Performance and modularity benefits of message-driven execution

Attila Gürsoy\textsuperscript{a,*} and Laxmikant V. Kale\textsuperscript{b}

\textsuperscript{a} Computer Engineering Department, Koç University, Rumelifeneri Yolu, Sarıyer, Istanbul 34450, Turkey
\textsuperscript{b} Computer Science Department, University of Illinois at Urbana-Champaign, 1304 W. Springfield Avenue, Urbana, IL 61801, USA

Received 20 September 1999; revised 9 November 2003

Abstract

Processor idling due to communication delays and load imbalances are among the major factors that affect the performance of parallel programs. Need to optimize performance often forces programmers to sacrifice modularity. This paper focuses on the performance benefits of message-driven execution, particularly for large parallel programs composed of multiple libraries and modules. We examine message-driven execution in the context of a parallel object-based language, but the analysis applies to other models such as multithreading as well. We argue that modularity and efficiency, in the form of overlapping communication latencies and processor idle times, can be achieved much more easily in message-driven execution than in message-passing SPMD style. Message-driven libraries are easier to compose into larger programs and they do not require one to sacrifice performance in order to break a program into multiple modules. One can overlap the idle times across multiple independent modules. We demonstrate performance and modularity benefits of message-driven execution with simulation studies. We show why it is not adequate to emulate message-driven execution with the message-passing SPMD style. During these studies, it became clear that the usual criteria of minimizing the completion time and reducing the critical path that are used in SPMD programs are not exactly suitable for message-driven programs.

\copyright 2004 Elsevier Inc. All rights reserved.

Keywords: Parallel programming; Message-driven; Message-passing; Modularity; Latency tolerance; Multithreading; SPMD

1. Introduction

Message-passing programming model and distributed memory machines are common platforms for running large parallel scientific and engineering applications. A significant performance bottleneck in such parallel computations is the processor idling due to communication latencies and load imbalances. Optimizing the performance often forces programmers to sacrifice modularity, however, reuse in the form of modules and libraries for developing large software is a widely used technique for better software systems. This paper focuses on the performance benefits of message-driven (data-driven) execution model for parallel programs composed of multiple libraries or modules where a module might correspond to a set of coordinated activities performing a particular task. In message-driven model, there are many entities (e.g., objects, user level threads, handlers, etc.) on each processor. The runtime system activates the entity when the data it is waiting for is available. The idea of triggering computation by the availability of data is not a new concept. In general, dataflow computational model has been applied and investigated in many areas of computer science such as processor design, multithreaded architectures, and parallel programming. In this paper, the focus is on performance benefits of message-driven execution model within the context of parallel programming of distributed memory machines. We argue that modularity and efficiency together can be achieved more easily in message-driven execution model than in message-passing SPMD style.

1.1. Latency

Physical reality dictates that access to remote information will be slower than that to local information. This slowdown generally occurs for two reasons: the delay introduced by the communication network and software overhead due to protocol stacks in the
communication software and operating system, which forms the communication delay. The other source of delay, which is often more significant than the communication delay, is the delay in the creation of the information that the remote processor, as illustrated in Fig. 1. The time interval between the arrival of the request and the completion of the requested service is the response delay. The response delay can be longer than the service time itself and is often unpredictable. The communication delay and the response delay together constitute the remote information access latency, or simply referred to as latency in the rest of the paper. This latency may be caused by (transient as well as sustained) load imbalances, chains of dependences possibly forming critical paths, in addition to hardware and operating system behaviors. Minimizing the impact of this latency on overall performance is a major objective in parallel programming, especially on large scalable parallel machines.

One can focus on reducing the overall communication delay at the system level. For example, better switching hardware and interconnection networks can reduce the transmission delay, although physical reality dictates that remote access would always be slower than local access. The software overhead can be reduced by better operating system support, and streamline implementations. VI architecture standard for PC clusters [7], and their precursors such as active messages [57], FM [45] and PM [53] are examples of such an approach. A complimentary approach to reduce the impact of the latency is to overlap it with useful work.

1.2. Hiding latency by overlapping

Overlapping latency with useful work by switching to another computation—that is having multiple contexts in which a processor switches between them in order to hide latencies—has been used at various levels in computer systems from the architecture level to the language level. Although our work focuses on explicit parallel programming of nonshared address space machines, there are significant research and work on overlapping latency to improve performance, particularly in dataflow and multithreaded execution models. The results of these studies, that is, impact of hiding latency on the application performance apply to our study as well. In parallel computing, the idea of triggering computations with the availability of data was used at the level of individual instructions in dataflow architectures [4,11]. Dataflow architectures are inherently parallel and they offer an elegant solution to the memory latency problem. However, pure dataflow approach is too fine-grained and this has limited its practical implementations.

Multithreading has been the most popular approach to hide latencies. HEP [50] was one of the first fine-grained multithreaded (interleaved multithreading) architectures which supports fast context switching at every cycle, so that when a thread stalls due to memory access, another thread can be resumed almost instantaneously. Many fine grain independent pieces of computation, or concurrency in a program, is needed to hide memory access latencies and this was a major limiting factor for utilizing the machine. TERA [3] is a more recent example of such machines with more advanced hardware mechanism combined with use of compilation techniques. Fine-grain multithreading improves performance [43], however, dependence on compiler technology to extract concurrent work from application programs remains to be a major issue [8].

The hybrid dataflow execution model [9,14,16,26] which combines conventional control-flow with dataflow applies dataflow approach at a coarser level where each thread is a sequence of instructions. The performance improvements due to hiding latencies have been reported by many performance and simulation studies [37,51]. The potential benefits are limited by context-switch and synchronization overhead, and locality. The impact of these factors and ways to deal with them are studied by many researches including [39–42].

Latency tolerance techniques for shared address space cache-coherent multiprocessors have been studied extensively. An excellent review can be found in [10] discussing various techniques such as prefetching, block-data transfer, precommunication, and multithreading to hide latencies as well as the fundamental limitations and factors reducing benefits of multithreading. Lack of concurrency in applications, network bandwidth can limit degree of multithreading and latency tolerance [8]. Simulation studies and experiments show that latencies can be hidden effectively with multithreading [35].

The architecture level solutions typically emphasize tolerating memory access delays, which is a special case of “communication delays” as described above. The broader delays constituting the remote information...
access delay (i.e. additional delays due to synchronization, load imbalances, etc.) experienced by parallel applications on distributed memory architectures can be dealt with better at the level of programming model or language level. Handling latency problem at the software level is a current problem especially for distributed memory (address space) machines.

At the software level, multithreading can be used in software distributed shared memory for improving performance through overlapping. In [56,38], it is shown that multithreading can result in reduced execution time of real applications. Some other examples of improving performance includes application level adjustments such as rearranging communication operations by issuing sends early and postponing receives as much as possible to overlap communication delays [10,23]. Multithreading can be used in the context of message passing and at the application level again for hiding latencies (using multithreading and MPI) [5].

