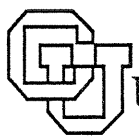
EFFICIENT COMPILE-TIME/RUN-TIME CONTRACTION OF FINE GRAIN DATA PARALLEL CODES

Richard Neves and Robert B. Schnabel

CU-CS-697-94 January, 1994



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Efficient Compile-Time/Run-Time Contraction of Fine Grain Data Parallel Codes[1]

Richard Neves[2] and Robert B. Schnabel[3]

CU-CS-697-94 January 1994

Appears in *Proceedings of Sixth Annual Languages and Compilers for Parallel Computing Workshop*, Portland, OR, August 1993

[1]Research supported by NSF grants ASC-9015577 and CDA-8922510.

[2]Department of Computer Science, Campus Box 430, University of Colorado, Boulder, Colorado 80309 U.S.A (neves@cs.colorado.edu)

[3]Department of Computer Science, Campus Box 430, University of Colorado, Boulder, Colorado 80309 U.S.A. (bobby@cs.colorado.edu)

Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the agency named in the acknowledgements section.

Abstract

This research studies the contraction problem for data parallel languages. That is, we seek to efficiently execute code written for a fine grain virtual parallel machine on a much coarser grain actual parallel machine. This paper identifies issues involved in solving this problem. Three issues are then addressed in detail: efficiently implementing tasks to emulate virtual processors, efficiently scheduling these tasks, and reducing space overhead while maintaining data consistency among these tasks. In implementing tasks, saving a reduced register state at context switches is addressed. For scheduling tasks, heuristics are proposed that minimize scheduling cost and promote data locality. Minimizing space overhead using assumptions about the communication paradigm, run-time techniques and compile-time analysis is also discussed. Finally, experimental results are presented concerning one of the main issues discussed, scheduling. The results show that the proposed scheduling heuristics are both cost efficient and promote data locality for three different problems when compared to three other scheduling policies.

1. Introduction

This research studies techniques needed to compile and execute fine grain, data parallel codes on coarser grain multiprocessors. This is known as the *contraction* problem. Fine grain data parallel codes are generated by programming models that assume a virtual parallel machine with many more processors than the targeted, physical machine. By allowing a language to assume a virtual parallel machine, the programmer can specify a data parallel algorithm at a granularity appropriate to an algorithm and problem, and disregard how many processors the algorithm will actually run on.

The research addresses a specific domain of languages: data parallel languages. These languages are intended mainly for numerical computation. The issues that are studied are most directly relevant to explicitly parallel languages such as PC [3], or DINO [25], but may also be relevant to implicitly data parallel languages such as Fortran D [14] or High Performance Fortran [27]. It is assumed that information regarding how data is mapped to virtual processors and how virtual processors are mapped to physical processors is provided. This assumption is generally satisfied by such languages.

Within the domain of data parallelism, the research addresses at least three communication paradigms: Block-SIMD (BSIMD) with communication points identifiable at compile-time, BSIMD with communication points only identifiable at run-time, and the general send/receive paradigm. These three paradigms span different approaches to data parallel programming. (BSIMD is sometimes referred to as loosely synchronous.) In BSIMD each piece of data is "owned" by a single virtual processor. Each processor only can modify the data that it owns, but processors can read data owned by any other processors. The language specifies a "block" (e.g. a parallel loop), and modifications of data in the current block are only viewable by other processors after the next block boundary. Semantically, data is exchanged *only* at block boundaries. Processors that do not own a piece of data, read the version of that data before the previous block boundary, while the processor that owns a piece of data reads the current value of that data. Kali [22], DYNO [31], Fortran D [14], and High Performance FORTRAN [27] are some examples of BSIMD. Fortran D and High Performance FORTRAN can be viewed as BSIMD in their use of loop parallelism. In most cases, communication points are identifiable at compile-time in these languages. DYNO is an example of BSIMD where communication points are determined at run-time. This is also true of Kali, Fortran D, and High performance FORTRAN when index arrays are utilized. The general send/receive paradigm is provided by vendors as the default means of programming their multiprocessors. The paradigm is also found in languages like DINO where it is augmented with distributed data mappings.

A distributed memory MIMD multiprocessor is assumed in this research. In addition, machines that support small, low latency messages are well-suited to contracting fine grain code, which will have a propensity to generate many small messages. Therefore, support for such messages [13, 24] is related to this research but it is not our focus in this paper. When a low latency interface is unavailable, aggregation of small messages becomes necessary. Though aggregation is also part of the research, it is only addressed briefly in this paper.

One possible approach to the contraction problem is for the compiler to generate an equivalent, efficient program whose number of processes matches the number of physical processors. This is done in the SIMD language Dataparallel C [16], for example, by using for-loops to emulate virtual processors. This allows several virtual processors to execute as one process on a physical processor. This contraction mechanism is possible because the SIMD semantics of the language

state that no interactions between virtual processors may exist. Our assumptions are considerably less restrictive.

In fact, we do not take this purely compile-time approach to contraction in this research, for two reasons. First, it is difficult or impossible to efficiently contract parallel programs at compile-time as the complexity of the program or of the communication paradigm grows. This is particularly true for general send/receive code. Second, it may be more efficient to allow multiple virtual processors (as tasks) per physical processor to mask communication costs, even in models such as BSIMD [12].

