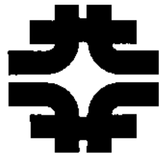

FERMILAB-Conf-93/076

A Multi-thread Approach to Coalescing Small Transfers Generated by Portable Tool Sets

Mark Fischler and Mike Uchima

Fermi National Accelerator Laboratory
P.O. Box 500, Batavia, Illinois 60510

April 1993

Submitted for presentation at *Supercomputing '93*,
Portland, Oregon, November 15-19, 1993

Operated by Universities Research Association Inc. under Contract No. DE-AC02-76CHO3000 with the United States Department of Energy

Disclaimer

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

A Multi-thread Approach to Coalescing Small Transfers Generated by Portable Tool Sets*

Mark Fischler† and Mike Uchima††

Fermilab**, Batavia, IL 60510

ABSTRACT

An approach to providing scientific users with the means to code complex algorithms in a natural way is to provide portable tool sets implementing the concepts of each scientific field. These tool sets facilitate implementation on massively parallel systems, but often tend toward large numbers of small inter-CPU data transfers, adversely impacting efficiency. We present techniques for mitigating this effect without affecting the way the user sees the tools. Efficiency improvements obtained by transfer coalescing via "multi-threading" are studied in the context of Lattice Gauge Theory computations.

---

* Paper submitted for presentation at Supercomputing '93, as a TECHNICAL PAPER.

† M. Fischler (presenting author): mf@fnal.fnal.gov (708)840-4339 FAX (708)840-8208

†† M. Uchima: uchima@fnal.fnal.gov (708)840-2671 FAX (708)840-8208

** Fermilab is operated by Universities Research Association, Inc. under contract with the U.S. Department of Energy

1. Introduction

Much scientific work using massively parallel computers has traditionally been done by scientists who have become expert in the techniques of efficiently utilizing each particular system [1]. This situation was appropriate when supercomputers were expensive and rare — the CPU time was a very critical resource. However, massively parallel systems are becoming more accessible; we are entering a realm where the scientists' time, effort, and creativity are the key to advances in many fields. Hardware power is improving tremendously, but there are areas where the largest gains will be achieved by improvements in algorithms.

Tools can be provided to help the scientist who is not expert in parallel programming for a particular system [2]. An important class of such tools leads naturally to implementations which are fairly efficient except for one effect: The pattern of interprocessor communication becomes dominated by small transfers. In most of today's systems, frequent transfers force unacceptably low efficiencies, invalidating this promising approach.

Fortunately, improvements to the implementation of these tools can mitigate this problem. The idea is to coalesce multiple transfers between processors, without affecting the way the user sees the transfers. When an algorithm has a greater degree of logical parallelism than that of the actual system, a useful coalescing scheme is that of "multi-threading". The benefits of these improvements depend on the nature of the system and the algorithms; we present representative studies based on calculations necessary for Lattice Gauge Theory [3,4].

1.1 Tool Sets and Small Transfer Lengths

To facilitate exploration of complicated algorithms, it is useful to separate the creative effort of algorithm development, from the effort of actually coding up the desired algorithms. The scientist should be able to code algorithms for massively parallel systems, without becoming expert in any particular system.

The scientist can code complex algorithms in a natural, portable way if provided with a concept oriented tool set. This implements an appropriate set of the concepts that the

scientist is familiar with, and allows programs based on these concepts to run on any massively parallel system supporting that tool set. Any particular tool set will not be useful for every application — the set of concepts would be be too broad to implement efficiently. But a well-chosen set of concepts can be applicable for a wide spectrum of algorithms. For example, for problems involving discretizing differential equations on a grid, some appropriate geometrical concepts would be those of the *site*, of *field* data associated with each site, of *directions* and *neighboring sites* in each direction. An execution concept is that of the *task*, to be performed over each site in some set, without regard to ordering.

Such a tool set should shield the user from details about the system — distribution of data and work, access of data residing on non-local processors, and so forth. The system need not perform automatic parallelization; the user — guided by the set of available concepts — supplies knowledge about the parallel nature of the algorithm. This knowledge is so instinctive to the scientist that its value is easy to overlook; but it provides the system with a much better view of the nature of the algorithm than could be derived automatically. The tool set can use that information to handle all the details involved in setting up for parallel execution.

These tools can be based on a paradigm of the massively parallel system: the distribution of memory within the system, the nature of interprocessor communication, and so forth. The underlying model makes it possible to port the concepts to various machines of the same general nature, but with widely differing hardware details.