Message-driven programming is a promising model in this regard, and is the one that we focus on in this paper. In the message-driven model, which is distinct from message-passing, there are typically many entities (e.g. objects, user level threads, handlers, etc.) on each processor. No entity is allowed to block the processor on which it is running while waiting for some remote data on. Instead, the system activates the entity when the data it is waiting for is available. This improves latency tolerance into ways. First, when one entity is waiting for data from remote processor, another ready entity may be scheduled for execution. Second, even a single entity may wait for multiple data items simultaneously, and continue execution whenever any of the expected items arrived.

One of the early implementations of message-driven execution was in the Charm++ programming language [27,30,31], which we use in the studies reported in this paper. The message-driven entities in Charm++ are “objects”, and remote data is brought to these objects in the form of asynchronous method invocations. Threads (either user-level threads, or system-level kernel threads) that pass messages to each other, as in Chant [21] constitute another example of message-driven systems.

Several other message-driven approaches include one-sided (handler based) communication in Nexus [13], the recent optimistic form of active messages [58], user-posted handlers in Multipol [59], dataflow like threaded procedures in [9,54].

Some recent performance results of applications developed with message-driven and data-driven multi-threaded systems have been reported emphasizing performance improvements due to computation/communication overlap [46,55,60]. In addition to the performance benefits, the message-driven approach has significant modularity advantages [1,29] over the prevalent message-passing model, as embodied in MPI [12] or PVM [52]. However, there has not been an extensive comparison of the message-driven approach with the traditional message passing programming model.

This paper aims at such a comprehensive qualitative and quantitative comparison of the two approaches. The next two sections of the paper define and compare the two programming models. The message-driven execution model allows user programs to adapt dynamically to run-time variations and unpredictable latencies. Thus, the message-driven models are well-suited for the next generation of parallel applications, which tend to model irregular, unpredictable, and dynamic behavior. The performance benefits of message-driven execution manifest themselves especially with programs composed of multiple modules or libraries. Reuse of software is desirable even in sequential context. Reuse of parallel software modules, which require more effort to produce, is even more desirable. As we discuss in Sections 2 and 3, message-driven execution enables efficient composition of software modules. More specifically, modular applications can be developed in the message passing model also; but it is not easy to overlap the idle times in one module with computations from one of the other modules. This ability to overlap idle times across multiple independent modules and libraries is a major advantage of the message-driven model. For example, transient load imbalances in one module can be ameliorated by adaptively and automatically overlapping computations from another module. One can attempt to impart such adaptive behavior to message passing programs by using nonblocking message passing primitives. Section 4 discusses why such an approach is not adequate. The quantitative performance comparison of the two approaches is presented in Section 5. It uses several synthetic benchmarks as well as real applications. To understand the performance comparison on the wide variety of current parallel architectures, as well as future generations of parallel machines, we utilize simulation studies. This allows us to understand, especially, how the comparison is affected by changes in communication to computation ratios. During these studies, it became clear that the usual criteria of minimizing the completion time and reducing the critical path that are used in SPMD programs might not be suitable for message-driven programs. A schedule which allows overlapping of idle times (not necessarily minimizing the critical path) can result in shorter execution time. Relevant and appropriate criteria in this context are developed as well. Proper scheduling and mapping of tasks to processors is vital for achieving high performance for message-passing or message-driven programs and is particularly challenging for irregular and dynamic problems [2,34]. In this work, the studies were limited to the default greedy scheduling of messages of the underlying execution model.
2. Traditional SPMD model and overlapping

The predominant paradigm used for programming distributed-memory machines is provided by the SPMD—single program multiple data—model which is supported by vendors of parallel machines in their operating system. PVM and MPI are examples of commonly used such message-passing libraries. The SPMD model simplifies program development by using a simple mechanism for internal synchronization and scheduling. The phrase SPMD has been used with somewhat different meanings by different authors [15,17,36]. In the SPMD model, as used in this paper, there is one process per processor with a single thread of control (usually all processes are executing the same program). Communication among processes (hence processors) is usually with blocking primitives. Messages have tags, and the `receive` primitive blocks the processor until a message with a specified tag arrives (of course, there is no reason not to use non-blocking communication occasionally if it does not complicate the code). Moreover, we use traditional SPMD model to mean strict usage of blocking receives. In Section 4, we consider SPMD message-passing programs with non-blocking receives and give arguments why SPMD with non-blocking receives also are inadequate for parallel modular software development.

A single thread of control and blocking receives makes the programming of these machines relatively easy. Most of the synchronization requirements of the computation are handled implicitly by the blocking receives. For example, consider the code in Fig. 2(a). The computation of $t_1$ and $t_2$ needs the remote information $a$ and $b$. Since the `recv` statements will not allow the processor to proceed until the required messages are received and made available to the code, the correct computation of $t_1$, $t_2$, and $t_3$ is guaranteed.

The simplicity of the flow-of-control attained in SPMD is at the expense of idling processors. After issuing a blocking receive, the processor must wait idly for the specified message to arrive. This wait may not always be dictated by the algorithm, i.e., the algorithm may have more relaxed synchronization requirements. Yet the usage of blocking primitives forces unnecessary synchronization and may cause idle time. This idle time can be decreased by rearranging the send and receive operations. This involves moving the sends ahead and postponing the receives as much as possible in the code. This type of structure is common in SPMD programs that are either written by a programmer or automatically generated by a compiler [15]. However, this strategy cannot handle cases with more complex dependencies and unpredictable latencies. For example, consider again the code in Fig. 2(a). The code contains two receives followed by computations $t_1$, $t_2$, and $t_3$. $t_1$ and $t_2$ are independent of each other and need data from different receives. One of the receives can be postponed to overlap latency of the other one. If the latencies for these receives are known a priori, then the one with longer latency may be postponed. However, in general, unpredictable communication delays make it difficult to know the latencies in advance. In case of such unpredictability, if the one that is postponed turns out to be the wrong one at runtime, then the idle time on the processor would be the same as the time for the original code. Thus, the traditional SPMD model cannot adapt to such runtime conditions by rearranging the send and receive operations.

2.1. Traditional SPMD is inadequate to develop efficient large programs

Although SPMD model can achieve limited performance improvements as discussed in the previous section, it cannot overlap computation and communication across modules and libraries. This is true even with the non-blocking receives. A module or library is defined as an independently developed program that can be called from other programs. For example, the code in Fig. 2(b) invokes two computations (`par_fft` and `par_sum`). In the SPMD style, invocation of another module passes the flow of control to that module. Until that module returns control, the calling program cannot continue. Therefore, the idle times that a module experiences cannot be overlapped with computation from another module.\footnote{See Section 4 for a discussion of how one can use non-blocking message passing primitives for this purpose in SPMD style and why this is not adequate.} In Fig. 3, for example, module $A$ invokes two other modules $B$ and $C$. Module $A$ cannot activate $B$ and $C$ concurrently even if the computations...
in B and C are independent of each other. As a result, the processor time is not fully utilized, as illustrated in the same figure.