For these reasons, we pursue a mixed compile-time/run-time approach to the contraction problem. In studying the contraction problem from this viewpoint, the goal of this research is to minimize the overhead resulting from contraction of fine grain computation. One source of overhead is the time it takes to switch between virtual processors. This includes the time needed to save the current state and restore the state of the next virtual processor. It is also important to reduce overhead due to scheduling virtual processors. It is imperative that the choice of which virtual processor to schedule next be made very inexpensively in almost all instances. Closely related to the scheduling issue is the overhead that can arise from poor use of the cache. Virtual processors executing on the same processor will have a high miss rate if the order in which virtual processors are scheduled does not account for locality of reference. Another source of overhead is the memory consumed in order to maintain consistent views of data among virtual processors. Virtual processors that communicate on the same processor will actually read and write data, as opposed to sending and receiving data over a network. One goal is to keep as few copies of this communicated data on the processor as possible. Finally, overhead may result from communication of small messages between virtual processors on different physical processors, if message latency on the parallel computer is high. Our goal is to minimize the above overhead so that it possible to contract codes where the amount of computation between context switches is as little as 20-40 flops.

The approach to task emulation taken in this research is to emulate each virtual processor with a task abstraction that is managed partly at compile time and partly at run time so that the time to switch between two tasks is minimized. The bulk of this switching cost is due to saving and restoring register state [1]. We consider two approaches to the issue of register optimization. Both of these approaches can be expressed using either continuations [11] or threads. Using different techniques, these two task implementations can result in the same register allocation, but will differ in how efficiently stack space is used.

In addition to implementing tasks, an inexpensive means of scheduling tasks is another key issue in this research. Scheduling accounts for a significant part of the overhead in contracting very fine grain tasks at run-time. We assert that, in this special context, scheduling can be performed at a fraction of the cost of traditional and more general approaches. To achieve this goal, we have designed a set of scheduling heuristics that attempt to minimize scheduling overhead by using communication patterns found in data parallel programs. Another goal of these heuristics is to maximize cache locality by scheduling tasks that share data consecutively when possible.

The proposed scheduling heuristics will be compared to scheduling policies that function independent of the application. We seek to show that the application dependent scheduling method as we propose is less costly and promotes locality as well or better than application independent methods. In fact, we seek to approach the data locality found in manually contracted code. In the "Preliminary Results" section, the proposed heuristics are compared to LIFO, FIFO, and random scheduling policies. LIFO and FIFO are inexpensive, yet provide no direct way of promoting cache locality among communicating virtual processors. A random policy will motivate the need to

address data locality in contraction. By comparing with these three policies, the proposed heuristic's success in attaining the seemingly conflicting goals of low cost and good data locality will be measured.

The solution of the contraction problem requires cooperation between the compiler and run-time system. In compiling virtual processor code, information needs to be provided to the run-time system. This information includes *data to virtual processor* and *virtual processor to physical processor* mappings. If it is possible, off-processor communication points should be identified at compile-time. Information regarding how data is stored is also helpful to aid the scheduling heuristics in improving locality of reference. In this research, we assume that this information is made available by the compiler since this is a reasonable assumption for many data parallel programming languages. Our future research may seek to relax some of these assumptions.

While this paper will focus on issues of task implementation, task scheduling, and reducing space overhead, additional compile time analysis which is beneficial to contraction will be addressed in greater detail in future research. In contracting virtual processor code, varying compiler transformations will be necessary. These include creating templates for data local to virtual processor code such that virtual processor instances can access this data in the heap, reducing stack space usage (when using threads), and performing the compiler analysis to transform code to a continuation-friendly form (when using continuations). Compile-time analysis also will be necessary to reduce space overhead and promote locality. For example, dependence analysis performed by the compiler can reduce the need for run-time duplication of data. Moving block points (when it is legal to do so) and postponing duplication until necessary are two means of accomplishing this. Both compile-time and run-time analysis will be needed in performing the message aggregation necessary for multiprocessors which do not support low latency, small message communication.

This paper is primarily a preliminary discussion of the issues involved in the approach to contraction outlined above. The next three sections discuss issues and approaches for task implementation, task scheduling, and reducing space overhead respectively. The final section presents preliminary experimental results on the costs of several scheduling strategies and their effects upon data locality.

2. Implementing Tasks

As stated in section 1, the approach that appears necessary for the contraction problem considered in this research is emulate virtual processors with a task abstraction mechanism managed partly at compile-time and partly at run-time to. The term task is used abstractly to refer to the emulation mechanism (e.g. threads or continuations) that is used. This compile-time/run-time combination is necessary because compiler involvement is needed to generate efficient context switches and reduce space overhead, while run-time involvement is essential to handle communication-dependent synchronization.

In implementing tasks, two key considerations are that context switch cost and stack space usage must be minimized. Minimal context switch cost is vital in supporting fine granularity. An important factor in reducing context switch cost is reducing the number of registers saved and restored at context switches. Minimizing stack usage is important because the potential for heavy stack space memory consumption is greater for fine granularity in multitasking models

Two interesting possibilities in implementing tasks at the user level are threads and continuations. In this paper, a thread refers to a non-preemptive, light-weight, user-level task which does not enter the operating system on a context switch. The second, continuation-based, task abstraction has its

roots in the language community [7, 17, 21, 29, 30] and has recently been applied as a portable, light-weight task abstraction in the Mach kernel [11]. Unlike the use of continuations in [11], this research considers only context switching within the same address space. The use of continuation-based tasks assumes that the code is restructured by the compiler such that for every receive (potential block point), there is a corresponding function that executes all the code following the receive. This function is called the *continuation* of current task. If the task blocks at the receive, the continuation function represents the functional state of the task. The task also has a data state which is stored in the heap.