For convenient programming models, the underlying paradigm tends to make optimistic assumptions about the hardware. To the extent that these assumptions are not accurate, applications built upon a tool set may be inefficient. A practical paradigm of a massively parallel model is a distributed memory, flat global access model. Main memory is divided among the local memories of the many processor nodes; data might reside on any node being used for the application, and may be accessed from any node in the system. This model is clean and powerful, and can be used as the basis for implementation of convenient tool sets.

These gains in user convenience are easiest to implement based on the assumption that frequent, small transfers will be acceptable (from an efficiency viewpoint). That is, the paradigm allows the user to think in terms of the natural chunk of execution for the problem, and moves data accordingly. In the above example, the natural transfer size would be field data associated with one site. Such small transfers exacerbate the impact of per-communication costs. While low communications overheads and transfer latencies are desirable, they are not easy to achieve when designing massively parallel systems. Many of today's powerful systems become inefficient for computations requiring many small transfers.

So a given concept oriented tool set will be more valuable if a means of coalescing many small transfers into fewer, larger transfers can be found. Such an enhancement must have no impact on the paradigm of the system as seen by the user, and minimal impact (if any) on coding issues; otherwise, it will sacrifice the benefits of these tools.

In section 2 we will present the paradigms and concepts of a particular tool set, suitable for grid-oriented problems. In section 3, we examine performance implications: For sample applications, efficiency degrades due to frequent internode transfers. In section 4 we present techniques for transfer coalescing, and discuss how this improves performance.

2. Canopy

2.1 Concepts

Canopy [5] is a tool set designed for exploring algorithms for Lattice Gauge Theory; it is applicable to many other scientific grid-oriented applications as well. The concepts selected are suitable for automated implementation on massively parallel systems. In fact, Canopy originated as the "language" for coding applications on the massively parallel 5 - 50 Gflop ACPMAPS systems built at Fermilab [6].

The user sees geometric and computational concepts. Geometric concepts include those of *sites*, *directions*, and *neighbors*. A collection of sites forms a *grid*; an application may define one or more grids. (Pre-packaged grids, e.g. n-dimensional rectilinear grids, are used for most applications, but arbitrary grids may be specified by supplying connectivity and coordinate functions). In theoretical physics a ubiquitous concept is that of a field — an object which has a value at each point in space. In Canopy, one or more *fields* can be defined on the grid; for each field, room for one instance of a structure is allocated at each site. (*Link fields,* defined on the links between neighboring sites, are also supported; these are a common concept in Lattice Gauge Theory.)

The large bodies of data in problems suitable for massively parallel systems are organized into fields. The field data is accessed only by Canopy routines such as field_pointer (to read a field) and put_field (to write field data).

The basic computational concept in Canopy is that of the *task*. This is a routine which is to be done by each site in a set, in principle in parallel. A set of sites can be defined explicitly, or an entire grid can be specified. In addition, compound (partially ordered) sets of sites are supported — a task done on such a set will logically run for sites on level 1 before those on level 2, and so forth. Orthogonal to the task concept is that of the control program, which does non-parallel computations for the job as a whole. The control program issues tasks to initiate parallel execution.

During a task, each site can be viewed as a virtual processor; the task routine is the program, and an instance of the field data is part of local memory. Since the number of sites for a problem is typically many times the number of physical processing nodes, the

responsibility for many sites will be allocated to each node. Although the work for each site can vary, balance in workload is achieved in a statistical sense. Canopy automates the distribution of data and work; the user need not be concerned with how many nodes are used for a particular job.

2.2 Model of the System

Task routines for a particular "home" site often access field data belonging to other (typically neighboring) sites. The user sees only the high-level geometric concepts, specifying the target site by directions, paths along the grid, and so forth. For example, the routine field_pointer(field, dir) supplies a pointer to the field data for a neighboring site in the specified direction. The tool set implementation must supply access to the desired data, wherever it resides on the system. When the target site is not on the same node as the home site, a copy must be made by transfering data from (or to) off-node locations, using lower-level communications primitives.

Canopy supports a broad range of applications, including non-lockstep algorithms. A task might even include data-dependent logic determining which site's data is to be accessed next. Since the access pattern is not known in advance by remote nodes, they cannot prepare data for subsequent transfer. That is why the communications model of the system must be that of "flat, global access" — any node can access the field data of any other node in the job, without the prior knowledge or cooperation of the remote node. The relevant low-level routines are remote_read and remote_write, specifying the target node and an address and length of the desired remote data.

At first glance, this underlying model might eliminate a very important class of machines — those with message-passing communications. From the hardware viewpoint, that style of communications can be cleaner to implement. For example, Intel systems (the iPSC series, the Delta and the Paragon) use message passing system calls, with the ability to associate a handler with specific incoming messages. Similar models are supported for other systems; fewer systems support direct flat global access.

