

Compiler Support for Implicit Process Sets

Ernesto Gomez* and L. Ridgway Scott†

25th October 2005

Abstract

Much of parallel computing in scientific computation has been limited to static parallelism. We introduce here support for a kind of dynamic, or adaptive, parallelism, arising from implicit process sets in SPMD execution.

Implicit process sets are formed by program logic which selects a subset of processes to execute some section of code that is not executed by other processes. The actual set of processes selected is implicit (rather than spelled out) when it depends on factors not known ahead of time to the programmer, such as input data or computed results. Communication in such sections of code is generally either avoided or given only limited support by parallel programming languages and tools. This restricts the kinds of algorithms commonly implemented to those in which communications can be performed only in global sections of code.

To support implicit process set, we introduce entities we call Merging Implicit Process Sets (MIPS), and a support framework for them. We test the practical possibilities of MIPS support by extending the PC (Parallel C) compiler from the University of Chicago to automatically insert MIPS code in IF and IF-ELSE statements. We display experimental results showing the efficiency of our approach.

*California State University San Bernardino, Department of Computer Science

†University of Chicago, Department of Computer Science

1 Implicit Process Sets

A common form of parallel and distributed execution involves concurrent execution of multiple copies of the same program code (MPI 1, for example, supports this model). We call this mode of execution SPMD (Single Program Multiple Data). SPMD execution implies task parallelism, because each copy of the program has different data, which can lead each process to follow a different computational path [11, 5].

Much of parallel computing in scientific computation has been limited to static parallelism. We introduce here support for a kind of dynamic, or adaptive, parallelism, arising from implicit process sets in SPMD execution. In this work we restrict ourselves to a fixed initial set of processes; we do not allow the processes to grow beyond the initial amount.

Program logic on non-uniform predicates creates subsets of processes that execute different sections of code. If predicates are dependent on data, process number or the results of computation, these process subsets may not be the same on repeated execution of the same program, and it may not be possible to anticipate the membership of these subsets before the program is executed. We call these process subsets “Implicit Process Subsets”, because they are implied but not spelled out in the program logic (unlike explicit process subsets, which must be defined by the programmer, for example by manipulating the MPI communicator) [16].

A key point when considering such process subsets is the identification of merges as well as splits. Process subsets can be merged when processes that are executing different code sections must again execute the same code. For example, processes executing an IF-ELSE construct may take different execution paths as a consequence of the IF predicate, but will again be executing the same code after the the conditional blocks contained in the IF structure. We propose a framework we call “Merging Implicit Process Sets” (MIPS) to support sets of processes that execute the same code concurrently [6].

Communication performed by processes in conditionally executed code is prone to error and may deadlock, because processes required to participate to ensure successful communication completion may not execute the appropriate code. As a result, many SPMD programming systems and languages either have artificial restrictions that prevent communication except in sections of code that can be guaranteed to be executed by all processes (BSP [17], Titanium [18]) or leave it up to the programmer to ensure correctness of

communication in code executed by subsets of processes, without providing any guarantees (MPI [12, 14], Planguages [16, 4]).

Many standard algorithms, for example the upper/lower parallel prefix algorithm described in [8] require or at least could benefit from collective communications between process subsets. If subsets are explicitly defined, support for implicit sets of subsets would be useful to verify that the programmer specified sets are correct. As an alternative, allowing program logic to implicitly define process sets could significantly simplify the programming task.

Imposing restrictions on communication statements depending on considerations of which processes might or might not execute code at runtime seems arbitrary, particularly where the syntax of such statements is otherwise correct and consistent with the language definition. Further, these restrictions impose limitations on programming style and on the kinds of algorithms that may be easily expressed. We resolve this inconsistency in current practice through the MIPS framework. We consider MIPS as a key support required for dynamic process sets, analogous to the way garbage collection supports dynamically allocated languages.

We expect three significant advantages to our approach:

1. Debugging and fault tolerance: communication errors due to missing processes can be flagged, and programmer specified explicit process sets can be verified.
2. Language consistency: by incorporating support for process sets at the compiler, we avoid situations in which the same syntax is either correct or incorrect depending on where placed in the program text or which particular processes execute it.
3. Algorithmic advantage: supporting implementation of algorithms which require or exploit sets of processes that dynamically depend on data or computational results; existing technology strongly discourages such algorithms.