Notice that this problem is independent of communication latency. Even if communication latency were to be zero, each of the modules may have idle times on individual processors due to critical paths and load imbalances in them. Despite its simplicity, the traditional SPMD model is far from being a programming model for developing efficient large parallel applications for these reasons. The message-driven execution helps to solve these problems.

3. Message-driven execution and overlapping

Message-driven execution, in contrast to the SPMD model, supports many small entities or objects per processor. These objects are activated by the availability of messages that are directed to them. At this level of description, it suffices to say that each object has a state and a set of functions (methods) for dealing with incoming messages.

A message-driven system can be thought as a collection processors where each processor has a pool of messages and a collection of objects. A message scheduler, running one copy on each processor, selects one of the messages from the pool, identifies the destination object and method from the message, and then invokes the method. Message-driven execution overcomes the difficulties experienced by the SPMD model. It can effectively overlap latency with useful computation adaptively. The rearrangement of send and receives in SPMD, for example as in Fig. 2(a), failed to achieve the overlap adaptively. Message-driven execution, on other hand, could compute either \( t_1 \) or \( t_2 \) depending on which message arrived first (or whichever message is made available for processing by the message-scheduler), hence, adapting itself to the runtime conditions.

A message-driven parallel object to perform this computation is illustrated in Fig. 4 using the Structured Dagger language (Structured Dagger and Dagger [19] are coordination languages in the Charm++ programming environment). A parallel object in Structured Dagger specifies pieces of computations (when-blocks) and dependences among computations and messages. A when-block is guarded by dependences that must be satisfied before of messages or completion of other constructs. In Fig. 4, the overlap construct calculates \( t_1 \) and \( t_2 \) in any order depending on the arrival order of messages. Once \( t_1 \) and \( t_2 \) are calculated and when a message to tag3 is received, then \( t_3 \) is calculated. Further details of the Structured Dagger language can be find in [28]. The purpose of this small example was to show how the message-driven parallel objects are programmed in Structured Dagger.

3.1. Message-driven execution and modularity

The message-driven paradigm, more importantly, allows different modules that might have some concurrent computations to share processor time. Consider the computation discussed in Section 2.1. Assuming that modules B and C have independent subcomputations (which is a reasonable assumption since the modules are developed independently and computing distinct subproblems), the idle times on a processor can be utilized by another module if it has some work to do. Such a scenario is illustrated in Fig. 5. Module C gets processor time (by virtue of having its message selected by the scheduler) while B waits for some data, and vice versa, thus achieving a better overlap than the SPMD program in Fig. 3.

In traditional SPMD model, library computations are invoked by regular function calls on each processor. The library call blocks the caller on all processors. After the completion, the library module returns the result and the control to the calling module. Some disadvantages of libraries in SPMD style are:

1. Idle times in the library computation cannot be utilized even if there are other independent computations.
2. Caller modules must invoke the library on all processors even when only a subset of the processors provide input, or receive output.
3. Library computations must be called in the same sequence on each processor, otherwise the program can enter into a deadlock.

![Fig. 4. Structured Dagger illustrating adaptive overlapping.](image1)

![Fig. 5. Message-driven modules share the processor time.](image2)
These drawbacks force SPMD style programmers to use flat codes (i.e., merging potential library modules into the main or caller module) for better performance. The second requirement, namely “every processor has to call”, might be a real bottleneck for developing large efficient programs. The communicator context approach of MPI which creates communication domains does not really solve this problem. It mainly solves the usage of the same message tag in different modules. The idle times of processors cannot be overlapped with computations of different modules because of the explicit receive command. On the other hand, message-driven style encourages creation of smaller and more reusable modules. Therefore, we expect libraries to be a major strength of message-driven systems in the future.

4. Emulating message-driven style in SPMD

In this section, we will discuss how message-driven execution can be emulated within the SPMD context and examine the adequacy of such an emulation. The emulation of message-driven execution involves usage of nonblocking message passing primitives, particularly the nonblocking receive primitive. The examples presented in this section will use a primitive operation called probe and the blocking receive primitive instead of nonblocking receive. The probe(tag1) function checks if a message with tag tag1 has arrived. A probe followed by a blocking receive (if probe succeeds) is equivalent to nonblocking receive for our purposes.

4.1. Using nested if blocks

A simple approach to incorporate message-driven ideas in SPMD programs is to use nested if statements and nonblocking receives to replace some of the “blocking receive and compute” sequences. These sequences are the part of the code where the rearrangement of receives does not help for dynamic situations—as in the example of Section 2.

To illustrate this, the SPMD code in Fig. 2(a) is rewritten as shown in Fig. 6. The if statement in the modified code first checks whether the message with tag tag1 has arrived. If the message has arrived, it computes t1 and blocks to receive the message with tag tag2. After this message arrives, it computes t2. However, the messages may arrive in the reverse order. Therefore, in the case the message tag1 has not arrived, the else part of the if repeats the same thing for the reverse order. If both messages have not arrived yet, the code must wait until they arrive (while statement) in order to continue with the rest of the code (computation of t3). The while loop must assure that both t1 and t2 have been computed in order to compute t3. This requires synchronization variables and/or counters. In this simple dependency case, a flag, not_done, is sufficient. However, in general, the synchronization can be quite complicated by the dependencies among the receives and computations. In addition, the complexity of the loop increases with the number of concurrent receives. In general, if there are n receives, then the code must handle n! permutations of receive sequences. Finally, this approach will not take full advantage of message-driven execution since these if code-blocks will be scattered around the code and the computations across the if code-blocks cannot be executed concurrently. For these reasons, the nested if approach is not sufficient to exploit the full power of message-driven execution.

4.2. Using a global while-switch loop

A more structured approach to approximate message-driven style is to use a global while-switch construct in a module as shown in the Fig. 7. This approach requires the SPMD program to be decomposed into code-blocks (functions f1(),..., fn()) such that these functions can be executed upon receiving a particular message. The
processor continuously checks if a message (with any tag) has arrived. Whenever a message arrives, the processor invokes the appropriate code-block, depending on the tag of the message. A simple (and clear) application of this strategy is when the SPMD program can be decomposed into functions such that each one depends on a single message and there are no dependencies among the functions. Otherwise (i.e., in the presence of complex dependencies), the functions or the loop itself have to be augmented with all the synchronization constructs. As a result, the code would not appear as readable as the one in the example. Although the global while-switch loop appears to be more powerful than the nested if approximation, it still has some fundamental limitations. These limitations can be summarized as follows:

**Difficulty in supporting multiple modules**: Every module must know the entities in other modules to prevent conflicts. Such global entities include the tags of messages defined in different modules. Adding a new message tag requires knowing all of the tags used across the modules, which destroys modularity. The MPI message passing interface [12] solves this tag problem by providing the notion of *contexts*. Another name conflict is the function names. Again, the modules must use different names for a multiple module compilation. These two name conflicts destroy modularity in the compiling phase.