Fortunately, the remote access model can be expressed in terms of the message passing paradigm, as long as handlers can be associated with incoming messages, and non-

blocking receive requests can be posted. The trick is that each node repeatedly posts receives for incoming "transfer request" type messages. (Implementing remote access routines as a layer over message passing does not seriously degrade performance.) Similarly, message passing (including handlers) can be implemented in terms of remote_read and remote_write, if a remote_interrupt capability is also available.

A tool set will be portable if its potential system dependencies are put in terms of simple-to-specify low level routines (including interprocessor communications). Then supporting a new machine involves porting the small number of "hardware interface" routines, rather than a large body of higher level user support functions. A side benefit is that several distinct tool sets might share one hardware interface definition; then porting one is tantamount to porting them all. Canopy is written on top of the Canopy Hardware Interface Package (CHIP). CHIP deals with remote access routines rather than message passing, since these are more convenient for the Canopy paradigm. By using CHIP (or an analogous interface for tools based on message passing [7]), tools written for one system can be ported to many different systems with little effort.

An important technical point in defining a remote access interface is that of access causality. The proper rule is that if processor A writes to B, and later A writes to C telling C to read B, then C must see the *newly written value* in B. Higher level tools tend to rely on this property in subtle ways, leading to undesired behavior if a particular access is delayed by the right interval to violate the above rule. In systems based on message-passing hardware, where a given message might have arbitrary transit time, implementations of remote_read and remote_write must be carefully guarantee access causality. For example, this requires an 'acknowledge' return message for remote_write.

3. Performance Implications

3.1 Communications Latency and Overhead

Canopy benefits from partitioning the data space and work into many relatively independent parts (sites). The natural approach to dealing with communications is to (logically) transfer data needed to process one site, when that data is required. This may necessitate an off-node access, to be done immediately.

This approach has performance implications, which can be illustrated using examples from lattice gauge theory applications. (The Fermilab theorists have been running a spectrum of applications on ACPMAPS; these results vary by a factor of two, depending on the algorithms used.) In the dominant tasks for typical applications, each site reads field data from nearby sites, and performs computations to alter data at the home site. The fields comprise small complex matrices; computations involve various matrix multiplies, accumulates, and other steps.

A well-implemented tool set will tend to cluster neighboring sites on the same node, but since the grid is 4-dimensional, the "internode boundaries" are not negligible. Depending on the size of the problem and the number of processors, about one field access in 4 requires off-node access. One representative task routine requires 8000 cycles and an average of 4 off-node field accesses. On systems used today, the internode data bandwidth is adequate for this flow, but transfer latencies and overheads are a serious concern.

The overhead of an off-node access has several components: the master node must verify that the target is sensible; the transfer must be set up (in software); the hardware protocol must proceed; the slave hardware (and possibly software) must respond to the transfer; and any completion handshakes must be done. To these costs must be added the loss of processing time on the slave node (for cooperation in the access). One less obvious effect is "cache backwash": When transfer processing code misses cache, in addition to the direct increase in communications latency, an almost equal cost will be incurred after the transfer is completed, to get useful data replaced into cache on the master and slave nodes.

For hardware and system design focusing on excellent communications latencies, the total per-access cost can be as small as a few hundred cycles. Then the inefficiency due to

numerous transfers is an acceptable 15%. (This was the case for the 5 Gflop ACPMAPS system, and should ultimately be true for the Intel Paragon, with transfers handled smoothly by the auxiliary communications processor on each board.) However, for many systems, (e.g. earlier Intel systems, and the 50 Gflop ACPMAPS system) a read access can cost 2,000 - 6,000 cycles. This leads to performance degradation by a factor of 2-4 — unacceptable for many key applications.

Much of the work done to date on massively parallel systems [1,8,9] implements algorithms with regular data access patterns, and fixes the distribution of field data in advance. With these restrictions, the user program can explicitly prepare large blocks of data to send, minimizing transfer overheads. Very high efficiency can be achieved, especially when running suitable algorithms on restrictive special architectures. This approach sacrifices the benefits of concept oriented tool sets: it requires the algorithm to fit a mold, and the user to consider explicitly how to distribute work and data.

Many potential scientific users would prefer to code in a more natural manner, and are willing to trade off some performance. Still, one would like to avoid unnecessarily large degradations. This can be accomplished if the implementation of the tool set finds some way — transparent to the user — to coalesce many transfers between the same two nodes. If the extra work required to coalesce the transfers is no more than a few hundred cycles per block, then communications latencies can be amortized over many blocks transfered together.