However, the automatic identification and support of implicit process sets requires interprocess communication to create the sets and verify their correctness and uniformity at all involved processes. A main concern of this paper is therefore to test the cost of this communication and evaluate the practical usefulness of our approach.

We begin with a brief review of previous work on explicit process sets. This is followed by a discussion of the basic concept of splits and merges. We follow with a description of a form of overlapping, here referring to the execution of code between communication statements and variable use at senders and receivers, which we introduced in [4] in the context of irregular problems, and which we believe to be important to the efficient implementation of the MIPS framework. We follow with performance results, analysis and conclusions

2 Explicit Process Sets

We briefly review explicit process sets as background for our discussion of implicit process sets. Previous work includes both language and library based approaches. Most systems provide an initial set of processes specified by the system (e.g. `MPI_COMM_WORLD` [12]). Function libraries such as MPI allow the creation of programmer-defined explicit process sets. Languages may restrict communication to sections of code executed by all processes (Titanium [4], Cray Fortran [14]). Other systems do not address the problem of communication between subsets of processes [9], or warn that communications in sections that may not be executed by all processes are dangerous without preventing such communications (Pfortran [3, 4]).

2.1 BSP

BSP (Bulk Synchronous Parallelism) [17] and systems that adhere to it have an execution model in which computation and communication stages alternate. All communication are restricted to specific communication blocks that execute at global barriers. This allows maximal utilization of network bandwidth, guarantees that correctly written communication patterns will not fail due to a missing process and simplifies performance analysis. However, BSP does not efficiently support algorithms that can not be expressed as long sections of independent computation, and any communication between a subset of processes at the global barrier must be specified explicitly.

2.2 Language based approaches

A number of parallel languages have features that could be used to support process subgroups; we are not however aware of any that provide a full solution. Some examples:

Fortran M and CC++ [10] provide mechanisms to link senders and receivers through channels; this mechanism can be used to define groups of processes. However, it is possible for a send or receive operation to incorrectly specify the participating processes communication statements are actually executed, and there is no verification that all supposedly participating processes actually execute the code.

Some systems have attempted to resolve synchronization issues with shared memory through a system of tags. For example, Linda [8] implements messages through a shared memory area; processes write and read to this area independently of each other. Coordination between senders and receivers is carried out through unique matching message tags. Synchronization code is not required; an intended receiver of a message makes repeated attempts to read until a message with the right tag appears. This still presents problems with process subgroups; if write and read statements appear in a section of code that is conditionally executed, then both sender and receiver processes must execute that section of code for communications to be correct.

2.3 Function libraries

Library based approaches allow the calling of communication functions from conventional sequential languages. The most accepted libraries implement the MPI standard [12, 13], which does include support for subsets of processes and communication between them. However, as a linkable library, MPI can not perform the analysis required for automatic identification of subgroups. In many cases, considerations of efficiency prevent MPI from even attempting to guarantee consistency of subgroups. For example, the creation of an MPI communicator as a subset of the set of all processes (in MPI 1) is a local operation; MPI provides no guarantee that the subgroup is consistent at all potential participants in a communicator.

The MPI 2 standard [13] provides operations on intercommunicators that allow creation of process subsets as a result of local program logic (MPI_COMM_SPLIT). It further specifies some calls on intercommunicators (e.g. *WINDOW* creation) that are collective operations. However the

standard does not modify the MPI 1 [12] statement that creating subset communicators is a local operation.

MPI 2 supports single-sided communications (*PUT* and *GET* semantics). Since these operations are on memory locations, they will not block as long as the process that owns the referenced memory is alive. However, synchronization is required to ensure that reads and writes occur in the correct order, and all processes in the process set that could access the referenced memory must execute the synchronization statement (e.g. the MPI 2 *FENCE* statement).

2.4 Fault tolerance

Process sets have also been considered in the context of fault tolerance, both from a theoretical point of view [2] and in the implementation of practical systems [15]. Again, as in all the above cases, these are explicit process sets defined by the programmer.

