Performance Benefits of Message Driven Executions

L.V. Kale and A. Gursoy
Department of Computer Science
University of Illinois at Urbana Champaign
1304 W. Springfield Ave., Urbana, IL-61801

Abstract

This paper argues that message driven execution is an effective and efficient way of dealing with communication latencies and irregularities in parallel computations. It describes how message driven execution is supported in Charm portable parallel programming system. It goes on to discuss a performance study on Intel's Paragon machine, that demonstrates the performance advantages conferred by message driven execution, in the context of global reduction operations. To simplify specifications of message driven programs, Charm supports a notation called Dagger and a visual editor that accompanies it which is also briefly discussed.

1 INTRODUCTION

Communication latency and unpredictable delays in remote response times constitute significant impediments to achieving high performance on massively parallel computers. A processor arriving at a global operation such as `gisum` must wait idly until all processors arrive at that point, the global reduction is then computed, and the result returned to the processor. Even when a processor is waiting for a message from just a single neighboring processor, the neighboring processor may not be in a position to send this message due to the state of computations on that processor. So, even if communication technology were to keep pace with improving processor speed, processor idling while waiting for messages will remain a significant factor. Message driven execution — as distinct from mere message passing — is a strategy designed to overcome this hurdle.

In message driven execution, a process does not block the processor it is running on while trying to receive a message. Instead, processes are scheduled for execution depending on the availability of the messages for them. Processes typically provide code in the form of entry functions or continuations and a way of associating them with specific incoming messages. With this information, the runtime system can invoke the appropriate code in the appropriate process to handle a particular incoming message. Charm is one of the first systems to embody message driven execution in a portable parallel programming system running on stock multicomputers.

In this paper, we will present a performance study that demonstrates the benefits of message driven execution. In Section 2, we will briefly introduce Charm. Section 3 defines the problem — a simplified version of a core routine in a parallel application, describes the

library for global operations supported in Charm and the programming solution we developed using it. It also presents and analyzes the performance data obtained with Charm and with the native reduction library provided by the system. Section 4 describes Dagger, a notation that simplifies the expression of message driven programs, and a visual editor for it.

2 CHARM

Charm [1, 2, 3] is a machine independent parallel programming system. Programs written using this system will run unchanged on MIMD machines with or without a shared memory. The programs are written in C with a few syntactic extensions. The system currently runs on Intel's Paragon, iPSC/860, iPSC/2, NCUBE, CM-5, Encore Multimax, Sequent Symmetry, ALLIANT FX/8, single-processor UNIX machines, and networks of workstations.

Programs consist of potentially small-grained processes (called chares), and a special type of replicated processes, called branch-office chares. Charm supports dynamic creation of chares, by providing dynamic (as well as static) load balancing strategies. There may be thousands of small-grained chares on each processor, or just a few, depending on the application. Chares interact by sending messages to each other and via specific information sharing modes.

A Charm program consists of chare definitions, message definitions, and declarations of specifically shared objects in addition to regular C language constructs (except global variables). A chare definition consists of local variable declarations, entry-point definitions and private function definitions as illustrated in Figure 1. Local variables of a chare are shared among the chare's entry-points and private functions. Private functions are not visible to other chares, and can be called only inside the owner chare. However, C functions that are declared outside of chares are visible to any chare. Entry-point definitions start with an entry name, a message name, followed by a block of C statements and Charm system calls. Details about these systems calls (such as `CreateChare, SendMsg`), and other features of the system (information sharing abstract data types) can be found in [4]. The Charm runtime system is message driven. It repeatedly selects one of the available messages from a pool of messages in accordance with a user selected queueing strategy, restores the context of the chare to which it is directed, and initiates the execution of the code at the entry point.

3 PERFORMANCE BENEFITS

Message driven execution helps enhance the performance of parallel applications in multiple ways. When there are multiple "processes" (chares) per processor — which is a feature Charm supports — and a process needs to wait for a message, control is automatically transferred to another process which has a message to process, thus keeping the processor utilized. Even a single chare can wait for multiple data items concurrently, processing whichever one

```
chare chare-name {
        local variable declarations
        entry EP1 : (message MSGTYPE *msgptr) {C code block} ...
        entry EPn : (message MSGTYPE *msgptr) {C code-block}
        private function-1() {C code block} ...
        private function-m() {C code block } }
```

Figure 1: Chare Definition

is available first. (When many messages are available, the system chooses one based on user selected priorities or scheduling strategies).