When implemented in the standard way, as general purpose run-time mechanisms, threads and continuations have contrasting properties. Threads have a fairly high context switch cost since register state must be saved and restored (which allows register optimization). Continuations have far lower context switch cost, but are far less advantageous for register optimization due to the fact that functions are inserted at every potential context switch. This prevents the compiler from associating the code which would normally be executed when the context switch does not occur with the code analyzed to that point. Although the approaches differ significantly when implemented as stated, both can be improved and the differences between them become far fewer, when they are modified using compile-time information available in the special context of this research.

To achieve the efficiency that is needed in the contraction context, the compiler must cooperate in implementing tasks. In the case of threads, compiler cooperation allows the thread context switch to avoid saving and restoring unused registers. For a continuation-based implementation, compiler cooperation allows the introduction of register optimization by including the code following a continuation function call in the register analysis. (This requires that continuation calls save and restore the required register state.) Additionally, in the thread-based task implementation, the compiler can move data allocated on a thread's stack to separately allocated memory, thus allowing a stack size that more closely matches that actually needed by the task. Note that a continuation-based task implementation does not require this optimization since each continuation shares a single stack. Once the compile-time register analysis is incorporated in continuations, it appears that the only difference between thread-based and continuation-based task implementations will be stack usage.

Another interesting issue is that to some extent, effective register optimization and fast context switches are conflicting goals in this research. The benefit of storing data in registers [6] may be offset by the cost of saving and restoring these registers at context switches [1]. The difficulty is that a context switch may not always occur. At compile-time, only the *potential* block points can be identified. Whether a context switch actually occurs may depend, for example, on how long a message takes to arrive. Because a context switch cannot be predicted, we are left with two choices. One choice is to treat the blocks of code before and after the context switch as disjoint during local register optimization. In doing so, no registers would need to be saved and restored during the context switch. Another choice is to treat the blocks of code before and after a context switch as always contiguous during execution. In this case, those registers allocated through register optimization would need to be saved at the context switch. That is, only those registers in active use would need to be saved [28]. It is likely that the best approach will be to combine the above choices by allowing the compiler to decide on a case by case basis which to use. This judgment would be made based on how well register optimization could be performed on the said code block, the cost of saving registers on the host architecture, and an assumption that context switches will occur. This will be a topic of future research.

Our research will implement and test both thread-based and continuation-based task mechanisms to assess the issues discussed above. It should be mentioned that, regardless of how tasks are implemented, some amount of compiler optimization is lost simply by programming at a fine granularity to begin with. Register optimization may prove ineffective when compared to the resulting context switch cost and frequency of context switches. Additionally, manually contracted code would make copious use of loops (in simple cases), which would allow the compiler to perform a number of loop optimizations that may not be possible in our automatic approach. Manual contraction would also attempt to promote locality directly, whereas we address this indirectly. Our research will evaluate the effects of these factors.

3. Scheduling Tasks

In addition to efficient task mechanisms, a key to efficient, automatic contraction is very efficient methods for scheduling tasks. Uniprocessor scheduling heuristics are needed that attempt to minimize scheduling cost and promote cache locality in the context of the contraction problem. It has been noted that scheduling order is important to data locality [23]. Simple scheduling heuristics such as FIFO (first in first out) and LIFO (last in first out) scheduling can be used (see e.g. [5]), but do not take advantage of the special characteristics of the scheduling problem in the contraction context, namely, the availability of information about communication partners. None of the specialized and more complex uniprocessor scheduling policies that we are aware of (such as those which are priority-based) are both applicable to this situation and computationally inexpensive. For this reason, we have designed new scheduling heuristics for use in the contraction problem.

In this section, we will motivate and describe the new scheduling heuristics for the contraction problem, and discuss how they compare to a FIFO scheduling policy. By comparing the proposed heuristics with a FIFO policy, the low cost of communication-based scheduling is illustrated. The comparison is also instructive in showing that the effort in promoting cache locality is worthwhile and need not come at a high scheduling cost. An experimental comparison with FIFO, LIFO and a random policy is given in Section 5.

To motivate our approach, first consider a FIFO ready list scheduling policy. When a task begins or resumes execution, it is removed from the ready list. If the task blocks, another task is taken from the head of the list and run. (We assume that a task blocks on exactly one receive at a time.) The former task does not return to the end of the ready list until the reason it blocked is no longer applicable (i.e. when a message the task is waiting for has been sent). This approach has two undesirable characteristics in the context of this research:

1. For each scheduling decision, the task at the head of the ready list is removed for execution. When an executing task sends data to a blocked task (which was waiting for the *send*), the blocked task is added to the end of the ready list. Thus, the ready list is managed twice (on average) per scheduling decision.
2. When adding tasks to the ready list, no effort is made to order the tasks such that data locality is promoted. Thus, the resulting scheduling order of tasks may be arbitrary with respect to data locality.

In the proposed heuristics, the ready list overhead found in the first point is avoided altogether in *most* scheduling decisions. The overhead described in the second point is reduced by attempting to consecutively schedule tasks that compute on shared data. Before describing these heuristics, we define some terminology.

The proposed heuristics differentiate between two types of tasks: *external* and *internal*. External tasks communicate at least once with tasks mapped to other (actual) processors, while internal tasks only communicate with tasks on the same processor. Note that the terminology allows external tasks to also communicate with tasks on the same processor. When a communication primitive (i.e. a send or receive) communicates off-processor, it is referred to as an external communication (e.g. an external send). Likewise, on-processor communications are referred to as internal (e.g. an internal receive). We say that an internal receive occurs when the receiving process runs and reads the new value.

