A Parallel, Portable and Versatile Treecode

Michael S. Warren* John K. Salmon†‡

Abstract

Portability and versatility are important characteristics of a computer program which is meant to be generally useful. We describe how we have developed a parallel N-body treecode to meet these goals. A variety of applications to which the code can be applied are mentioned. Performance of the program is also measured on several machines. A 512 processor Intel Paragon can solve for the forces on 10 million gravitationally interacting particles to 0.5% rms accuracy in 28.6 seconds.

1 Introduction

Programs on the cutting edge of scientific computation are often aimed at the solution of a single problem on a single machine. Concepts such as portability and versatility are often left aside in the quest to "get the problem solved." Even your humble authors have been guilty of this approach. The initial attempt at a parallel treecode [7], as well as our parallel N-body algorithm which won a Gordon Bell Prize in 1992 [8], were applicable only to the gravitational N-body problem in rather restricted domains. This code was portable to several parallel machines, but the portability was at the expense of the readability of the code, which was littered with conditional compilation directives. In our early efforts to extend the code to take advantage of algorithmic improvements and newer machines, it became clear that we were attempting to renovate a code which had serious limitations, and could not handle the demands which would be placed upon it. It was time to start over. This paper describes the fundamental differences between a code that works, and a code that works while being portable and versatile.

There are many ways to achieve portability of computer software. Perhaps the easiest, and most common is reliance on "standard interfaces" between application code and OS services. Unfortunately, industry-wide standards have only recently begun to emerge in the field of parallel computing. We were not content to wait for real message passing standards to take root and be fully, reliably and efficiently implemented on the computers which we intend to use. Rather than rely on industry-wide standards, we approach portability from another direction. When writing applications we use an extremely small set of standard routines for ALL communication-related activities. These standard routines are then defined for each particular machine in a single system dependent file. This approach to portability is discussed below in section 3. Versatility depends largely on a simple, well-defined and sufficiently general interface between the parts of the code which are not physics related (i.e. data structures, load balancing, I/O) and the problem dependent modules. We describe our definition of this interface in section 2. Section 4 is devoted to a discussion of debugging techniques, and section 5 describes the performance of the code on several machines. We begin the next section with a very brief overview of fast N-body methods.

*Fluid Dynamics, Los Alamos National Laboratory, Los Alamos, NM

†Australian National University, Canberra, Australia

‡Physics Department, California Institute of Technology, Pasadena, CA

2 Overview

Several methods have been introduced which allow N-body simulations to be performed on arbitrary collections of bodies in time much less than $O(N^2)$, without imposition of a lattice [1, 2]. They have in common the use of a truncated expansion to *approximate* the contribution of many bodies with a single *interaction*. The resulting complexity is usually determined to be $O(N)$ or $O(N \log N)$, which allows computations using orders of magnitude more particles. These methods represent a system of N *bodies* in a hierarchical manner by the use of a spatial *tree* data structure. Aggregations of bodies at various levels of detail form the internal nodes of the tree (*cells*). Making a good choice of which cells to interact with and which to reject is critical to the success of these algorithms [4].

One of two tasks generally takes the majority of the time in a particle algorithm: (1.) Finding neighbor lists for short range interactions. (2.) Computing global sums for long-range interactions. These sub-problems are similar enough that a generic library interface can be constructed to handle all of the aspects of data management and parallel computation, while the physics-related tasks are relegated to a few user-defined functions. Using this generic design we have implemented a variety of modules to solve problems in galactic dynamics [6] and cosmology [13] as well as fluid-dynamical problems using smoothed particle hydrodynamics [10], a vortex particle method [5] and a panel method [11, 12]. Further applications in the areas of molecular dynamics, chemistry, electromagnetic scattering and generation of correlated and constrained random fields are under development.

Treecodes present interesting problems for parallelization because they are highly adaptive, irregular and dynamic. Our parallel treecode library infrastructure is intended to allow someone who is relatively unfamiliar with parallel programming to avoid these problems entirely, and focus on solving their own particular physics problem. A new force law can be implemented by the definition of data structures for the particles and cells, and a few functions which are described below.