These advantages are particularly significant in the context of global operations. In traditional SPMD programs, a global reduction adds a barrier: all processors must arrive at the barrier before anyone is allowed to proceed beyond the barrier. Many application programs involve multiple global operations (and their associated pre and post computations) that are independent of each other. 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. Message driven execution provides a solution in this context. We will illustrate its performance advantages by means of an example.

This example is abstracted and modified from a real application — a core routine in parallelized version of a molecular mechanics code, CHARMM. 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 (`gssum`) preceded by the computation of the array on every processor. Alternatively, one can divide the array A into k parts, and in a loop, compute each partition and call the reduction library for each segment separately. Each call to the reduction library is a blocking one, i.e. the code cannot initiate the local computation belonging to the block before receiving the result of the current reduction. Therefore, in either case, each reduction call makes the processor wait until the reduction is complete. As the reduction involves a critical path of at least $\log p$ messages (either via a spanning tree, or via a virtual hypercube-topology based algorithms), the idle time on all processors is proportional to $\log p$ and to the size of A. An approximate expression for the completion time for the blocking SPMD model can be written as:

$$T_{\text{blocking}} = nt_l + (nt_r + k\alpha + n\beta)\log p$$

where t_l is the computation per data item, t_r is the reduction computation per data, α is the communication startup time, p is the number of processors, and β is the communication cost per data item being sent.

However, the computations for each partition are completely independent. In particular, computation of the next k items (i.e., the next partitions) are not dependent on the result of the reduction, and so 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 to either 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 time to completion in this case is given by:

$$T_{\text{message\_driven}} = nt_l + nt_r + kt_o + (\alpha + (\beta + t_r)\frac{n}{k})\log p$$

where t_o is the overhead per partition in a message driven execution model. This overhead includes context switching, scheduling and dispatching of the messages in addition to overhead introduced at the user level algorithm. Overlapping is reflected by the fact that the communication term is only proportional to the size of one partition ($\frac{n}{k}$) instead of the size of whole array (n). In effect, one is required to wait only for the result of the last reduction. One pays for this benefit with the overhead t_o because of the larger number of messages involved. A performance study is needed to determine if this tradeoff is worthwhile.

To implement the message driven strategy for the performance study, we used the concurrent reduction library provided in Charm which is described next.

3.1 Reduction Library

The Charm reduction library supports non-blocking global reduction operations. Due to the asynchronous nature, the interface between the reduction module function and the user program is different from the blocking version of the reduction (e.g., gshigh). First, the caller must provide a return address for the result. Secondly, if there is a need for more than one reductions that may overlap in time, they have to be carried out without mixing them up. This can be achieved by having different instances of the reduction object. The library provides the following functions in order support concurrent reductions:

id = ReductionType::Create(size) This call creates an instance of the reduction object and returns a unique identifier.

ReductionType::depositData(id,x,y,fptr,caller_id) A processor participates in a reduction by depositing its data to the reduction object. In this function call, the data to be reduced is `x`. `fptr`, and `caller_id` is the return address. The reduction object calls the function `fptr` of the calling chare `caller_id` when the reduction is over and the result is placed in `y`.

ReductionType::depositDataMsg(id,x,ep,caller_id) This is another version of the deposit call. The difference is that the result is returned in a message to the entry point `ep`.

