Data Locality in Large Scale Multiprocessors

Jim P. Sorgo

Supervisor: Derek L. Eager

Abstract

In a large scale multiprocessor, accessing data stored in a remote memory can take orders of magnitude longer than accessing data stored in local memory. Good data locality is achieved when a program has a minimal number of accesses to data stored in remote memories. This paper is a survey of methods to achieve good data locality in programs written in imperative programming languages for large scale MIMD multiprocessors.

The scope of the survey is limited to systems that offer the programmer a shared-memory programming paradigm, as distinguished from a message passing programming paradigm.

1 Introduction

In this paper, a *large scale multiprocessor* is a MIMD machine with the following architecture:

- It consists of many *processing nodes*, where each of these *nodes* is a uniprocessor, i.e. each node is a processor and cache connected to a memory via a bus.
- The nodes are connected by some interconnection network.

Since there are many nodes, with each node containing a memory, an obvious way to organize the memories is as a mirror image of the architecture: a separate address space for each node. This is how the memories are organized in a *multiple address space* large scale multiprocessor; the compiler sees a separate address space for each node. An example multiple address space machine is a network of workstations.

Currently, the most popular way to program a multiple address space machine is with message passing. However, the message passing programming paradigm (or any other architecture dependent programming paradigm such as in [17]) is not considered in any part of this paper; the scope of this paper is limited to the shared-memory programming paradigm. Programs are easier to write and maintain in the shared-memory programming paradigm than in the message passing programming paradigm. For a multiple address space machine to offer a shared-memory programming paradigm, however, the compiler must convert the multiple address spaces into a single address space.

A *single address space* large scale multiprocessor, on the other hand, automatically offers a shared-memory programming paradigm because either the hardware or operating system takes the multiple memories of the nodes and organizes them into a single, large, global address space. Thus, the compiler sees a single address space. Example single address space machines are the KSR-1 [9] and DASH [11]. These are both hardware implemented single address space machines as opposed to operating system implemented single address space machines.

In a large scale multiprocessor, memory references are either local memory references or remote memory references. A local memory reference is the processor of node i accessing data that is stored in the memory of node i. A remote memory reference, on the other hand, is a processor accessing data that is stored in the memory of another node. In a remote memory reference, the data travels across the interconnection network from the remote node to the processor's own node.

Remote memory references take much longer to complete than do local memory references. In fact, even in single address space machines a remote memory reference can take orders of magnitude longer to complete than a local memory reference. Communication times across the system interconnection network are a bottleneck in modern large scale multiprocessors [13].

Good data locality is achieved when a program has a minimal amount of remote memory references. This paper is a survey of methods to achieve good data locality in programs written in imperative programming languages for large scale multiprocessors.

The rest of this paper is organized as follows:

- Section 2: Locality through Blocking
- Section 3: Locality through Object Placement and Method Scheduling
- Section 4: Locality through Loop Scheduling
- Section 5: Locality through Array Allocation and Owner Computes
- Section 6: Future Work — Locality through Adaptive Prefetching/Poststoring
- Section 7: Conclusions

2 Locality through Blocking

In blocking, a single "unit of computation" (e.g., loop iteration) operates on an entire block of data. Even though this block of data may have to be fetched from a remote memory, the cost of doing so is amortized because the elements of the block are reused many times while the block is cached locally. Thus, there may not be a need for the programmer to tune the distribution of data or computation; it may not matter on which nodes data resides because the cost of remote memory accesses are amortized.

An example study of blocking is that by Lam *et al.* [10], who present a blocked algorithm for matrix multiply. Their performance experiments were run on a uniprocessor; however, blocking is a technique that can be applied to any type of large scale multiprocessor.

This algorithm includes a *blocking factor*. Performance is highly dependent on the problem size and the choice of blocking factor. The same blocking factor can yield widely varying cache miss rates for very similar problem sizes. Suggestions as to how to choose the blocking factor are offered in [10].

3 Locality through Object Placement and Method Scheduling

Parallel programming using an object-oriented paradigm is attractive not only because object-oriented programming is popular, but also because the goal of achieving data locality is facilitated by the object-oriented programming paradigm.