3.2 Contention for Communications Channels

Transfer coalescing techniques can gain even when their CPU costs are not small. Several nodes may share the same communications resources, which can become bandwidth bottlenecks. If those resources are occupied during part of the transfer overhead, then frequent transfers exacerbate the bottleneck. In this case, the reduced number of transfers will gain, and extra CPU overhead to implement coalescing is moot with respect to the critical resource.

This effect is important on the ACPMAPS 50 Gflop system. Sixteen i860 processors nodes reside in each crate, sharing bandwidth to neighboring crates. This intercrate

bandwidth is a limitation for many large jobs. The per-transfer overhead incurred with the channel open is about twice the actual data transfer time for a typical field. So, transfer coalescing becomes more important as problem size increases.

3.3 Performance Without Transfer Coalescing

The applications studied illustrate various aspects of lattice gauge computation. Two are local, highly structured algorithms typical of applications brought up on massively parallel systems without the Canopy tools: a gauge field Monte Carlo (MC), and a conjugate gradient quark propagator computation (CG). Another is a gauge-fixing application, accelerated by a highly non-local FFT (GF).

The last algorithm looked at is a propagator computation, inverting an operator using a Minimum Residual method with Lower/Upper decomposition preconditioning (LU). This is a highly non-lockstep algorithm involving significant synchronization issues and is exceedingly difficult to program for parallel systems without the aid of Canopy. It inevitably runs at a low efficiency when measured in Flops done per second. It is worth pursuing because it converges to the correct result much more rapidly than alternative methods, for situations which take the most time: operators with large condition number [10]. Studies of systems with light quarks fall into this category.

Each application was coded using Canopy, and studied on ACPMAPS across a range of problem sizes, scaling the number of nodes with lattice size. The results are shown in Figure 1.

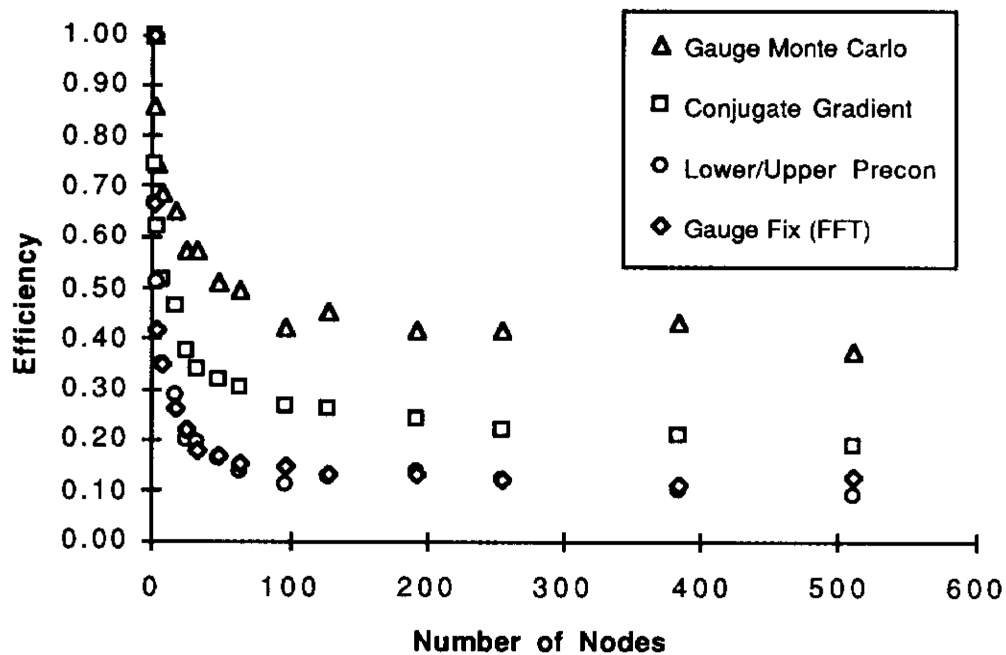

Figure 1: Efficiencies (relative to single-node performance) of several algorithms without transfer coalescing.

Although for more than 16 nodes communications channel contention plays a major role on ACPMAPS, this behavior illustrates the general situation: Frequent communications severely degrade performance. Algorithms which are communications-intensive and non-local perform particularly poorly on large systems.

4. The Multi-thread Strategy

4.1 Theory

Communications intensive tasks usually involve many logical block transfers between a given pair of nodes. The idea is to coalesce multiple transfers between processors, without affecting the way the user sees the transfers. Ideally, the user would face no changes or restrictions associated with transfer coalescing; in our implementation, the restrictions are minimal and do not hurt the usefulness of the tool set.