3 Splits and Merges

In the SPMD model with a fixed number of processes, we consider that all processes start together as members of a single implicit process set. Splits, in which a given set of processes divides into disjoint subsets, are identifiable in the source code; they are conditional control statements that determine what will be executed depending on some predicate. For example, in C, we identify selection statements (*if*, *switch*), loop control statements (*while*, *for*) and conditional jumps (*if* followed by *goto* or *break*) as statements that may produce a split.

A “closest merge” corresponding to a split is the first statement that must be executed by all processes that have passed through the split. For example, in the case of *if*, *if-else* statements, the merge occurs at the first statement following the conditionally executed code block or blocks. The analysis for merges in more general cases is done on the control flow graph ([1]); it is described in [6] and in a forthcoming paper.

We have demonstrated that, given an N - way split, we can always find at most $N - 1$ matching merges, at which the merging process set can be computed from a knowledge of the process sets that were formed at the split. Therefore no communication is required.

We need an *all to all* vector transfer at splits to allow each participating process to compute which other processes are taking the same branch in the code. This vector transfer is potentially expensive; it should be avoided where possible. Given a compiler, two immediate optimizations should be applied. The simplest follows from noting that the purpose of keeping track of process subsets is to support communications. If the code that could be executed between a split and its matching merge contains no communication statements, it is not necessary to track the process set, and no code is required. The second optimization is overlapping, which we now review.

4 Overlapping

The kind of task parallelism we seek to support, particularly when it involves communication by some subsets of processes and not by others, can easily lead to large differences in the execution time of different process subsets between splits and corresponding merges. This is not a problem at the merge itself, since no communication is involved. It can, however, introduce large synchronization waits at communication statements that follow a merge, when processes that have taken different paths need to communicate.

We have previously considered this problem in [7]. We there proposed an overlapping execution protocol, where we use the term to refer to overlapping time intervals between variable definition and use at producer and consumer processes, rather than the more conventional overlapping of computation with communication. Overlapping uses a runtime system to dynamically schedule communications periods between data definition and use at producers and consumers. In that paper, and later in (Dissertation), we demonstrated that the overlapping protocol works to minimize the need to block either producer or consumer processes, and tends to hide the synchronization delays of asymmetric processing. To ensure that processes will be blocked if communications are not completed in time to satisfy a data dependence, the overlapping system maintains a Finite State Automaton (FSA) corresponding to each communication statement, and needs to keep track of usage of communicated variables at each process. This requires calls to a runtime system inserted ahead of each statement where a communicated variable is used; these calls can be inserted automatically by a compiler.

5 Experimental results

A first concern is the efficiency of our approach. We here describe experiments we have carried out to test the implementation of support for the simplest MIPS case, in which process sets are created by *IF* statements.

We have implemented MIPS support as the SOS (Subset support, Overlapping and Short-cutting) function library. (Overlapping and short-cutting are described in short-cutting paper). SOS is a library built on top of MPI communications, callable from C or Fortran. All SOS communication is non-blocking; a communication statement is a declaration that the specified action can happen. Additional SOS calls inform the runtime system about variable usage in the program; the runtime system uses this information to block execution of a particular process if some pending communication must actually occur to preserve data dependences.

We have incorporated the SOS library into an experimental variant of version 2.1 of the PC compiler [16]. (The standard PC compiler, including the test program suite, is available at <http://planguages.cs.uchicago.edu/>. The experimental PC/SOS and PC/MIPS compilers are available on request by email to egomez@csci.csusb.edu). The baseline PC/SOS compiler implements only overlapping support, but not MIPS. A test version of the compiler inserts MIPS code for *IF* and *IF-ELSE* constructs. The compiler performs the optimization of adding code only for *IF* or *IF-ELSE* constructs that have a communication statement in their scope.

In order to isolate performance of MIPS code from other factors, we chose to perform tests on a program that does not in fact split into different process subsets. We chose program `recursion.pc`, part of the standard PC test suite, since it has irregular communication patterns.

Since the PC compiler generates overlapping communications and all communication statements are non-blocking, timing individual communication statements is meaningless. We instead measure total program runtime with and without MIPS support code, using the MPI timer routine.