The new heuristics avoid the manipulation of a run-time data structure, such as a ready list, entirely in most situations. In particular, this can be done when it is possible to switch to a task's communication partner. Suppose that task A blocks on a *receive* from an internal task B. If task B is scheduled next, it is possible that the *send* from task B that task A is blocked on will be executed when task B next executes. If this is the case, then task A will be able to resume execution any time after task B suspends execution. So, instead of choosing a task from the ready list to schedule next when task A blocks, a reasonable strategy is to schedule the task specified in task A's receive directive. Note that this scheduling decision can be made without having to manage a ready list. In fact, the only computation necessary is to determine whether B is an internal, ready task. This information can be made available at minimal computational expense during the course of internal communication by setting the status of a receiving task to "ready" when the sending task sends the required data item. A mechanism similar to I-Structures [2] can be used for this. (Issues in implementing I-Structures in similar context are discussed in [26].) This heuristic will be referred to as "scheduling the sending task", and will be the first scheduling heuristic. Clearly, it has made significant use of the special information about communications that are available in this contraction context.

"Scheduling the sending task" does not account for every scheduling decision. Suppose a task blocks on an *external receive* (i.e. from an off-processor task). It is not possible to schedule the sending task since the sending task resides on another actual processor. As another example, suppose a task blocks on an *internal receive*, but the sending task itself is blocked. It is not possible to scheduling the sending task in this case either. A possible task to schedule next, which is also a communication partner, would be a previous internal receiving task. By this we mean an internal task which the blocked task has sent data to in the past, *and* which has not yet received that data. Note that there could be more than one task fitting this description, so the task the blocked task sent data to most recently is chosen. This heuristic will be referred to as "scheduling a receiving task" and is the second scheduling heuristic. Its implementation cost will be discussed later.

In the discussion thus far, two scheduling approaches have been utilized: 1) scheduling the sending task and 2) scheduling a previous receiving task. Both approaches are insufficient when:

> "Scheduling the sending task" is impossible because the current task is blocked on an *external receive,* or the current task is blocked on an internal receive and the sending task is blocked.
>
> and
>
> "Scheduling a previous receiving task" is impossible because all previous send partners are blocked (on other receives).

Another scheduling option is necessary when these situations occur. So far, the heuristics have neglected external tasks which have become ready due to an external communication. Such a task

motivates another scheduling heuristic. By keeping external tasks on their own *external ready list*, the list can be used as a third scheduling option when "scheduling the sending task" or "scheduling a previous receiving task" fails. After an external task has blocked, it is added to the external ready list when the internal or external send the it blocked on has been executed. (This implies that an asynchronous receive handler for off-processor communication exists.) External tasks are removed from the external ready list when they are scheduled to run. Since this external ready list only contains external tasks, the cost of maintaining it is not as expensive as maintaining a list for every task (since this would imply managing the list at every context switch). In addition, scheduling external tasks in a separate ready list coincides with off-processor communication except in those cases where external tasks block on receiving data from internal tasks. Thus, the overhead in managing the ready list is often dominated by the cost of off-processor communication incurred when external tasks block on external receives (unless message latency is extremely low relative to managing a linked list). Also, external communications are assumed to constitute a small percentage of all communications in a fine-grain data parallel program, so this list is expected to be used for only a small portion of the scheduling decisions. Thus, the third scheduling approach is the use of an external ready list.

We expect to show in a future paper that the above three heuristics are sufficient for a broad class of data parallel codes meeting certain requirements (except in the termination phase of execution). These requirements are not formalized here, but this contention is supported by preliminary simulation results (see Section 5).

In order to account for all possible codes, however, a fourth approach is needed as a safety net. The fourth approach (referred to as *lookup*) involves finding ready tasks that the above heuristics were unable to uncover. It is described in more detail below. This fourth step is the most expensive, but should account for a very small minority of most scheduling decisions.

In summary, the proposed scheduling heuristics determine which task to schedule next as follows:

(1) If the current task is blocked on an **internal receive** and the sending internal task is not blocked then
    switch to the corresponding **sending task**
(2) else if there exists an internal ready task that has not yet received a message sent by the current task since the last context switch then
    switch to this **receiving task**
(3) else if the external ready list is not empty then
    switch to the head of the external ready list
(4) else *lookup* a ready task

These heuristics impose an implicit execution order on tasks that is based on communication dependencies. They attempt to utilize very inexpensive scheduling methods for most context switches, and to use the least expensive means of switching between tasks first. A goal of this research is to show that the scheduling of tasks using this approach has low overhead, and that the overhead is considerably lower than an approach which maintains a ready list of all tasks. Now we briefly discuss the cost of these heuristics relative to ready list management overhead.

Step #1 must test that the receive is internal and that the sending task is not blocked. The sending task is then scheduled. A comparison of the cost of this step to the ready list management code, on two representative workstations, shows that step #1 costs approximately 1/2 of managing a ready list.

Step #2 requires that each time a send occurs, if the corresponding receive has not occurred, then the receiving task identification be stored for future reference. Thus, when the task blocks, it need only check that this task identification refers to a ready task. Including the conditional branch in step #1, the cost, measured in the same way as for step #1, is approximately 2/3 the cost of managing a ready list.

The cost of step #3 is approximately 1/2 greater than ready list management when one includes the conditional branches in steps one and two. But since each utilization of step #3 corresponds to an external communication which has far higher overhead, this extra cost is not significant. Also, the ratio of external to internal tasks is typically quite low, meaning that step #3 is used infrequently.