**Centralized changes**: Any change in the modules, such as addition of more messages, may result in modifying the global loop structure. One may try to provide one individualized loop per module to increase modularity. But then, passing the flow of control across modules becomes difficult (the example discussed in Section 2.1). When module B receives a message that belongs to module C, then it should call the appropriate function in C. This further complicates a modular design of message-driven programs.

**Dependencies must be reflected in the loop**: The dependencies among the functions and messages must be handled either inside the function or in the loop, presenting a further programming difficulty.

**No message scheduling**: The basic while-switch loop has no scheduling control strategy. The messages are processed in the order in which the processor provides them to the global loop (which is FIFO).

**Difficulty in supporting dynamic objects (computations)**: Finally, this approach in SPMD does not support dynamic creations of objects (here the functions with their local state).

These limitations prevent the SPMD approach from being a programming model for developing large, efficient parallel applications. A message-driven language seems to be a better method for developing such applications. However, such a pick-and-process message loop is a step in the right direction. In fact, such loops are used in the underlying implementation (or runtime system) of message-driven languages, which are often done on top of an SPMD system. A programmer, in the absence of a message-driven language, should use such a loop to derive the performance benefits of message-driven execution.

5. A quantitative performance study

There are numerous factors affecting the performance of parallel programs that cannot be captured by simple analytical models. These are factors such as scheduling policy, irregular computation times, irregular interactions among sub-computations, and load imbalances. In order to take such factors into account, we have developed a simulation system and conducted a simulation study to determine the impact of latency on both message-driven and SPMD programs and to quantify the benefits of message-driven in the form of overlapping idle times with useful computation (particularly across modules). The simulation study allowed us to vary various machine parameters systematically such as network latency and communication coprocessors involvement. The studies investigated the effect of the following cases: network latency, random variations in network latency, dependences among computations, idle times due to load imbalances, and the effect of a communication coprocessor. While studying these benchmarks, it became clear that the usual criteria of minimizing the completion time and reducing the critical path that are used in algorithm design are not exactly suitable for message-driven programs. Relevant and appropriate criteria in this context are developed in the context of an example in Section 5.4.

5.1. Simulation of message-driven programs

Simulation of message-driven programs are challenging due to their non-deterministic behavior (message-driven execution is based on the ability to perform computations in different orders depending on the arrival of messages/data). This non-determinism requires an execution-driven simulation technique for accurate results. Execution-driven techniques, on the other hand, are very costly since they execute parts of the program during the simulation. We have developed a trace-driven simulation method to avoid large computational costs over multiple simulation runs. The difficulty with the trace-driven simulation method is that the traces gathered from a particular run may not be valid under a new environment due the changes in the order of message arrival. The simulator cannot reconstruct a new sequence of events if the only information is the traces of execution of code blocks from one particular execution. Usually, parallel programs define
a partial order of events. If this partial order is available to the simulator, then it becomes possible to reconstruct a new total order of events that does not violate the partial order. In [20], we identify the information that is necessary to carry out such simulations and the necessary conditions for an accurate simulation and describe a method for extracting such information from program executions. The simulator is based on the Dagger programming language. Dagger facilitates the reconstruction of a new total order of events by (a) tracing the execution at the level of basic blocks (when blocks) rather than only messages, and (b) providing information about dependences among messages and computations. We conducted the simulation studies using this trace-driven simulator system. The parallel machine modelled by the simulator is a collection of processing nodes connected with a communication network. Each processing node consist of an application processor (AP), a communication coprocessor (CP), and local memory. The information is exchanged through messages between the nodes. For the simulation results presented here, the important architectural parameter is the delay that a message experiences. The communication delay does not only take place in the network. The software overhead on the processing nodes is usually more significant. The total delay between the time an AP starts sending a message and the time the message becomes available at the destination AP is broken down into various delays. These delays occur in each component: sender AP, sender CP, network, receiver CP, and receiver AP. The delays at each component is modelled with the widely used $a + nb$ expression where $a$ is the startup time (for each component), $b$ is the time per unit data, and $n$ is the total length of the message. The simulation studies for the effect of latency and communication processor were conducted by systematically changing these parameters.

5.2. Selection and description of benchmarks

This section describes the benchmark algorithms and programs that have been selected to conduct the simulation study. As it is shown before, message-driven programs benefits from the availability of multiple concurrent computations. Message-driven formulation of parallel programs differs from SPMD formulation to exploit such cases. Therefore, along with their description, we will describe the SPMD and message-driven formulation of the benchmark algorithms. The selected algorithms can be grouped in the following classes:

- The first group consists of synthetic benchmarks, which are prototype algorithms which contained features that create opportunities for overlap. Two benchmark programs, Wave and Milib, were developed to demonstrate the advantages of message-driven execution. Although these computations are synthetic, they reflect possible practical applications (particularly large numerical applications [44]). The purpose of the synthetic computations is to examine and quantify the performance advantages of message-driven programs in situations where opportunities for overlap exist.
  - The second group contains the concurrent global reduction operations. Global reduction operations (such as inner product, global sum, etc.) are very common in many computations. The global communication nature of the reduction operation creates idle times in processors that cause performance degradation in parallel applications. Many algorithms with reduction operations have been modified to reduce this bottleneck. Among them, asynchronous reductions [48] and pipelined reductions [47] can be counted, as well as reducing the number of reductions by combining a few of them together [6]. Due to the widespread necessity and importance of such reduction operations, the concurrent reduction benchmark was chosen to represent this class of computations.
- The third benchmark is the core part of a parallel implementation of a CFD algorithm—Harlow–Welch [22].

5.2.1. Milib

This benchmark models an application that invokes two or more collective operations (implemented as separate parallel libraries/modules). The collective operation has its own computation and communication phases. Each processor performs its local computation and invokes a number of independent collective operations. Eventually, each collective operation is completed and results are returned to each processor. After all the collective operations are completed, the same procedure is repeated for a number of times. Fig. 8 illustrates the message-driven and SPMD formulation of the Milib benchmark for two-modules case. In the message-driven formulation, parlib_reql(req1) is a non-blocking call. It starts the first collective operation and the result is to

![Fig. 8. Milib-message-driven (Structured Dagger) and SPMD formulations.](image)
be returned with a message of type \( M \) to the entry \( \text{req1} \) (which is used in the \( \text{when} \) clause of the Structured Dagger language). Due to the message-driven model the processor switches between modules. The SPMD version, on the other hand, invokes the parallel library and passes the control to the library module (since communication is done explicitly with receive operations in the library).

