On the Utility of Threads for Data Parallel Programming*

Thomas Fahringer† Matthew Haines‡ Piyush Mehrotra‡

†Institute for Software Technology and Parallel Systems
University of Vienna
Liechtensteinstrasse 22, A-1092, Vienna, Austria

‡ICASE
NASA Langley Research Center, Mail Stop 132C
Hampton, VA 23681-0001

Abstract

Threads provide a useful programming model for asynchronous behavior because of their ability to encapsulate units of work that can then be scheduled for execution at runtime, based on the dynamic state of a system. Recently, the threaded model has been applied to the domain of data parallel scientific codes, and initial reports indicate that the threaded model can produce performance gains over non-threaded approaches, primarily through the use of overlapping useful computation with communication latency. However, overlapping computation with communication is possible without the benefit of threads if the communication system supports asynchronous primitives, and this comparison has not been made in previous papers. This paper provides a critical look at the utility of lightweight threads as applied to data parallel scientific programming.

*Research supported by the National Aeronautics and Space Administration under NASA Contract No. NASA-19480, while the authors were in residence at ICASE, NASA Langley Research Center, Hampton, VA 23681.

1 Introduction

Threads provide a useful programming model for asynchronous behavior because of their ability to encapsulate units of work that can then be scheduled for execution at runtime, based on the dynamic state of a system. For example, the threaded model is used in the event-driven world of network protocols, for client-server windowing applications, and for runtime systems implementing *fork* semantics in task parallel programming languages [4, 11, 1]. The utility of threads in these domains is in simplifying the complexities of asynchronous programming, and is well documented.

Recently, the threaded model has been applied to the domain of data parallel scientific codes [2, 6]. These initial reports indicate that the threaded model can produce performance gains over non-threaded approaches, primarily through the use of overlapping computation with communication latency. However, overlapping computation with communication is possible without the benefit of threads if the communication system supports asynchronous primitives [8], and this comparison has not been made in previous papers. Overlapping computations with communication is asynchronous programming, so the potential for threads to simplify this approach may be valid. To what extent, then, are threads useful in the realm of data parallel scientific programming?

This paper provides a critical look at the utility of lightweight threads as applied to data parallel scientific programming. We employ a lightweight thread package for distributed memory multiprocessors, called Chant [5], to encode 2 data parallel scientific applications: a simple Jacobi program written in Fortran and a particle-in-cell (PIC) simulation program written in C. We compare the threaded performance of these applications with the non-threaded performance, and discuss the implementation and "ease-of-programming" issues that arise. Our initial study indicates that employing the threaded model in this domain raises several significant programming challenges, and the performance gained by overlapping computations with communications is often either negated by the increased overhead of the threaded system or can be matched using non-threaded, asynchronous programming techniques. However, there are indications that, as with the other asynchronous programming domains, threads simplify the task of overlapping communications with computations and provide simultaneous access to other benefits, such as load balancing capabilities.

In Section 2 we provide background information on the threaded programming model and its application to data parallel programming. Section 3 then provides the details of the threaded implementations of our applications, including performance results and analysis, and we conclude with remarks on the potential benefits of threads for data parallel programming in Section 4.

2 Lightweight Threads

A *thread*, as commonly defined, is an independent, sequential unit of computation that executes within the context of a kernel-supported entity, such as a Unix process. Threads are often classified by their "weight", which corresponds to the amount of context that must be saved when a thread is removed from the processor, and restored when a thread is reinstated on a processor (i.e. a *context switch*). Operating processes and threads are typically referred to as *heavyweight* or *middleweight* because of their large context and the need to cross the kernel interface for all operations. By exposing all context and thread operations at the user-level, a minimal context for a particular application can be defined, and operations to manipulate threads may avoid crossing the kernel boundary. As a result, user-level (*lightweight*) threads can be switched in the order of

tens of microseconds, which is at least an order of magnitude better than current operating system processes and threads.