The basic algorithm may be divided into several stages. Our discussion here is necessarily brief. A much more detailed description of the implementation can be found elsewhere [9, 10]. First, particles are domain decomposed into spatial groups. Second, a distributed tree data structure is constructed. In the main stage of the algorithm, this tree is traversed independently in each processor, with requests for non-local data being generated as needed. In our implementation, we assign a `Key` to each particle, which is based on Morton ordering. This maps the points in 3-dimensional space to a 1-dimensional list, which maintaining as much spatial locality as possible. The domain decomposition is obtained by splitting this list into N_p (number of processors) pieces (see Fig. 1).

In the tree building stage, one must implement `Cofm()`, which takes the particle data and creates parent cells with the required information, such a mass and center of mass and size for a gravitational calculation, or center and bounding radius for a neighbor finding calculation. In the tree traversal stage, the `Walk()` function takes as arguments a `source` tree which contains the bodies and their multipole moments that create the gravitational or electrostatic field. The `sink` tree contains the positions where the field is to be evaluated. In the language of [2], a sink is the center of a "local expansion." The tree traversal is controlled by the `Inherit()` and `Interact()` functions, which encode all of the "physics" of the application. These two functions are completely independent of the complexities of parallelism, domain decomposition and distributed tree management. The `Interact()` function has two missions: to compute "interactions" between sources and sinks, or, if that is not possible due to some "physics" constraint (e.g., accuracy, geometry, etc.) to determine whether the sink or source should be split to satisfy the appropriate constraints.

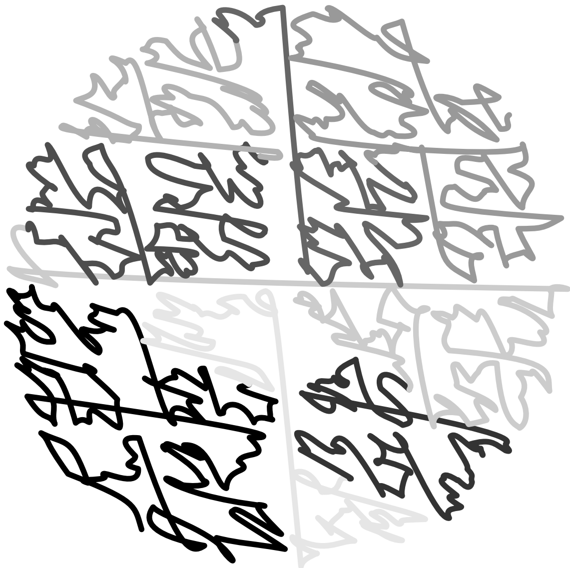

FIG. 1. *The self-similar space filling curve (Morton order) connecting the particles which is used to obtain the domain decompostion. Eight processor domains are shown in different levels of gray.*

3 Portability

Rather than rely on industry-wide standards, we approach portability from another direction. When writing applications we use an extremely small set of standard routines for ALL communication-related activities. We call these routines either "`MPMY`", or the `Salmon-Warren Message Passing Interface` (`SWAMPI`). It's no coincidence that the names sound a lot like the name of another message passing interface. While it is not true that `MPMY` is a subset of `MPI`, it is hoped that programmers familiar with `MPI` will have no trouble guessing the semantics of the `MPMY` routines. The `MPMY` routines in turn are implemented in system-specific libraries for each vendor's parallel operating system. One can immediately identify several features of this approach: (1.) It involves more programming than using industry-wide standards because it is necessary to translate `MPMY` calls into system-specific calls. (2.) There is a strong incentive to keep the `MPMY` interface as simple as possible so that the extra programming is not overwhelming. (3.) It is important to design the `MPMY` interface so that it can be implemented efficiently no matter how "brain damaged" the vendor interface. (4.) It is possible to run on new hardware even if the hardware only supports a non-standard interface. It is not necessary to wait until the vendor or a third party ports one of the standard interfaces. We have been able to implement our parallel N-body code with the use of only five primitive functions:

```
int Isend(const void *buf, int cnt, int dest, int tag, Comm_request *reqp);
int Irecv(void *buf, int cnt, int src, int tag, Comm_request *reqp);
int Test(Comm_request req, int *flag, Status *stat);
int Nproc(void); int Procnum(void);
```

There are additional functions which can be written in generic form from these primitives, but it is often more efficient to use a native system function. For instance, a `Wait` call which blocks until a message is received can be written as a loop over the `Test` function, but it is less demanding of system resources to use the native `Wait` function if it is available. Various collective communication functions also fall under this heading.