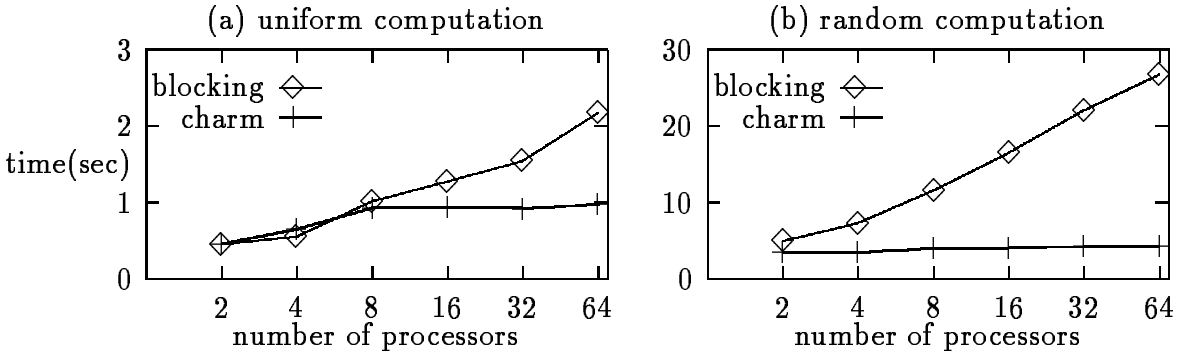

Figure 2: Concurrent Reductions, k=160,n=40960

3.2 Performance Results

We will present some performance results of the message driven implementation of the above computation. The computation was programmed in C (using the Intel supplied native reduction library) for the blocking SPMD version and in Charm for the message driven version. The runs were made on an Intel/Paragon machine.

In Figure 2-(a) shows the performance results of the case k=160, n = 40960, and up to 64 processors. The effect of pipelining of reductions in Charm is apparent from the flattening of the curve beyond eight processors. The increase up to eight processors with Charm can be attributed to the increase in the branching factor of the spanning tree used by the reduction library.

Each processor in the above experiment did a fixed amount of computation (one floating point addition) per element of the array before calling the reduction. In real applications, this computation is likely to be larger, and more important, likely to vary from processor to processor. Figure 2-(b) shows results of the computation for the same parameters but with a random amount of computation added to each partition. The performance benefits of message driven execution becomes more significant when there exist irregularities in the computation. The blocking version makes every processor wait at a barrier for the last processor to arrive at the barrier, thus making the completion time the *sum of maxima* (for all partitions) as opposed to the *maximum of the sum* for the message driven version.

To explore and understand the behaviour of the two versions over a broader range of parameters, we performed more detailed studies. These involved varying k (the number of partitions) and exploring a larger range of processors. Part of the data we obtained is shown in Figure 3-(a), which shows the performance of the cases $k = 16$ and 64, $n = 16384$ and up to 256 processors. As the performance of the blocking version is optimum with $k = 1$, we also include the data for this case. For the Charm program, the execution time increases very slowly with the number of processors, while the time for the blocking version increases significantly. The reduction function provided by the system appears to be proportional to $\sqrt{p}$ where p is the number of processors. This could be due the preliminary nature of the current version of the Paragon software (or our use of it) and can be expected to be improved

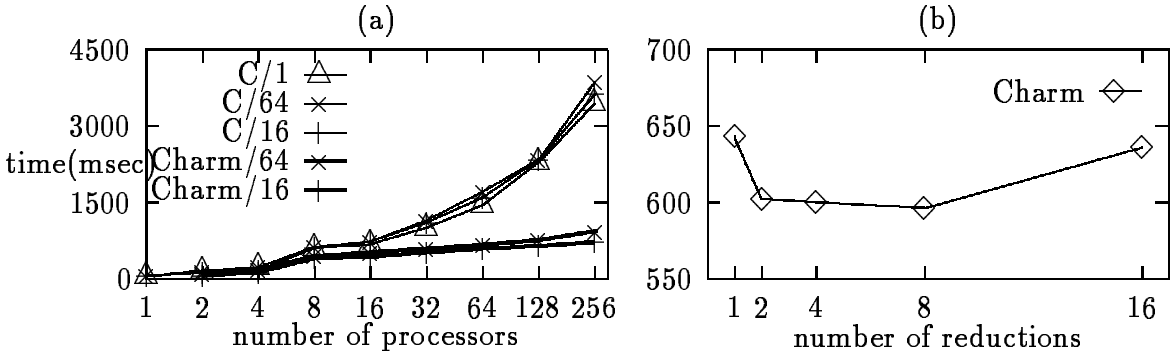

Figure 3: Concurrent Reductions, k=16,64, n=16384

in future releases. However, the fundemental advantage of message driven execution will still be valid as the time for the blocking version must rise proportional to at least $\log p$.