3.1 COOL

COOL [2, 3] is an acronym for "Concurrent Object-Oriented Language." It has been studied in the context of a hardware implemented single address space large scale multiprocessor. COOL is C++ supplemented with some additional abstractions. There are abstractions to express each of the following three elements of parallel programs:

- parallelism
- the two basic elements of synchronization — mutual exclusion and event sychronization
- data locality

3.1.1 Parallelism and Synchronization

To express parallelism, there is only a single abstraction, *parallel*. In the function declaration, a programmer simply puts the keyword *parallel* in front of the function name. Many instances of such a function can execute simultaneously and asynchronously. Each instance of the function can be thought of as a *task*.

For example, if the function "laplace" is a *parallel* function, then the following code creates a million tasks that can execute simultaneously and asynchronously:

```
for (j = 0; j < 1000000; j++)
    region[j].laplace;
```

The main way of expressing mutual exclusion is to declare a parallel method to be either *mutex* or *nonmutex*. The main abstraction for event synchronization is the condition variable. These abstractions fit nicely into the language.

3.1.2 Data Locality

The abstractions most relevant in this report, however, are those that express data locality. These abstractions are referred to as *affinity hints* in [2, 3]. Affinity hints are optional since they only affect scheduling decisions; affinity hints do not affect semantics.

In "default object affinity scheduling", the COOL runtime system schedules a task on the processing node that contains in its local memory the object that the task is invoked upon. If there are several tasks invoked upon the object, then COOL also uses "default *task* affinity scheduling." Because of default object affinity scheduling, all these tasks will be scheduled on the same node. The default task affinity scheduling policy executes these tasks *back to back* to achieve cache locality. The programmer can override default object affinity scheduling with an object affinity hint. The runtime system then schedules the same as it does with the above described default object affinity scheduling except now the affinity is for an explicitly specified object rather than for the object the task is invoked upon. Similarly, the programmer can override default task affinity scheduling with a task affinity hint. This task affinity hint specifies an object. Back to back execution of tasks that reference that object occurs (for tasks that are all scheduled on the same node).

In the case where a task operates on only a single object, an affinity hint is not necessary; the default affinity can work well. For example, in the above counted loop the runtime system knows that the *jth* "laplace" task is invoked upon the *jth* "region" object. By default, the runtime system schedules the *jth* "laplace" task on the processor that contains the *jth* "region" object in its local memory. If the *jth* "region" object is the only data object that the *jth* "laplace" task operates on then data locality is achieved because when a memory reference misses in the cache, the cache is filled from local memory rather than from the memory of a remote node.

If all the nodes of the machine contain an equal number of "region" objects in their respective local memories, then the runtime system schedules an equal number of "laplace" tasks on each processor. This results in good load balance, provided all the "laplace" tasks have the same execution times.

The job of distributing the "region" objects among the nodes of the machine, however, is left to the programmer. In COOL, the programmer is always burdened with mapping data to the physical topology of the machine. This is unfortunate from the point of view of ease of programming, especially when a task operates on more than one data object.

Additionally, the programmer has to make decisions about granularity. The choice of "what is an object" may be dictated more by the fact that the object is the unit of distribution and parallelism, rather than by the inherent structure of the problem. For example, the inherent structure of some problem may make it natural to select an object to be large. Unfortunately, a large object can make it difficult to write a COOL program with all three of parallelism, data locality, and load balance: the object is the unit of distribution, so the large object cannot be distributed among more than one node. Thus, the programmer may be forced into choosing something smaller to be an object, even though this smaller entity may be unnatural with respect to the inherent structure of the problem.

In the case where a task operates on more than one object, and these objects are not all located on the same processor, the task has affinity for more than one processor.[1] The programmer must understand which objects a parallel function operates on, how the function operates on these objects, and how these objects are distributed among the nodes of the machine. The programmer then uses this understanding to decide which processor a task has *the most*, or at least *a lot of*, affinity for and then gives the proper affinity hint to make the runtime system schedule tasks as per his/her decision.

3.2 Others

Fowler et al. present a concurrent object-oriented language in [6]. Their language is not as advanced as COOL, however. For one thing, their language is studied in the context of a small scale multiprocessor.