Step #4 requires that a ready task be found among all tasks. It may appear that a linear search is necessary to find such a task, but in practice this is not the case. It can be shown that Step #4 will only be needed when the only ready tasks are those which *have not* and *will not* communicate with any other tasks, or when the only ready tasks are those which *will not* communicate with any other tasks. Our intuition from limited experience, is that these situations occur only during the final phase of most data parallel algorithms. Thus, both of these situations can efficiently be addressed by keeping track of which tasks have begun execution and which tasks have terminated, thereby reducing the computation needed to search for a ready task. Keeping track of this incurs a one-time startup cost and a one-time termination cost for each task, which is approximately equivalent to the cost of managing a ready list for each context switch in a FIFO policy. In addition to this one time cost, the cost of using step #4 is amounts to finding a ready task in this termination list. In practice we find that the first dereference yields a ready task. This research aims to show that this one time cost is easily amortized over all context switches, and that the use of step #4 is a very small percentage of all context switches for most data parallel algorithms.

In summary, we have discussed heuristics that reduce scheduling overhead by switching among tasks in a communication-based ordering. This should allow inexpensive scheduling heuristics to be used most of the time. In addition, since the proposed heuristics promote switching between communicating internal tasks and since communicating tasks often access as adjacent portions of the main data structures in parallel programs, the heuristics are likely to encourage cache locality in data references. To assess these heuristics, we are conducting experiments using a wide variety of communication patterns for the three communication paradigms considered in this research. Section 5 gives a preliminary indication of the performance of these heuristics on four applications using careful estimates of the cost of each step and simulating the resulting data locality.

4. Space Overhead

In discussing the scheduling and implementation of tasks in the previous two sections, the time overhead incurred in using tasks to emulate virtual processors was the primary concern. The critical issues included context switch time, efficiency due to register optimizations, and efficiency of the scheduling heuristics.

Contraction of a fine-grain virtual machine to a much coarser grain actual parallel machine can also imply a great deal of space overhead due to data duplication. In this section we briefly motivate this problem, and discuss some possible approaches to reducing space overhead.

Data duplication is often necessary to maintain consistent versions of distributed data among virtual processors. Assume first that in a general send/receive paradigm is being used, and consider a situation where a virtual processor (VP) sends the value of a variable to another VP on the same physical processor. Suppose the sender continues to modify the variable just sent. If only one copy of the variable were kept on the actual processor, the receiving VP would see the

subsequent modifications that the sender made to the variable, which is not the intention. In this situation, one copy of the variable is not sufficient to maintain a consistent view among the sender and receiver. It is not difficult to envision a situation where n VPs communicate m pieces of data to each other and m*n copies are needed. Of course, such a situation assumes that a VP communicates a piece of data only once. An indeterminate number of copies may be needed if the VPs send/modify data over and over again.

However, the amount of data duplication that can be required is dependent on the communication paradigm. As illustrated above, a general send and receive paradigm could require enormous amounts of space to maintain consistency. On the other hand, the BSIMD model puts constraints on when virtual processors can modify distributed data, thus reducing the number of copies required. In this model, each data item is "owned" by exactly one VP. If a VP does not own a piece of data, any reads of that data item refer to the version that existed at the previous synchronization point (this version will be the same for all non-owner VPs). During a block, the owner of the data in the BSIMD model can modify its own copy of the data, but the values of this items that the other VPs use is not affected. This means that at most two copies of each piece of distributed data are needed (one pre-block copy and one local copy for the owner VP) regardless of how many other VPs read the data, as long as there is global synchronization at block boundaries.

In general, the space overhead that arises in automatically contracting a data parallel program can be reduced by adding synchronization, using dependence information to postpone duplication of a data item, and constraining the programming model. We now briefly discuss each of the first two possibilities, and then discuss the third in a a little more detail.

First, if memory is limited, the contraction system can put a limit on how much duplication is allowable by forcing the computation to synchronize. For example, suppose that one is using an unrestricted send/receive model, and that no more than six copies of any one piece of data were allowed on the actual processor. Every time a VP encountered a send, a new copy of data would need to be created. When the number of copies reaches six, the run-time system could block the sending VP until one of its previous recipients of the same data item no longer needs its copy (because it received a new version, or it terminated, etc.). The tradeoff is that eventually this limit on data duplication may become a limit on parallelism. (Also, there is a theoretical possibility of introducing deadlock through this limit, but the circumstances that would be required to cause this seem highly unlikely to occur in real parallel programs.) If all of the recipients are unable to relinquish a copy of the data in question because one of them is waiting on an off processor communication, parallelism is lost.

Second, dependence information can be used to reduce space overhead. A simplistic approach to duplicating data is to create a new version of an item each time it is sent. This insures that the sender will have a private value to modify after the send. If the run-time system knew that the sender was not going to modify the data until much later in the code, the duplication could be postponed until that point. When this point is reached, it may not be necessary to duplicate the data because the sender yielded to receiver in the interim and the receiver is no longer reading the data. In other words, the data item should only be duplicated when it is a *live* data item.

Third, space overhead can be reduced by constraining the programming model. BSIMD illustrates a model where synchronization restrictions reduce data duplication. Only VPs which own a specific data item can modify that data item. This keeps the number of VPs creating versions of data items to one VP per data item instead of many VPs per data item. Even without global synchronization, this constrains the maximum number of copies of a data item to the number of times the same data item is sent by a VP. As mentioned previously, if the semantic constraint

imposed by BSIMD, that each VP must synchronize at block boundaries, is strictly implemented, this further reduces the number of copies of each data item to two. In this case data is exchanged between the virtual processors only at the block boundaries. VPs cannot continue into the next block until each of them has reached the block boundary and communicated the relevant data with other VPs.

It should be noted, however, that implementing this global synchronization that is semantically imposed by BSIMD, as in the more general case above, potentially reduces parallelism. Furthermore, synchronizing at block boundaries is not always necessary. A VP can continue computation past a block boundary until it first reads data:

- which it doesn't own, and
- which has not yet been written to for the last time by the owner in a previous block (when a VP modifies data for the last time in a block, it can send this data to the VPs which read the data in the next block).