To study the effect of the degree of pipelining, we varied k for a fixed number of processor in the Charm version. In Figure 3-(b) we plot the elapsed time as a function of the number of reductions (k) on 128 processors. The time decreases up to 8 reductions, and then it starts to increase. Here, at $k = 8$ the overhead starts to become dominant, and the execution time increases. The crossover point will shift to the right for larger values of n, for more processors, for larger communication latency per byte, or for smaller Charm scheduling overhead.

4 DAGGER

Although message driven execution improves performance and modularity, its expression in a program may sometimes get cumbersome. Such programming has some similarities to event based programming used for X-windows or Microsoft windows programs, but is more complex due to dependencies among messages and the nature of this asynchrony. This sometimes leads to bugs due to a message processing sequence that is different than those anticipated by the user program. We have developed a notation called *Dagger* [5], and an associated graphical program editor that helps programmers deal with this complexity.

The Dagger language augments the Charm language with a special form of chare called a *dag chare*. A dag chare (dag: directed acyclic graph) specifies pieces of computations (when-blocks) and dependences among computations and messages. A when-block is guarded by some dependences that must be satisfied to schedule the when-block for execution. These dependences include arrival of messages or completion of other when-blocks.

In the Figure 4, a template for a dag chare is shown. In addition to entries, a dag chare may declare some other data local to that dag in the local variable declaration section. The local variables are shared among when-blocks and private functions of the dag chare. Private functions are regular C functions which may contain Charm or Dagger statements/calls, and they can be called only within the static scope of the dag chare.

dag chare example {
local variable declarations
condition variable declarations
entry declarations
when depn_list_1 : {when_body_1} ...
when depn_list_n : {when_body_n}
private functions }

Figure 4: Dagger Chare Template

As in a chare definition, there are no explicit receive calls in a dag chare. The dag chare declares entry points, and messages are received at these entry points. The entry point declaration, which is in the form:

```
entry entry_name : (message msg_type *msg)
```

defines an entry with the name `entry_name`, and associates a variable with a specified message type with that entry. Messages can be sent to entry points by supplying the `entry_name` in the Charm system calls such as `SendMsg`. The variable `msg` is a pointer to the message received by the entry.

Receiving a message at an entry point is not sufficient to trigger a computation. (In contrast, in Charm, arrival of a message triggers a computation which is associated with that entry point.) The computation must be in a state where it is ready to process the message. A Dagger program tells the Dagger runtime system when it is ready to process a message by using the `expect` statement:

```
expect(entry_name)
```

If a message arrives before an `expect` statement has been issued for it, Dagger will buffer the message. The message becomes available only after the `expect` statement is executed. A dag chare may have a special type of variable, condition variable. A condition variable is declared as follows:

```
CONDVAR cond_var_name
```

The condition variable is used to signal completion of a when-block. In other words, it is used to express the dependences among when-blocks which belong to the same dag chare. A when-block can send a message to an entry which is defined in the same dag chare, however to utilize a shared variable (condition variable) is more efficient. A condition variable is initialized to the *not-ready* state when it is declared. It is set to the *ready* state by the `ready` statement:

```
ready(cond_var_name)
```

Once a condition variable is set, the Dagger may schedule the when-blocks which are waiting for that condition variable to be set. A when block is a computation which is guarded by a list of entry names and condition names:

`when` $e_1, \ldots, e_n, c_1, \ldots, c_m$ `: {when-body}`

where e_i is an entry name, and c_i is a condition variable. In order to initiate the execution of the when-block, the dependence list of the when-block must be satisfied. The dependence