The implementation takes advantage of two key properties of applications which do frequent transfers: There are usually many sites per processor node, and communications tend to be local. That is, the allocation of sites to nodes can usually be arranged such that most nodes will frequently access the same several "neighboring" nodes. For local algorithms in gauge theory on a 4-dimensional lattice, almost all accesses involve a small number of "neighboring processors" for a given node. Even for non-local algorithms, the pattern of communications from a given node will typically favor several nodes as transfer targets.

The strategy of multi-threading is as follows:

When an off-node field access is done, the underlying tools record information about the requested transfer, and allocate space for incoming data, but *do not actually do the remote access*. Instead, the processor moves on to do the task routine for another site — another "thread" of execution. The actual accesses are done later, when no further work can be done without the required data. At that time, threads which have had their requests satisfied will be marked as eligible to resume processing. The key point is that of the many access requests accumulated, several will involve the same remote node. These requests can be coalesced into a single transfer.

Three conditions must exist for this strategy to be worthwhile: (1) There must be efficient remote scatter/gather communications routines. (2) The CPU cost of a thread switch must be small enough that it does not impact performance. (3) The degree of coalescing — the number of blocks to be accessed from each remote node — must be large enough to make the effort worthwhile.

4.2 Scatter/gather

Coalesced transfers require data blocks from many specified addresses on the target node to be read in to the accessing node, and placed at specified addresses. This is a remote gather (with local scatter) operation. Since the remote node will not have prepared data for transfer, at least the gather aspect on the remote node must be performed by the communications primitive.

On an arbitrary system without explicit remote scatter/gather hardware, it might not be possible to accomplish remote gather efficiently. Fortunately, for systems using the message passing paradigm as the underlying mechanism for remote access transfers, it is easy to create a remote_gather routine — a control block is passed, and the desired data is sent back.

Multi-threading will lead to short data blocks within the remote_gather access, and there is some per-block "bookkeeping" overhead associated with such transfers. But this overhead is small: On the ACPMAPS system, transfers of multiple 20-word blocks proceed at about 80% of the maximum possible transfer rate. Therefore, no new communications bandwidth problems are introduced by multi-threading.

4.3 Light context switching

Although multi-threading is a very simple form of multi-tasking, the "threads" of execution in this approach are as "lightweight" as can be. This is not a pre-emptive multi-tasking environment. Threads can lose the CPU only at well-defined points in their execution (when off-node data access is requested). All that needs to be done is:

- Save any registers which are to be preserved across subroutine calls.
- Set up a new stack context, by pointing to an available stack area and storing the return address.
- Select the next site available for execution.
- A few internal variables, such as the pointer to the current home site, must be saved and set to their new values.

Much of the usual work associated with context switching is avoided. For example, the task routine is already in memory. New memory mapping and allocation are not needed — the new stack pages are selected from among a set which were allocated at the start of processing. Flushing and restoring the floating point pipeline is not necessary. The thread context switch involves neither traps nor system calls.

The number of threads which may be active at one time is established when multi-threading is initiated; thread stacks are created at that time. Each thread requires memory for stack space and for local allocation to place incoming data. Remote data transfers proceed when the number of threads is saturated (or every site has a thread) and no remaining threads are eligible for execution — every thread is waiting for data.

4.4 Degree of Coalescing

The degree to which multi-threading succeeds in coalescing transfers can be defined by a ratio:

$$\frac{\text{number of internode transfers done in the absense of coalescing}}{\text{actual number of transfers done with coalescing}}$$

Obviously, this ratio will be dependent on the application, the number of threads available, the distribution of sites among the nodes, and the order in which sites are processed. Although *a priori* estimates of the degree of coalescing may be useful, we have also studied the actual degree of coalescing by running applications using various numbers of nodes and numbers of threads. The results are shown in Table 1.

Number of nodes	Number of threads	Algorithm			
		MC	CG	GF	LU
96	20	7.7	4.6	26.3	12.5
96	50	14.8	9.1	49.6	18.0
96	100	22.8	15.5	87.4	20.6
256	100	25.8	16.1	84.5	18.1
96	250	49.6	35.6	186.1	21.9
96	500	90.6	59.0	260.6	23.8

Table 1: Degree of coalescing for four representative algorithms.

(Most trials were on 96 nodes; some 256-node trials were examined to verify that the degree of coalescing is not strongly dependent on number of nodes.)

For structured local algorithms, the degree of coalescing goes up linearly for large number of threads — geometric arguments would predict this. The highly non-local FFT benefits substantially from "vertical coalescing" (discussed in 4.7 below); multiple transfers associated with a task are grouped before moving to the next thread, giving excellent coalescing. (In this case, the degree of coalescing can exceed the number of threads.)