Threads are typically used for representing asynchronous computations within a single process, since the scheduling is dynamic and can be influenced by the runtime conditions of a system. Thus, threads are most useful when the scheduling of independent tasks is dependent on runtime conditions, such as event-driven applications. For data parallel programming, threads are most commonly used in two ways:

1. For latency tolerance. Accessing distant memories in a parallel computer (whether by message passing or direct addressing) is typically a long-latency operation with respect to accessing local memory. Often, there is an order of magnitude separating the access times for local and remote memories. Therefore, it is highly desirable to either avoid or hide the latency of remote memory references. The former is often accomplished by using intelligent partitioning and caching techniques that eliminate the need to access a remote memory location. The latter is accomplished by overlapping useful computation with communication, thereby "hiding" the latency.

   A threaded system overlaps computation with communication by creating multiple threads on each processor (the actual number depends on the relative speeds of the underlying thread and communication systems and on the remote access pattern of the application) and allows a thread to run until a remote memory reference has been initiated, at which time it switches to another ready thread rather than wait idle for the communication request to complete.

2. For resource management. For data parallel programming, the parallelism is obtained by distributing the program data over a set of processors (memories), where each processor can operate on its portion of the data independent of the others. It is often convenient to write data parallel programs under the assumption that there are a large (or infinite) number of "virtual" processors available, so that the decision over distribution is controlled only by the algorithmic constraints. However, this requires that the virtual processors be mapped onto the physical processors at some point.

   A threaded system can be used to implement virtual processors (VPs) [9], where each thread implements a single VP. However, this is not a natural role for threads to play, as this requires that each thread maintain its own address space, something typically not supported by underlying thread packages. As a result, the programmer is forced to manually separate global data structures into separate regions for each thread, which is difficult to do for even small programs.

For this paper we focus on the use of threads in data parallel programs, mainly for overlapping computation with communication. It is also possible, however, to employ threads for other purposes, such as task parallelism and load balancing, but these issues are beyond the scope of this paper.

2.1 Chant

In this section we outline the distributed threads pacakge, Chant, that is used for our experiments. Though a detailed examination of distributed threads is beyond the scope of this paper, we will

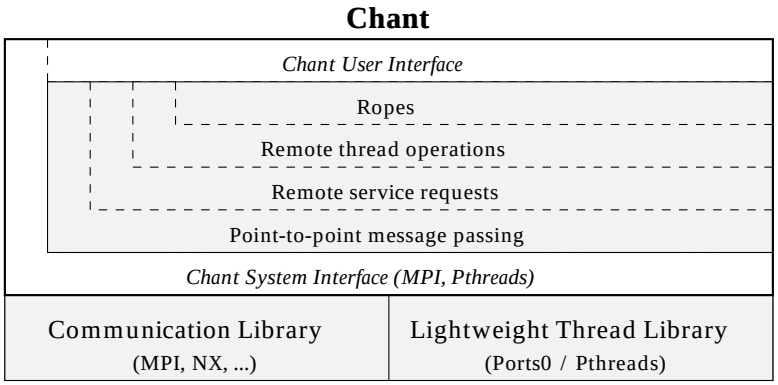

Figure 1: Chant runtime layers and interfaces

nonetheless provide an introduction to Chant. Please refer to [5] for a more detailed description of Chant and the issues faced in supporting distributed threads.

The POSIX committee has recently established a standard for the interface and functionality of lightweight threads within an operating system process, called *pthreads* [7]. Since threads are defined within the context of a process, they share a single address space, and communication among threads is only defined in terms of shared memory primitives, such as events and locks. Thus, the interaction of pthreads in a distributed environment is undefined. Likewise, the Message Passing Interface Forum (MPI) has recently established a standard for communication between processes [3]. Although various extensions to the standard have already been proposed [13, 14], communication between lightweight threads within processes has yet to be supported by MPI. Therefore, Chant was designed to provide a simple mechanism for combining lightweight threads with interprocessor communication.

Chant is designed as a layered system (as shown in Figure 1), where efficient point-to-point communication provides the basis for implementing remote service requests and, in turn, remote thread operations. Chant relies on a *system interface* to achieve a high degree of portability, where the underlying thread and communication systems are ports0 (a subset of pthreads; see [12]) and MPI, respectively.

