Communication Optimizations for Parallel Computing Using Data Access Information

Martin C. Rinard
Department of Computer Science
University of California, Santa Barbara
Santa Barbara, California 93106
martin@cs.ucsb.edu

Abstract

Given the large communication overheads characteristic of modern parallel machines, optimizations that eliminate, hide or parallelize communication may improve the performance of parallel computations. This paper describes our experience automatically applying communication optimizations in the context of Jade, a portable, implicitly parallel programming language designed for exploiting task-level concurrency. Jade programmers start with a program written in a standard serial, imperative language, then use Jade constructs to declare how parts of the program access data. The Jade implementation uses this data access information to automatically extract the concurrency and apply communication optimizations. Jade implementations exist for both shared memory and message passing machines; each Jade implementation applies communication optimizations appropriate for the machine on which it runs. We present performance results for several Jade applications running on both a shared memory machine (the Stanford DASH machine) and a message passing machine (the Intel iPSC/860). We use these results to characterize the overall performance impact of the communication optimizations. For our application set replicating data for concurrent read access and improving the locality of the computation by placing tasks close to the data that they access are the most important optimizations. Broadcasting widely accessed data has a significant performance impact on one application; other optimizations such as concurrently fetching remote data and overlapping computation with communication have no effect.

1 Introduction

Communication overhead can dramatically affect the performance of parallel computations. Given the long latencies associated with accessing data stored in remote memories, computations that repeatedly access remote data can easily spend most of their time communicating rather than performing useful computation. A variety of optimizations can either eliminate communication or ameliorate its impact on the performance of the program.

- **Replication:** Replicating data close to reading processors allows them to concurrently read the data from local memories instead of serially reading the data from a remote memory.
- **Locality:** Executing tasks close to the data that they will access can improve the locality of the computation, reducing the amount of remote data that tasks access.
- **Broadcast:** Broadcasting widely accessed pieces of data upon production can parallelize the data distribution and eliminate fetch overhead.
- **Concurrent Fetches:** Concurrently fetching the multiple pieces of data accessed by a given task can parallelize the data transfer overhead.
- **Latency Hiding:** Using excess concurrency to fetch remote data for one piece of computation while executing another can hide the latency of accessing remote data.

While all of these optimizations may improve the overall performance of the application, it is unclear how to best apply the optimizations. Requiring the programmer to implement the optimizations directly may lead to an explosion of low-level, machine-specific communication constructs distributed throughout the source code [15]. Many parallel languages, on the other hand, fail to give the implementation the data usage information that it needs to automatically apply sophisticated communication optimizations on a variety of computational platforms [10, 2, 6].

This paper describes our experience with communication optimizations in the context of Jade [19, 20], a portable, implicitly parallel programming language designed for exploiting task-level concurrency. Jade programmers start with a program written in a standard serial, imperative language, then use Jade constructs to describe how parts of the program access data. The Jade implementation analyzes this information to automatically extract the concurrency and execute the program in parallel. As part of the parallelization process the implementation exploits its information about how tasks will access data to automatically apply the communication optimizations appropriate for the hardware platform at hand. Jade implementations exist for shared memory machines (the Stanford DASH machine [14]), message passing machines (the Intel iPSC/860 [3]) and heterogeneous collections of workstations. Jade programs port without modification between all platforms.

The first contribution of this paper is a set of implementation techniques for automatically applying communication optimizations on both shared memory and message passing machines. On shared memory machines the Jade implementation exploits its knowledge of how the program will access data to employ a scheduling heuristic that attempts to enhance the locality of the computation by scheduling tasks on processors close to the data they will access. Message passing machines give the software more control over the movement of data, and the message passing implementation of Jade exploits this control and its knowledge of how the program will access data to apply all of the communication optimizations listed above.

The second contribution of this paper is an empirical evaluation of the performance impact of each communication optimization on a set of complete scientific and engineering applications. The actual performance impact of a specific communication optimization on a given application depends both on the capabilities of the underlying hardware and on the characteristics of the application itself. We experimentally evaluate the communication optimizations by running several complete Jade applications and collecting data that allows us to characterize the performance impact of each optimization. We present experimental performance results for the applications running on both the Stanford DASH machine and the Intel iPSC/860.

Although all of the research presented in this paper was performed in the context of Jade, the results should be of interest to several sectors of the parallel computing community. We expect future implementations of parallel languages to have substantial amounts of information about how computations access data, with the information coming either from the compiler via sophisticated analysis or from the programmer via a high level parallel language. The experimental results presented in this paper provide an initial indication of how the potential communication optimizations enabled by such information affect the performance of actual applications. The results may therefore help implementors to choose which optimizations to implement and language designers to decide which optimizations to enable.

The remainder of the paper is organized as follows. In Section 2 we briefly describe the Jade programming language. In Section 3 we present the optimization algorithms for both hardware platforms. In Section 4 we describe the Jade applications. In Section 5 we present performance results for the applications that allow us to evaluate the performance impact of the different optimizations. In Section 6 we survey related work; we conclude in Section 7.

2 The Jade Programming Language

This section provides a brief overview of the Jade language; other publications contain a complete description [17, 18, 20]. Jade is a set of constructs that programmers use to describe how a program written in a sequential, imperative language accesses data. It is possible to implement Jade as an extension to an existing base language; this approach preserves much of the language-specific investment in programmer training and software tools. Jade is currently implemented as an extension to C.

Jade provides the abstraction of a single mutable shared memory that all tasks can access. Each piece of data allocated (either statically or dynamically) in this memory is a shared object. The programmer therefore

implicitly aggregates the individual words of memory into larger granularity shared objects by allocating data at that granularity.

Jade programmers explicitly decompose the serial computation into tasks by using the `withonly` construct to identify the blocks of code whose execution generates a task. The general form of the `withonly` construct is as follows:

```
withonly { access specification section } do (parameters) { task body }
```

The `task body` contains the code for the task; `parameters` is a list of task parameters from the enclosing context. The implementation generates an access specification for the task by executing its `access specification section`, which is a piece of code containing access specification statements. Each such statement declares how the task will access an individual shared object. For example, the `rd(o)` access specification statement declares that the task will read the shared object `o`; the `wr(o)` statement declares that the task will write `o`. The task's access specification is the union of the executed access specification statements.

In many parallel programming languages tasking constructs explicitly generate parallel computation. Because Jade is an implicitly parallel language, Jade tasks only specify the granularity of the parallel computation. It is the responsibility of the Jade implementation to dynamically analyze tasks' access specifications to determine when they can execute concurrently. This analysis takes place at the granularity of shared objects, with the implementation preserving the dynamic data dependence constraints. If one task declares that it will write a shared object and another task declares that it will access that object, there is a dynamic data dependence between the two tasks. In this case the implementation executes the two tasks serially, preserving the execution order from the original serial program. Tasks with no dynamic data dependences may execute concurrently.

In the basic model of parallel computation described so far, all synchronization takes place at task boundaries. A task does not begin its execution until it can legally perform all of its declared accesses; once a task starts its execution, it does not give up the right to access a shared object until it completes. Jade eliminates these limitations by providing a more advanced construct and additional access specification statements [17]. These language features allow programmers to express more advanced concurrency patterns with multiple synchronization points within a single task.

3 Communication Optimization Algorithms

Access specifications give Jade implementations advance information about how each task will access data. Both the shared memory and message passing implementations exploit this advance information to optimize the communication. We next present overviews of the two implementations and describe the communication optimization algorithms that each implementation uses.

3.1 Overview of the Shared Memory Implementation

The shared memory implementation has three components: a synchronizer, a scheduler and a dispatcher. The synchronizer uses a queue-based algorithm to determine when tasks can execute without violating the dynamic data dependence constraints [17]. The scheduler takes the resulting pool of enabled tasks generated by the synchronizer and assigns them to processors for execution, using a distributed task stealing algorithm to dynamically balance the load. The dispatcher on each processor serially executes its set of executable tasks.

3.2 Shared Memory Communication Optimizations

The shared memory implementation relies on the hardware to implement the abstraction of a single address space. A running program therefore performs all communication on demand as it references remote data. Although the lack of software control over the communication limits the optimization opportunities, the shared memory scheduler does apply a locality heuristic. This heuristic attempts to improve the locality of the computation by executing tasks close to the data that they will access. We have implemented several variants of the locality heuristic, each tailored for the different memory hierarchies of different machines [17]. In this paper we discuss the locality heuristic used on machines such as the Stanford DASH machine with physically distributed memory modules (each associated with a processor or cluster of processors) and hardware coherent caches.

3.2.1 The Shared Memory Scheduler

The scheduler assigns tasks to processors using a distributed task queue algorithm. There is one task queue for each processor; the implementation structures this queue as a queue of object task queues. There is one object task queue associated with each object; each object task queue is in turn a queue of tasks. Figure 1 contains a picture of these data structures. Each object task queue is owned by the processor that owns the corresponding object (i.e. the processor in whose memory module the object is allocated). Each processor's task queue contains all the non-empty object task queues owned by that processor.