The LU preconditioning algorithm has limited coalescing for large number of threads. This is due to a special site distribution, which minimizes task startup and synchronization losses. For smaller numbers of threads, the order in which sites are processed tends to bunch transfers to the same several nodes. Because of this effect, and synchronization issues, LU decomposition runs best on ACPMAPS with about 50 threads.

In all these instances, overhead associated with internode transfers can be amortized over 10-50 or more blocks, so the cost of a transfer has been replaced by a few percent of that cost, plus one thread switch. For systems on which the transfer cost is large and/or the thread switch is fast, this can mean considerable savings.

4.5 Impact on User Code

To take advantage of multi-threading, the user code indicates how many threads to use and how much stack space to reserve for each active thread. Although large local stack-allocated arrays can be created in C, most task routines use very little stack memory for one site. (The huge field data structures associated with a large problem are not on the stack.) For most applications, a default value of 8K suffices for stack space. (The system can warn the user if this stack space is ever overflowed.) The number of threads is chosen, keeping in mind the need for one thread stack apiece — multi-threading does involve a trade of memory space for speed. On systems with 16 Mbytes or more per node, 256 or more threads are commonly possible, but as few as 50 threads gives an adequate degree of coalescing, so it is easy to choose a suitable number of threads.

Another impact on the user program, involving global variables, occurs in relatively few codes, but is more noxious when it appears. Global variables make sense in massively parallel computing when they are written (and broadcast) only by the control program in between parallel tasks, and are treated as read-only by the distributed tasks. If a task

routine were to write to a global variable and broadcast it to other nodes, it would be ill-defined as to whether the same task routine running for another site would read the old or new value. Thus Canopy discourages the use of task-written globals.

However, there is one case of a task-written global variable which will work properly in the absence of multi-threading: A "task global", which has a scope of one task routine, and is logically meaningful only during the execution of that task for a single site. For instance, such task globals can be used instead of passing computed values as arguments down a chain of subroutines. (While some consider this poor coding practice, it is occasionally convenient.) When the task routine ends, the value of the variable is discarded.

Such task global variables are logically dangerous, since task routines are in principle executed in parallel. They work properly in a single-threaded environment because each node processes sites one at a time. They will *not* work properly in the multi-thread environment: The task might set a global, then request a remote access and lose its thread. Another task can then set the same global to a different value. The context switching mechanism has no way of preserving the original value, because there is nothing to identify which global variables are being used in this manner.

Fortunately, there is an easy fix for this problem. Task globals are turned into elements of an array dimensioned by the number of threads, and indexed by a special variable CAN_my_thread, which is always set to the executing thread number. This fix does force the user to modify code, which is undesirable. We could automate the modifications by a pre-processor, with the user only identifying task globals. However, the user would still need to identify task globals, and watch out for caveats associated with the automated procedure. It is easier to apply the fix by hand in the rare cases where it is necessary.

4.6 Effect on Performance

The effects of transfer coalescing were studied for lattice gauge theory applications running on ACPMAPS — configuration generation, gauge fixing, and propagator calculation. As shown in 3.5 above, these applications perform poorly in the absence of multi-threading, as the problem size and number of nodes grow together. (Both communications overhead and channel contention effects are mitigated by transfer coalescing.)

We ran the identical programs, selecting a moderate and a large number of threads. The effects of multi-threading are shown in figures 3 and 4. It can be seen that this technique gains considerably, especially for large problem and system sizes.

