Locality 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 improve locality by executing tasks close to data that they will access may improve the performance of parallel computations. This paper describes our experience automatically applying locality 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 locality optimizations. We present performance results for several Jade applications running on the Stanford DASH machine. We use these results to characterize the overall performance impact of the locality optimizations. In our application set the locality optimization level has little effect on the performance of two of the applications and a large effect on the performance of the rest of the applications. We also found that, if the locality optimization level had a significant effect on the performance, the maximum performance was obtained when the programmer explicitly placed tasks on processors rather than relying on the scheduling algorithm inside the Jade implementation.

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. Improving the locality of the computation by executing tasks close to the data that they will access can improve the communication behavior of the program by reducing the amount of remote data that tasks access.

This paper describes our experience automatically applying locality optimizations in the context of Jade [10, 11], 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 a locality heuristic. This heuristic attempts to enhance the locality of the computation by scheduling tasks on processors close to the data they will access.

This paper presents the algorithm that implements the locality heuristic. It also evaluates the performance impact of the locality heuristic by presenting the results obtained by executing several complete Jade applications. The collected data allow us to characterize the performance impact on this set of applications. Our execution platform is the Stanford DASH machine [5].

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 locality 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 locality optimization algorithm. In Section 4 we describe the Jade applications. In Section 5 we present the performance results. 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 [8, 9, 11]. 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 [8]. These language features allow programmers to express more advanced concurrency patterns with multiple synchronization points within a single task.

3. Locality Optimization Algorithm. Access specifications give the Jade implementation advance information about how each task will access data. The locality heuristic exploits this advance information to optimize the communication. We next present an overview of the Jade implementation for shared memory machines such as the Stanford DASH multiprocessor and describe how the locality heuristic is integrated into the implementation.

3.1. Implementation Overview. 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 [8]. 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. Locality Optimization. We have implemented several variants of the locality heuristic, each tailored for the different memory hierarchies of different machines [8]. 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.

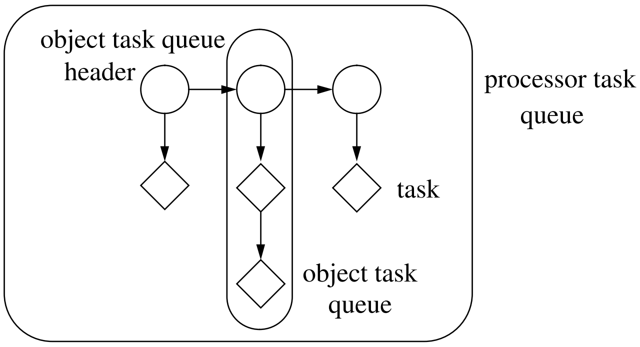

Fig. 1. *Task Queue Data Structures*

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. The goal is to satisfy the task's references to the locality object 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 at that processor. 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. The goal is to satisfy the task's accesses to its locality object 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.

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 [4], 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 [13] 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[2]. 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 the locality heuristic. Each experiment isolates the effect of the hueristic on a given application by running the application first with the optimization turned on then with the optimization turned off. In each case we report results for the applications running on 1, 2, 4, 8, 16, 24 and 32 processors. 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 implementation uses the scheduling algorithm described in Section 3.2.1.
- **No Locality:** The implementation distributes enabled tasks to idle processors in a first-come, first-served manner using a single shared task queue.

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.

Our evaluation of the locality optimizations 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.

5.1. Task Locality Percentage. Figures 2 through 5 plot the *task locality percentage* for the different applications 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.