Measurements were performed on the Cray XD1 at the Gauss Laboratory of the University of Puerto Rico, Rio Piedras. This machine has 6 dual AMD Opteron nodes. Operating system is a 64-bit version of SUSE Linux 9.2. In addition to PC, the Gnu C compiler was used, together with version 1.2.6 of MPICH. Further tests were conducted on the Raven cluster at California State University San Bernardino, a commodity cluster of 13 dual processor, 1.4GHz, 32 bit Pentium machines, connected through a Gigabit Ethernet

switch. Raven runs Red Hat 9.2 Linux, and uses the 32 bit counterparts of the compilers we used on the Cray.

On each computer, we measured three cases:

5.1 Baseline case: no MIPS code.

Without MIPS code, basic program time is essentially all communications, function calls and program logic; the program performs no significant computation. We also ran tests with an artificial computational load of 10^5 repetitions of $x = \log(y) * \log(x) + x$ (for integer y , double x) in each function call.

5.2 Naive MIPS implementation.

A first implementation of MIPS support uses direct MPI calls to implement an all to all vector transfer. The call to perform this transfer at the split is inserted by the compiler immediately after an IF statement (duplicate calls are inserted at the start of each code block that could be selectively executed as a result of the IF). The call to SOS to initialize the stream does not return until the vector combine completes. Since it is hard-coded, the order in which processes participate in the vector combine is fixed. The split code in this case behaves like a barrier.

Instrumenting the code shows that for 8 processes, program recursion.pc performs 39 splits. Since the MIPS group does not actually change, all 8 processes participate in each split; that is we have 39 vector combines of 8 processes each.

We subtract the time this program takes without split code from the runs with splits, and divide by the number of splits to find the time per split. We then added an artificial load of 10^5 repetitions (as above), and repeated the runs. Although the same communications are performed as when the program is run without the artificial load, slight load imbalances and execution time differences on different nodes introduce added synchronization waits, increasing the delay incurred by each vector transfer.

5.3 Overlapping, dynamic scheduling

A better implementation of MIPS splits uses overlapping with dynamic scheduling, utilizing the same message queue implemented by SOS for data transfer.

Case	ms, run, no load	μ sec/split	ms, run with load	μ sec/split
no MIPS	.478 \pm .006	0	124.777 \pm .075	0
static	2.198 \pm .007	50	127.200 \pm .012	62
overlap	1.493 \pm .010	28	125.900 \pm .210	28

Table 1: **Cray XD1, 8 processes** Times are averages of 10 runs, rounded to nearest microsecond (μ sec). Run times are in milliseconds (ms). Error estimate is standard deviation for 10 runs.

In this implementation, the split code actually uses an SOS global reduction operation to implement the vector combine. As a result, the communications required for the vector combine are dynamically scheduled and only forced to complete when the next SOS communication or MIPS statement is encountered. In its current implementation, the SOS runtime code operates in the same execution thread as the main program, interrupting it. That is, communication and computation alternate execution rather than running in parallel. We performed the same sequence of tests, with and without added computational load, as for the first two cases.

Overlapping implementation of split code have the same direct communication cost with and without computational load, but overlapping partly conceals the synchronization delays.

We summarize our results on the Cray in Table 1.

5.4 Experiments on commodity cluster

We have carried out additional experiments on Raven, a commodity cluster located at California State University, San Bernardino. The configuration of Raven is similar to the Cray XD1; both have multiple independent dual-processor nodes.

On Raven, we ran tests with 4, 8 and 16 processes, and we also experimented with varying the assignment of processes to nodes, specifically to test the possibility of differences between running a single process on each dual processor node or running two processes per node. Results are displayed in Table 2. The communications performed by this program do not approach the bandwidth limits of Gigabit Ethernet, but latency effects invisible on the Cray might appear on Raven's slower network. With 4 processes, the program performs 5 recursive calls and 13 splits, with 8 processes there are 15 recur-

P	ppn	Case	run, no load	split	run w. load	split
4	1	no MIPS	.8±.08	0	132.1±.97	0
4	1	static	4.8±.06	.3	132.6±1.4	.3
4	1	overlap	4.8±.11	.3	133.9±.14	1.3
8	1	no MIPS	2.1±.11	0	396.6±3.8	0
8	1	static	21.9±.12	.5	398.5±2.5	.5
8	1	overlap	21.9±.12	.5	401.7±2.0	1.3
8	2	no MIPS	2.0±.05	0	397.1±4.3	0
8	2	static	21.1±.22	.5	398.7±6.4	.4
8	2	overlap	21.1±.28	.5	398.4±4.2	.3
16	2	no MIPS	7.2±.34	0	1031.0±4.0	0
16	2	static	79.6±3.2	.7	1050.0±4.3	1.8
16	2	overlap	78.9±.91	.7	1045.8±3.6	1.4