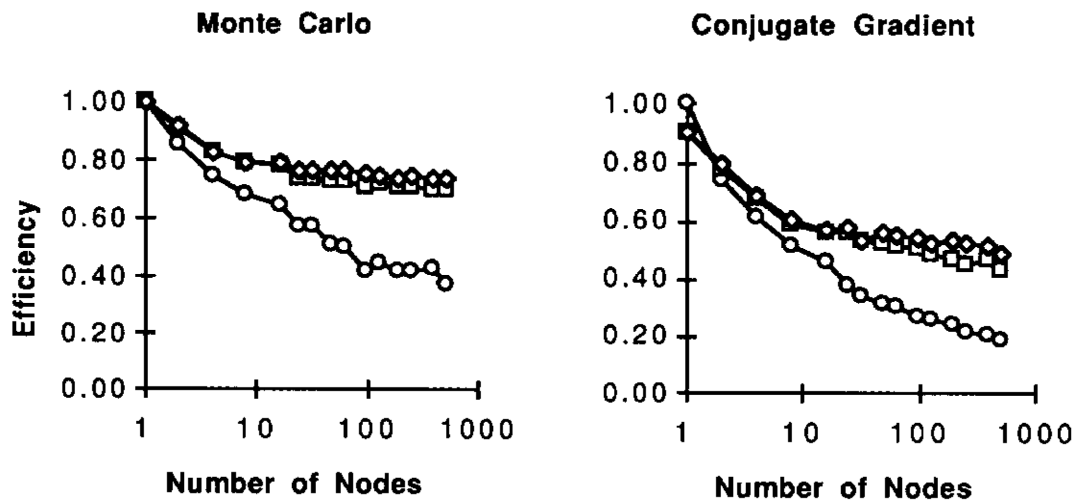

Figure 3: Effect of number of threads on performance of algorithms with local communications patterns. The top, middle and bottom curves show 512 threads, 64 threads and single-threaded (no coalescing) performance.

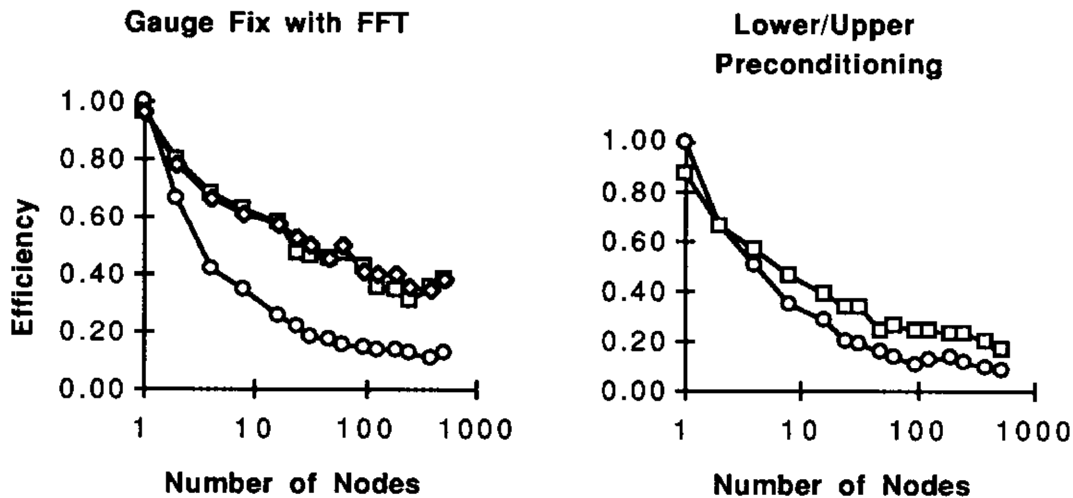

Figure 4: Performance of algorithms with non-local or complex patterns of communication. The bottom curve shows non-coalesced performance. LU preconditioning was not done for 512 threads.

On ACPMAPS, transfer coalescing typically gains a factor of about two; for structured yet highly non-local communication patterns such as FFT, the gain is more than a factor of three. Very large numbers of threads are not important; the gains achieved for 64 threads are almost identical to those for 512.

4.7 Special Features and Techniques

Multi-threading may enhance system performance beyond the basic savings of communications overhead. For instance, transfers can now be non-blocking: If the channel to a remote node is unavailable, pending accesses to other nodes can be attempted. If one pending transfer succeeds, the threads which had requested data from that node can be activated. This effect, though beneficial, is small because the non-blocking feature is only important when the communications channels are nearly saturated, and gains CPU time, rather than communications bandwidth. Suppressing the non-blocking feature was found to cost up to 6%.

In multi-threading, routines to *read* remote field data participate in coalescing. It is more natural in typical applications to read data from other sites, rather than to change it. However, a simple technique suffices to coalesce data to be written to remote nodes. The put_field routine can mark (and/or copy) data for transfer, but delay the transfer until the

task completes for each site. For programs which do a lot of remote writing, this can improve performance. Algorithms using the Fast Fourier Transform benefit from this improvement, by 35% or more for large problems. (One subtlety — to avoid data coherency problems, it is best to "flush" pending remote writes whenever a remote read is initiated, and vice versa. Otherwise, a field can be written at another site and then read back as its old value.)

Another technique which can be important but requires some simple user guidance is that of "vertical coalescing". Many task routines need to pull in several data fields from various sites, before manipulating that data. The basic multi-threading scheme would involve a context switch as each off-node access is requested, even though the data is not needed until later. With user guidance, the system can be informed about when the data is needed, by bracketing the field_pointer() calls with calls indicating the beginning and end of a vertical coalescing block. This not only saves the context switching overhead, but also increases the degree of coalescing, by making more transfers available to coalesce for a given number of threads. Opportunities to apply vertical coalescing are common in lattice gauge theory, and save an average of 9% on algorithms sampled.