Next, Chant supports *point-to-point communication* (i.e., send/recv) between any two threads in the system by utilizing the underlying message passing system (MPI). Issues to be addressed at this level include naming global threads in the system, avoiding intermediate copies for message buffers, and efficient polling for outstanding messages. Chant uses the concept of a *context* to represent an addressing space within a processor, where contexts represent a linear ordering of processes in the system as maintained by the underlying communication system (e.g., MPI uses `rank` in `MPI_COMM_WORLD`). Global threads within Chant are therefore identified using the doublet `<context_id,thread_id>`.

Atop efficient point-to-point message passing, Chant supports *remote service requests* by instantiating, in each context, a service thread which is responsible for handling all incoming remote service requests (asynchronous messages) and delivering any necessary replies. Using the remote service request mechanism, Chant can easily support *remote thread operations*, such as remote thread create, by invoking the specified thread request on the desired processor and, possibly, by adding some software "glue" to make it work.

Next, Chant supports collective operations among thread groups using a scoping mechanism called *ropes*. Ropes allow a user to specify a collection of threads that will participate in a global, collective operation such as a broadcast or barrier. Ropes also provide an alternate naming scheme that allows all threads within the rope to be addressed using their relative index within the rope.

Finally, Chant provides a *user interface* that is an extension of the *pthreads* standard, where access to each of the underlying layers can be made directly or indirectly. Thus it is still possible to access the underlying MPI or pthreads interfaces from within a Chant thread.

3 Experiments

In this section we discuss two experiments used to evaluate the benefits of data parallel programming using threads. The first experiment analyzes the Jacobi relaxation iterative method. The second experiment deals with a particle-in-cell code, which is an irregular code. The threaded versions are then compared against a conventional non-threaded programs employing communication/computation overlapping techniques based on asynchronous message passing primitives.

3.1 Jacobi

The Jacobi relaxation iterative method is used to approximate the solution of a partial differential equation discretized on a grid. For this experiment we consider only the main Jacobi kernel routine as shown below:

```
      DO 10 Q=1,ITER
        ...
L:      DO 20 J=2,N-1
          DO 20 I=2,N-1
            UHELP(I,J)=(1-OMEGA)*U(I,J)+OMEGA*0.25*
     *        (F(I,J)+U(I-1,J)+U(I+1,J)+U(I,J+1)+U(I,J-1))
20      CONTINUE
        ...
10    CONTINUE
```

Note that this code represents a regular Jacobi implementation, which means that each grid element operation requires the same computational effort. Furthermore, for all experiments the Jacobi kernel is placed within an outer loop over *ITER* iterations in order to obtain reasonably large runtimes.

In order to evaluate the advantage of using threads for this kernel we manually encode three different data parallel versions using a column-wise distribution of the arrays *UHELP*, U, and F on a Intel Paragon machine with 128 nodes:

- A *blocking version* which exchanges all data required to do local computations outside of loops incorporating blocking receive operations.
- A *hand overlapped version* which statically overlaps computation with communication as may be done by a data parallel compiler (c.f. Kali compiler [8]). First, all send operations are done. Second, all the local loop iterations which do not require non-local data are processed. Third, the corresponding receive operations are executed in blocking mode. Finally, all loop iterations requiring non-local data are processed.

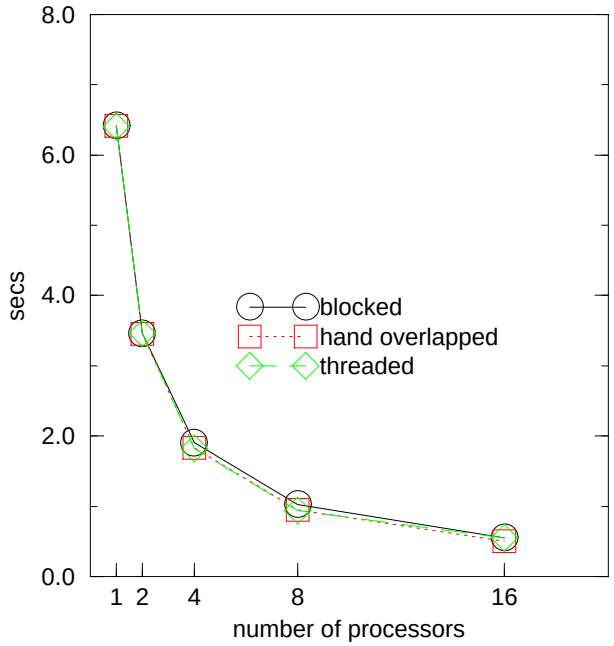

Figure 2: Measured runtime for parallel regular Jacobi program versions with N=512, ITER=20, B=1