Table 2: **Raven** Times are given in milliseconds for both run times and split times and represent averages of 10 runs, rounded to the nearest tenth of a millisecond. Error estimates are the standard deviation for 10 runs. P is the number of processes, and ppn is the number of processes per node.

sive calls and 39 splits and with 16 processes there are 39 recursive calls and 103 splits.

5.5 Evaluation of results

Our first conclusion is that, although the communication costs of split code are significant when a program performs little computation, the absolute costs are small. On the Cray, times per split are in tens of microseconds; on the commodity cluster they are on the order of a millisecond (for the number of processes we have been able to test).

Although we see a general upward trend in time per split as number of processes increase, our data does not allow any firm conclusions on what is the rate of increase. Looking at the time per split for the no-load case, the increase appears roughly logarithmic in the number of processes. This is what we would expect for both static and overlapping communications in SOS, which follows a tree pattern. The results with computational load show greater and less consistent variation with number of processes. This is prob-

ably due to slight speed differences between processes running on different machines, forcing some processes to wait longer at each collective communication, a conclusion supported by an observed larger variation between run times for all runs with computational load.

We note that none of the runs without load on Raven showed any differences in time per split for static split code versus overlapping split code. This is somewhat surprising, since the overlapping runtime system requires additional messages that the static code does not require. The static split code implemented the all to all vector transfer required for a split as a reduction followed by a broadcast, requiring $2(N - 1)$ messages, where N is the number of processes. The overlapping code is structured the same way, but it requires an additional protocol message for each data message (short-cutting paper), so the number of messages is doubled. The protocol messages are short (32byte payload), so they do not efficiently use bandwidth and are more affected by latency. Overlapping gains its advantage by exploiting irregularities in timing between different processes. The protocol messages are used to opportunistically schedule sends as destination processes are able to receive them, rather than on a static, fixed schedule. It appears that there exists enough irregularity between processes to gain some advantage from this opportunistic scheduling even without added computational load, so that the extra time needed by protocol messages is never visible. (We note that the present implementation of the SOS library in fact uses a single thread of execution, with the runtime system interrupting the process every 2 microseconds to do its work. Therefore we do not believe any of the advantage of overlapping can be due to parallel execution of computation and communication).

On the Cray (Table 1), the time each split adds to the total computation is always less (by a factor of about one half) for the overlapping code than for the static code. On the commodity cluster, timing differences between overlapping and static code appear only on the runs with computational load, and the advantage consistently depends on how processes are assigned to nodes. If each process is assigned to a different compute node, so that all communications are carried out over the Gigabit Ethernet, then the static code runs faster (Table 2 with ppn=1). If pairs of processes are assigned to each node, so we have one process per CPU, then half the messages are actually transferred in memory and do not go over the network. In this case, the overlapping code is always faster (Table 2 with ppn=2).

We believe these results show the sensitivity of overlapping to the greater

machine	0 bytes	32 bytes	64 bytes	128 bytes
Cray XD1 - same node	1.75	1.87	1.88	2.07
Cray XD1 - network	1.72	1.88	1.93	2.13
Raven - same node	20.21	20.47	20.42	20.60
Raven - network	47.32	48.48	50.05	51.92

Table 3: **Timing of short messages** All times in microseconds.

latency for the required short protocol messages over the network. In support of this conclusion, we have measured latency for short messages on the Cray and on Raven, using the *mpptest* program supplied with the MPICH distributions on each machine. Results are shown in Table 3.

We note that the latency for short protocol messages between Raven nodes is twice the latency for messages on the same node. This somewhat reduces the average message latency when two processes are scheduled on each dual processor node, rather than one process per node. As a result, the overhead of overlap protocol messages is reduced and the system can respond more quickly to pairs of processes that are ready to communicate. The advantage of overlapping is more significant on the Cray, where the latency is one tenth that of Raven and there is no significant difference between same node and different node message times.