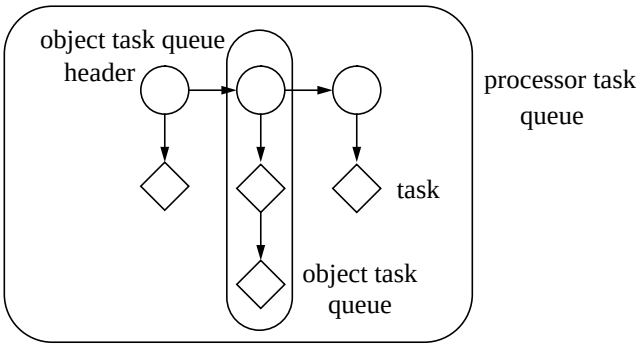

Figure 1: Task Queue Data Structures

Each task has a locality object; in the current implementation the locality object is the first object that the task declared it would access. The implementation will attempt to execute each task on the owner of its locality object. In this case the task's references to the locality object will be satisfied out of local instead of remote memory.

When a task is enabled, the scheduler inserts the task into the object task queue associated with its locality object. If the object task queue was empty before the task was inserted, the scheduler inserts the object task queue into the processor task queue of the processor that owns that object task queue.

When a processor finishes a task, the dispatcher needs a new task to execute. The scheduler first tries to give the dispatcher the first task in the first object task queue. If the task queue is empty, the scheduler cyclically searches the task queues of other processors. When the scheduler finds a non-empty task queue it steals the last task from the last object task queue and executes that task. Once a task begins execution it does not relinquish the processor until it either completes or executes a Jade construct that causes it to suspend. There is no preemptive task scheduling.

3.2.2 Rationale

On a distributed memory machine like the Stanford DASH machine it takes longer to access data stored in remote memory than in local memory. The task queue algorithm therefore attempts to execute each task on the processor whose memory module contains the task's locality object. In this case the task's accesses to its locality object will be satisfied out of local instead of remote memory.

Although the locality heuristic is primarily designed to enhance locality at the level of local versus remote memory, it may also enhance cache locality. The locality heuristic attempts to execute tasks that access the same locality object consecutively on the same processor. This strategy may enhance cache locality if the locality object becomes resident in the cache. The first task will fetch the object into the cache, and subsequent tasks will access the object from the cache instead of local memory. If the heuristic interleaved the execution of tasks with different locality objects, each task might fetch its locality object into the cache, potentially ejecting other tasks' locality objects and destroying the cache locality of the task execution sequence.

3.3 Overview of the Message Passing Implementation

Like the shared memory implementation, the message passing implementation has a synchronizer, a scheduler and a dispatcher. But because the message passing implementation is also responsible for implementing the Jade abstraction of a single address space in software using message passing operations, it has an additional component: a communicator that generates the messages required to implement the abstraction of a single address space. Before a processor executes a task, the communicator ensures that its memory contains up to date versions of the shared objects that the task will access.

3.4 Message Passing Communication Optimizations

Message passing machines give the software much more control over communication than do shared memory machines. The message passing implementation exploits this control to implement a wider range of communication optimizations. Some of these optimizations (replication and migration) mirror those performed by the hardware in most shared memory machines, while others (concurrent fetches, broadcasting and overlapping communication with computation) have not been supported in traditional hardware shared memory machines.

3.4.1 Replication and Concurrent Fetches

The communicator operates at the granularity of shared objects. When a task declares that it will access a remote object, the communicator fetches the entire object from the owner in one message into the memory of the processor that will execute the task. If multiple tasks read the same object, the communicator replicates the object so that the tasks can concurrently read their own local copies instead of serially reading a unique copy. The computation that determines which processor owns the object is integrated into the Jade synchronization algorithm [17]. Each remote object fetch therefore generates two messages: a small message from the processor that will execute the task to the owner requesting the object and a potentially much larger message containing the object sent from the owner back to the executing processor. If a task declares that it will access multiple remote objects, the communicator parallelizes the communication by fetching the objects in parallel.

3.4.2 Adaptive Broadcast

Broadcasting a widely accessed object can improve the communication performance by parallelizing the transfer of the broadcasted object and eliminating fetch overhead. The communicator therefore switches to a broadcast algorithm when it encounters a widely accessed object: if all processors ever access the same version of a given object, the communicator broadcasts all succeeding versions of that object to all processors. The owner of each version of each object detects when all processors have accessed its version by recording the set of processors that request its version.

The adaptive broadcast algorithm does not always optimize the communication. If a processor does not access a broadcasted version of an object, the communication required to transfer the object to that processor is wasted. If a broadcasted version is not widely accessed, the adaptive broadcast algorithm may impair the overall communication performance.

3.4.3 Locality and Latency Hiding

The message passing scheduler uses a centralized dynamic load balancing algorithm augmented with a locality heuristic that attempts to minimize the amount of shared object communication. As in the shared memory implementation, each task has a locality object. In the current implementation the locality object is the first object that the task declared it would access. Each object also has an owner (the last processor to write the object); the owner is guaranteed to have a copy of the latest version of the object. The owner of a task's locality object is called the task's target processor. Executing a task on its target processor eliminates the need to fetch the locality object from a remote processor — the task can simply access the locally available copy. The scheduler attempts to enhance the locality by executing each task on its target processor.

The scheduler also attempts to keep each processor supplied with multiple tasks. The goal is for each processor to be able to overlap computation and communication by fetching remote objects for one task while executing

another. The scheduling policy is to keep assigning tasks to processors for execution until each processor has a target number of tasks to execute.

When a task created on the main processor (the processor executing the main thread of the computation) becomes enabled, the scheduler first checks if any processor has fewer tasks than the target number. If not, it puts the task into the pool of unassigned tasks at the main processor. It will eventually send the task to a processor for execution when the load drops. If any processor has fewer than the target number of tasks, the scheduler will assign the enabled task to one of the least-loaded processors (i.e. the processors with the fewest tasks). If the target processor is one of the least-loaded processors, it will get the task. Otherwise, the implementation chooses one of the least-loaded processors arbitrarily.

When the remote processor receives an enabled task, the interrupt handler that received the message containing the task immediately sends out messages requesting the remote objects that the task will access. In the best case the processor is executing another task when the new task arrives, and it resumes the execution of this old task when it finishes sending out the request messages. By the time the old task finishes and the processor is ready to execute the new task, the remote objects may have already arrived and the new task can immediately execute.

When a remote processor finishes a task it got from the main processor, it informs the main processor. The main processor then checks its pool of unassigned enabled tasks. If the pool is not empty the implementation will assign one of the tasks in the pool to the remote processor for execution, giving preference to tasks whose target processor is the remote processor.

The scheduling algorithm is optimized for the case when the main processor (the processor running the main thread of control) creates all of the tasks in the computation. Although the computation will execute correctly if other processors create some of the tasks, the algorithm may fail to generate a reasonable load balance. The choice of a centralized scheduling algorithm does not affect the experimental results: in the current set of Jade applications the main processor creates all tasks.

4 Applications

The application set consists of three complete scientific applications and one computational kernel. The complete applications are Water, which evaluates forces and potentials in a system of water molecules in the liquid state, String [11], which computes a velocity model of the geology between two oil wells, and Ocean, which simulates the role of eddy and boundary currents in influencing large-scale ocean movements. The computational kernel is Panel Cholesky, which factors a sparse positive-definite matrix. The SPLASH benchmark set [23] contains variants of the Water, Ocean and Panel Cholesky applications. We next discuss the parallel behavior of each application.

- **Water:** Water performs an interleaved sequence of parallel and serial phases. The parallel phases compute the intermolecular interactions of all pairs of molecules; each serial phase uses the results of the previous parallel phase to update an overall property of the set of molecules such as the positions of the molecules. Each parallel task reads the array containing the molecule positions and updates an explicitly replicated contribution array. Replicating this array at the language level allows tasks to update their own local copy of the contribution array rather than contending for a single copy. At the end of the parallel phase the computation performs a parallel reduction of the replicated contribution arrays to generate a comprehensive contribution array. The locality object for each task is the copy of the replicated contribution array that it will write.

- **String:** Like Water, String performs a sequence of interleaved parallel and sequential phases. The parallel phases trace rays through a discretized velocity model, computing the difference between the simulated and experimentally observed travel times of the rays. After tracing each ray the computation backprojects the difference linearly along the path of the ray. Each task traces a group of rays, reading an array storing the velocity model and updating an explicitly replicated difference array that stores the combined backprojected difference contributions for each cell of the velocity model. At the end of each parallel phase the computation performs a parallel reduction of the replicated difference arrays to generate a single comprehensive difference array. Each serial phase uses the comprehensive difference array generated in the previous parallel phase to

generate an updated velocity model. The locality object for each task is the copy of the replicated difference array that it will update.

- **Ocean:** The computationally intensive section of Ocean uses an iterative method to solve a set of discretized spatial partial differential equations. Conceptually, it stores the state of the system in a two dimensional array. On every iteration the application recomputes each element of the array using a standard five-point stencil interpolation algorithm.

  To express the computation in Jade, the programmer decomposed the array into a set of interior blocks and boundary blocks. Each block consists of a set of columns. The size of the interior blocks determines the granularity of the computation and is adjusted to the number of processors executing the application. There is one boundary block two columns wide between every two adjacent interior blocks.

  At every iteration the application generates a set of tasks to compute the new array values in parallel. There is one task per interior block; that task updates all of the elements in the interior block and one column of elements in each of the border blocks. The locality object is the interior block.