4.8 Other Tool Sets

The merits of transfer coalescing are not restricted to lattice gauge theory, or to problems on a static grid. The properties of many virtual processes per node and nearly-local transfer patterns are not unique to Canopy. Tool sets involving dynamic grids and re-allocation of work, or using non-grid paradigms, can still profit from the trick of using light-weight controlled context switches to enable better structuring of communications, in a user-transparent manner.

5. Summary

Tool sets which provide implementations of appropriate scientific concepts are invaluable, but they can logically imply frequent internode communications. Our objective was to minimize the performance degradation in situations where many small transfers are generated, by coalescing internode traffic into larger blocks. To avoid sacrificing the advantages of the tool set, these techniques must have minimal effect on the users' view of the tools.

The Canopy tool set, which is appropriate for problems on arbitrary but static grids, was used as a testing ground for approaches to this issue. Canopy was ideal for this purpose because it implements high-level concepts, and quite a few lattice gauge applications — written without regard to transfer coalescing issues — were available for study. The strategy used to coalesce transfers was "multi-threading", involving light-weight context switches without system calls.

Transfer coalescing requires the addition of remote scatter/gather to the set of communications capabilities required to support the tool set on various systems. Creating these new "primitives" is straightforward in systems based on message passing.

The impact of multi-threading on user code is minimal — the paradigm of considering each site as a virtual processor remains intact. Some attention must be paid to the use of read/write global variables during task routines which are logically executed in parallel. Multi-threading also places an additional small burden on memory.

For the applications studied, multi-threading achieved a high degree of coalescing, and improved performance significantly. The gains are largest on jobs running on many processor nodes. Typical applications gained a factor of two; in extreme cases, applications which could not sensibly be run on large numbers of nodes without transfer coalescing were improved by factors of 3 or more.

The key point in implementing transfer coalescing techniques is to keep the user model clean. Assumedly, a tool set being considered for performance improvements contains a carefully constructed set of concepts and interfaces for its intended users. The tools were created to make programming easier, more natural, and more portable. When modifying

the tools to improve efficiency, these concepts and interfaces must remain unchanged, to retain those advantages. Fortunately, automated coalescing is often possible within that constraint.

Acknowledgments

The Lattice Gauge Theory group at Fermilab were valued participants in these performance studies: The physics production applications studied were created by E. Eichten, G. Hockney, A. Kronfeld, A. El-Khadra, and P. Mackenzie. Dr. Hockney was also a key contributor to the implementations of Canopy and of the multi-thread enhancements.

References

[1] Several examples of such applications appear in "Proceedings of the First Intel Delta Workshop" edited by T. Mihaly and P. Messina, CCSF-14-92, Caltech Concurrent Supercomputing Facilities.

[2] D. Cheng, "A survey of Parallel Programming Languages and Tools" (available from NASA Ames Research Center) describes a number of tool sets appropriate for various scientific fields.

[3] E. Eichten, G. Hockney and H. Thacker, "Lattice Calculation of the B-Meson Decay Constant", Nucl. Phys B (Proc. Suppl.) 20 (1991) 500.

[4] A. El-Khadra, G. Hockney, A. Kronfeld and P. Mackenzie, "A Determination of the Strong Coupling Constant from the Charmonium Spectrum", Phys. Rev. Letters 69,729 (1992).

[5] M. Fischler, G. Hockney, and P. Mackenzie, "Canopy 5.0", Fermi National Accelerator Laboratory technical document. The Canopy manual is available (as a LATEX source doucument) over anonymous ftp at fncrd7@fnal.fnal.gov.

[6] M. Fischler, "The ACPMAPS System: A Detailed Overview", FERMILAB-TM-1780, 1992.

[7] R. Butler and E. Lusk, "User's Guide to the p4 Programming System", ANL-92/17, 1992. The p4 library defines primitives expressing both message passing and shared memory paradigms.

[8] Z. Dong and N. Christ "QCD Phase Structure with 8 Light Quark Flavors", Nucl. Phys B (Proc. Suppl.) 26 (1992) 314.

[9] F. Butler, H. Chen, J. Sexton, A. Vaccarino, and D. Weingarten, "Infinite Volume Continuum Limit of Valence Approximation Hadron Masses", Nucl. Phys B (Proc. Suppl.) 26 (1992) 287.

[10] G. Hockney, "Comparison of Inversion Algorithms for Wilson Fermions", Nucl. Phys. B (Proc. Suppl.) 17 (1990) 301-304. Several varieties of pre-conditioners and orderings of sites for the LU pre-conditioner were explored.