5.2.2. Wave

This program models an application where each processor performs some local work \( L_i \) and collaborates in a parallel operation \( G_i \) by providing input from its local work \( L_i \). The result of \( G_i \) is needed by only one of the processors for its next local phase \( L_{i+1} \) while the other processors proceed to compute their \( L_{i+1} \) phase without waiting for the result of \( G_i \). The processors form a 2D mesh connection logically. The data dependences are such that \( G_i \) is accomplished by a diagonal wave that starts at the upper-left processor and sweeping all the processors. In order to propagate the wave, a processor receives two messages from it (north and west side, for instance), processes them, and sends messages out in the wave direction (south and east). This type of dependency pattern is common in many parallel computations such as triangular systems [24].

After completing \( L_i \), each processor invokes a parallel library for \( G_i \). In the SPMD formulation, this invocation transfers the control to the library. Therefore, the processors that do not need the result of \( G_i \) cannot continue with \( L_{i+1} \) until \( G_i \) is finished. On the other hand, the message-driven formulation allows a processor to work on \( L_{i+1} \) and \( G_i \) concurrently, if the processor does not need the result of \( G_i \). Wave is a simple example of applications that some processors do not need the result of collective (parallel) libraries but all processors cooperate in the execution of the libraries.

5.2.3. Concurrent global operations

Many application programs involve multiple global operations (and their associated pre- and post-computations) that are independent of each other. In traditional SPMD programs, a global reduction adds a barrier: all processors must arrive at the barrier before any one is allowed to proceed beyond the barrier. Here, the barrier created by the global operations are completely artificial. It is simply an artifact of the blocking style of control transfer embedded in the underlying SPMD programming model.

This example is abstracted and modified from a real application—a core routine in parallelized version of a molecular mechanics code. Each processor has an array \( A \) of size \( n \). The computation requires each processor to compute the values of the elements of the array and to compute the global sum of the array across all processors. Thus, the \( i \)th element of \( A \) on every processor after the operation is the sum of the \( i \)th elements computed by all the processors. In the traditional SPMD model, this computation can be expressed with a single call to the system reduction library (such as \texttt{MPI\_REDUCE}) preceded by the computation of the array on every processor. Alternatively, one can divide \( A \) into \( k \) parts, and in a loop, compute each partition and call the reduction library for each segment separately, thus pipelining the computations, since computation of each partition is completely independent from others. The computations of the next \( k \) items (i.e., the next partitions) do not depend on the result of the reduction so that they could be started even before the reduction results from the previous partitions are available. With this (message-driven) strategy, one process that has just finished computing a partition is willing either to process the result of the reduction of any previous partition or compute the next partition. Thus the wait for reduction results for a partition is effectively overlapped with the computation of other partitions. The message-driven and SPMD formulation of the concurrent reductions is listed in Fig. 9.

5.2.4. Harlow–Welch

Harlow–Welch is a fractional step method [33] to solve 3D unsteady incompressible Navier–Stokes equations. As a preliminary part of this research, a parallel implementation of this method was carried out [18]. A brief description of the parallel algorithm is given here. The computational domain involves uniform grid spacing in one direction and nonlinear on the other two directions. The algorithm consists of multiple time-steps. At each step, intermediate velocities are calculated, at each grid point, then fast Fourier transform (FFT) is applied on the uniform axis. After FFTs, \( n \) independent 2D linear systems (penta-diagonal) are solved, and inverse FFT operations are performed along the uniform axis again. The computational domain is partitioned into rectangular boxes which extend along the uniform axis as shown in Fig. 10(a). Due to this decomposition, FFT calculations become local operations. However, the \( n \) linear systems, called

```java
class concrdn {
    stategy concrdn (MSG *msg) {
        overlap {
            for (i=1; i<k; i++)
                atomic { deposit(i) }
            for (i=1; i<k; i++)
                atomic { deposit(i) }
            for (i=1; i<k; i++)
                atomic { deposit(i) }
        }
    }
}
for (i=1; i<k; i++)
    MPI\_REDUCE(msg, comp(n));
```

Fig. 9. Concurrent reductions - message-driven (Structured Dagger) and SPMD formulations.
messages have only the network delay $x_{\text{net}} + n\beta_{\text{net}}$ which can be completely overlapped by useful computation if there is any. The programs are simulated by varying $x_{\text{net}}$ systematically. In the second experiment, the computation phases are varied randomly to observe the effects of load imbalances. For investigating the effects of communication processors, a set of simulations were conducted by varying the components of the message delay (i.e., delays in network, AP, and CP) but keeping the total delay fixed. The numbers are presented in cpu cycles. For a meaningful comparison, the average time for basic computation blocks (that is, computations triggered with the arrival of messages) are provided with the discussion of the results, and the values for latencies are selected from a range to be close to actual parallel machines.

5.3.1. Effects of network latency

Figs. 11(a)–(c) show the completion time of the Mlib benchmark for $k = 1, 2, 3$ concurrent collective operations on 16 processors. The average subcomputation time is 10,000 cpu cycles ($\text{comp}1$ and $\text{comp}2$), and the latency is varied between zero and 10,000 cpu cycles. At zero latency, the completion time of the message-driven and SPMD versions are the same for all cases, $k = 1, 2, 3$, because processors do not experience any idle time due to communication. For $k = 1$ (Fig. 11(a)), the completion time of message-driven and SPMD versions increase equally with the latency because there is no other computation that can be performed during the idle time caused by latency. For $k = 2$ and 3 (Figs. 11(b) and (c)), the SPMD program starts performing more poorly as the latency increases. The behavior of the message-driven program resembles the one in the simple analytical model. For small values of latency, the curve is almost straight. After some point, it starts to increase though with a smaller slope than the SPMD program. The straight part of the curve is longer with three concurrent libraries since there are more opportunities to overlap the latency than the one with fewer library invocations. Fig. 11(d) shows a similar behavior for the Wave benchmark. However, the SPMD and message-driven versions differ at zero latency which will be discussed in Section 5.3.2.

Fig. 12(a) shows the simulation studies for the concurrent reductions benchmark of size $n = 2048$ (per processor). The traces were gathered from runs on 256 processors with various values of $k$ (number of partitions/pipelining). Each curve depicts the performance of message-driven and SPMD programs with different numbers of partitions ($k = 1, 8$ and 64). The mean value of sequential computation blocks are 73,000, 15,000, and 4800 cpu cycles for $k = 1, 8$ and 64, respectively (as calculated from the traces). For $k = 1$, the message-driven and SPMD programs perform equally, since there is only one reduction. For a given

xy-planes in the figure, to be solved are decomposed among processors as shown in Fig. 10(b). These linear systems can be solved by various numerical algorithms. Jacobi’s method is one of the common iterative algorithms for such systems. The easy and efficient parallelization of the method makes it a prototype parallel algorithm and its communication pattern is similar to many other iterative solvers. Therefore, the result of this experiment is likely to be applicable to similar solvers. The Jacobi’s method calculates the values of the unknown vector at iteration $n + 1$ by using their values from iteration $n$. For five-point stencil and mesh-based decomposition, each processor exchanges the boundary points with its neighbors and calculates the next iteration in parallel. This requires four neighbor messages. After every iteration a convergence is performed to determine if the solution has been reached. The convergence test requires a global operation across processors, such as sum or max type of reduction operations.