4 Locality through Loop Scheduling

Loops are the largest source of parallelism in most applications. Loop scheduling distributes the iterations of a loop among many processors thereby allowing the computation of iterations to proceed in parallel. All the methods of achieving locality through loop scheduling that are presented in this section have been studied in the context of hardware implemented single address space machines.

4.1 Some Fundamentals of Loop Scheduling

There are both static and dynamic approaches to loop scheduling. In a static approach to loop scheduling, the distribution of loop iterations among the available processors is decided during compilation. Sometimes it is impossible for the compiler to determine the relative amounts of work contained within the loop iterations (i.e. whether some iterations will take longer to execute than other iterations).[2] At these times the best a compiler can do using static scheduling is to give each available processor the same number of iterations and hope that each processor receives a similar amount of work. If, however, some iterations do in fact take longer to execute than other iterations, then each processor may not receive a similar amount of work and load imbalance will probably arise. *Load imbalance* means that some processors are busy executing iterations while other processors are idle for lack of work. Another problem with static scheduling is that it is not robust with respect to variations in the number of available processors.

In a dynamic approach to loop scheduling, iterations are distributed among the available processsors during run-time [5, 12, 14, 15, 16]. Dynamic loop scheduling methods address the above mentioned problems with static scheduling. A dynamic loop scheduling method has either a single, central work queue or many, distributed work queues. Basically, whenever a processor is idle, it removes a chunk of iterations from a work queue and then executes those iterations. This continues until all iterations have executed.

Ideally, the loop iterations get distributed among processors in a way that not only results in good load balance, but also minimizes both scheduling overhead and communication overhead.

Scheduling overhead is processor cycles spent on scheduling operations. A *scheduling operation* is a processor removing a chunk of iterations from a work queue. Fewer scheduling operations does not necessarily mean less scheduling overhead because some scheduling operations are more expensive than others. For example, a scheduling operation in which there is contention for a lock protecting the work queue is more expensive than one without contention.

Communication overhead occurs when a memory reference misses in the cache and the required cache block is located in the memory of a remote node rather than in local memory. Communication

[1] Presumably, the default affinity usually does not work very well in this case as [2, 3] usually does not give results for when the default affinity is used in this case.

[2] These are the times when the amount of work contained by an iteration depends either on data read at run-time or on expressions that are too complicated for the compiler to evaluate.

overhead is the cycles a processor spends idling between when it issues the reference and when the data finally arrives in the cache.

The goals of load balance and low scheduling overhead conflict with each other. The best possible load balance is achieved when any processor finishes within one iteration of any other processor. To achieve this best possible load balance, even when some iterations take much longer to execute than other iterations, simply make the chunk size equal to one. A processor always removes just a single iteration from a work queue. This maximizes the number of scheduling operations, however.

Guided self-scheduling (gss) [5, 16] is a dynamic loop scheduling method that changes the size of chunks at runtime. It allocates large chunks of iterations at the beginning of a loop so as to reduce the number of scheduling operations, and then allocates small chunks towards the end of the loop to balance the workload. As the loop executes, the chunks get progressively smaller until near the end of the loop, a chunk is just a single iteration. This is accomplished by having a processor take some constant fraction (e.g. 1/(2P) where P is the number of processors) of the remaining iterations to be executed.

4.2 Minimizing Communication Overhead using Affinity Scheduling

4.2.1 A Single Parallel Loop

Like gss, the affinity scheduling algorithm of Marketos and LeBlanc [12, 14, 15] attempts to minimize scheduling overhead while achieving load balance. Unlike gss, however, this affinity scheduling algorithm also addresses the goal of minimizing communication overhead (maximizing data locality).

A limitation on the algorithm is that it requires a parallel loop to be nested within a sequential loop. It also requires an iteration, say iteration j, of the nested parallel loop to access the same data each time the parallel loop executes. The basic idea is that the first time the parallel loop executes, the data for iteration j will be loaded into a cache. The next time the parallel loop executes, iteration j will have affinity for the processor on which it last ran because that processor's cache will contain the data for iteration j (provided the cache is sufficiently big such that the data has not been flushed).