Parallelism may be lost when it is possible for a VP to compute past a block boundary and it instead waits on another VP to reach that boundary.

A solution is to relax the synchronization requirement and allow VPs to continue into the next block as long as the above conditions are satisfied. It is important to require that the synchronization is relaxed only when computation in the current block by other VPs on the same actual processor is not possible. This requirement insures that data is only duplicated in those situations where duplication is necessary for the sake of parallelism. Relaxing the synchronization requirement exposes a tradeoff between space overhead and parallelism.

It should be noted that even with this tradeoff, the BSIMD programming model still reduces space overhead. Ownership of data insures that only one VP per data item need make local copies of data. More significantly, the model forces each VP to lexographically encounter the same number of block boundaries. This allows the run-time system to recognize when duplicates of data can be discarded. This problem is significantly more difficult in the send/receive paradigm.

In summary, by introducing synchronizations, taking advantage of dependence information, and using constraints inherit to the communication paradigm, the space overhead connected with contracting a massively parallel program can be reduced, and probably be kept to an acceptable level. Our research is exploring these issues, starting with the space versus parallelism tradeoff in contracting BSIMD programs that was described above. These results are not part of the preliminary results reported in this paper.

5. Preliminary Results

This section presents and analyzes results of preliminary tests that we made to asses the scheduling heuristics discussed in Section 3. Recall that the goals of the proposed scheduling heuristics are two-fold. First, the heuristics attempt to minimize the computational overhead in choosing the next task to execute. Second, the heuristics attempt to promote overall data locality such that the resulting cache miss rate is comparable to that of manually contracted code.

Three test applications were used to evaluate the scheduling overhead and cache behavior of the proposed heuristics when compared to other policies. These applications are simple Jacobi iterative solvers using these different stencils: five-point star, nine-point cross, and nine-point square. These applications are very common kernels in numerical methods, and are typical BSIMD applications. The results separated in this section are for a 100x100 grid per actual processor, with one virtual processor per grid point. Smaller grids were also tested, and led to roughly the same conclusion

except for unrealistically small grids. (One grid point per virtual processor is the natural way to program these algorithms.)

In evaluating scheduling overhead, we compare the new heuristics to FIFO and LIFO scheduling policies. Both the FIFO and LIFO policies demonstrate the use of a single data structure to store ready tasks. We seek to show that maintaining such a data structure is unnecessary and that by avoiding its use, the overall scheduling overhead can be reduced significantly.

In investigating data locality, we compare the miss rates resulting from applying the new heuristics to the miss rates resulting from the use of FIFO and LIFO scheduling policies. In addition, a random scheduling policy is simulated for each application to provide worst case cache miss rates. We show that data locality doesn't suffer from the use of inexpensive scheduling heuristics.

The test applications were run within a contraction simulator that we constructed. The simulator requires as input a virtual machine, a set of virtual processors and their data, and a mapping of the data to the virtual machine. After initialization, the simulation is driven by the communication between virtual processors. A multitasking library, Awesime [15], is used to implement the virtual processors. Awesime is an object-oriented library for parallel programming and process-oriented simulation. Although Awesime provides extensive support for simulation, modifications were needed to support LIFO and random scheduling policies. The Awesime package also needed to be modified to provide the contraction simulator control over scheduling. Awesime also provides an efficient means of attaining cache miss rates for various cache configurations since it directly integrates a version of the TYCHO [18] cache simulation software.

In obtaining the results below, the following assumptions were made. First we assume that the compiler has annotated where sends and receives between virtual processors occur, and which virtual processors are internal versus external. Second, in our measurements of cache locality, we do not include the data accesses by the run-time system. We expect that their inclusion would affect all the policies about equally. (In fact they might hinder the FIFO and LIFO policies more since these policies access more data in the course of scheduling). Third, we use a tool called pixie [10] to instrument the scheduling code and analyze the resulting execution trace to obtain the cost, in cycles, for each of the scheduling steps in the new heuristics. The code used to measure this cost is a conservative estimation of the code which will actually be used in a working contraction prototype since it lacks a few planned optimizations. The cost of the ready list manipulations performed by the FIFO and LIFO policies are also measured in cycles using pixie. The code used in obtaining the FIFO and LIFO scheduling is derived from the scheduler found in [4] and the resulting cost should be considered a very optimal estimation. The costs we arrived at are 24, 32, 67, and 53 cycles for steps 1-4, respectively, and 45 cycles for the FIFO and LIFO scheduling (which includes removing and adding an elements to the ready list). In the new heuristics, Step #4 requires an additional cost which is dependent on how many searches are necessary before finding a ready task. This cost is 17 cycles per search.

Table 1 shows the frequency and cost (in cycles) for each of the steps in the proposed heuristics, for 50 iterations of the 100x100 grid on each of the three problems. As mentioned in Section 3, a major assumption in designing the heuristics is that the first steps will be used more frequently than the later steps. In particular, it is important that the last step (#4) be utilized as infrequently as possible. The results show that, for these three application, step #4 occurs less than one percent of the time for the last two problems and 1.3% for the first. It is also important that, when step #4 is used, the number of tasks examined before finding a ready task be as small as possible. As mentioned in Section 3, we attempt to minimize these searches by recognizing that step #4 is usually only utilized in the termination phase of the algorithm (for these applications, the last

iteration). The number of searches is reduced by examining only the tasks which are still active. For the problems below, this approach has been successful. Two of the three problems always find a ready task without any extra searches (i.e. "searches" in Table 1 equals zero), while the 5 point test case has to examine two tasks on average every time it must use step #4. Note that even in the worst case, the first problem, step 4 accounts for only 3.9% of the total cost.