- **Panel Cholesky:** The Panel Cholesky computation decomposes the matrix into a set of panels. Each panel contains several adjacent columns. The algorithm generates two kinds of tasks: internal update tasks, which update one panel, and external update tasks, which read a panel and update another panel. The computation generates one internal update task for each panel and one external update task for each pair of panels with overlapping nonzero patterns. The locality object for each task is the updated panel.

In any application-based experimental evaluation the input data sets can be as important as the applications themselves. In each case we attempted to use realistic data sets that accurately reflected the way the applications would be used in practice. The data set for Water consists of 1728 molecules distributed randomly in a rectangular volume. It executes 8 iterations, with two parallel phases per iteration. These performance numbers omit an initial I/O and computation phase. In practice the computation would run for many iterations and the amortized cost of the initial phase would be negligible. The data set for String is from an oil field in West Texas and discretizes the 185 foot by 450 foot velocity image at a 1 foot by 1 foot resolution. It executes six iterations, with one parallel phase per iteration. The performance numbers are for the entire computation, including initial and final I/O. The data set for Ocean is a square 192 by 192 grid. The timing runs omit an initial I/O phase. For Panel Cholesky the timing runs factor the BCSSTK15 matrix from the Harwell-Boeing sparse matrix benchmark set[7]. The performance numbers only measure the actual numerical factorization, omitting initial I/O and a symbolic factorization phase. In practice the overhead of the initial I/O and symbolic factorization would be amortized over many factorizations of matrices with identical structure.

5 Experimental Results

We performed a sequence of experiments designed to measure the effectiveness of each communication optimization. We ran the shared memory experiments on the Stanford DASH machine and the message passing experiments on the Intel iPSC/860. Each experiment isolates the effect of a given optimization by running the applications first with the optimization turned on then with the optimization turned off. Except where noted all other optimizations are turned on. In each case we report results for the applications running on 1, 2, 4, 8, 16, 24 and 32 processors.

There are several exceptions to the experimental approach described above. We evaluate the effect of replication and using excess concurrency to hide latency by analyzing the dynamic behavior of the applications rather than performing experiments. Because it is possible to understand the effect of the concurrent fetch optimization by analyzing the dynamic behavior of the applications with the optimization turned on, we present no results from runs with the optimization turned off.

5.1 Replication

In the message passing implementation the memory of the processor executing each task must contain a copy of each object that the task declares it will access before the task can execute. Concurrent execution of multiple tasks

that read the same object therefore requires replication: all of the memories of the processors that concurrently execute the task must contain a copy of the object.

In the current application set replication is a crucial optimization. All of the applications contain at least one shared object read by all of the tasks in the important parallel sections; such shared objects typically contain parameters from input data files or pointers to other shared objects that the tasks manipulate. Eliminating replication would serialize all of the applications.

5.2 Locality Optimizations

On both DASH and the iPSC/860 we ran the applications at three locality optimization levels.

- **Task Placement:** In Ocean and Panel Cholesky the programmer can improve the locality of the computation by explicitly controlling the placement of tasks on processors. For Panel Cholesky the programmer maps the panels to processors in a round-robin fashion omitting the main processor and places each task on the processor with the updated panel. For Ocean the programmer maps the interior blocks to processors in a round-robin fashion omitting the main processor and places each task on the processor with the interior block that it will update. The programmer omits the main processor because both Panel Cholesky and Ocean have a small grain size and create tasks sequentially. For such applications the best performance is obtained by devoting one processor to creating tasks. For Water and String the programmer cannot improve the locality of the computation using explicit task placement.
- **Locality:** The DASH implementation uses the scheduling algorithm described in Section 3.2.1; the iPSC/860 implementation uses the scheduling algorithm described in Section 3.4.3.
- **No Locality:** The implementation distributes enabled tasks to idle processors in a first-come, first-served manner. The DASH implementation uses a single shared task queue; the iPSC/860 implementation uses a single task queue at the main processor.

We report results for Ocean and Panel Cholesky running at all three locality optimization levels and for Water and String at the Locality and No Locality optimization levels. On the iPSC/860 each run has the adaptive broadcast, replication and concurrent fetch optimizations turned on and the latency hiding optimization turned off (i.e. the target number of tasks for each processor is set to one).

5.2.1 Locality Optimizations on DASH

Our evaluation of the locality optimizations on DASH starts with a measurement of the effectiveness of the locality heuristic, continues with an assessment of how executing tasks on their target processors affects the communication behavior, then finishes with an evaluation of how the communication behavior relates to the overall performance. Figures 2 through 5 plot the *task locality percentage* for the different applications running on DASH at the different locality optimization levels. Each line plots, as a function of the number of processors executing the computation, the number of tasks executed on their target processors divided by the total number of executed tasks times 100. At the Locality optimization level the scheduler will always execute each task on its target processor unless the task stealing algorithm moves the task in an attempt to balance the load. The task locality percentage therefore measures how well the locality heuristic achieves its goal of executing each task on the processor that owns its locality object.

Several points stand out in these graphs. The task locality percentage at the Locality optimization level for both String and Water is 100 percent, which indicates that the scheduler can balance the load without moving tasks off their target processors. The task locality percentage at Locality for Panel Cholesky and Ocean, on the other hand, is substantially less than 100 percent, indicating that the dynamic load balancer in the Jade scheduler moved a significant number of tasks off their target processors. At Task Placement the task locality percentage goes back up to 100 percent, which indicates that the programmer chose to place tasks on their target processors. At No Locality the task locality percentage drops quickly as the number of processors increases for all applications.

We next measure how the task locality percentage differences relate to the communication behavior of the applications. On DASH all shared object communication takes place during the execution of tasks as they

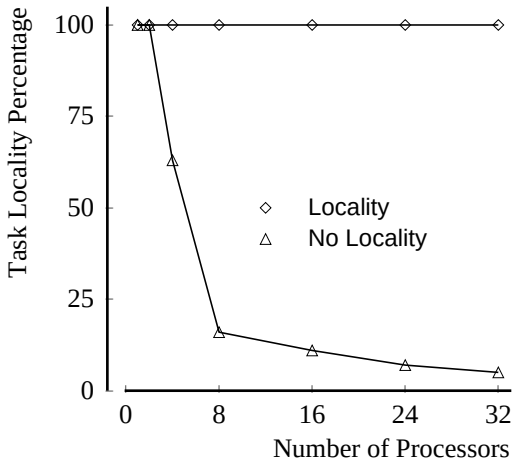

Figure 2: Percentage of Tasks Executed on the Target Processor for Water on DASH

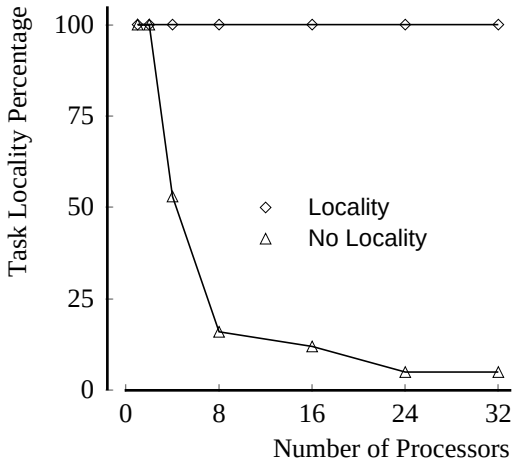

Figure 3: Percentage of Tasks Executed on the Target Processor for String on DASH

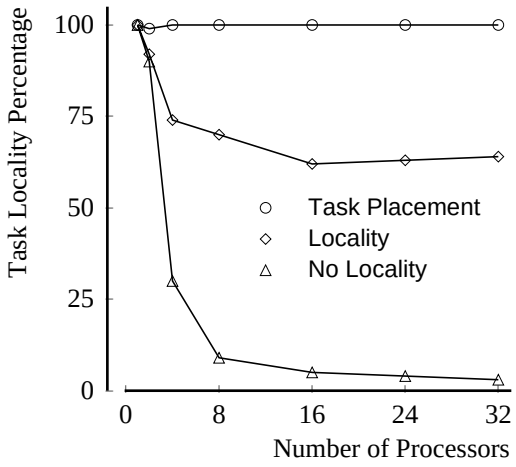

Figure 4: Percentage of Tasks Executed on the Target Processor for Ocean on DASH

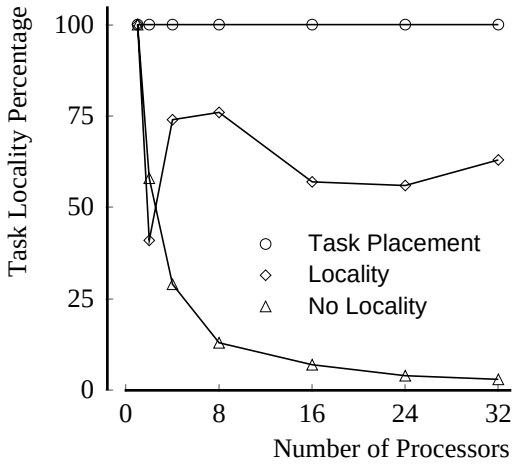

Figure 5: Percentage of Tasks Executed on the Target Processor for Panel Cholesky on DASH

access shared objects: differences in the communication show up as differences in the execution times of the tasks. We therefore measure the effect of the locality optimizations on the communication by recording the total amount of time spent executing task code from the application (as opposed to task management code from the Jade implementation). We compute the time spent in tasks by reading the 60ns counter on DASH[14] just before each task executes, reading it again just after the task completes, then using the difference to update a variable containing the running sum of the total time spent executing tasks. Figures 6 through 9 plot the time spent executing tasks for each of the applications. The total task execution times increase with the number of processors executing the computation because the total amount of communication increases with the number of processors.