6 Conclusions and future work

As a debugging tool, the MIPS framework is useful to verify correctness of explicit process sets defined by the programmer. This eases the implementation of algorithms that require communication in such process subsets; a compiler switch could then be used to remove MIPS support in the final code.

Absolute costs of split code appear to be small enough to justify the application of MIPS code not only in a debugging context (to verify the correctness of communication in process subsets), but also in general parallel programming. Given MIPS support in a compiler, algorithms in which data-dependent subsets of processes communicate become easy to write. We believe the development of such algorithms has been restricted by lack of compiler support; our results suggest that correct and efficient process subset communications requires substantial analysis and code insertion, not practi-

cal to do by hand.

We intend to continue work in developing a compiler that fully supports MIPS for all types of splits and merges. We will also investigate extending the framework to support increasing the number of processes beyond the initial set. We continue to investigate algorithms that could exploit the capabilities MIPS support opens up.

References

- [1] AHO, A. V., AND ULLMAN, J. D. *Compilers - Principles, Techniques and Tools*. Addison-Wesley, 1986.
- [2] ALAN FEKETE, N. L., AND SHVARTSMAN, A. Specifying and using a partitionable group communications service. *ACM Transactions on Computer Systems* 19, 2 (2001), 171–216.
- [3] BAGHERI, B., CLARK, T., AND SCOTT, L. R. Pfortran: a parallel dialect of Fortran. *Fortran Forum* 11 (Sept. 1992), 20–31.
- [4] BAGHERI, B., CLARK, T., AND SCOTT, L. R. Pfortran: a parallel extension of Fortran (the Pfortran reference manual). Tech. Rep. 124, University of Houston, 1992. revised march 1999, Available at <http://planguages.cs.uchicago.edu/> .
- [5] FOSTER, I. Task parallelism and high performance languages. *IEEE Parallel and Distributed Technology* (1995).
- [6] GOMEZ, E. *Single Program Task Parallelism*. PhD thesis, University of Chicago, 2004.
- [7] GOMEZ, E., AND SCOTT, L. R. Overlapping and short-cutting techniques in loosely synchronous irregular problems. *Lecture Notes In Computer Science* 1457 (August 1998), 116–127.
- [8] HARRY F. JORDAN AND GITA ALAGHBAND. *Fundamentals of Parallel Processing*. Prentice-Hall, 2003.
- [9] HOARE, C. A. R. Communicating sequential processes. *Communications of the ACM* 21 (1978), 666–777.

- [10] IAN FOSTER. *Designing and Building Parallel Programs*. Addison-Wesley, 1995.
- [11] JORDAN, H. F. The force. In *The Characteristics of Parallel Algorithms*, L. H. Jamieson, D. B. Gannon and R. H. Douglass, eds. (1987), MIT Press.
- [12] MESSAGE PASSING INTERFACE FORUM. *MPI: A message passing interface standard, Version 1.1*. 1995. Available at <http://www.mcs.anl.gov/mpi>.
- [13] MESSAGE PASSING INTERFACE FORUM. *MPI: A message passing interface standard, Version 2.0*. 1997. Available at <http://www.mcs.anl.gov/mpi>.
- [14] NUMRICH, R. W. F-: a parallel extension to cray fortran. *Scientific Programming* 6, 3 (1997), 275–284.
- [15] RICCIARDI, A. M., AND BIRMAN, K. P. Using process groups to implement failure detection in asynchronous environments. Tech. Rep. O-8971-439-2/91/007/0341, ACM, 1991.
- [16] SCOTT, L. R., CLARK, T. W., AND BAGHERI, B. *Scientific Parallel Computing*. Princeton University Press, 2005.
- [17] SKILLICORN, D. B., HILL, J. M. D., AND MCCOL, W. F. Questions and answers about BSP. Tech. Rep. TR-15-96, Oxford University Computing Laboratory, 1996.
- [18] YELICK, K., SEMENZATO, L., PIKE, G., MIYAMOTO, C., LIBLIT, B., KRISHNAMURTHY, A., HILFINGER, P., GRAHAM, S., GAY, D., COLELLA, P., AND AIKEN, A. Titanium: A high-performance Java dialect. In *ACM 1998 Workshop on Java for High-Performance Network Computing* (New York, NY 10036, USA, 1998), ACM Press.