- A *virtual threaded* version that creates three different threads for each processor: two boundary threads responsible for communicating with the other boundary threads and performing the boundary iterations, and a computation thread responsible for performing all local iterations. Each processor gets $\frac{N-2}{|P|}$ (where $|P|$ divides $N-2$) iterations of the J loop assigned. P is the set of processors employed to execute the virtual threaded version. Each boundary thread will perform B iterations, where $2 * B \leq \frac{N-2}{|P|}$. The computation thread will perform $\frac{N-2}{|P|} - 2 * B$ iterations. By changing B we can control the work distribution among boundary and computation threads within a specific processor. For our analysis both boundary threads execute the same number (B) of threads. An experiment (see Figure 6) will be shown which shows the effect of varying B. Every local processor of P respectively creates and terminates its boundary and computation threads at the beginning and at the end of the program, and the threads are free to execute simultaneously, as there is no dependence among them. Threads within the same processor access shared memory in order to prevent intra-processor communication with messages.

Figure 2 shows the runtimes for all three program versions based on a regular Jacobi implementation. We observe that the hand-overlapped and threaded versions are slightly better than the blocked version due to their ability to overlap computation and communication, but that the difference is nearly negligible. This is because the balanced computation and communication behavior of a regular Jacobi version produces very even communication patterns in which all processors exchange about the same amount of data at about the same time, and processors rarely wait long for messages to be received. Also, since communication is rather small as compared with computation (problems of size N yield an $O(N^2)$ increase in computation as compared with an $O(N)$ increase of communication), effects of reducing communication costs are not dramatic. Finally, all processors are responsible for the same amount of computation. Therefore, both computation and

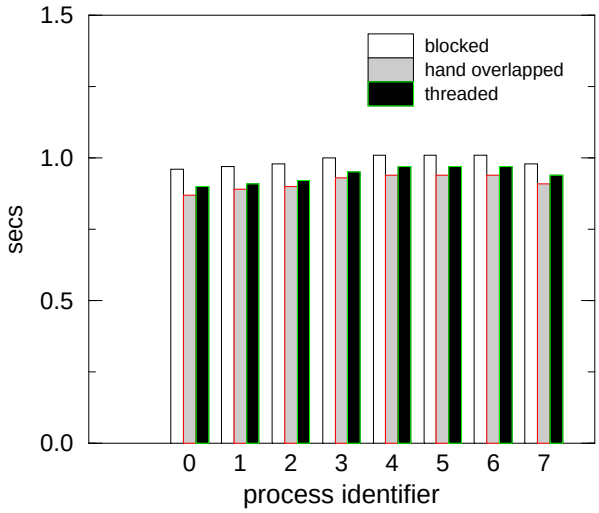

Figure 3: Workload of processors for parallel regular Jacobi program versions

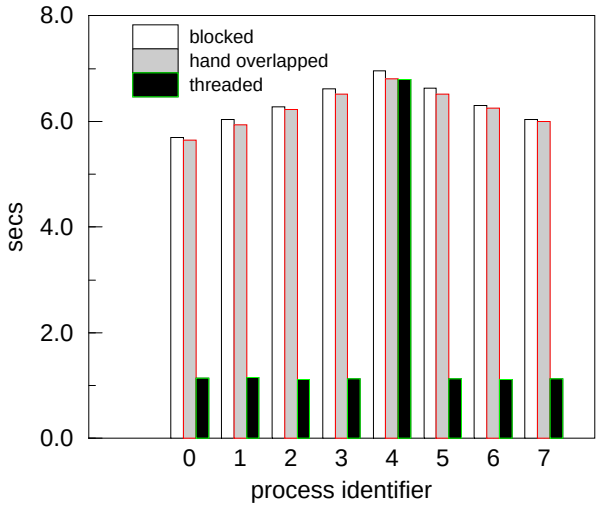

Figure 4: Workload of processors for parallel irregular Jacobi program versions

communication phases occur in concert. This explains a nearly linear speedup for smaller number of processors.

