Tolerating Latency with Dagger

Attila Gürsoy and **L.V.Kale**
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana IL 61801, USA
{gursoy,kale}@cs.uiuc.edu

Abstract

The communication latency is a major issue that must be dealt with in parallel computing. The parallel computation model therefore must provide the ability to tolerate such latencies. Communication using blocking receives is the commonly used mechanism in parallel programming today. Message driven execution is an alternate mechanism which does not use receive style statements at all. The message driven execution style promotes the overlap of computation and communication: Programs written in this style exhibit increased latency tolerance. However, they are often difficult to develop and debug. We present a coordination language called Dagger to alleviate this problem. The language has a mechanism which is called - expect, that replaces the receive statement. It has been implemented in the Charm parallel programming system, and runs programs portably on a variety of parallel machines.

1. INTRODUCTION

Communication latency, and the idea that remote data will take longer to get hold of than local data, is a fact that must be dealt with in parallel programming on most scalable parallel machines. Dealing with this latency is therefore a major objective in parallel processing. On the hardware side, this is being addressed by designing architectures that reduce the latency to the minimum. The ALLCACHE architecture of the KSR-1 machine, and the message-processor architecture of J-Machine [2] are examples of these attempts - as well as the continuous evolution of communication hardware in the traditional architectures of Intel and NCUBE machines. However, physical reality dictates that remote access will always be significantly slower than local access. Software techniques for *tolerating* latency are therefore essential.

Communication can be blocking or nonblocking. Communication using blocking receives is the commonly used mechanism in parallel programming today. By postponing receives beyond some useful computations, one can attempt to overlap communication with computation. However, such programs can not adopt the runtime variations of communication delays and can not exploit the opportunities presented by nondeterministic message arrival. On the other hand, blocking style is easy to use because they impose strong synchronization among concurrent activities which may be relaxed.

Message driven execution is a promising technique in this regard. In message driven execution style (which is distinct from mere message-passing), user programs don't block on a *receive-message* call. Instead, the system activates a process when there is a message for it. Therefore, it gives the ability to overlap computation and communication. This helps latency tolerance in two ways: First, when one process is waiting for data from a remote process,

another ready process may be scheduled for execution. Secondly, even a single process may wait for multiple data items simultaneously, and continue execution whenever any of the expected items arrive.

Message driven execution has its own problems. Messages may arrive in an unexpected order. In addition to that, *split-phase* style of programming that it requires complicates the flow of control. We propose a coordination language called Dagger which retains the benefits of the message-driven execution, while reducing the complexity of the resultant programs. Dagger programs run on top of Charm [8] which is a message-driven system. Dagger allows specification of processes in terms of dependences between messages and pieces of computations. These dependences form a partial order [1] which clarifies the flow of control. The Dagger runtime system buffers messages until they can be processed, and automatically maintains all the flags and counters needed to ensure that the partial order is adhered to.

The next Section defines the latency. In Section 3, Charm system is described. The programming difficulties in message driven systems are discussed in Section 4. Dagger language, its simulation model and the preliminary performance results are presented in Sections 5 and 6. Related work is reviewed in Section 7. Conclusions are drawn in Section 8.

2. LATENCY TOLERANCE

The important issue in message passing systems is the latency incurred in communication and cooperation. We divide latency into two categories: communication latency, and the unpredictable delay in the remote response.

The communication delays are due to software overheads, link propagation time and the network bandwidth. The software overhead usually attributed to the CPU, however the CPU is free to do other work during the network latency time, i.e., the network latency can be tolerated if the CPU performs some useful work during that time. The network latency usually is modelled approximately by $t = \alpha + \frac{n}{\beta}$ where α is the link propagation time, β is the bandwidth, and n is the size of the message.

The communication latency is measured after a message has been sent from a remote site. The creation of this message at remote site itself may be delayed due to numerous runtime conditions. This delay is often unpredictable and application dependent. The remote response latency can be handled along with the network latency if the CPU continues to do some useful work adaptively until remote responses arrive. Many parallel applications contain sufficient parallelism to exploit this fact. Therefore, the ability of tolerating these latencies is an important issue in a parallel programming language.

3. CHARM - A MESSAGE DRIVEN SYSTEM

Charm [8] 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 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,

[1] The dependence graphs are allowed to have cycles in them, and so strictly speaking, are not partial orders (i.e DAGs). However, the back-edges correspond to iteration, and can be considered as an abbreviation mechanism for denoting an unfolded (and unbounded) partial order.

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 [11]. 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.

---

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

4. PROGRAMMING DIFFICULTIES IN MESSAGE DRIVEN STYLE

Although the message-driven execution allows the overlap of communication and computation, it extracts a price in the form of apparent program complexity. The *split-phase* or *continuation-passing* style of programming that it requires is sometimes non-intuitive, and obfuscates the flow of control. As the system may execute messages in the order it receives them (as opposed to a deterministic order imposed by sequential receive statements), the programmer must deal with all possible orderings of messages, often with flags and counters. These are accompanied by reasoning which can sometimes get complex, about which message-orderings will not arise, which are harmless, and which must be dealt with by buffering, counters, and flags.