The first part of the algorithm uses a static approach to loop scheduling: if there are P processors, then divide the iterations of the nested parallel loop into P equally sized chunks and allocate a chunk to each processor. This first part of the algorithm is the part that addresses the goal of achieving data locality. It does so by always placing the *ith* chunk on the local work queue of processor i. This way, every time the parallel loop executes (i.e. for each iteration of the outer sequential loop), each processor receives the same set of iterations to execute. This doesn't necessarily mean that processor i executes all the iterations in the *ith* chunk, however; there is a dynamic aspect to this algorithm as well.

The dynamic aspect is to allow load sharing. When a processor is idle, it removes a chunk of iterations from its local work queue. As in gss, chunks start out large and then progressively get smaller. The load sharing aspect is as follows: if a processor's own local work queue is empty, the processor takes a chunk from another processor's work queue.

If the first part of the algorithm divides up the iterations such that each of the original P chunks contain similar amounts of work, then not much load sharing will take place; processor i will mostly execute the iterations in the *ith* chunk. In other words, a processor will execute the same set of iterations every time the parallel loop executes. Thus in the second, and all subsequent executions of the parallel loop, a processor does not fetch data out of the memory of a remote node; all the data is in the cache. (This assumes, however, that the cache is large enough to contain all this data.)

Sometimes the first part of the algorithm divides up the iterations such that a lot of load sharing takes place during the execution of the parallel loop. This happens when not all iterations contain the same amount of work. For example, sometimes most of the work of the parallel loop is contained in the first few iterations. In this case, the first processor receives the bulk of the work so that all processors except the first finish their work quickly. These processors obtain the rest of their work by load sharing. Load balancing is achieved, but data locality is probably not because processors probably do not execute the same iterations of the parallel loop every time the parallel loop executes.

Subramaniam studies this case in [18]. There, a processor remembers how many iterations it actually executed during the last iteration of the outer sequential loop. This information is then used to readjust chunk sizes. He finds that it does not take too many iterations of the outer sequential loop before chunk sizes converge to something that results in little load sharing.

4.2.2 Multiple Parallel Loops

The previous affinity scheduling algorithm considers communication overhead for only a single parallel loop. Often parallel loops execute in sequence, i.e. the first parallel loop executes, then the second parallel loop executes, etc., and these parallel loops all access the same data with regular access patterns. If the second parallel loop does not go through the data in the same way that the first parallel loop went through the data, then there will probably be communication overhead at the beginning of the second parallel loop.

For example consider the following sequence of two parallel loops:

```
for (j = 0; j < 1000000; j++)
    a[j] = ...;
for (j = 0; j < 1000000; j++)
    ... = a[999999 - j];
```

Suppose these loops are executed by 10 processors and the first loop is simply divided up into 10 chunks with one chunk given to each processor. If chunks are distributed so that the *ith* chunk is given to the *ith* processor, then after the first loop has finished executing, processor 1's cache will contain a[0] to a[99999], processor 2's cache will contain a[100000] to a[199999], etc. Now if the second loop is divided up the same way, each processor will need to access different data, and a large communication overhead will occur.

Affinity regions [9] are a means for a programmer to reduce this kind of communication overhead. Basically, subject to some limitations, a programmer declares the sequence of parallel loops to be an affinity region.[3] The runtime system breaks the loops into chunks and schedules these chunks to try to avoid communication overhead between loops. In the above example the runtime system would schedule the last chunk of the second loop on the same processor as it scheduled the first chunk of the first loop.

The feasibility and profitability of affinity regions is discussed in [1].

5 Locality through Array Allocation and Owner Computes

5.1 Motivation

Array allocation and owner computes is a method designed for a multiple address space large scale multiprocessor with a slow interconnection network, but this general idea (of partitioning arrays and letting this drive the computation distribution) can be applied in the single address space context as well, typically through manual programmer placement of data and of computation. This report will focus on the multiple address space context, however, an approach wherein the compiler derives the computation distribution from the data distribution.