We have implemented the `MPMY` interface for the following parallel platforms: Thinking Mahines CM-5, Intel Paragon (running NX or SUNMOS), Intel Delta (using gcc or the PGI compiler), Ncube-2, and IBM SP-1. Additionally, we have implemented `MPMY` using `PVM` and `MPI`, which allows us to run on machines which support those standards (such as the Cray T3-D which uses `PVM` as its native message passing library). We have also implemented our message passing functions using UNIX User Datagram Protocol (UDP) sockets, which allows us to compute on any network of workstations supporting that communication protocol. The most important benefit of being able to run on a network of workstations (or as multiple processes on a single machine) has been the availability of state-of-the-art debugging tools during program development. Sequential machines on which `MPMY` has been tested in single processor and networked modes include Sun Sparc, SGI, DEC alpha, and HP workstations, Cray Y-MP, and even an i486 laptop running linux.

4 Debugging

It has been our experience that programs on parallel machines do not work much of the time. If one tries to develop a code without a plan to employ when things go wrong, parallel program development can stagnate. One does not have to wonder for long why parallel programming is harder than sequential programming. The basic message-passing programming paradigm we have chosen is by nature more prone to failure. Deadlock, race conditions, and other non-deterministic behavior is possible. Not to mention, the hardware technology is immature, the programming concepts are new, and the operating systems are under continual development. There are several reasons for failures: Programmer error (which can include improper constructs due to misunderstanding of a particular message passing system), incorrect or undefined compiler behavior, improper or undocumented operating system behavior, or hardware failure. Diagnosing and correcting these problems are a very important (and perhaps the most important) part of parallel program development. The problem becomes particularly acute when one is porting a program which works correctly on one system to another, since by then most "easy" errors have been filtered out, leaving one with extremely complicated and hard to find failure modes.

Debugging is notoriously difficult on parallel machines. Until very recently, vendors were still delivering machines without even an `adb` equivalent. Perhaps one day powerful parallel debuggers will be standard equipment on all machines. Until then, one must resort to the pedestrian approach of littering one's code with print statements. This is far harder in parallel than in serial environments, because one must tag or redirect every `printf` in order to know which particular processor it came from. A second, more serious, problem is that the addition of the print statements can change the behavior of the program, possibly making the problem you are looking for disappear. In addition, a code full of `#ifdef VERBOSE` statements (which are necessary to recover any kind or performance when the debugging is finished) is an eyesore, and doesn't really offer enough control over which messages are printed.

We have implemented a `Msg` library to avoid the pitfalls mentioned above. The `Msg` function allows us to print debugging and status information to individual files, and to control which information is provided via runtime arguments. We have found this mechanism far preferable to the *ad hoc* addition of `printf` statements to the code when trying to track down a problem. In another mode, each processor stores the messages in a circular memory buffer, which is emitted in response to an external stimulus (for instance, a message on a socket). This makes reading the messages somewhat tricky, but writing the messages is kept completely inside a processor, thereby minimizing the possibility that external interactions or synchronizations will alter program behavior. This can be useful when the very act of delivering a message to the I/O device alters the timing sufficiently to bypass the bug. In addition, it can sometimes provide information about what happened shortly before some strange event (such as the machine being silent when output is expected).

5 Performance

In Table 1, we show benchmark results for several representative machines and problem sizes. The benchmark problem is N particles of mass $1/N$ distributed randomly in a sphere of radius 1. The error bound for each partial interaction is set to 1% of M/R^2, which is .01 in this case. To confirm the accuracy of the method, the potential and force were calculated for a dataset with 100,000 particles using an exact N^2 algorithm, and these values were used to calculate rms and maximum force errors, which were .00477 and .0213 respectively.

machine	*N particles*	*N processors*	*time* (sec)
Cray Y-MP	100,000	1	153.7
Paragon	100,000	1	151.1
Sparcstation/10	100,000	1	64.1
CM-5	100,000	32	10.4
Paragon	100,000	32	7.3
CM-5	2,000,000	64	74.6
Paragon	2,000,000	64	47.7
SP-1	2,000,000	64	20.5
Paragon	2,000,000	256	12.9
CM-5E	2,000,000	256	10.8
Paragon	10,000,000	512	28.6

TABLE 1

A tabular comparison of several parallel and sequential machines for the canonical test problem.