In this implementation, every iteration basically has four phases: for all planes, (a) send boundaries, (b) receive boundaries, (c) calculate, (d) and do the convergence test. If message transmission time is longer than the time the cpu spends on sending the messages, then it must wait idly in order to continue with the calculation. Similarly, during the reduction operation, it must wait for the result of the reduction. Since the linear systems (xy-planes) are independent of each other, they can be be solved concurrently in order to utilize the idle times. With message-driven formulation, a generic parallel solver library can be invoked for each xy-plane concurrently without bringing the coding complexity of the solver into the main application.

5.3. Simulation studies and results

These studies were conducted on the SPMD and message-driven formulations of the benchmark programs to investigate how the two approaches handle variation in communication latencies and load imbalances. In the first experiment, the AP and CP components of the message delays are set to zero, i.e.,
$k > 1$, the completion time of the message-driven program, $t^k_{md}$, is always smaller than that of SPMD program, $t^k_{spmd}$. If we compare $t^k_{spmd}$ to $t^k_{md}$, their relative performance depends on the latency, number of processors and problem size. For this problem environment, message-driven program for $k = 8$ and 64 performs better than even the best SPMD program. In general, as latency increases or the number of processors

Fig. 11. Effects of network latency: (a) Mlib with one concurrent module, (b) Mlib with two concurrent modules, (c) Mlib with three concurrent modules, and (d) wave.

Fig. 12. Effects of network latency: (a) Conc. reduction with $k$ partitions and (b) Harlow–Welch with $k$ partitions.
increases, the message-driven program’s relative performance improves (because those two factors contribute to the computation time of a reduction). For $k = 64$, however, the message creation and handling overhead cancels partially benefits from overlapping.

Experiments were conducted for the Harlow–Welch on 64 processors. The size of the computational grid per processor was chosen to be $8 \times 8 \times 64$ (the whole grid was $64 \times 64 \times 64$). The subdomain is further divided into $k$ partitions (1, 2, 4, and 8). The simulated completion time of the programs, $t_{md}^k$ and $t_{spmd}^k$, is plotted for various values of $\beta_{net}$ in Fig. 12(b). The average computation time is 78,000, 41,000, 22,000, and 12,000 cpu cycles for $k = 1, 2, 4, 8$. The $t_{spmd}^k$ and $t_{md}^k$ are the same, as expected, and they increase equally with the network latency. When we partition the domain (thus pipeline the operations), the time curve of message-driven program becomes flatter than that of the SPMD program. However, for larger values of $k$, the increase in the overhead due to the increase in the number of messages pushes the curve upward. For a given value of $k$, the performance of message-driven program is always better than the corresponding SPMD program. Also, the message-driven program does better than the best SPMD case ($k = 1$) when $k = 2$ and 4 (or $\beta_{net}$ is very large).

### 5.3.2. Effects of delays due to dependences

Even in the absence of network latency ($\beta_{net} = 0$), the message-driven wave performs better than the SPMD wave as shown in Fig. 11(d) ($\beta_{net}$ component is negligible for this program). This is because wave has inherent idle times in it due to the nature of dependences and critical path. The time needed to propagate a wave one step is 10,000 cpu cycles (the computation done at each step of the wave), and 8 steps are required to complete the wave (number of blocks on the diagonal of $4 \times 4$ mesh times 2). During one wave completion (there are multiple concurrent waves), only a small number of processors are busy at a time. Therefore, processors experience idle times due to the dependences in the algorithm. Note that the SPMD version has to make a blocking receive call on all processors to cooperate in the wave computation. However, the message-driven program switches to other available computations adaptively. These include production of data for subsequent iterations and handling of other waves. In the message-driven case, there may exist multiple waves active at a given time. Thus, the message-driven program interleaves the computations of multiple waves along with the computations of subsequent iterations.

### 5.3.3. Effects of load imbalances

In the wave problem, the SPMD program took longer than the message-driven one, even at zero latency, due to the delays created by computations on other processors. Similar delays may occur in the case of irregular or varying computation requirements. In the Mlib case, if the computation times of the libraries vary across processors randomly, then we expect the message-driven program to adapt itself to the variation and outperform the SPMD program. In order to validate this observation, the latency experiment is repeated by varying the size of subcomputations (in parallel libraries invoked) between the communication phases randomly on each processor (exponentially distributed with a mean of 10,000 cpu cycles). The results comparing uniform case (each computation phase is 10,000 cpu cycles) and random case are shown in Fig. 13 for two and three concurrent modules. Now, at zero latency, the message-driven program outperforms that of the SPMD for the same value of $k$, because it overlaps idle times caused by delays on the other processors by scheduling the computations adaptively. Due to the variations, SPMD cannot make correct scheduling decisions statically.

![Fig. 13. Effects of load imbalances: Mlib with varying computation load: (a) Mlib-random with 2 modules and (b) Mlib-random with three modules.](image-url)
5.3.4. Effects of communication processor

In this section, the effect of a communication processor (CP) is studied. In many parallel architectures, a major part of the communication delay occurs in the network interfacing. The communication related processing during this phase can be performed by the processor itself. Alternatively, a separate CP may take part of the load from the application processor [25, 48, 49]. Having a CP obviously increases the availability of the processor for useful computation. Therefore, overall performance of parallel computations is expected to be better. In this section, the impact of CPs on the SPMD and message-driven programs will be studied. The experiments are conducted as follows: the network delay and the sum of the delay which takes place in AP (application processor) and CP is kept constant. However, the part of the delay that belongs to AP and CP has been varied as illustrated in the Fig. 14(a).