An example multiple address space context is a network of workstations where the interconnection network is an ordinary (slow) local area network. On this type of machine, the communication overhead of a remote memory access is extremely high. Therefore the goal of array allocation and owner computes is to bring the number of remote memory accesses down to the absolute minimum. As an example of what is meant by the *absolute minimum* number of remote memory accesses, consider the following comparison of this method with the previously discussed Marketos and Leblanc affinity loop scheduling algorithm.

[3] The programmer also specifies tile indices, but space limitations do not permit a discussion of tiles here.

Recall the first step of the Marketos and LeBlanc algorithm: if there are P processors, then divide the iterations of the nested parallel loop into P equally sized chunks and allocate a chunk to each processor. Note that only *iterations* are thus statically distributed among the processors; *data* is *not* statically distributed.

Instead, data is dynamically distributed during the first iteration of the outer sequential loop. In this first iteration, not only do memory references miss in the cache, but since there has been no static data distribution, many of these cache misses are satisfied from remote memories rather than from local memory. The process of filling all the caches, often from remote memories, is a dynamic (i.e. during runtime) data distribution. Substantial communication overhead is incurred as most of the processors often wait for data to travel across the interconnection network from remote nodes to their caches.

The reason this algorithm can perform well, though, is that this substantial communication overhead need not be incurred on every iteration of the outer sequential loop. If the first part of the algorithm distributes the iterations of the nested parallel loop among the processors such that little load sharing takes place during the execution of the parallel loop, then the bulk of dynamic data distribution gets done during the first execution of the parallel loop. Not much dynamic data distribution takes place in subsequent executions of the loop because memory references are then mostly satisfied in the cache. Therefore, the communication overhead of dynamic data distribution is mostly incurred during the first execution of the parallel loop and is then amortized over the subsequent executions of the loop.

The performance experiments for this algorithm, however, were carried out on a hardware implemented single address space machine where the communication overhead of a remote memory access is not nearly as high as it is with a network of workstations connected by a local area network. In the latter environment, the cost of dynamic data distribution can be substantial. The array allocation and owner computes method tries to bring the number of remote memory accesses down to the absolute minimum by explicitly controlling (mostly statically) the distribution of both data and iterations, so as to achieve greater data locality. The main disadvantage is that, in order to achieve this greater data locality, the programmer typically gets burdened with the responsibility of (mostly) statically distributing both data and computation.

5.2 Basic Techniques

Basically in the array allocation and owner computes method the user writes a sequential program except that the program is supplemented with data distribution directives. These directives tell the compiler how the data will be distributed among the nodes of the machine. The compiler takes that sequential program and creates a parallel program by creating code for each node. Each node will typically operate on different portions of large shared arrays (i.e., exploiting data parallelism). References to remotely stored data are translated into message passing statements by the compiler.

Although the compiler codes the messages, it is still the programmer's responsibility to use data distribution directives and thus explicitly distribute data. The way the data gets distributed dictates the way the computation gets distributed using the owner computes rule: the compiler looks at the left hand side of each assignment statement, determines on which node the data being written resides on — this node is the "owner" of the left hand side, and assigns that piece of computation (typically an iteration of some loop) to this node. Thus, the programmer distributes both data and computation; the programmer distributes data explicitly and computation implicitly.

Arrays can be distributed both statically (initial placement of arrays, this is the typical case) and dynamically (changes in distribution of arrays during execution of the program). Dynamic distribution can be both *with* or *without* movement of existing data. Suppose there are four processors and array "A" has been statically distributed by column. Thus at the beginning of execution, each processing node has a column of "A." Suppose the programmer wishes to, at some point in the execution, have the distribution of "A" dynamically change to being distributed by row (wants each processor to have a row). This can be accomplished without moving the existing data as long as there is no need for the old values. There is no need for the old values if the program is just going to immediately write new values anyways. The only thing that needs to change is the location function that indicates where that new write is going to go.

Distributing data and computation in simple cases does not present too much of a burden upon the programmer; the decision is straightforward. Once data access patterns get complicated, however, so do data distribution decisions. Some remote memory accesses become unavoidable and the programmer has to evaluate more than one possible data distribution in order to decide which is best.