A few details related to optimization should be mentioned. For the Paragon code, the $1/\sqrt{r^2}$ operation was coded in assembly language using a Newton-Raphson iteration. For the SP-1 code, the same operation was coded in `C` using a Chebychev polynomial approximation and one Newton iteration (see [3] for details). On the CM-5, no attempt was made to use the vector units. The YMP performance could possibly be improved a great deal by tuning of a few critical functions where vectorization was inhibited.

6 Conclusion

Parallel computers offer an effective way to apply large amounts of computational resources to simulation problems. We have shown that treecodes can be successfully implemented on a variety of parallel machines and make efficient use of several hundred processors. By careful implementation of a generic "tree" library and a simple message passing interface we have been able to meet the twin goals of versatility and portability.

Acknowledgments

We would like to thank David Edelsohn for providing the benchmark results on the SP-1, and IBM Watson Research center for providing time on that machine. The authors wish to acknowledge the Advanced Computing Laboratory of Los Alamos National Laboratory, Los Alamos, NM 87545; this work was performed in part on computing resources located at that facility. Time on the CM-5E was provided by the Naval Research Laboratory. This research was performed in part using the

Intel Paragon operated by Caltech on behalf of the Concurrent Supercomputing Consortium. This research was supported by a grant from NASA under the HPCC program.

References

[1] J. Barnes and P. Hut, *A hierarchical O(NlogN) force-calculation algorithm*, Nature, 324 (1986), p. 446.

[2] L. Greengard and V. Rokhlin, *A fast algorithm for particle simulations*, J. Comp. Phys., 73 (1987), pp. 325–348.

[3] A. H. Karp, *Speeding up N-body calculations on machines without hardware square root*. (preprint), 1992.

[4] J. K. Salmon and M. S. Warren, *Skeletons from the treecode closet*, J. Comp. Phys., 111 (1994), pp. 136–155.

[5] J. K. Salmon, M. S. Warren, and G. S. Winckelmans, *Fast parallel treecodes for gravitational and fluid dynamical N-body problems*, Intl. J. Supercomputer Appl., 8 (1994), pp. 129–142.

[6] M. S. Warren, P. J. Quinn, J. K. Salmon, and W. H. Zurek, *Dark halos formed via dissipationless collapse: I. Shapes and alignment of angular momentum*, Astrophysical Journal, 399 (1992), pp. 405–425.

[7] M. S. Warren and J. K. Salmon, *An* $O(N \log N)$ *hypercube N-body integrator*, in Proceedings of the Third Conference on Hypercube Computers and Applications, G. C. Fox, ed., New York, 1988, ACM Press, p. 971.

[8] ——, *Astrophysical N-body simulations using hierarchical tree data structures*, in Supercomputing '92, Los Alamitos, 1992, IEEE Comp. Soc., pp. 570–576.

[9] ——, *A parallel hashed oct-tree N-body algorithm*, in Supercomputing '93, Los Alamitos, 1993, IEEE Comp. Soc., pp. 12–21.

[10] ——, *A portable parallel particle program*, Computer Physics Communications, (1994). (in press).

[11] G. S. Winckelmans, J. K. Salmon, and M. S. Warren, *The fast solution of three-dimensional boundary integral equations in potential flow aerodynamics using parallel and sequential tree codes*, (1994). (in preparation).

[12] G. S. Winckelmans, J. K. Salmon, M. S. Warren, and A. Leonard, *The fast solution of three-dimensional fluid dynamical N-body problems using parallel tree codes: vortex element method and boundary element method*, in Seventh SIAM Conference on Parallel Processing for Scientific Computing, SIAM, 1995.

[13] W. H. Zurek, P. J. Quinn, J. K. Salmon, and M. S. Warren, *Large scale structure after COBE: Peculiar velocities and correlations of cold dark matter halos*, Astrophysical Journal, (1994).