Fig. 14(b)–(d) show the result of the simulations. The completion time of the programs are plotted versus to the ratio of the delay spent in the CP to the sum of AP and CP processor delay (i.e., \(0 \leq \frac{x_p + \beta_p}{x_p + \beta_p + x_c + \beta_c} \leq 1\)) in Figs. 14(b)–(d). The impact of the CP is significant. As a larger part of the communication cost is handled by the CP, the completion time decreases for both SPMD and message-driven programs. This is because the processor carries out less work. The decrease in the completion time for the SPMD case may appear counter-intuitive at first (i.e., one may reason that message transfer time is fixed, therefore the SPMD computation will experience the delay eventually). Indeed, an SPMD program with RPC-like communications (i.e., every send is followed by an awaited response from another processor) will not benefit from the CP. However, SPMD programs with consecutive sends or receives, as well as pipelined SPMD algorithms, will benefit from this because the CP takes some of the processing and the processor may continue with the next computation. The message-driven programs benefit more, as shown in the results, since there exist more opportunities to utilize the time freed up by the CP. The effect of the CP is more dominant for the communication bound cases: (1) Harlow–Welch since it sends four more messages per partition (compared to the concurrent reductions), and (2) concurrent reductions with 64 partitions since communication/computation ratio is

![Fig. 14. Effect of the communication processor: (a) adding a communication coprocessor, (b) concurrent reductions, (c) concurrent reductions, and (d) Harlow–Welch 4 partitions.](image-url)
high. For these cases, pushing the communication cost to CP helps improve performance of both SPMD and message-driven computations. However, after some point, SPMD computations start to suffer from communication latencies, whereas those which are message-driven will not suffer as long as some other task exists. The CP may sometimes become a bottleneck (at least in transient phases of the program). The delays in the CP bottleneck are analogous to remote information service delay, which message-driven execution can handle effectively.

5.3.5. Effects of random latency variations

This section presents the effect of variable delays in the communication latencies. Such delays may be caused by the load in the network; they may also be caused by other unrelated factors on machines such as networks of workstations where the network resources are being utilized by unrelated processes. The simulation experiments are conducted similar to the network latency study. During the simulation, the network latency \((x_{\text{net}})\) for each message is randomized (exponentially distributed with a mean of \(x_{\text{net}}\)) as opposed to a constant (uniform) \(x_{\text{net}}\) for each message. The completion time of the benchmarks are plotted as a function of average network latency \((x_{\text{net}})\) in Figs. 15 and 16. The same experiments are repeated with constant network latency for comparative purposes. The times for the constant and the random cases are plotted for each SPMD and message-driven program as well as the difference between the constant and the random cases (time for random minus time for constant) on the same plot to be able to see better how SPMD and message-driven versions handle variations. As shown in the graphs, message-driven programs tolerate the variations in the network delays better. In other words, variation in latencies has more impact on the SPMD programs. This result is not surprising. As expected, message-driven computations utilize the additional idle times if possible.

5.4. Load balance versus critical path

As has been observed so far, the message-driven execution overlaps idle times of concurrent computations successfully. This property leads to different algorithm design techniques for message-driven execution. In an SPMD computation, the idle time which a processor experiences depends heavily on the critical path of computations (which execute on some remote processor or processors). Because the SPMD model cannot effectively utilize the idle time across modules, the algorithms must be designed in such a way that the critical path of computations is minimized.

However, minimizing critical paths may not be always good for message-driven programs. Consider a situation where a program can call two independent modules M1 and M2. When running in isolation, the running time for M1 is \(t_1\) and that of M2 is \(t_2\). In an SPMD program, the two modules are called one after the other, and therefore the completion time for the caller is \(t_1 + t_2\) (as \(t_1\) and \(t_2\) are critical paths in M1 and M2, respectively). It is clear that one must minimize the critical path to obtain good performance.

In a message-driven program, however, the completion time of the combined algorithm is \(t_1 + t_2 - t_{\text{overlap}}\), as discussed in earlier chapters. The overlapped time depends on a variety of factors and may range from zero to \(\min(t_1, t_2)\). Therefore, it is feasible to change the algorithms in M1 and M2 in such a way that the critical paths \(t_1\) and \(t_2\) are increased, while the opportunities for overlap are also sufficiently increased, thus increasing \(t_{\text{overlap}}\) and reducing the overall completion time.

If M1 were to keep a particular processor busy for all its time \(t_1\) and M2 also would keep the same processor busy for its whole duration, then clearly the overlapped time would be zero. Conversely, if the two modules utilize the processors in a complementary way so that the busy processors in M1 are relatively idle in M2 and vice versa, then the opportunities for overlap are higher. Thus, load balance would seem to be another property of the algorithm that is as important as the critical path, and one can consider increasing the critical path for decrease in load imbalance.

This issue is brought to the surface by a study involving the concurrent reductions experiment discussed in previous sections. To further understand this issue, a series of experiments were conducted which are described in the next sections.

5.4.1. Load balanced spanning trees and message-driven execution

We create opportunities for a tradeoff between critical path and load balance by choosing variants of the reduction algorithms. A reduction operation is generally implemented by using a spanning tree to combine the data that reside on each processor. The spanning tree that was used in the studies was a hypercube-specific spanning tree. It guarantees that a node and its immediate descendants are its direct neighbours. The maximum branch factor on each node was 4. The work done on each processor is proportional to the number of its children. The processors are thus divided into five load groups: processors that have 0, 1, ..., 4 children. Another possible tree is a similar tree with maximum branching factor 2. In this case, processors are grouped into three load groups. If we compare those two spanning trees, the first one has shorter critical path, but has more unbalanced load.

Experiments with these two trees were performed on 64 processors. The concurrent reductions and Harlow–Welch method were executed with different branching factors. Table 1 shows the completion time of the
concurrent reductions \((k = 64)\) and the Harlow–Welch program \((k = 8)\). As shown in the table, the completion time of the message-driven programs decreases as the branching factor decreases. On the other hand, the completion time of the SPMD programs increases as the branching factor is reduced.

Although the critical path of the spanning tree with branching factor 2 is longer, the delays on the critical path are overlapped with other computations. The more balanced load, therefore, shortened the execution time. Fig. 17 shows the individual processor utilization (simulated) for the concurrent reduction program. The spanning tree with branching factor 2 has better load balance. The grouping of processors into classes with approximately equal utilization (as seen on the figure) corresponds to their number of children in the tree. However, there exists some load imbalance due to the leaf nodes. In the case of multiple reduction operations, this imbalance can be further improved by using complementary spanning trees.

5.4.2. Complementary spanning trees for multiple reductions

In order to further reduce the load imbalance in concurrent reductions, one can use two separate spanning trees. The leaf processors in a reduction
operation can be an internal node for another reduction operation which takes place concurrently. The Harlow–Welch program was modified to examine the effect of further load balance on the completion time. Two reduction library modules were programmed. The second one complemented the first one (i.e., leaf nodes in the first one were an internal node in the second one).

Table 2 shows the result of complementary trees. For simplicity, the complementary trees were implemented using a natural sequencing instead of hypercube specific ordering (with wormhole routing, the immediate connection is not that important in any case). The data for the hypercube ordering are duplicated from the previous table for comparison. The completion time of the Harlow–Welch goes further down with the complementary trees as shown in the table.

Fig. 16. Effect of randomly varying network latency: (a) conc reductions 8 partition-SPMD, (b) Harlow–Welch 2 partitions-SPMD, (c) conc reductions 8 partitions data driven, (d) Harlow–Welch 2 partitions-data driven, (e) conc reductions 8 partitions-difference, and (f) Harlow–Welch 2 partitions-difference.