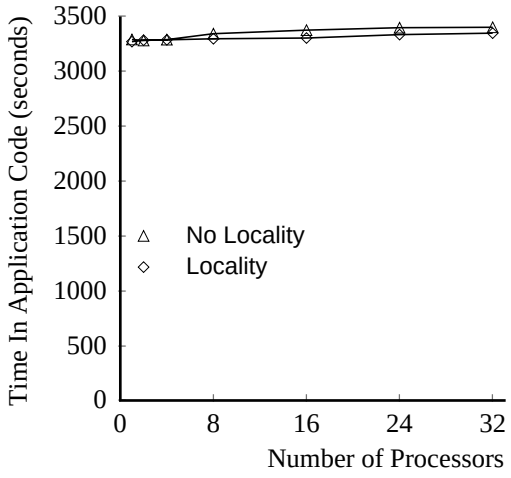

Figure 6: Total Task Execution Time for Water on DASH

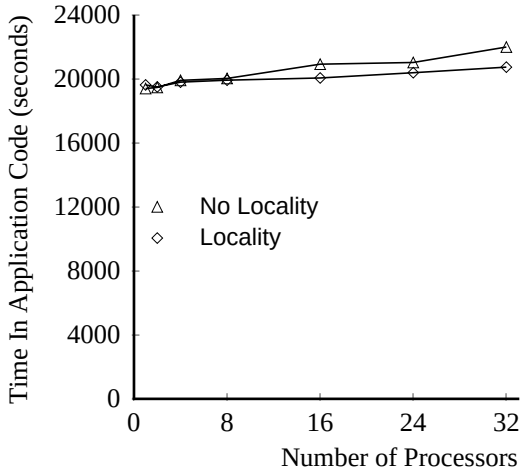

Figure 7: Total Task Execution Time for String on DASH

For String and Water the task locality percentage differences translate into very small relative differences between the task execution times. The tasks in both of the applications perform a large amount of computation for each access to a potentially remote shared object. Enhancing the shared object locality has little effect on

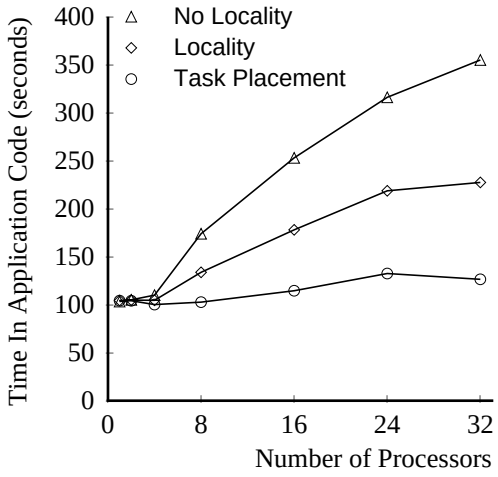

Figure 8: Total Task Execution Time for Ocean on DASH

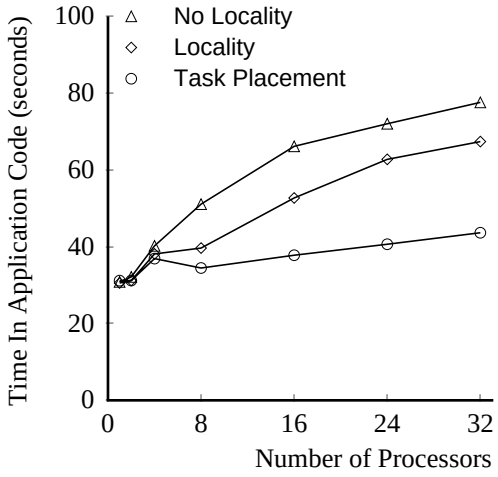

Figure 9: Total Task Execution Time for Panel Cholesky on DASH

the task execution times because there is little communication relative to the computation. For Ocean and Panel Cholesky the task locality percentage differences translate directly into large relative differences in the task execution times. These applications access potentially remote shared objects much more frequently than the other two applications. Without good shared object locality they both generate a substantial amount of communication relative to the computation.

We next present the execution times for the different applications. Table 1 presents the execution times for two sequential versions of each application running on DASH. The serial version is the original serial version of the application with no Jade modifications. The stripped version is the Jade version with all Jade constructs automatically stripped out by a preprocessor to yield a sequential C program that executes with no Jade overhead. The difference between the two versions is that the stripped version includes any data structure modifications introduced as part of the Jade conversion process.

	Water	String	Ocean	Panel Cholesky
Serial	3628.29	20594.50	102.99	26.67
Stripped	3285.90	19314.80	100.03	28.91

Table 1: Serial and Stripped Execution Times on DASH (seconds)

Tables 2 through 5 present the execution times for the Jade versions running on DASH at several different locality optimization levels. A comparison of the execution times of the Jade versions running on one processor with the serial and stripped execution times presented above reveals that overhead from the Jade implementation has a negligible impact on the single processor performance of all applications except Panel Cholesky.

We next consider the performance impact of the locality optimizations. The locality optimization level has little impact on the overall performance of Water and String — all versions of both applications exhibit almost linear speedup to 32 processors. The locality optimization level has a substantial impact on the performance of Ocean and Panel Cholesky, with the Task Placement versions performing substantially better than the Locality versions, which in turn perform substantially better than the No Locality versions. When comparing the performance of the Locality and Task Placement versions, recall that the Jade implementation and the programmer agree on the target processor for each task. For these applications the difference is that the dynamic load balancing algorithm in the Locality version moves tasks off their target processors in an attempt to balance the load.

	1	2	4	8	16	24	32
Locality	3270.71	1648.96	833.19	423.14	220.63	153.03	119.48
No Locality	3290.47	1648.60	832.91	434.36	229.84	160.82	124.74

Table 2: Execution Times for Water on DASH (seconds)

	1	2	4	8	16	24	32
Locality	19621.15	9774.07	5003.69	2534.62	1320.00	903.95	705.84
No Locality	19396.12	9756.71	5017.82	2559.44	1350.06	948.73	769.21

Table 3: Execution Times for String on DASH (seconds)

Even with explicit task placement, neither Ocean nor Panel Cholesky exhibit close to linear speedup with the number of processors. For Ocean the serialized task management overhead on the main processor is a major source of performance degradation. We quantitatively evaluate the task management overhead by executing a work-free version of the program that performs no computation in the parallel tasks and generates no shared object communication. This version has the same concurrency pattern as the original; with explicit task placement corresponding tasks from the two versions execute on the same processor. The *task management percentage* is the execution time of the work-free version of the program divided by the execution time of the original version.

	1	2	4	8	16	24	32
Task Placement	105.21	105.36	36.36	16.14	9.24	8.39	10.71
Locality	105.33	99.22	37.79	25.30	17.58	14.52	13.26
No Locality	104.51	99.20	38.97	31.21	22.31	18.88	17.31

Table 4: Execution Times for Ocean on DASH (seconds)

	1	2	4	8	16	24	32
Task Placement	35.71	33.64	15.24	7.82	5.95	5.61	5.76
Locality	34.94	17.99	11.77	7.53	7.30	7.43	7.86
No Locality	35.09	18.99	12.97	9.29	7.88	8.00	8.48

Table 5: Execution Times for Panel Cholesky on DASH (seconds)

Figure 10 plots the task management percentages for Ocean on DASH. This figure shows that the task management overhead rises dramatically as the number of processors increases. The task management overhead delays the creation of tasks, which in turn extends the critical path of the computation.

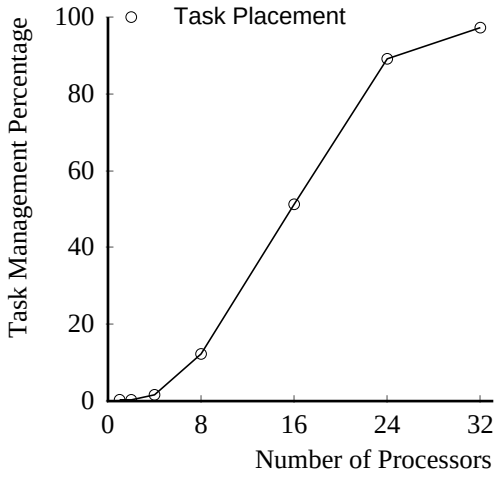

Figure 10: Task Management Percentage for Ocean on DASH

For Panel Cholesky several factors combine to limit the performance, among them an inherent lack of concurrency in the basic parallel computation [21] and the task management overhead, which lengthens the critical path[17]. Figure 11 presents the task management percentage for Panel Cholesky. This figure shows that, as the number of processors increases, the Jade implementation spends a substantial amount of time managing tasks.