Scheduling Statistics for Heuristics	5 point	9 point cross	9 point square
Step #1: Schedule the sending VP.	375,452 (9.01)	386,889 (9.29)	468,488 (11.2)
Step #2: Scheduler the receiving VP.	126,521 (4.05)	171,921 (5.50)	48,905 (1.56)
Step #3 Schedule an external VP from the ready list.	17,112 (1.15)	40,521 (2.71)	20,540 (1.38)
Step #4 Look up a ready VP.	6,898 (0.57)	4,663 (0.25)	5,152 (0.27)
Extra searches required for step #4.	11,832	0	0

Table 1. The frequency and cost of the proposed heuristics for each stencil on a 100x100 system running for 50 iterations. The first number in each box is the number of occurrences of this heuristic; the number in parentheses is the total cost in millions of cycles.

Tables 2, 3, and 4 show the results of comparing the data locality and scheduling cost of the FIFO and LIFO policies with the proposed heuristics on each of the problems. The cache results only address data locality for a specific cache configuration: a 16k two-way set associative configuration. A number of other cache sizes and associativities also were simulated, but their presentation here would add little in differentiating the cache behavior among scheduling policies, since the patterns in cache behavior among scheduling approaches were the same for the problem sizes tested. In the FIFO and LIFO cases it is necessary to choose a means of initializing the ready list. To this end, we show the difference in cache behavior and scheduling cost when FIFO is initialized along rows and along columns. The tables also show two results for both 2 iterations and 50 iterations. The results for two iterations clearly reflect the influence of the initialization whereas the results for 50 iterations are more indicative of the steady state.

5 point dimension = 100	**Proposed Heuristics**		**FIFO** Init by **row**		**FIFO** Init by **column**		**LIFO** Init by **row**	
Iterations	2	50	2	50	2	50	2	50
Cache Miss Rate (percent)	5.48	3.81	5.05	5.20	5.14	5.18	2.60	3.54
Scheduling Cost (millions of cycles)	1.22	14.77	1.80	23.28	1.80	23.28	3.14	42.20

Table 2. Comparison of scheduling policies using a 5 point stencil. (For this problem, manually contracted code yields a miss rate of 5.04%, random scheduling yields a miss rate near 40%.)

The cache behavior presented in these tables shows that the proposed heuristics are very competitive with the miss rates that one would obtain from the expected manual contraction, using a row-wise traversal of the data structure. In addition, the cache behavior using the proposed heuristics is close to the miss rates obtained using either FIFO policy, superior on the first problem and within 31% in the other two. LIFO cache locality is very good in all cases. In LIFO, as soon as a task becomes ready (i.e. the data it was waiting for has been sent), it is put at the beginning of the ready list where it is promptly scheduled. This has the effect of scheduling neighboring communication partners (similar to the proposed heuristics). In the worst case (Table 2), the heuristics are 58% worse than the LIFO miss rate, and are within 29% in the FIFO cases. Thus, the miss rates for all approaches are roughly comparable, with some advantage to LIFO and FIFO.

9 point square dimension = 100	**Proposed Heuristics**		**FIFO** Init by **row**		**FIFO** Init by **column**		**LIFO** Init by **row**	
Iterations	2	50	2	50	2	50	2	50
Cache Miss Rate (percent)	3.32	3.81	2.82	5.20	5.14	5.18	2.60	3.54
Scheduling Cost (millions of cycles)	1.25	14.77	1.80	23.28	1.80	23.28	3.14	42.20

Table 3. Comparison of scheduling policies using a 9 point square stencil. (For this problem, manually contracted code yields a miss rate of 2.82%, random scheduling yields a miss rate near 35%.)

The scheduling cost, unlike the behavior in data locality, differs much more among the scheduling policies. The proposed heuristics are 28 - 62% faster than the FIFO policies and a factor of 2.5 - 4.6 faster than the LIFO policy. Not only is the difference in scheduling cost greater (in most cases), but this cost has a greater weight than the cache miss rate when comparing policies. For example, while performing 50 iterations in the 5 point, 9-point cross and 9-point square stencils and assuming a 13 cycle cache miss penalty (as suggested in [19]), the proposed heuristics would require miss rates of 26%, 10%, and 15% respectively (instead of the achieved 3-4% cache miss rate) to match the larger scheduling overhead of the nearest competitor, FIFO.

9 point cross dimension = 100	**Proposed Heuristics**		**FIFO** Init by **row**		**FIFO** Init by **column**		**LIFO** Init by **row**	
Iterations	2	50	2	50	2	50	2	50
Cache Miss Rate (percent)	3.67	4.02	2.82	3.12	3.36	3.43	1.49	3.54
Scheduling Cost (millions of cycles)	1.41	17.75	1.80	23.34	1.80	23.34	5.74	42.20

Table 4. Comparison of scheduling policies using a 5 point stencil. (For this problem, manually contracted code yields a miss rate of 5.04%, random scheduling yields a miss rate near 45%.)

These results show that the proposed scheduling heuristics do a good job of maintaining data locality by scheduling among communication partners based upon data layout information from the compiler. While having data locality comparable to manually contracted code, the heuristics are computationally inexpensive. The low cost of these heuristics (an average 45% improvement over traditional approaches on the problems tested) allows finer granularity in contracting data parallel codes and will be an important tool in our continuing research on the contraction problem.