Figure 3 displays the even workload across all processors of the three program versions analyzed. The threaded version is slightly worse than the hand overlapped version due to its context switch overhead, and the blocking version has the largest runtime due to the small waiting time induced by blocking receive operations.

Figure 5 shows the runtimes for all three Jacobi versions, however this time we employ an irregular workload. In this version, we put a different load on a specific processor (in this case processor 4). This conforms to some realistic situations in which different relaxation algorithms are applied depending on the location of each grid point. The runtime of the irregular code significantly increases due to the imbalanced load, and there is still no significant difference among the overall runtime of the three program versions. However, there is an important difference in the *idle* time

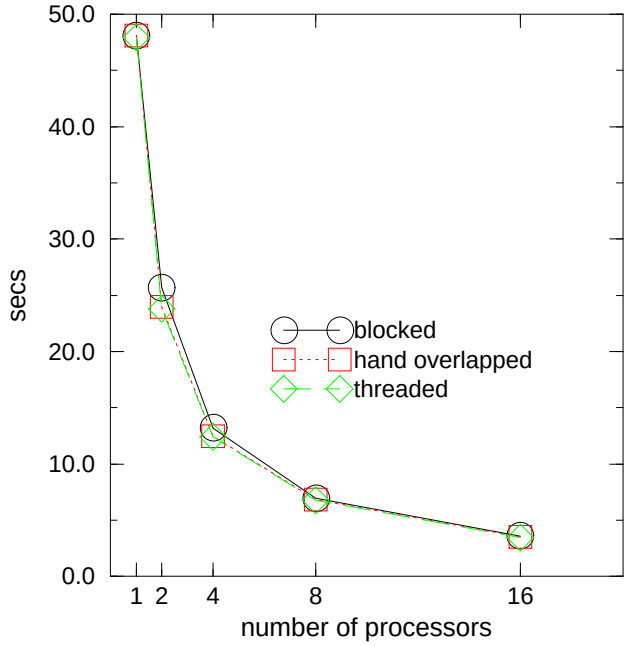

Figure 5: Measured runtime for parallel irregular Jacobi program versions with N=512, ITER=20, B=1

of the individual processors for the three methods.

Figure 4 depicts the runtime for each processor of an 8-processor Jacobi execution. It can be clearly seen that processor 4 (due to the increased workload) dominates the runtime of the entire program, and that all three program versions have a very similar runtime with respect to processor 4 since each iteration is synchronized. However, the interesting aspect of this experiment is that the threaded version implies a much smaller runtime for all other processors. The blocking version requires each processor to wait for some data at the beginning of each kernel iteration. The hand-overlapped version at one point of the execution also blocks for non-local data to be received. In both of these versions all processors have to wait at a specific kernel iteration until processor 4 finishes the previous iteration and sends its corresponding data. In contrast, the computation thread of the threaded version never has to wait for any of its boundary threads to complete. Communication and blocking time of the boundary threads is always overlapped by useful work to be done by the computation thread. Only the boundary threads between neighboring processors depend on each other. However, the computation thread of processor 4, which is responsible for the majority of work, does not considerably slow down its boundary threads. Since the Paragon thread scheduling strategy assigns equal time slots to each thread in a round-robin fashion, once the boundary threads of processor 4 are done, all other processors but 4 are finished too.

Finally, we want to investigate the effect of varying B, the number of iterations assigned to the boundary threads. Note that the overall number of iterations assigned to the threads of a processor (1 computation and 2 boundary threads) is fixed. By changing B, we control the work distribution among the threads for each processor. Figure 6 displays the runtime of three different threaded regular Jacobi versions with varying values for N, $ITER$, and B. This experiment clearly shows that modifying B does not change the overall runtime for any threaded regular Jacobi version.