5.2.2 Locality Optimizations on the iPSC/860

We start our discussion of the locality optimizations on the iPSC/860 with the task locality percentages in Figures 12 through 15. As for DASH, the task locality percentages for the Locality versions are 100 percent for Water and String, and somewhat less for Ocean and Panel Cholesky. For the Task Placement versions they go up to 100 percent for Ocean, and to 92 percent for Panel Cholesky. The task locality percentage for Panel Cholesky on the iPSC/860 is less than 100 percent because the computation starts out with the current version of all panels owned by the main processor, which just initialized them. The target processor for the first task to access each

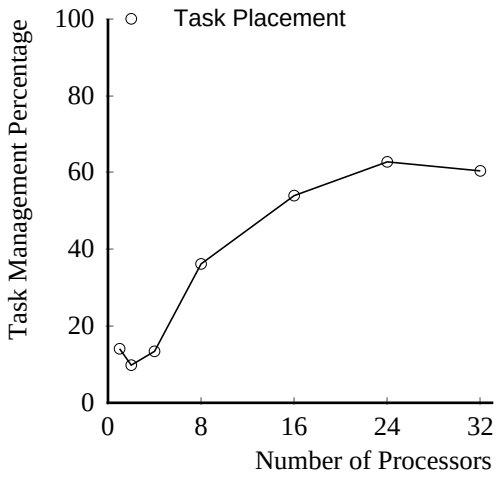

Figure 11: Task Management Percentage for Panel Cholesky on DASH

panel is therefore the main processor, and that task fails to execute on its target processor when the programmer explicitly places it on one of the other processors.

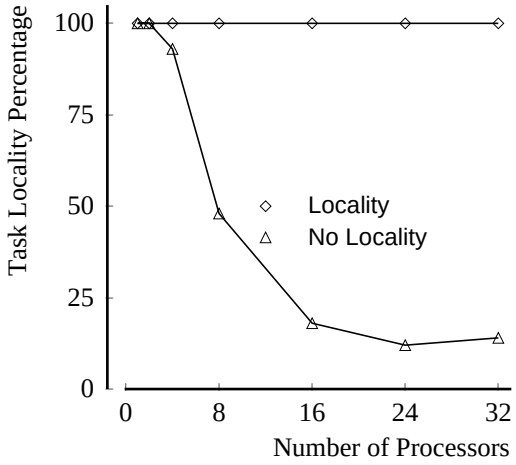

Figure 12: Percentage of Tasks Executed on the Target Processor for Water on the iPSC/860

We next measure how the changes in the task locality percentage relate to the communication behavior. On the iPSC/860 we measure the communication volume directly by recording the total size of the messages that transfer shared objects between processors. We also use a method similar to that used in the DASH implementation to calculate the total time spent executing tasks (unlike the DASH runs the task execution times include no communication time), and divide the total size of the messages (in Mbytes) by the total task execution time (in seconds) to obtain the communication to computation ratio. Because the total task execution times on the iPSC/860 do not vary significantly with the optimization level, changes in the communication to computation ratio correspond directly to changes in the communication.

Figures 16 through 19 plot the communication to computation ratios for the different applications. Improvements in the communication show up as a reduction in the communication to computation ratio. For all of the

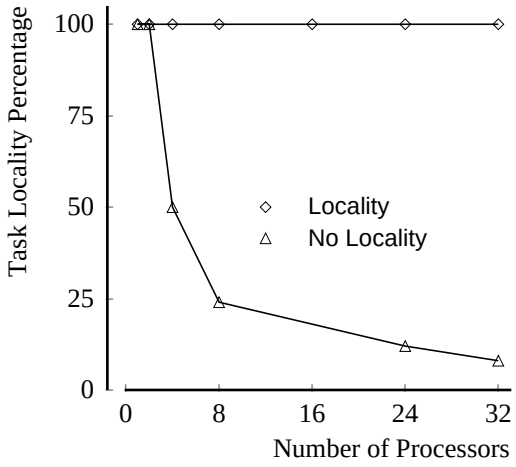

Figure 13: Percentage of Tasks Executed on the Target Processor for String on the iPSC/860

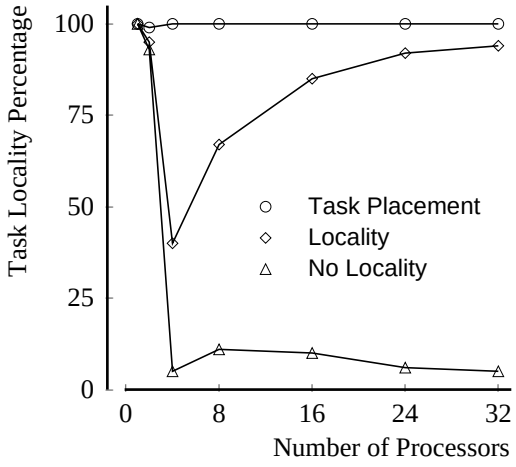

Figure 14: Percentage of Tasks Executed on the Target Processor for Ocean on the iPSC/860

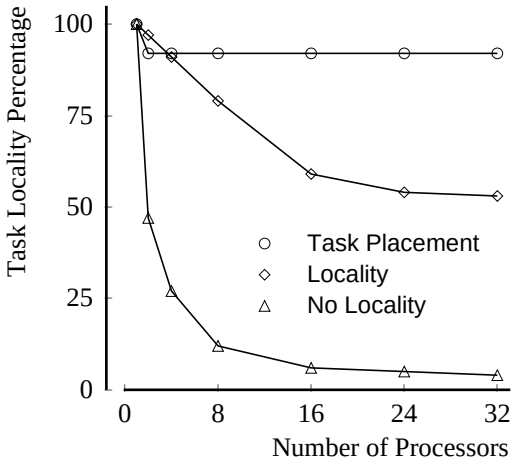

Figure 15: Percentage of Tasks Executed on the Target Processor for Panel Cholesky on the iPSC/860

applications the locality optimization level substantially changes the communication to computation ratio, with lower ratios directly related to higher task locality percentages. The magnitudes of the ratios, however, vary dramatically from application to application. The Water and String applications have very small ratios relative to the communication bandwidth on the iPSC/860 (2.8 Mbytes/second per link), while Ocean and Panel Cholesky have much larger ratios. One would therefore expect the locality optimization level to have a much larger effect on the Ocean and Panel Cholesky applications than on Water and String.

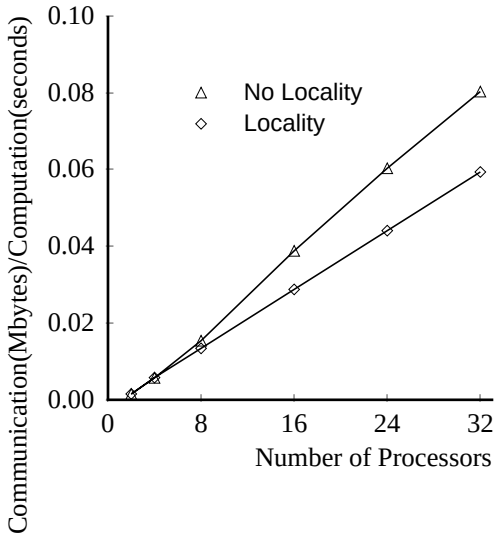

Figure 16: Communication to Computation Ratio for Water on the iPSC/860

Table 6 contains the execution times for the two sequential versions of the applications. Tables 7 through 10 show how the execution times vary with the locality optimization level. For Water and String, as expected, there is almost no effect. For Ocean the locality optimization level has a substantial effect on the performance, with the effect most pronounced at 8 processors. At larger numbers of processors the Jade task management overhead plays a progressively larger role in the performance of the application — the iPSC/860 does not support the fine-grained communication required for efficient task management[3]. Figure 20 presents the task management