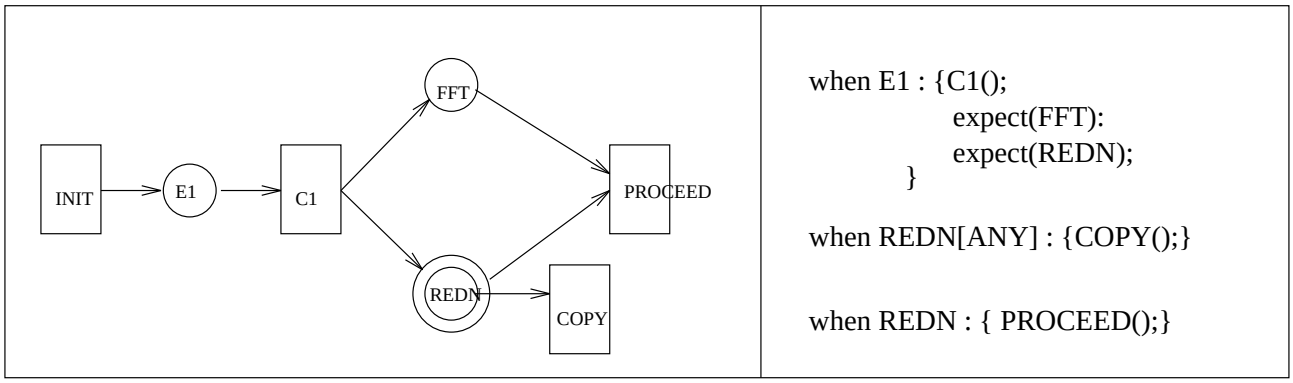

Figure 5: A Dagger Example

list is satisfied if :

- a message has been received and `expect` statement has been executed for each entry e_i in the dependence list,
- for each condition variable c_i in the list, a `ready` statement has been executed.

The `when-body` is a block of C code possibly including Charm system calls, and `expect` and `ready` Dagger statements. The messages received by the entries e_i's are accessed inside the `when-body` through the message pointers defined in the entry declarations.

As an example, consider the graph shown in Figure 5. The rectangles in this graph represent subcomputations and circles represent messages. Note that all the subcomputations are part of a single process executing on a single processor. Arrows from messages to subcomputations represent dependencies while arrows to messages represent readiness to receive messages. Double circles represent entry points which may receive multiple messages. The arrow from inner circle point to computations that are performed when any one message to that entry point arrives. Arrows from the outer circle point to computations that are performed when all expected messages for this entry point have been received. Thus the process specified by this graph carries out the initial subcomputation then expects the `E1` message. When this message is received it carries out the `C1` subcomputation, and is ready to receive messages to either the `REDN` or `FFT` entry point. When all the messages to both these entry points have been received it initiates the `proceed` computation. The dagger code fragment corresponding to this graph is shown to right in Figure 5. Notice an extension to the Dagger as defined above for the `REDN` entry point: `ANY` is a keyword that signifies that the when-block should be triggered every time a message for `REDN` arrives, whereas the "when `REDN` " block is triggered only after all the expected messages for `REDN` have arrived. By programming in this notation the need to specify counters, flags, and buffers to store messages, which would otherwise be necessary to implement the same algorithm, is eliminated.

The visual notation shown in Figure 5 is very intuitive, and shows the flow of control within a process (a chare) clearly. We have developed a visual program editor based on this notation. One can create such graphs, provide necessary definitions and labels, and add

program text to the boxes using this editor. The visual program is automatically translated to Dagger/Charm notation.

There are computations where the concurrent phases of a dag exist (in time). An example of this is a dag augmented with a loop where different iterations of the loop may be executed concurrently. Another example is a client-server type of computation. Client processes may send multiple requests concurrently to a server dag. The server dag performs the same computation for different requests concurrently. This type of computations are supported by a reference number mechanism. Further details about this feature and Dagger can be found in [5].

5 CONCLUSION

The advantages of message driven execution for multiple independent global operations can be seen to arise from two separate factors.

1. the pipelining effect, which eliminates the critical path proportional to at least $\log p$ during which processors would have remained idle in a blocking SPMD style program.
2. the relaxation of the need for every processor to wait at the barrier formed by the reduction operation.