References

(1) T. E. Anderson, H. M. Levy, B. N. Bershad, E. D. Lazowksa, "The Interaction of Architecture and Operating System Design", In *International Symposium on Computer Architecture*, 1991.

(2) Arvind, R. S. Nikhil, K. K. Pingali, "I-Structures: Data Structures for Parallel Computing", *ACM Transactions on Programming Languages and Systems*, Vol. 11, No. 4, October 1989, Pages 598-632.

(3) B. Bagheri, B. Raghavachari, and L. R. Scott, "PC, a Parallel Extension of C, the PC Reference Manual", Technical Report, Department of Computer Science, Pennsylvania State University, 1990.

(4) B. N. Bershad, E. D. Lazowska, H.M. Levy, "PRESTO: A System for Object-Oriented Parallel Programming", Software Practice and Experience, Vol. 18, No. 8, August 1988, pages 713-732.

(5) L. Bic, A. C. Shaw, 1988, *The Logical Design of Operating Systems*, Prentice Hall, Englewood Cliffs, New Jersey.

(6) F. C. Chow, J. L. Hennessy, "The Priority-Based Coloring Approach to Register Allocation", *ACM Transactions on Programming Languages and Systems*, Vol. 12, No. 4, October 1990, pages 501-536.

(7) E. C. Cooper, J. G. Morrisett, "Adding Threads to Standard ML", Technical Report 186, School of Computer Science, Carnegie Mellon University, December 1990.

(8) D. Culler, S. Goldstein, K. Schauser, T. von Eicken, "TAM - A Compiler Controlled Threaded Abstract Machine", Technical Report, Computer Science Division, University of California, Berkeley.

(9) T. Derby, E. Eskow, R. Neves, M. Rosing, R. B. Schnabel and R. P. Weaver, "The DINO User's Manual," University of Colorado Technical Report CU-CS-501-90.

(10) Digitil Equipment Corporation, "PIXIE", Unix-style Manual Page for PIXIE, Ultrix V4.2, Rev 96, September 1991.

(11) R. Draves, B. Bershad, R. Rashid, R. Dean, "Using Continuations to Implement Thread Management and Communication in Operating Systems", In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, October 1991, pp. 122-136.

(12) T. von Eicken, D. Culler, S. Goldstein and K. Schauser, "Active messages: a Mechanism for Integrated Communication and Computation", In *Proc. of the 19th Int'l Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

(13) E. W. Felton and D. McNamee, "Improving the performance of message-passing applications by multithreading", *Proceedings of Scalable High Performance Computing Conference*, IEEE Press, 1992, pp. 84-89.

(14) G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu, "Fortran D language specification", Center for Research on Parallel Computation Technical Report CRPC-TR90079, 1990.

(15) D. Grunwald, "A User's Guide to Awesime: An Object Oriented Parallel Programming and Simulation System", University of Colorado Technical Report CU-CS-522-91.

(16) P. J. Hatcher, M. J. Quinn, 1991, *Data-Parallel Programming on MIMD Computers*, The MIT Press, Cambridge, MA.

(17) C. T. Haynes, D. P. Friedman, "Engines Build Process Abstractions", In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pages 18-23, August 1984.

(18) M. D. Hill, "Tycho", University of Wisconsin, Unix-style manual page.

(19) M. D. Hill, "Evaluating Associativity in CPU Caches", In *IEEE Transactions on Computers*, Vol. 12, No. 38, pages 1612-1630, 1989.

(20) S. Kleiman and D. Williams, "SunOS on SPARC", *Sun Technology*, Summer 1988.

(21) B. W. Lampson, J. G. Mitchell, E. H. Satterthwaite, "On the Transfer of Control Between Contexts", In *Lecture Notes On Computer Science: Proceedings ofthe Programming Symposium*, pages 181-203, Springer-Verlag, 1974.

(22) P. Mehrotra and J. Van Rosendale, "Programming distributed memory architectures using Kali", Institute for Computer Applications in Science and Engineering (ICASE), Technical Report No. 90-69, 1990.

(23) J. C. Mogul, A. Borg, "The Effect of Context Switches on Cache Performance", In *International Symposium on Computer Architecture*, 1991.

(24) M. Rosing and J. Saltz, "Low latency messages on distributed memory multiprocessors", Institute for Computer Applications in Science and Engineering (ICASE), Technical Report 92-95, 1992.

(25) M. Rosing, R. B. Schnabel, and R. P. Weaver, "The DINO parallel programming language," *Journal of Parallel and Distributed Computing* 13, 1991, pp. 30-42.

(26) E. Spertus, S. C. Goldstein, K. E. Schauser, T. von Eicken, D. E. Culler, W. J. Dally, "Evaluation of Mechanisms for Fine-Grained Parallel Programs in the J-Machine and the CM-5", In *International Symposium on Computer Architecture*, 1993.

(27) G. L. Steele, Jr., "High performance Fortran: status report", in *Proceedings of Workshop on Languages, Compilers, and Run-Time Environments for Distributed Memory Multiprocessors, SIGPLAN Notices*, Jan. 1993.

(28) D. W. Wall, "Global register allocation at link time", In *ACM SIGPLAN Symposium on Compiler Construction*, June 1986.

(29) M. Wand, "Continuation-Based Multiprocessing", In *Conference Record of the 1980 LISP Conference*, pages 19-28, August 1980.

(30) S. A. Ward, Jr. R. H. Halstead, "A Syntactic Theory of Message Passing", *Journal of the ACM*, 27(2):365-383, April 1980.

(31) R. P. Weaver, "Supporting dynamic data structures at the language level on distributed memory machines", Ph.D. Thesis, Department of Computer Science, University of Colorado, Boulder, Nov. 1992.