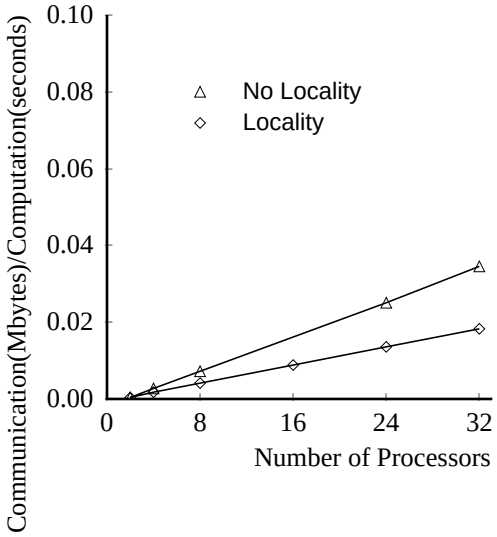

Figure 17: Communication to Computation Ratio for String on the iPSC/860

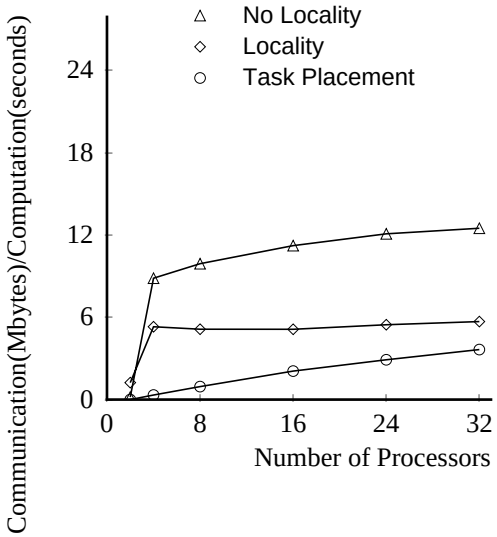

Figure 18: Communication to Computation Ratio for Ocean on the iPSC/860

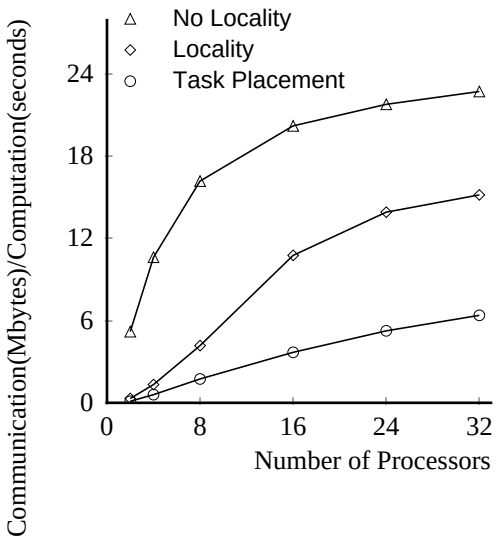

Figure 19: Communication to Computation Ratio for Panel Cholesky on the iPSC/860

percentages on the main processor for this computation. This figure shows that, at 16 processors and above, the task management overhead is the limiting factor on the overall performance.

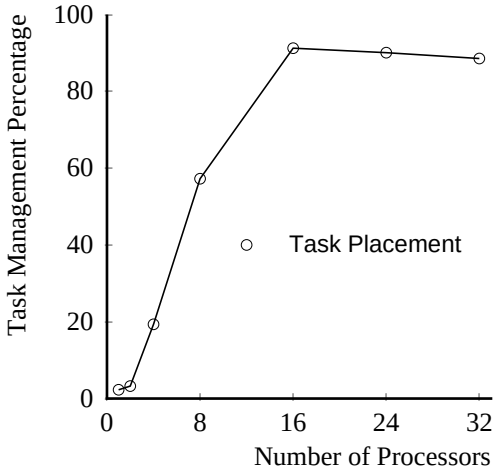

Figure 20: Task Management Percentage for Ocean on the iPSC/860

For Panel Cholesky the locality optimization level has an effect, but the effect is diluted by other sources of inefficiency such as the task management overhead [17]. As for the DASH versions, moving tasks off their target processors in an attempt to balance the load fails to improve the overall performance. Figure 21, which plots the task management on the main processor, shows that the task management overhead significantly limits the overall performance.

5.3 Adaptive Broadcast

Both Water and String can potentially benefit from the adaptive broadcast optimization. In both applications all processors in every parallel phase access an object updated in the preceding serial phase. The performance

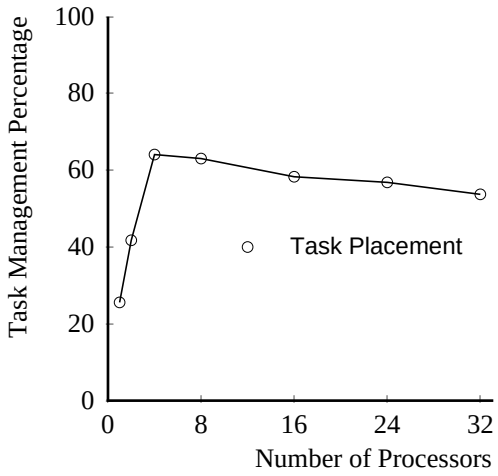

Figure 21: Task Management Percentage for Panel Cholesky on the iPSC/860

	Water	String	Ocean	Panel Cholesky
Serial	2482.91	20270.45	54.19	27.60
Stripped	2406.72	19629.42	60.99	28.53

Table 6: Serial and Stripped Execution Times on the iPSC/860 (seconds)

	1	2	4	8	16	24	32
Locality	2435.16	1219.71	617.28	315.69	165.64	118.09	91.53
No Locality	2454.78	1231.91	623.34	318.34	167.77	119.72	93.11

Table 7: Execution Times for Water on the iPSC/860 (seconds)

	1	2	4	8	16	24	32
Locality	17382.07	9473.24	4773.02	2418.75	1249.69	873.14	678.55
No Locality	18873.86	9529.52	4765.96	2424.12	-	869.27	680.94

Table 8: Execution Times for String on the iPSC/860 (seconds)

	1	2	4	8	16	24	32
Task Placement	77.44	68.14	28.75	18.77	24.16	37.18	51.87
Locality	77.71	93.74	95.95	57.28	39.50	44.48	55.96
No Locality	78.03	100.29	159.77	88.86	56.33	55.56	63.58

Table 9: Execution Times for Ocean on the iPSC/860 (seconds)

	1	2	4	8	16	24	32
Task Placement	54.56	50.18	31.56	32.50	34.41	36.38	38.17
Locality	54.54	34.17	33.65	35.97	43.73	47.62	50.83
No Locality	54.43	107.43	99.39	75.84	59.02	56.41	59.45

Table 10: Execution Times for Panel Cholesky on the iPSC/860 (seconds)

impact of the optimization depends on the relative sizes of the broadcasted object and the tasks in the parallel phases. Without the adaptive broadcast optimization the updated object(s) are serially transferred from the main processor to all of the other processors using point-to-point messages at the beginning of the parallel phase. The main processor is involved in the message send, so its computation is delayed by the time required to send the updated object(s) to all of the other processors. Because the critical path of the computation goes through the main processor, serially transferring the updated object(s) instead of using a broadcast lengthens the critical path by increasing the amount of time that the main processor spends communicating.

We quantify this argument by analyzing the relative sizes of the serial sends, the broadcast operation, and the sizes of the parallel phases. In Water the main updated object is 165,888 bytes long, which means that it takes .07 seconds to send it to another processor. For the 32 processor runs it takes 31 * .07 = 2.17 seconds to send it serially to all other 31 processors, and .31 seconds to broadcast it to all processors. Each parallel phase begins with either the broadcast of the updated object or the creation of the first task in the parallel phase and ends with the resumption of the task in the next serial phase. The mean length of the parallel phases are 5.4 without adaptive broadcast and 7.3 with broadcast. The corresponding numbers for String are 383,528 bytes for the updated object, .16 seconds for one serial send, 31 * .16 = 4.96 seconds for the serial sends to all other processors and .7 seconds for the broadcast. The mean parallel phase length is 106 seconds without adaptive broadcast and 108 seconds with broadcast. Even with adaptive broadcast the first two of the six parallel phases do not broadcast the updated object because it has yet to be accessed by all processors. For both applications the difference in overall performance is caused by the difference in the distribution times for the updated object. Tables 11 and 12 contain the execution times for Water and String. Each run has the locality, replication and concurrent fetch optimizations turned on and the latency hiding optimization turned off. As expected, the adaptive broadcast optimization has a much larger impact on Water than on String. Above two processors neither Ocean nor Panel Cholesky ever accesses the same version of an object on all processors, so the adaptive broadcast algorithm has no effect at all on the execution above one processor. The single processor Ocean and Panel Cholesky runs expose a degenerate case in the adaptive broadcast algorithm — every processor involved in the computation (i.e. the one processor executing the application) has a copy of every object. The algorithm therefore generates a broadcast operation every time an object is updated, which degrades the performance. In practice we expect programmers to use the stripped version of the program described in Section 5.2.1 for their serial runs.

	1	2	4	8	16	24	32
Adaptive Broadcast	2435.16	1219.71	617.28	315.69	165.64	118.09	91.53
No Adaptive Broadcast	2459.87	1233.98	625.27	323.84	180.15	140.59	122.74

Table 11: Execution Times for Water on the iPSC/860 (seconds)

	1	2	4	8	16	24	32
Adaptive Broadcast	17382.07	9473.24	4773.02	2418.75	1249.69	873.14	678.55
No Adaptive Broadcast	18877.42	9469.36	4765.68	2425.82	1255.29	874.18	689.57

Table 12: Execution Times for String on the iPSC/860 (seconds)

	1	2	4	8	16	24	32
Adaptive Broadcast	77.44	68.14	28.75	18.77	24.16	37.18	51.87
No Adaptive Broadcast	63.14	65.54	28.73	19.11	25.68	39.99	55.71

Table 13: Execution Times for Ocean on the iPSC/860 (seconds)

	1	2	4	8	16	24	32
Adaptive Broadcast	54.56	50.18	31.56	32.50	34.41	36.38	38.17
No Adaptive Broadcast	37.25	49.76	31.29	32.01	34.92	35.87	38.16

Table 14: Execution Times for Panel Cholesky on the iPSC/860 (seconds)

5.4 Hiding Latency with Concurrency

In the current versions of Water, String and Ocean the programmer matches the amount of exposed concurrency to the number of processors executing the computation. There is no excess concurrency available to hide the latency of fetching remote objects. For Water and String simply generating a finer-grain task decomposition with more tasks than processors would not enable the implementation to effectively apply the optimization. Even with a finer-grain task decomposition all of the parallel tasks that execute on a given processor would still access the same remote objects, so the implementation would be unable to execute any of the multiple tasks assigned to a given processor until the remote objects required by all of the tasks on that processor arrived. The communication and computation would still execute serially. To enable the latency hiding optimization the programmer would have to artificially decompose both the objects and the tasks into finer-grain units. Decomposing the objects would impose substantial programming overhead because the programmer would have to change the object indexing algorithm used throughout the program. Even if the programmer generated finer-grain objects and tasks, the overall performance impact of the optimization would be negligible for Water and String — both applications already exhibit almost linear speedup.