5.3 Automatic Mapping

It is unfortunate to have to burden the programmer with mapping data and computation to the physical topology of the machine. Gupta and Banerjee address this in [7]. There, they consider all the issues that must be resolved in order to have the compiler automatically distribute data and computation.

A limitation on the automatic mapper is that it only works for applications with a regular and static structure that is fully analyzable at compile time. Many applications have this characteristic, although many others do not.

Another drawback is that even if all the required input is obtained, determining an optimal mapping is NP-hard. Heuristics must therefore be used, which means that there is no guarantee about the optimality of the mapping.

6 Future Work — Locality through Adaptive Prefetching / Poststoring

Adaptive prefetching is a method of dynamically distributing data. Its goal is to dynamically distribute data in a way that incurs low communication overhead. The following example is in the context of a hardware implemented single address space machine such as DASH [11].

As previously discussed, in the Marketos and LeBlanc algorithm, data is dynamically distributed during the first iteration of the outer sequential loop. Substantial communication overhead is incurred as most of the processors often wait for data to travel across the interconnection network from remote nodes to their caches.

To be more precise, if most nodes do not initially (before the loop begins executing) store any data required by the loop, then most processors fetch a block of data from a remote memory upon every cache miss. Thus, communication overhead is incurred upon every cache miss. Furthermore, there are a lot of cache misses because a cache miss occurs every time a processor has finished using the data from the previously fetched block.

A *prefetch* is a remote memory access that is issued before the data is needed. If the processor issuing the prefetch performs useful computation during the entire time it takes the prefetch to complete, then no communication overhead will be incurred for that remote memory access. Thus, prefetching can reduce the communication overhead of dynamic data distribution. However, it depends on knowledge, or at least a good guess, of which memory locations a processor is going to access in the future. It also depends on prefetched data not usually being either invalidated or flushed between when it arrives in the cache and when the processor accesses it in the cache.

A *poststore* occurs when a processor that produces data sends the data it produced to some number of consumer processors *as soon as the data has been produced*; the producing processor "updates" some number of memory locations as soon as possible.

One level from which prefetching can be controlled is the programmer/compiler level. This kind of prefetching is not adaptive, however, since the prefetch will (a) take place during runtime at exactly the point in the instruction stream explicitly specified by the programmer or compiler and (b) fetch exactly the amount of data explicitly specified by the programmer or compiler.

In order to be *adaptive*, prefetching must be controlled from a level lower than the compiler. At runtime, statistics are generated on the memory behavior of processors. Example statistics include a history of memory locations accessed and the success or failure of previously made adaptive prefetches. These statistics are used to predict, among other things, which memory locations a processor will access in the future. Prefetches are made to these locations.

Adaptive prefetching controlled by hardware is studied in [4]. Adaptive prefetching/poststoring controlled from the level of the runtime/operating system is an attractive research area. Since this type would be software controlled (as opposed to hardware controlled), there is potential for a high degree of sophistication here.

7 Conclusions

Blocking is a technique that can be used to achieve data locality on any type of large scale multiprocessor. It is sound programming practice and therefore a technique that may last a long time.

Currently, the only realistic implementation of object placement and method scheduling is COOL. In COOL, the programmer explicitly distributes data and either explicitly or implicitly distributes computation. COOL is very similar to array allocation and owner computes, where the programmer explicitly distributes data and thus implicitly distributes computation. The two are very similar because they both require the programmer to fully control distribution of both data and computation. Since the programmer has such control, they are both applicable to all types of large scale multiprocessors. The only important difference between the two with respect to how locality is managed is in the syntax through which the programmer achieves control, however both employ data distribution directives and additional implicit or explicit abstractions to distribute computation according to how data has been distributed.

Another difference is that an automatic mapper has been studied in the context of array allocation and owner computes, but not in the context of COOL. This is a minor difference because the automatic mapper is probably something that will not be practical for a long time: removing the need for the programmer to consider data locality is too lofty a goal to be achieved at the compiler level in the near future. The only hope of achieving this goal in the near future is at levels below the compiler. Adaptive prefetching is presented as a possible part of achieving this goal soon.