Of these, the former can be reduced or eliminated with special purpose hardware (for example, a shared memory machines or a machine with a reduction network), with its associated cost. However, the latter is a fundamental advantage which becomes significant particularly when there is variations and irregularities in the application programs. As more "real" applications are explored, we believe such irregularities will occur more commonly and thus establish the importance and necessity of message driven execution. A system such as Charm can play an important part in this transition to message driven execution.

Message driven execution was proposed and studied in the Actor model by Hewitt[6], and later by Agha[7]. Charm is one of the first systems to embody message driven execution in a portable parallel programming system running on stock multicomputers (in 1986-87 [8]), along with the reactive kernel of Seitz et. al. [9]. The recent work on Active Messages [10] provides an efficient substrate for message driven execution, while the Split-C language [11] developed at Berkeley is a recent message driven language that uses Active Messages. The data flow approaches clearly embody message driven execution although they were based on specialized hardware. Message driven execution is similar in spirit to macro data flow approaches designed for general purpose parallel purpose machines.

Thread packages used on individual processors can also provide part of the benefit of message driven execution. While a thread is blocked for a message, another thread may continue. However, a thread is usually blocked waiting for only one message, unlike chares which can wait for multiple messages concurrently. Thread scheduling is somewhat arbitrary,

and outside the control of the programmer, whereas Charm can schedule messages based on priorities or user selectable control strategies. Stack management and efficient implementation are some of the additional problems for threads. More transiently, Charm is available on many parallel machines whereas threads are not.

The original problem in the computation in CHARMM, used a basis for the performance experiments in Section 3, was solved efficiently by Brooks. This solution is based on the fact that all the reduction results are not needed on all the processors in the application. For every partition, only one distinct processor needs to know the result of its reduction. The dimensional exchange algorithm that reduces by half the data to be exchanged in every phase was used based on this observation.

We plan to augment and optimize the reduction library in Charm in many ways. In particular, we plan to support the reductions whose results are needed only on one (or a few) processor. The algorithm for this purpose is based on multiple distinct spanning trees rather than on dimensional exchange.

The Charm parallel programming system which embodies message driven execution is a well-developed and stable system with a rich collection of primitives. Branched (replicated) chares and specifically shared objects provide a highly expressive method for specifying parallel programs. As a result, the Charm runtime has much detailed and specific information about the state of the computation. Intelligent performance feedback tools and debugging tools can be built to exploit this knowledge. We have started in this direction with a preliminary performance feedback tool called Projections, and are building the next generation of the tools.

Acknowledgement We are grateful to Sandia National Lab for the use of their Intel/Paragon computer for conducting the performance experiments described in this paper.

References

[1] L. V. Kale. The Chare kernel parallel programming language and system. volume II, pages 17–25.

[2] W. W. Shu and L. V. Kale. Chare Kernel - a runtime support system for parallel computations. *Journal of Parallel and Distributed Computing*, 11:198–211, 1990.

[3] L. V. Kale and W. Shu. The Chare Kernel base language: Preliminary performance resul ts. pages 118–121.

[4] W. Fenton, B. Ramkumar, V. Saletore, A. Sinha, and L. V. Kale. *The Chare Kernel Programming Language Manual.*

[5] A. Gursoy and L. V. Kale. Dagger: Combining the benefits of synchronous and asynchronous communication styles. Technical Report 93-3, Parallel Programming Laboratory, Department of Computer Science, University of Illinois, March 1993.

[6] G. Agha and C. Hewitt. *Concurrent Programming Using Actors: Exploiting Large-Scale Parallelism*, volume Lecture Notes in Computer Science, 206, pages 19–40. Springer-Verlag (Berlin-Heidelberg-New York), October 1985. Editors: S. N. Maheshwari.

[7] G. A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT press, 1986.

[8] L. V. Kale. The Design Philosophy of the Chare Kernel Parallel Programming System. Technical Report UIUCDCS-R-89-1555, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1989.

[9] W. C. Athas and C. L. Seitz. Multicomputers: Message-passing concurrent computers. In *Computer*, volume Volume 21, No. 8, August 1988.

[10] T. vonEicken, D. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. *ACM*, pages 256–266, 1992.

[11] D. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. vonEicken, and K. Yelick. Parallel programming in split-c. Technical report, University of California, Berkeley.