For Ocean the limiting factor on the performance is the task management overhead. Because increasing the number of tasks would increase the task management overhead, a finer-grain object and task decomposition would degrade the overall performance of the application.

Panel Cholesky does generate more tasks than processors, and it may initially seem plausible that the optimization would have an effect on the performance. But turning the optimization on (setting the target number of tasks per processor to two) has virtually no effect on the performance. We believe there are several factors that may contribute to the lack of effect. First, there is an imbalance between the mean object transfer latency and the mean task execution time used to hide that latency — the mean object transfer latency is over twice the mean task execution time. Second, the task management overhead limits the number of outstanding executable tasks, which in turn limits the amount of exploitable excess concurrency. The inherent lack of concurrency in the application also limits the amount of excess concurrency available to hide latency. Finally, almost all of the tasks assigned to each processor access the same remote object. The vast majority of the tasks in the computation are external update tasks, which read a remote panel and update a local panel. The implementation creates all of the tasks the read a given remote panel in a consecutive group, and the tasks tend to become enabled as a group. Processors will therefore tend to get task sequences in which consecutive tasks read the same remote panel and update different local panels. Because none of the tasks that access the same remote panel can execute until it arrives, such task sequences limit opportunities to overlap communication and computation.

5.5 Concurrent Fetches

We evaluate the effectiveness of the concurrent fetch optimization by comparing the task latency (the sum over all tasks of the time between when the messages requesting remote objects were sent out and when the last requested object arrived) with the object latency (the sum over all object requests of the time between when the message requesting the object was sent out and when the object arrived). An object latency substantially larger than the task latency indicates that the implementation may have effectively parallelized the overhead required to fetch remote objects.

At the highest locality optimization level the ratio of the object latency to the task latency is very close to one for all applications, indicating that fetching objects concurrently fails to improve the communication behavior. An analysis of the applications reveals one reason why this happened. Almost all of the tasks in String, Ocean and Panel Cholesky fetch at most one remote object per communication point, and there is no opportunity to parallelize the communication. In Water almost all communication points fetch one large and one small object

from the same processor, which serializes the communication.

5.6 Discussion

Replication, which was required to enable concurrent execution in all of the applications, was by far the most important optimization. The adaptive broadcast optimization had no substantial effect on Ocean, Panel Cholesky and String, but did significantly improve the performance of Water. The locality optimizations only improved the performance of Ocean and Panel Cholesky, and for both applications the programmer had to explicitly place tasks to achieve the maximum performance. To interpret this result, it is important to understand that the programmer and the locality heuristic in the Jade implementation always agree on the target processor for every task except for Panel Cholesky on the iPSC/860. Even in this case the programmer and locality heuristic agree on the target processor over 90 percent of the time. It should therefore be possible to improve the Jade scheduler by making it less eager to move tasks off their target processors in an attempt to improve the load balance.

The latency hiding optimization had no effect, but we view this as a weak result, in part because the current implementation fails to exploit the full range of opportunities to overlap communication with computation. For example, the implementation does not transfer an object until a task actually declares that it needs the object to proceed. The implementation therefore fails to overlap computation and communication in cases where one task produces a shared object that another concurrently executing task will eventually consume. Ideally, the implementation would eagerly transfer the object from the producer to the consumer as soon as it was produced. The current implementation, however, does not perform the transfer until the consumer declares that it will access the object and waits for the object to arrive. Ocean contains such missed opportunities to overlap communication with computation.

Another limitation of the latency hiding optimization is that the Jade implementation communicates at the granularity of shared objects. This communication strategy prevents the implementation from exploiting the finer-grain opportunities to hide latency that arise when tasks traverse large objects. In such cases it is possible to overlap the computation that accesses one part of an object with the communication required to fetch other parts. Because the Jade implementation performs all communication at the granularity of complete shared objects, it does not exploit this potential source of latency hiding.

Despite its lack of applicability in the current application set, we believe it is appropriate to include the concurrent fetch optimization in future implementations. It fits naturally into the execution model, should in practice never impair the performance, and may prove useful for some other applications.

6 Related Work

Chandra, Gupta and Hennessy [6] have designed, implemented and measured a scheduling algorithm for the parallel language COOL running on DASH. The goal of the scheduling algorithm is to enhance the locality of the computation while balancing the load. COOL provides an affinity construct that programmers use to provide hints that drive the task scheduling algorithm. The programmer can specify object affinity, which tells the scheduler to attempt to execute the task on the processor that owns the object, task affinity, which allows the programmer to generate groups of tasks that execute consecutively on a given processor, and processor affinity, which allows the programmer to directly specify a target processor for the task. The behavior of the COOL scheduler on programs specifying object affinity roughly corresponds to the behavior of the Jade scheduler using the locality heuristic from Section 3.2.1; with processor affinity the behavior roughly corresponds to explicit task placement. COOL versions of Panel Cholesky and Ocean running on the Stanford DASH machine with object affinity and no affinity hints exhibit performance differences that roughly resemble the performance differences that we observed for the Jade versions of these applications running at the different locality optimization levels.

There are several major differences between the Jade research presented in this paper and the COOL research. COOL programs do not contain enough information about the concrete pieces of data that the tasks will access for the implementation to apply any communication optimization except locality optimizations. The Jade implementation, on the other hand, has a complete, concrete specification of which objects each task will access and how it will access them. It can therefore automatically apply a full range of optimizations, including replication, broadcast, concurrent fetches and using excess concurrency to hide latency.

Second, COOL is not portable: it only runs on shared memory machines. Jade runs on a wide variety of hardware platforms, including both shared memory and message passing machines. This paper evaluates the performance impact of a full range of communication optimizations for a set of complete applications running on both shared memory and message passing platforms. The research presented in [6] only evaluates locality optimizations on a shared memory platform.

Fowler and Kontothanassis [9] have developed an explicitly parallel system that uses object affinity scheduling to enhance locality. The system associates objects with processors. Each task and thread has a location field that specifies the processor on which to execute the task or thread. Under affinity scheduling the location field is set to the processor associated with one of the objects that the task or thread will access. This system differs from the COOL affinity scheduler and the Jade locality heuristic in that it has a single FIFO queue per processor. There is no provision to avoid interleaving the execution of multiple tasks with different affinity objects.

SAM [22] provides the abstraction of a single shared object store on message passing machines. The SAM implementation automatically replicates data for concurrent read access; experimental results from SAM applications illustrate the importance of this optimization [22]. SAM also provides lower-level primitives that programmers can use to explicitly control the movement of data. These primitives allow programmers to overlap communication and computation by asynchronously prefetching data and eagerly transferring data from producing to consuming processors. The major philosophical difference between SAM and Jade in relation to communication optimizations is that the Jade implementation automatically applies communication optimizations while SAM provides constructs that allow programmers to directly control the movement of data.

There are several more subtle differences between the communication optimizations that SAM supports and those that the Jade implementation automatically performs. Prefetching is a single mechanism that allows a SAM programmer to both parallelize communication (by issuing several prefetches in a row) and overlap computation and communication. As described in Sections 3.4.1 and 3.4.3, the Jade implementation uses a different mechanism for each optimization.

SAM allows programmers to overlap communication and computation and eliminate fetch latency by eagerly transferring data from a producer to a single consumer — there is no support for broadcast. By contrast, the *only* standard Jade optimization that eagerly transfers data from producers to consumers is the adaptive broadcast optimization. Although we have built a Jade implementation that uses an update protocol to eagerly transfer data from producers to potential consumers, this implementation did not generate uniformly positive results. While the protocol worked well for applications such as Water and String with regular, repetitive communication patterns, it degraded the performance of other applications by generating an excessive amount of communication.

Tempest [8, 12] is a collection of communication mechanisms designed for implementation on a range of architectures. Programmers can apply communication optimizations by selecting an appropriate communication mechanism from a library or by coding their own custom protocol. Munin [5] is a page-based distributed shared memory system that provides several different coherence mechanisms. The programmer selects a coherence protocol for each piece of data; the idea is that the selected coherence protocol is the most efficient for the access pattern of that piece of data. Both systems support a minimal tasking model: each processor has a single thread of execution that it runs from the beginning until the end of the computation.

The major philosophical difference between these two systems and Jade with respect to communication optimizations is that these systems provide low-level mechanisms that the programmer can use directly to optimize the communication, while Jade encapsulates a set of policy decisions to automatically apply a range of communication optimizations. Jade also differs from these systems in that it has a mature task model with an associated task management subsystem. Supporting tasks as first-class entities in the execution model enables the Jade implementation to automatically apply optimizations (for example, the locality and latency hiding optimizations) that require the coordination of the communication and task management algorithms.