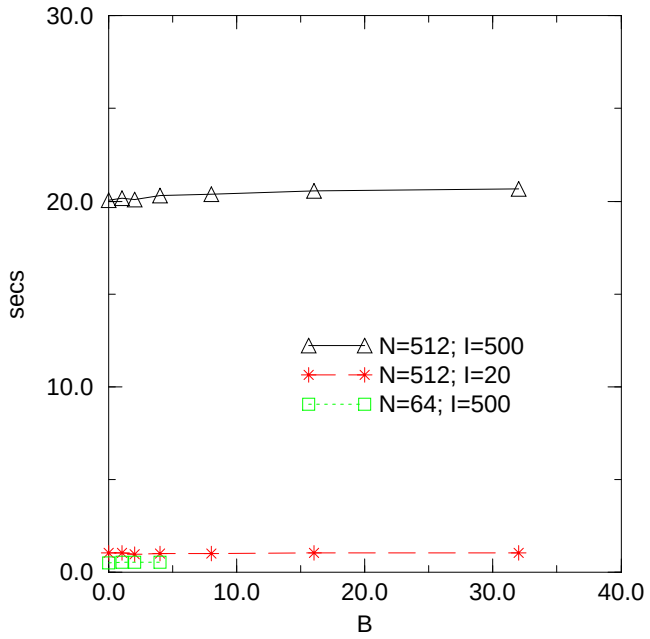

Figure 6: Threaded regular Jacobi version with 8 processors, 2 boundary and 1 computation thread for various values of B

As we have noted above, the regular Jacobi version is highly balanced such that it seems to be irrelevant how the loop iterations are distributed across the threads within one processor.

3.2 PIC

The Particle-in-Cell code (PIC) determines the motion of a group of interacting particles starting with some initial configuration of positions and velocities in a specified volume of space. The standard PIC version is an outer loop over time, with an inner loop alternating between two computations:

1. update of positions and velocities from the dynamic equations (particle push phase)
2. compute the solution of electro-magnetic partial differential (field solution phase)

Lubeck and Faber [10] described a parallel implementation of PIC, where both the spatial grid and the set of particles are regularly decomposed onto a set of processors. Each processor k keeps track of three different groups of particles:

- Particles which are owned by k and reside in its own region of the spatial grid,
- Particles which are owned by k and reside in some other processor's region of the spatial grid, and
- Particles which reside in its region of the spatial grid and are owned by some other processor.

By keeping track of the particles in this manner, it is possible to reduce the volume of communication needed to locate particles at each time step, but only at the extra expense of maintaining these groupings.

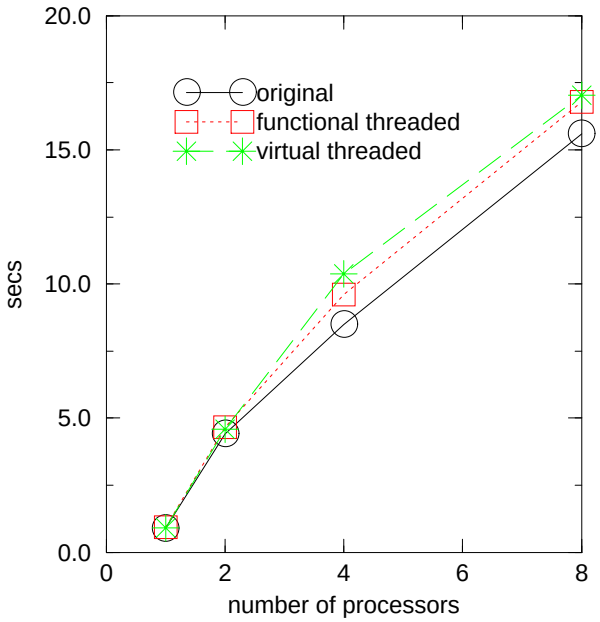

Figure 7: Measured runtime for parallel PIC program with varying number of processors

For the purposes of our experiment, we evaluate three different PIC versions written in C on a network of Sun-10 workstations:

1. The *original message passing version*, derived from a code given to us by the University of Colorado. This version has manually-overlapped computation and communication phases, and the communication phases are highly sequential based on a single communication buffer that each processor maintains. During the particle push phase, each processor sends and receives a set of particles to other processors, and these variable-sized messages are read into a single, common buffer on each processor. Since the messages are variable-sized, and a single buffer is being used, the communication cannot be parallelized.