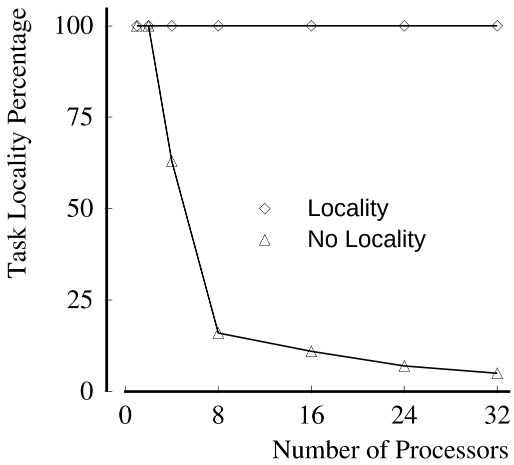

FIG. 2. *Percentage of Tasks Executed on the Target Processor for Water*

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

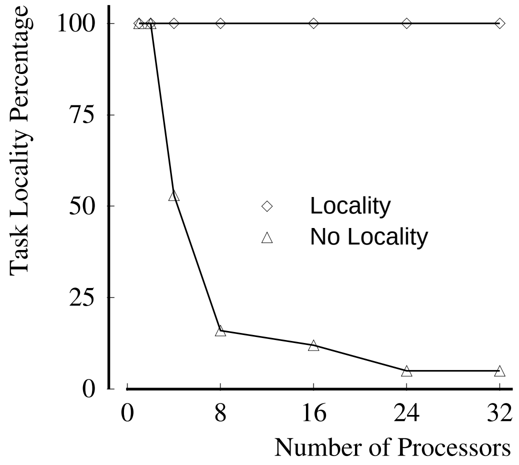

FIG. 3. *Percentage of Tasks Executed on the Target Processor for String*

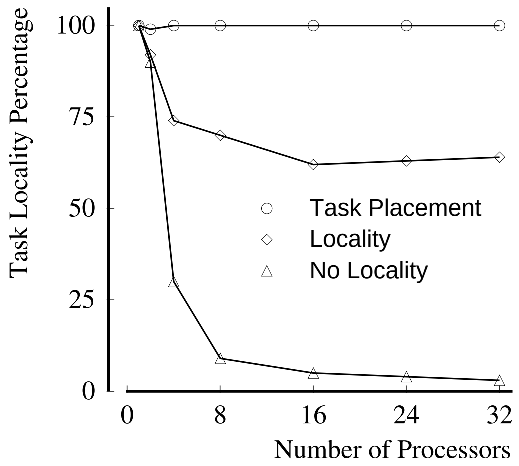

FIG. 4. *Percentage of Tasks Executed on the Target Processor for Ocean*

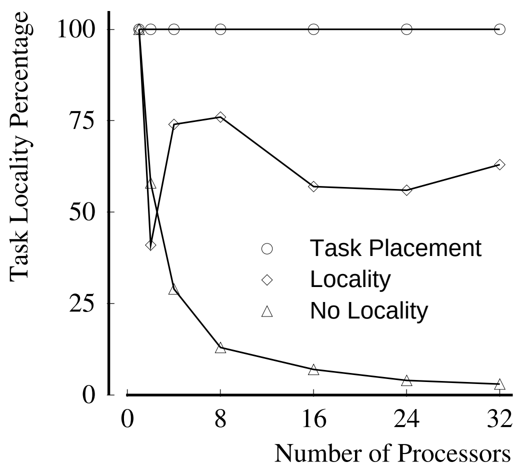

FIG. 5. *Percentage of Tasks Executed on the Target Processor for Panel Cholesky*

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. As expected, at No Locality the task locality percentage drops quickly as the number of processors increases for all applications.

5.2. Communication Behavior. 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 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[5] 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.

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 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.

5.3. Load Balancing. For Ocean and Panel Cholesky, the dynamic load balancer moves a significant number of tasks off their target processors in an attempt to improve the load balance. We evaluate this strategy by presenting the *total idle time* of each of these two applications. When the load becomes more balanced, the total idle time should decrease. Figures 10 and 11 present the idle time measurements for Water and Panel Cholesky. These figures show that the dynamic load balancer did not succeed in improving the load balance - in fact, the general trend is that moving tasks off of their target processors actually increases the idle time. The decrease in locality associated with moving tasks off of their target processors increases the execution times of the moved