Mowry, Lam and Gupta have evaluated the performance impact of prefetching in the context of a shared memory multiprocessor that transfers data at the granularity of cache lines. The prefetch instructions are inserted either directly by the programmer [15], or, for statically analyzable programs, by the compiler [16]. We see this communication optimization as orthogonal to the Jade communication optimizations. It takes place at a fine granularity, with communication operations that transfer a single cache line typically issued every few iterations of a loop. The Jade communication optimizations, on the other hand, are designed to operate with coarse-grain tasks that access large shared objects.

Olden [4] and Prelude [13] attempt to improve locality using computation migration. When a piece of computation accesses a remote piece of data, it may migrate to the processor that owns the data. Olden also contains a data movement mechanism that copies data to accessing processors instead of moving computation to data. Olden chooses between data and computation migration based on path affinities, which allow the programmer to specify the probability that data reached via a given pointer will be local or remote. Both the Olden and Prelude programming models are built on the assumption that tasks access one object at a time. Jade is built on the assumption that tasks may tightly interleave their accesses to multiple objects. The Jade programming model therefore allows programmers to declare that each task may access multiple objects, which in turn allows the Jade implementation to fetch the remote objects in parallel and delay the execution of a task until all of the objects are available locally. The more advanced Jade constructs also support pipelined access to objects [17].

Parallelizing compilers for message passing machines analyze a program written in a serial programming language and generate parallel code [1]. Part of the compilation task entails the generation of message passing operations that transfer data produced on one processor to all of the processors that consume the data. Because the generated code communicates the data as soon as it is produced, it may overlap the communication with computation at the consumer. The main focus of this research is on programs that exhibit regular, statically analyzable loop-level concurrency. Jade is designed for irregular programs that exhibit more complex task-level concurrency.

7 Conclusion

This paper summarizes our experiences automatically applying communication optimizations in the context of Jade. We showed how the Jade language design gives the implementation advance notice of how tasks will access data, and described how Jade implementations can exploit this advance notice to apply communication optimizations. We presented experimental results characterizing the performance impact of the optimizations on two computational platforms: a shared memory machine (the Stanford DASH multiprocessor) and a message passing machine (the Intel iPSC/860). The most important optimization was replication — without this optimization the applications would all execute serially. Locality optimizations had a significant impact on two of the four applications, and the adaptive broadcast optimization had an effect on one of the applications.

These results inject a note of caution into the discussion of communication optimizations for parallel computing. While it is important to obtain experience with a wider range of applications before drawing definitive conclusions, our experience suggests that several of the more esoteric communication optimizations may not in practice deliver performance improvements. The results in this paper, combined with results from additional applications, should allow implementors to analyze potential performance gains before investing the effort required to implement complex communication optimization algorithms.

Acknowledgements

This research was supported in part by a fellowship from the Alfred P. Sloan Foundation and by DARPA contracts DABT63-91-K-0003 and N00039-91-C-0138. The author would like to thank the anonymous reviewers for their many useful comments and suggestions.

References

[1] S. Amarasinghe and M. Lam. Communication optimization and code generation for distributed memory machines. In *Proceedings of the SIGPLAN '93 Conference on Program Language Design and Implementation*, Albuquerque, NM, June 1993.

[2] H. Bal, M. Kaashoek, and A. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3), March 1992.

[3] R. Berrendorf and J. Helin. Evaluating the basic performance of the Intel iPSC/860 parallel computer. *Concurrency: Practice & Experience*, 4(3):223–240, May 1992.

[4] M. Carlisle and A. Rogers. Software caching and computation migration in Olden. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.

[5] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, October 1991.

[6] R. Chandra, A. Gupta, and J. Hennessy. Data locality and load balancing in COOL. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993.

[7] I. Duff, R. Grimes, and J. Lewis. Sparse matrix problems. *ACM Transactions on Mathematical Software*, 15(1):1–14, March 1989.

[8] B. Falsafi, A. Lebeck, S. Reinhardt, I. Schoinas, M. Hill, J. Larus, A. Rogers, and D. Wood. Application-specific protocols for user-level shared memory. In *Proceedings of Supercomputing '94*, November 1994.

[9] R. Fowler and L. Kontothanassis. Improving processor and cache locality in fine-grain parallel computations using object-affinity scheduling and continuation passing. Technical Report 411, Dept. of Computer Science, University of Rochester, June 1992.

[10] R. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[11] J. Harris, S. Lazaratos, and R. Michelena. Tomographic string inversion. In *60th Annual International Meeting, Society of Exploration and Geophysics, Extended Abstracts*, pages 82–85, 1990.

[12] M. Hill, J. Larus, and D. Wood. Tempest: A substrate for portable parallel programs. In *Proceedings of the 1995 Spring COMPCON*, March 1995.

[13] W. Hsieh, P. Wang, and W. Weihl. Computation migration: Enhancing locality for distributed-memory parallel systems. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993.

[14] D. Lenoski. *The Design and Analysis of DASH: A Scalable Directory-Based Multiprocessor*. PhD thesis, Stanford, CA, February 1992.

[15] T. Mowry and A. Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87–106, June 1991.

[16] T. Mowry, M. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, Boston, MA, October 1992.

[17] M. Rinard. *The Design, Implementation and Evaluation of Jade, a Portable, Implicitly Parallel Programming Language*. PhD thesis, Stanford, CA, 1994.

[18] M. Rinard and M. Lam. Semantic foundations of Jade. In *Proceedings of the Nineteenth Annual ACM Symposium on the Principles of Programming Languages*, pages 105–118, Albuquerque, NM, January 1992.

[19] M. Rinard, D. Scales, and M. Lam. Heterogeneous Parallel Programming in Jade. In *Proceedings of Supercomputing '92*, pages 245–256, November 1992.

[20] M. Rinard, D. Scales, and M. Lam. Jade: a high-level, machine-independent language for parallel programming. *Computer*, 26(6):28–38, June 1993.

[21] E. Rothberg. *Exploiting the memory hierarchy in sequential and parallel sparse Cholesky factorization*. PhD thesis, Stanford, CA, January 1993.

[22] D. Scales and M. S. Lam. The design and evaluation of a shared object system for distributed memory machines. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation*, Monterey, CA, November 1994.

[23] J. Singh, W. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared memory. *Computer Architecture News*, 20(1):5–44, March 1992.

A The iPSC/860

The iPSC/860 is a distributed-memory machine consisting of i860 processing nodes interconnected by a hypercube network. Each processing node consists of a 40MHz i860 XR processor with a 4KByte instruction cache and an 8KByte write-back data cache, each with 32-byte lines. Each node can deliver a maximum of 40 MIPS and 30 double-precision MFLOPS. (There are special dual-operation instructions that allow a floating-point add and multiply to execution in parallel, thus raising the peak rate to 60 MFLOPS, but these instructions are not used in our compiled code.)

The processing nodes are connected by a hypercube network that scales from 8 to 128 processors in powers of 2. The network is circuit-switched and provides a bandwidth of 2.8 megabytes per second between neighboring nodes. Jade uses the NX/2 message-passing library on the iPSC/860, which provides primitives for sending arbitrary-sized messages between any two nodes. The NX/2 library automatically does buffering on the send and receive sides. We have measured a minimum time to send a short message as 47 microseconds.

B The Stanford DASH Machine

The Stanford DASH machine[14] is a cache-coherent shared-memory multiprocessor. It uses a distributed directory-based protocol to provide cache coherence. It is organized as a group of processing clusters connected by a mesh interconnection network. Each of the clusters is a Silicon Graphics 4D/340 bus-based multiprocessor. The 4D/340 system has four processing nodes, each of which contains a 33MHz R3000 processor, a R3010 floating-point co-processor, a 64KByte instruction cache, 64KByte first-level write-through data cache, and a 256KByte second-level write-back data cache. Each node has a peak performance of 25 VAX MIPS and 10 double-precision MFLOPS. Cache coherence within a cluster is maintained at the level of 16-byte lines via a bus-based snoopy protocol. Each cluster also includes a directory processor that snoops on the bus and handles references to and from other clusters. The directory processor maintains directory information on the cacheable main memory within that cluster that indicates which clusters, if any, currently cache each line.

The interconnection network consists of a pair of wormhole routed meshes, one for request messages and one for replies. The total bandwidth in and out of each cluster is 120 megabytes per second. The latencies for read accesses to shared data in DASH vary depending on the current state of the data in the caches of the local processor, the processors in the local processor, and the processors in the remote clusters. A processor takes 1, 15, and 29 cycles to access data that is in its first-level cache, in its second-level cache, or in the cache of another processor in the cluster, respectively. If the data is not available in the local cluster, a request must be made to the home cluster of the data, causing a latency of 101 cycles if there is no contention. If the data is dirty in a third cluster, the request must be forwarded to that cluster and the latency is 132 cycles.

Biographical Sketch

Martin Rinard is an Assistant Professor in the Department of Computer Science at the University of California at Santa Barbara. He received the Sc.B. in Computer Science, Magna cum Laude and with Honors, from Brown University in 1984. He received the Ph.D. in Computer Science from Stanford University in 1994. Professor Rinard is a member of the Parallel Computing Group at UCSB. His main research interests are in parallelizing compilers, parallel programming languages and parallel and distributed systems. In 1995 Professor Rinard was selected as a Sloan Research Fellow.

Professor Rinard's World Wide Web home page can be found at the URL http://www.cs.ucsb.edu/ martin/.