2. The *functional threaded version*, which splits each PIC phase into three threads, corresponding to sending, computing, and receiving. The PIC phases are executed sequential but the threads within a phase are executed simultaneously. The threads are created and terminated outside of the outer (time) loop and synchronized via mutex variables inside of the loop in order to prevent excessive thread creation and termination overhead.

3. The *virtual threaded version*, which is identical to the original version except that a naive partitioning scheme is employed to further sub-divide the spatial grid on each processor into a number of virtual processors (threads).

Figure 7 plots the measured runtimes for three different PIC versions, as described above, with 1024 particles and for varying number of processors. Each processor corresponds to a Sun-10 workstation in a workstation cluster. It can be clearly seen that the original version implies a slowdown in performance due to the fact that the number of send/receive operations (cf. Figure 8) for the original implementation is increasing linearly with the number of processors incorporated.

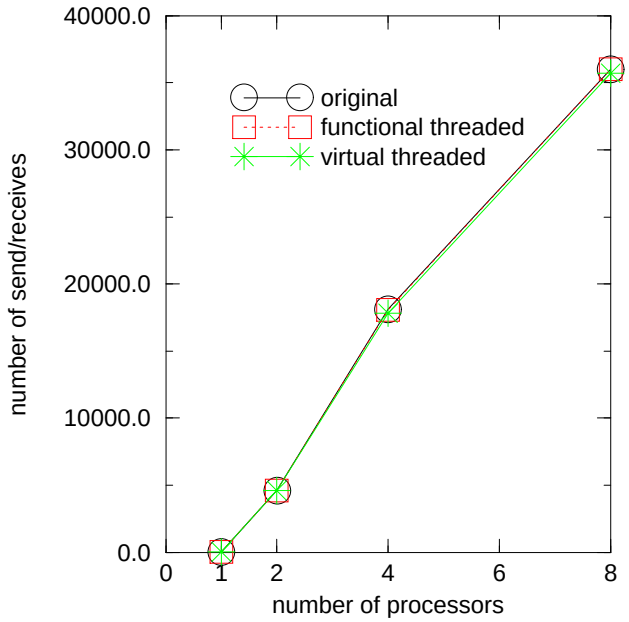

Figure 8: Number of send/receive operations for parallel PIC program with varying number of processors

Furthermore, the bookkeeping effort to maintain the particle groupings is significant for larger number of processors. Three tables, one for each class of particles as described above, need to be updated and organized. Additionally, the particles may cluster in only a few regions, implying a load imbalance of particles, consequently increasing both communication and bookkeeping efforts.

Figure 7 also shows that the threaded PIC versions slightly decrease the performance as compared to the original version. This is due to the fact that threads only improve execution speed if they can overlap computation and communication beyond what is being done in the non-threaded version. Since our original PIC code already overlapped computation with communication, the threaded approach provided only overhead. The functional threaded version is slightly better than the virtual threaded program because the former version is able to exploit both data and functional parallelism, while the later version is restricted to data parallelism only.

In summary, we note that the original version manually overlapped computation with communication phases, thus exploiting the main performance benefit of threads. Using threads, however, offers advantages with respect to ease of synchronization. In the original version computation and communication phases have to be synchronized manually, while the threaded versions only required the definition of a set of threads, and the thread-scheduler automatically schedules those threads ready to execute based on dynamic conditions. The underlying system automatically takes care of scheduling the threads such that blocking time is overlapped with computation. On the other hand, it is not trivial to detect functional parallelism for the functional threaded version, in particular in the presence of a highly complex C program with many side-effects. We encountered two principal problems for the virtual threaded version: First, it is necessary to manually separate the data of a single processor into thread private and thread global data, where thread global data is shared among all threads which reside in the same processor. Second, communication among threads on the same processor could be done via message passing or shared memory access (thread global

data). In the first case, the underlying system automatically takes care of the communication at the cost of additional message passing layer overhead. This overhead depends on whether the underlying thread system recognizes that a communication occurs between two threads within the same processor or the message is passed to the message passing layer (e.g. MPI layer), which in the worst case might even try to send the message to the network. In the second case, shared memory synchronization is inevitable.

4 Conclusion

The potential for lightweight threads to simplify asynchronous programming is realized in many applications, such as event-driven simulations and client-server applications. The extend to which they are useful in the realm of data parallel scientific programming is, however, still debated.