Table 1
Effects of branching factor

<table>
<thead>
<tr>
<th>Branching factor</th>
<th>Concurrent reductions</th>
<th>Harlow–Welch</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Data-driven SPMD</td>
<td>Data-driven SPMD</td>
</tr>
<tr>
<td>4</td>
<td>305 566</td>
<td>1159 1463</td>
</tr>
<tr>
<td>3</td>
<td>266 591</td>
<td>— —</td>
</tr>
<tr>
<td>2</td>
<td>230 656</td>
<td>1151 1576</td>
</tr>
</tbody>
</table>

Further load balance on the completion time. Two reduction library modules were programmed. The second one complemented the first one (i.e., leaf nodes in the first one were an internal node in the second one). Table 2 shows the result of complementary trees. For simplicity, the complementary trees were implemented using a natural sequencing instead of hypercube specific ordering (with wormhole routing, the immediate connection is not that important in any case). The data for the hypercube ordering are duplicated from the previous table for comparison. The completion time of the Harlow–Welch goes further down with the complementary trees as shown in the table.
5.5. Results

We set out to determine and quantify the performance benefits of message-driven execution. Briefly, the conclusions can be summarized as follows. Overall, the message-driven execution imparts substantial performance benefits in a wide variety of cases. Message-driven execution derives these benefits from its ability to adaptively schedule messages. Its performance benefits do not come from the presence of communication latencies alone; they also arise because of the idle times in individual subcomputations due to the critical paths and load imbalances.

1. In fact, for applications where the sole benefit for message-driven execution is due to overlapping communication latencies, the advantage is limited to a bounded region because of the presence of overhead on one side and relative insignificance of the communication latencies on the other side. The extent of this region depends on the particular application and also depends on the technological parameters, such as the ratio of processor speed to communication latencies. As this ratio will probably become worse in the future, even this region is likely to increase its extent as faster processors are used.

2. Whenever there is irregularity and variance, either in the network latency or in computation times, etc., the message-driven execution adapts better and gives superior performance.

3. The message-driven strategy also exploits the potential of a coprocessor better than an SPMD style, as it can effectively utilize the processor time freed up by the coprocessor.

4. The most significant advantage of message-driven execution occurs when there are multiple independent subcomputations within a parallel program. Suppose two parallel algorithms, A1 and A2, are written in two modules, M1 and M2, such that the main program makes two independent calls to M1 and M2 simultaneously. There are three cases in which advantages of message-driven execution can be analyzed.

Case 1 is when the two modules are such that each individually has no idle time at zero communication latency (for example, the Mlib program with uniform subcomputations). If two such algorithms are composed using a message-driven model, then the performance advantage comes only if there is a significant communication latency. So the combined algorithm can tolerate the communication latency somewhat better than each individual algorithm. However, this is often tempered by the overhead introduced by the message-driven execution. In this case, there is a narrow range in which the message-driven execution can benefit.

Case 2 is when the two algorithms individually have variations in their subcomputations (i.e., the time for subcomputations on different processors that belong to the same algorithm may vary). Such situations may create unpredictable idle times on processors due to dependences in the algorithms (for example, the Mlib program with non-uniform subcomputations). In this case, message-driven execution helps, because it can adaptively switch between the algorithms depending.

Case 3 is when algorithm A1 as well as algorithm A2 (or at least one of them) has significant idle time (due to dependences) when run by itself even in presence of no communication latency (for example, the wave program). In such cases, message-driven execution has a major advantage. It allows one to adaptively utilize the idle times within one algorithm for advancing the other algorithm and vice versa.

In SPMD style, once another module is called, the caller is blocked until the callee returns. The only way to prevent this, in SPMD style, is to flatten the code (i.e., combine code for multiple modules). Thus message-driven execution has an essential advantage that cannot be duplicated in SPMD program when overlap across multiple modules is involved.
The simulation studies presented here is substantiated by a recent implementation of a large molecular dynamics simulation program, NAMD [32], scaling to thousands of processors. NAMD is implemented on top of Charm++ and consists of many parallel modules. A recent performance study [46] shows how message-driven execution helps to reduce impact of idle times across modules.

6. Conclusion

The purpose of this work was to study message-driven execution as a mechanism for improving the efficiency of parallel programs. Message-driven execution can potentially lead to improved performance either in presence of idle times of communication latencies or when multiple algorithms are composed where each algorithm has its own idle time due to its critical path and load imbalances. Message-driven libraries are easier to compose into larger programs, and they do not require one to sacrifice performance in order to break a program into multiple modules. One can overlap the idle times across multiple independent module invocations.

A simulation study was conducted involving a few synthetic benchmarks and a few taken from real applications. The performance benefits of message-driven execution are enhanced when there are variations in communication latencies or individual computations. The performance advantages are not limited to communication latencies alone. If there are idle times in the individual algorithms, then message-driven programs are able to overlap and utilize processors better in such cases. It was also shown that a communication coprocessor can be exploited much better by a message-driven program than an SPMD program. As observed in the performance studies, the overhead per message in a message-driven system can be important in that it can offset the performance benefits of message-driven programs for cases where the average grain size of the computation (useful work per message) and the overhead is similar.

Criteria for designing message-driven algorithms were identified and were shown that they tend to be somewhat different from those for SPMD algorithms. In particular, in SPMD programs, the critical path is sole important criterion, whereas for message-driven algorithm load imbalance also acquires substantial importance. This is because load imbalance affects the degree of overlap one can attain with another independent module.

We expect that the inherent performance benefits and compositionality of message-driven execution will lead to its becoming a predominant style for writing large parallel applications.

References


Attila Gursoy received a B.S. in computer engineering from Middle East Technical University, Ankara, Turkey, in 1986 and an M.S. in computer science from Bilkent University, Ankara in June 1988. He continued his research in parallel programming environments, earning a Ph.D. in computer science from the University of Illinois at Urbana-Champaign in 1994. He spent 2 years as a postdoc at the Theoretical Biophysics group, Beckman Institute, Urbana. He was with Bilkent University between 1997 and 2002. He is currently an assistant professor of computer engineering department at Koc University. His research interests include parallel algorithms, computational biology and bioinformatics. He is a member of the IEEE, IEEE Computer Society, and ACM.

Laxmikant V. Kale received a B.Tech in electronics engineering from Benares Hindu University, Varanasi, India, an M.E. in computer science from the Indian Institute of Science in Bangalore, and a Ph.D. in computer science from the State University of New York, Stony Brook. He spent 2 years at the Tata Institute of Fundamental Research. In 1985 he went to the University of Illinois at Urbana-Champaign, where he is now full professor of computer science. His research interests include parallel programming tools, performance feedback and debugging, scientific and engineering applications of large scale parallel computing, and parallel processing for AI. He is a member of the IEEE, IEEE Computer Society, and ACM.