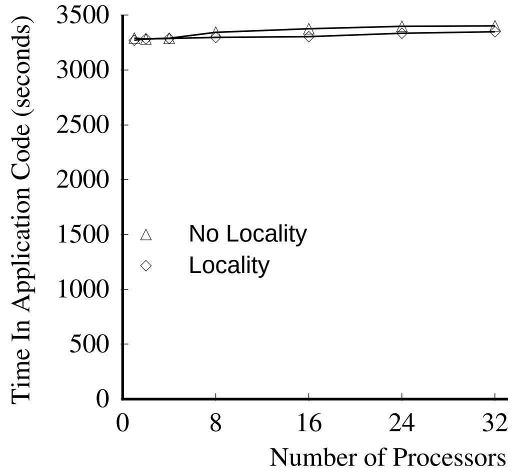

FIG. 6. *Total Task Execution Time for Water*

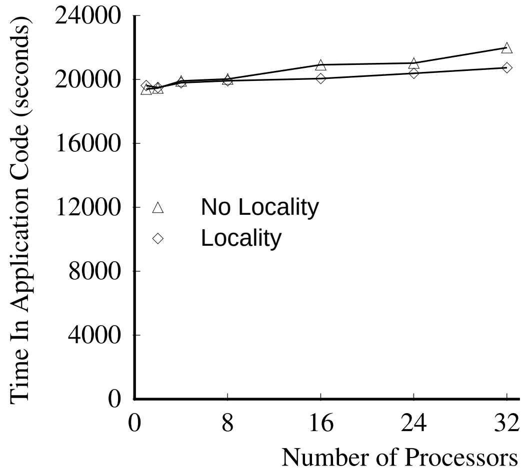

FIG. 7. *Total Task Execution Time for String*

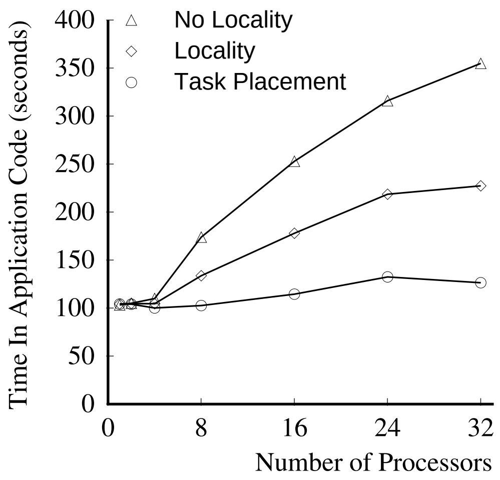

FIG. 8. *Total Task Execution Time for Ocean*

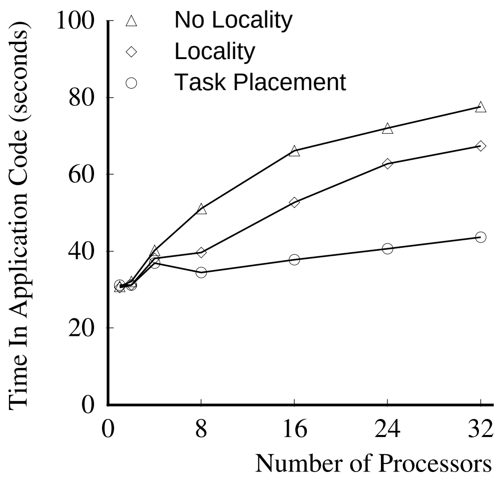

FIG. 9. *Total Task Execution Time for Panel Cholesky*

tasks, which in turn increases the length of the critical path. Section 5.5 discusses the reasons behind this anomaly in more detail.

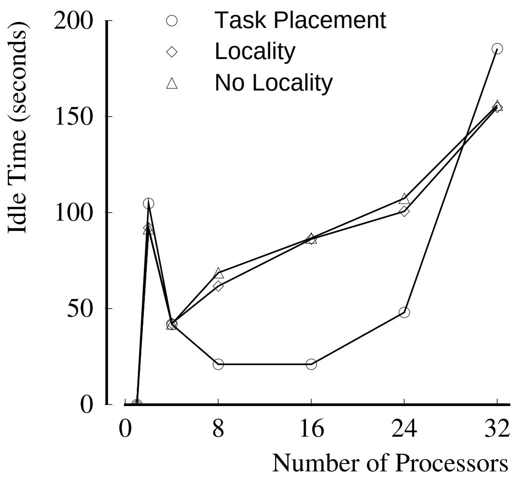

FIG. 10. *Total Idle Time for Ocean*

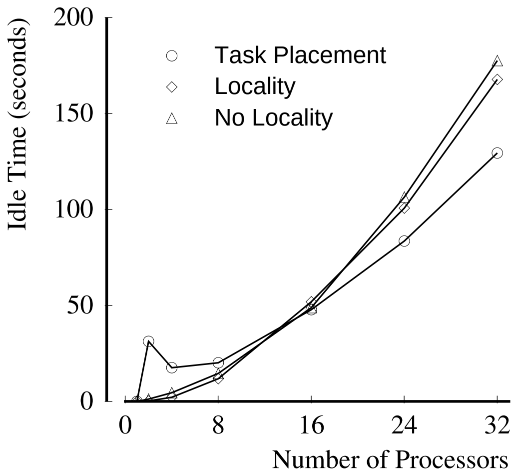

FIG. 11. *Total Idle Time for Panel Cholesky*

5.4. Execution Times. 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.

Tables 2 through 5 present the execution times for the Jade versions 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.

	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 (seconds)

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 (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 (seconds)

5.5. Performance Analysis for Ocean. 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. Figure 12 plots the task management percentages for Ocean. 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.