In this paper we illustrate the use of lightweight threads for two applications: a Jacobi relaxation code and a particle-in-cell (PIC) code. Based on our experiments, we can make several observations. First, adding lightweight threads to a data parallel application must be done with care so as not to disrupt the communication volume. As was demonstrated with the PIC code, adding threads can sometimes increase the overall communication volume, degrading the performance of an application. Second, employing lightweight threads for overlapping computations and communication is sometimes not possible, and sometimes not necessary. Again referring the PIC code, if the communication phase is written to be sequential, then employing multiple threads yields no benefit since the threads will be serialized by the sequential communication operations. Also, if the overlap is trivial (such as in Jacobi), then it is possible to encode the asynchronous communication without threads. However, as we see in the Jacobi experiment, threads free resources that are otherwise busy, providing the potential for load balancing to improve the execution or for running threads from another job. Third, implementing "virtual processors" using threads is possible but not well-supported, since threads assume a single addressing space within the same process, while virtual processors require separate address spaces. If this type of addressing is not supported by the underlying threads package (as is often the case), then the programmer must provide the support, which is difficult and slow.

In summary, we observe that lightweight threads do have utility in the domain of data parallel programming, but not to the extent as reported in previous papers when hand-overlapped communication is factored in. Also, the utility of threads is not necessarily in the increased performance of an application, but in the simplification of asynchronous programming and the ability to release idle resources for other work. While we do not attempt to provide the final word on the utility of lightweight threads for data parallel programming, we do hope to add fuel to the fire for this ongoing debate.

References

[1] Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.

[2] Edward W. Felton and Dylan McNamee. Improving the performance of message-passing applications by multithreading. In *Proceedings of the Scalable High Performance Computing Conference*, pages 84–89, April 1992.

[3] Message Passing Interface Forum. *Document for a Standard Message Passing Interface*, draft edition, November 1993.

[4] I. T. Foster and K. M. Chandy. Fortran M: A language for modular parallel programming. Technical Report MCS-P327-0992 Revision 1, Mathematics and Computer Science Division, Argonne National Laboratory, June 1993.

[5] Matthew Haines, David Cronk, and Piyush Mehrotra. On the design of Chant: A talking threads package. In *Proceedings of Supercomputing 94*, pages 350–359, Washington, D.C., November 1994. Also appears as ICASE Technical Report 94-25.

[6] J. Holm, A. Lain, and P. Banerjee. Compilation of scientific programs into multithreaded and message driven computation. In *Proceedings of the Scalable High Performance Computing Conference*, pages 518–525, Knoxville, TN, May 1994.

[7] IEEE. *Threads Extension for Portable Operating Systems (Draft 7)*, February 1992.

[8] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.

[9] Ravi Konuru, Jeremy Casas, Robert Prouty, Steve Otto, and Jonathan Walpole. A user-level process package for PVM. In *Proceedings of Scalable High Performance Computing Conference*, 1994.

[10] O.M. Lubeck and V. Faber. Modeling the performance of hypercubes: A case study using the particle-in-cell application. *Parallel Computing*, 9(1):37–52, December 1988.

[11] Piyush Mehrotra and Matthew Haines. An overview of the Opus language and runtime system. In *Proceedings of the 7th Annual Workshop on Languages and Compilers for Parallel Computers*, New York, November 1994. Also Appears as ICASE Technical Report 94-39.

[12] Portable runtime systems (ports) consortium. http://www.cs.uoregon.edu:80/paracomp/ports/.

[13] Anthony Skjellum, Nathan E. Doss, and Kishore Viswanathan. Inter-communicator extensions to MPI in the MPIX (MPI eXtension) library. Technical report, Computer Science Department and NSF Engineering Research Center, Mississippi State University, July 1994. Submitted to ICAE Journal Special Issue on Distributed Computing.

[14] Anthony Skjellum, Nathan E. Doss, Kishore Viswanathan, Aswini Chowdappa, and Purushotham V. Bangalore. Extending the message passing interface (MPI). Technical report, Computer Science Department and NSF Engineering Research Center, Mississippi State University, 1994.