposition speed–up curve shows a sharp "roll–off" for more than 64–128 processors because of the fixed N scaling (independent of P) of the algorithm's communication. The force–decomposition method is faster by a factor of three on

1024 processors.

We have used the faster method to simulate longer timescales than were previously possible in several interesting organic and catalytic systems. Figure 7 shows the results of a 500,000 timestep MD simulation of liquid–crystal molecule conformation. A 250–atom molecule was simulated in three physical environments: by itself in vacuum, surrounded (in vacuum) by a cluster of 26 other like molecules, and in an array of 27 molecules with periodic boundary conditions. The equilibrium shape of the molecule obviously depends on its environment, indicating the need to do larger scale simulations in more realistic environments in order to deduce molecular properties.

4 Summary

We have discussed two different approaches for performing simulations on irregular geometries on massively parallel machines. For PDEs we described graph partitioning methods to minimize interprocessor communication, and for many–body problems we discussed the force–decomposition approach. Both of the methods produce domain decompositions and data assignments to processors that enable a high degree of parallelism to be exploited on distributed memory parallel machines programmed with explicit message passing of data between processors (MIMD or SPMD programming). To illustrate their performance benefits we have used these methods in simulations running on thousand–processor machines.

While the applications discussed here have been developed at Sandia, research at many other laboratories and institutions has also shown the power of the explicit message–passing model of parallel computing in a variety of simulations and parallel computations. While innovative new parallel methods and algorithms continue to be developed, we believe that the ensemble of tools that has already been developed for MIMD simulations will enable a wide variety of previously unapproachable computational problems to be tackled on the new generation of MP supercomputers. These machines include the Intel Paragon, Thinking Machines CM–5, and Cray T3D machines – all of which are capable of operating in an explicit message–passing MIMD mode. With scalable, highly–efficient algorithms in place, the remaining obstacle to high performance on these machines is to achieve near–peak single–processor flop rates on the sophisticated RISC and vector chips that are the common floating–point denominator of all the newest MP machines. Since this problem was recognized, and to a great extent solved for vector supercomputers, it is in this direction we hope the MP compiler writers and super–scalar chip designers will turn their focus.

5 Acknowledgments

We wish to thank our many colleagues at Sandia who have either carried out or collaborated with us in performing much of the research summarized here. We particularly thank Doug Cline for useful discussions concerning his work that we discussed in Section 3.1, Jeff Jortner and Ron Oldfield for their graphics efforts for several of the figures, and Soumya Patnaik and Ruth Pachter of Wright Patterson Air Force Base for collaborations in the liquid–crystal simulations described in 3.3.

References

[1] M. P. Allen and D. J. Tildesley. *Computer Simulation of Liquids*. Clarendon Press, Oxford, 1987.

[2] S. Barnard and H. Simon. A fast multilevel implementation of recursive spectral bisection for partitioning of unstructured problems. In *Proc. 6th SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, 1993.

[3] B. M. Boghosian. Computational physics on the Connection Machine. *Comp. in Phys.*, Jan/Feb, 1990.

[4] H. S. Chan and K. A. Dill. The protein folding problem. *Physics Today*, 46:24–32, 1993.

[5] T. W. Clark, J. A. McCammon, and L. R. Scott. Parallel molecular dynamics. In *Proc. 5th SIAM Conference on Parallel Processing for Scientific Computing*, pages 338–344. SIAM, 1992.

[6] D. D. Cline and J. A. Schutt. Large–scale simualtion of the three–dimensional Navier–Stokes equations. *Computer Systems in Engineering*, 3:423, 1992.

[7] S. E. DeBolt and P. A Kollman. AMBERCUBE MD, Parallelization of AMBER's molecular dynamics module for distributed–memory hypercube computers. *J. Comp. Chem.*, 14:312–329, 1993.

[8] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving Problems on Concurrent Processors: Volume 1*. Prentice Hall, Englewood Cliffs, NJ, 1988.

[9] J. L. Gustafson, R. E. Benner, M. P. Sears, and T. D. Sullivan. A radar simulation program for a 1024–processor hypercube. In *Proc. Supercomputing '89*, pages 96–105. ACM Press, New York, 1989.