The task management percentages for Ocean also provide insight into the task locality percentages presented in Section 5.1 and the idle time measurements presented in Section 5.3. Recall that even though the dynamic load balancer moves a significant number of tasks off their target processors in an attempt to balance the load, this task movement does not systematically decrease the idle time — the overall idle time for the Locality version of Water is usually greater than for the Task Placement version. We attribute this behavior to the following properties of the computation. Ocean creates many small tasks, and especially for larger numbers of processors the machine can, in general, execute tasks faster than they are created. So the computation tends to run in a concurrency-starved state in which there are always idle processors. Even though the locality heuristic makes the scheduler generate a balanced distribution of tasks to processors, small perturbations in the execution times of the tasks create transient imbalances. In the main computationally intensive section of the computation, for example, the scheduler distributes tasks to processors in a well balanced, round-robin fashion. But variations in the task execution times may lead to situations in which one processor has yet to finish its last task while other processors are idle. When the next task for the busy processor is created, it will be executed on one of the

	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 (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 (seconds)

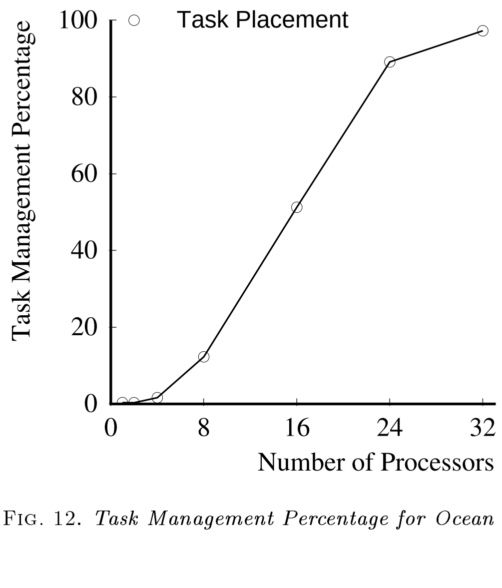

FIG. 12. *Task Management Percentage for Ocean*

idle processors even though a task will soon be created whose target processor is the idle processor. Each such premature task movement typically generates a cascading sequence of task movements as newly created tasks find their target processors busy. The net effect is a decrease in locality. Furthermore, the corresponding increases in the execution times of the moved tasks increase the length of the critical path of the computation.

5.6. Performance Analysis for Panel Cholesky. For Panel Cholesky several factors combine to limit the performance, among them an inherent lack of concurrency in the basic parallel computation [12] and the task management overhead, which lengthens the critical path[8]. Figure 13 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.

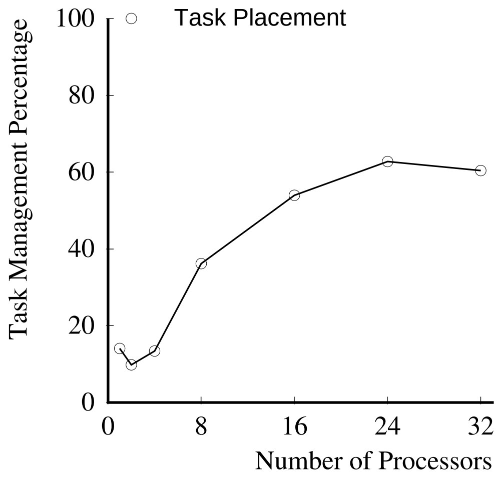

FIG. 13. *Task Management Percentage for Panel Cholesky*

The task management percentages for Panel Cholesky also provide insight into the task locality percentages presented in Section 5.1 and the idle time measurements presented in Section 5.3. For reasons similar to those discussed in Section 5.5, moving tasks decreases the locality of the computation without decreasing the idle time.

5.7. Discussion. The Water and String applications perform well regardless of the locality optimization level - they have a very small communication to computation ratio and improvements in the communication behavior to not translate into improvements in the overall performance. The locality optimizations did improve the overall performance of both Ocean and Panel Cholesky, but 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. These results suggest that it should 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.

6. Related Work. Chandra, Gupta and Hennessy [1] 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 is an explicitly parallel language, and the annotations in COOL programs are designed solely to allow the programmer to influence the scheduling algorithm. The annotations in Jade programs, on the other hand, provide a complete specification of which objects each task will access and how it will access them. The Jade implementation uses this information for multiple purposes: both to extract the concurrency and to apply locality optimizations.

Fowler and Kontothanassis [3] 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.

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 [6], or, for statically analyzable programs, by the compiler [7]. 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.

7. Conclusion. This paper summarizes our experiences automatically applying locality 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 locality optimizations. We presented experimental results characterizing the performance impact of the optimizations on the Stanford DASH multiprocessor.

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.

REFERENCES

[1] 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.

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

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

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

[9] 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.

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

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

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

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