Perhaps array allocation and owner computes will become more important than COOL because array allocation and owner computes has already been implemented on multiple address space machines, which are currently becoming popular. However, the syntax through which the programmer achieves control in COOL may allow finer control than the syntax in array allocation and owner computes does.

The Marketos and LeBlanc affinity loop scheduling algorithm as modified by Subramaniam is limited in scope of applicability, but within that scope it has the potential to last a long time because it does not require the programmer to consider data locality. This algorithm considers data locality for only a single parallel loop. Affinity regions are a means for a programmer to reduce communication overhead that occurs between two parallel loops that execute in sequence and access the same data with regular access patterns.

References

[1] Appelbe, B. and Lakshmanan, B. Optimizing Parallel Programs Using Affinity Regions. In *Proceedings of the 1993 International Conference on Parallel Processing*, St. Charles, Illinois, August 1993, pp. II 246-II 249.

[2] Chandra, R., Gupta, A., and Hennessy, J. L. Data Locality and Load Balancing in COOL. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, San Diego, California, May 1993, pp. 249-259.

[3] Chandra, R., Gupta, A., and Hennessy, J. L. COOL: An Object-Based Language for Parallel Programming. *Computer* 27, 8, (August 1994), 13-26.

[4] Dahlgren, F., Dubois, M., Stenstrom, P. Combined Performance Gains of Simple Cache Protocol Extensions. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, Chicago, Illinois, (April 1994), pp. 187-197.

[5] Eager, D. L. and Zahorjan, J. Adaptive Guided Self-Scheduling. Tech. Rep., 92-01-01, Department of Computer Science and Engineering, University of Washington, January 1992.

[6] Fowler, R. J. and Kontothanassis, L. I. Improving Processor and Cache Locality in Fine-Grain Parallel Computations using Object-Affinity Scheduling and Continuation Passing. Tech. Rep., 411, Computer Science Department, University of Rochester, June 1992.

[7] Gupta, M. and Banerjee, P. Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compilers on Multicomputers. *IEEE Transactions on Parallel and Distributed Systems* 3, 2, (March 1994), 179-193.

[8] Hiranandani, S., Kennedy, K., and Tseng, C. Compiling Fortran D for MIMD Distributed-Memory Machines. *Commun. ACM* 35, 8, (August 1992), 66-80.

[9] Kendall Square Research. *KSR Parallel Programming.* Waltham, Massachusetts, 1991.

[10] Lam, M. S., Rothberg, E. E., and Wolf, M. E. The Cache Performance and Optimizations of Blocked Algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, California, (April 1991), pp. 63-74.

[11] Lenoski, D., Laudon, J., Gharachorloo, K., Weber, W. -D., Gupta, A., Hennessy, J., Horowitz, M., and Lam, M. The Stanford DASH Multiprocessor. *Computer* 25, 3, (March 1992), 63-79.

[12] Markatos, E. P. and LeBlanc, T. J. Locality-Based Scheduling for Shared-Memory Multiprocessors. Tech. Rep., 420, Computer Science Department, University of Rochester, May 1992.

[13] Markatos, E. P. and LeBlanc, T. J. Shared-Memory Multiprocessor Trends and the Implications for Parallel Program Performance. Tech. Rep., 420, Computer Science Department, University of Rochester, May 1992.

[14] Markatos, E. P. and LeBlanc, T. J. Using Memory (or Cache) Affinity in Loop Scheduling on Shared-Memory Multiprocessors. Tech. Rep., Computer Science Department, University of Rochester.

[15] Markatos, E. P. and LeBlanc, T. J. Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems* 5, 4, (April 1994), 379-400.

[16] Polychronopoulos, C. D. and Kuck, D. J. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Transactions on Computers* C-36, 12, (December 1987), 1425-1439.

[17] Snyder, L. Type Architectures, Shared Memory, and the Corollary of Modest Potential. *Annual Review of Computer Science*, Annual Reviews Inc., Palo Alto, California, Vol. 1, (1986), pp. 289-317.

[18] Subramaniam, S. Affinity Scheduling of Unbalanced Workloads. M.Sc. thesis, Department of Computational Science, University of Saskatchewan, August 1993.