[10] Bruce Hendrickson and Robert Leland. An improved spectral graph partitioning algorithm for mapping parallel computations. *SIAM J. Sci. Stat. Comput.*, 1994. To appear.

[11] B. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 29:291–307, 1970.

[12] S. L. Lin, J. Mellor-Crummey, B. M. Pettit, and G. N. Phillips Jr. Molecular dynamics on a distributed–memory multiprocessor. *J. Comp. Chem.*, 13:1022–1035, 1992.

[13] S. J. Plimpton, B. A. Hendrickson, and G. S. Heffelfinger. A new decomposition strategy for parallel bonded molecular dynamics. In *Proc. 6th SIAM Conference on Parallel Processing for Scientific Computing*, pages 178–182. SIAM, 1993.

[14] A. Pothen, H. Simon, and K. Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Matrix Anal*, 11(3):430–452, July 1990.

[15] A. C. Robinson, A. L. Ames, H. E. Fang, D. Pavlokos, C. T. Vaughan, and P. Campbell. Massively parallel computing, C++ and hydrocode algorithms. In *Proceedings of 8th Conf. on Computing in Civil Engineering*, pages 519–526. ASCE, 1992.

[16] R. Williams. Performance of dynamic load balancing algorithms for unstructured mesh calculations. *J. Par. and Dist. Comp.*, 13:257–272, 1991.

Bill Camp is the manager of the Parallel Computational Sciences Department at Sandia National Laboratories and also technical manager of its Massively Parallel Computing Research Laboratory. Aside from program development responsibilities his current research interests include statistical physics computations on parallel computers.

Steve Plimpton is a staff member in the Parallel Computational Sciences Department at Sandia. His research interests are in using parallel particle methods for materials modeling (molecular dynamics, Monte Carlo) and for simulating low–density fluid flows.

Bruce Hendrickson is a staff member in the Applied and Computational Mathematics Department at Sandia. His research interests include scientific computation, parallel algorithms and combinatorics.

Rob Leland is a staff member in the Parallel Computing Sciences Department at Sandia. His main research interests are linear algebra, iterative solution of linear and eigen systems and electromagnetics.

Table I: Summary of partioning algorithm performance on airfoil mesh.

Method	*8 Processors*		*64 Processors*	
	cuts	hops	cuts	hops
Inertial	317	396	1166	1855
Spectral Bisection	212	286	997	1661
Spectral Bisection + KL	190	261	871	1472
Spectral Octasection + KL	197	200	911	1287
Multi-Level Bisection	197	276	855	1467
Multi-Level Octasection	240	240	991	1414

Table II: Performance of partitioning algorithms on machine–part mesh.

Method	*Sandia Mesh*	
	cuts	hops
Inertial	4652	5594
Spectral Bisection	3425	5084
Spectral Bisection + KL	2562	3847
Spectral Octasection	4138	4735
Spectral Octasection + KL	3140	3420
Multi-Level Bisection	2577	3780
Multi-Level Octasection	3365	3514

Figure 1: A potion of a two-dimensional domain decomposed into sub–domains, each of which are owned by a processor. Processor i owns the center sub–domain and must periodically receive border information from the 8 surrounding processors which influence its calculations.

Figure 2: View of a well-developed Kelvin–Helmholtz instability at a shear plane between two parallel flows.

Figure 3: Partitioning of a 8034–element airfoil grid into eight sets using spectral octasection plus Kernighan Lin.

Figure 4: Partitioning of 3–D machine part mesh into eight sets using spectral octasection plus Kernighan Lin.

Figure 5: Block decomposition of the $N \times N$ force matrix. Each processor owns an $(N/\sqrt{P}) \times (N/\sqrt{P})$ sub–block.

Figure 6: Execution time on two parallel supercomputers for a MD simulation of a 6750–atom liquid crystal system using two different parallel algorithms.

Figure 7: Equilibrium conformation of a liquid crystal molecule simulated in three environments (from left to right): in a periodic array, in a cluster of like molecules, in vacuum.