We will explain this by a simple example. Assume that a particular processor is waiting for three messages: *m1, m2*, and *m3* where they trigger the computations *C1, C2*, and *C3* respectively. In addition, *C1* and *C2* can be executed in any order, but *C3* must be executed after both *C1* and *C2*. Messages may arrive in any order. If the message *m3* arrives before *m1* and *m2*, then it triggers *C3*. However, the computation C3 can not be executed yet. Therefore, *C3* must check this situation, buffer the message *m3* if necessary, and wait for the completion of other computations. When the messages *m1* and *m2* arrive, *C1* and *C2* are executed (in any order). *C1* and *C2* must contain the code that checks if the message *m3* has already arrived, if so, one of them must trigger the computation *C3*. We developed a coordination language,

Dagger, on top of Charm to hide these details from the user, and still to allow message driven execution.

5. DAGGER

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.

---

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 2: Dagger Chare Template

In the Figure 2, 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.

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 variables, condition variables. 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

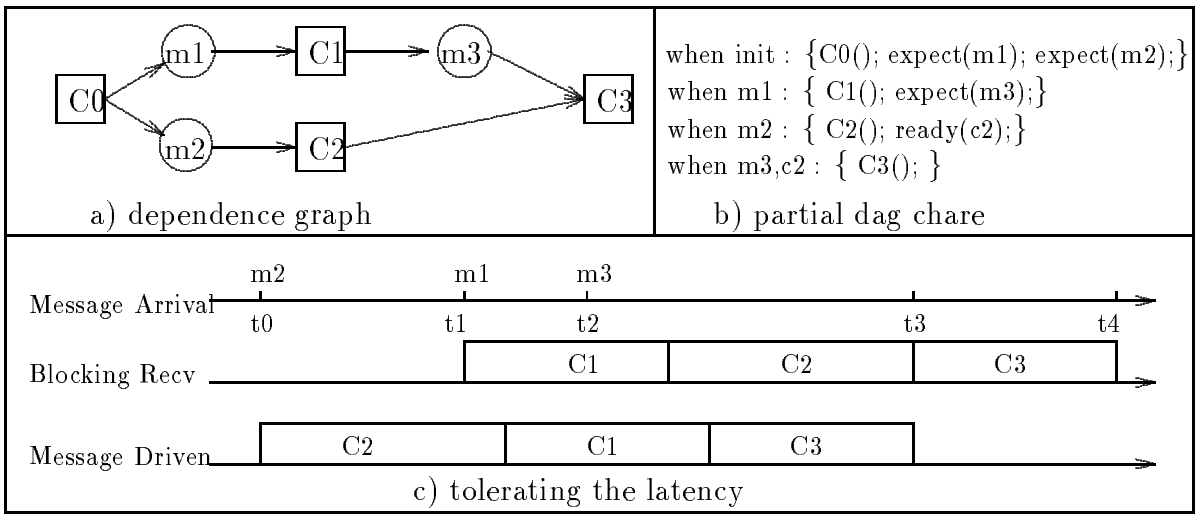

Figure 3: A Dag Chare Example

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 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, we will consider the computation which is discussed in Section 4. The dependences among the messages and computations for this example are shown in Figure 3-(a), and in part (b), the corresponding part of the dag chare is listed. Note that, this computation executes on one processor, the whole parallel algorithm is a collection of dag computations each running on a different processor. In part (c), the comparison of a particular execution instance of two versions of this computation is depicted: one implemented with blocking receives, and the other one implemented in Dagger. Assume that the blocking version waits for messages in the order (*m1, m2, m3*), and the actual message arrival order is (*m2, m1, m3*). The Dagger version completes earlier than the blocking version since it can execute *C1* and *C2* in any order. If there is opportunity, the programs written in Dagger will tolerate communication latencies and unpredictable delays occurring in other processors by overlapping the computation and communication.

So far, we explained the basic Dagger coordination language for ease of exposition. Supporting full generality of parallel programming requires two extensions embodied in Dagger.

These are: reference numbers, and entry points with multiple messages. There are computations where the concurrent phases of a dag exist (in time). An example of that 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 the reference number mechanism. Details about these features can be found in [6].

5.1. Preliminary Performance Results

In this section we present some preliminary performance results obtained by using programs written in Dagger language. Global reduction operations (such as finding maximum) are very common in many scientific applications. We present results of multiple reduction operations which is part of a fluid flow computation on NCUBE/2 parallel machine. Each processor has an array which is partitioned into blocks of size 512 words. A reduction (maximum) operation is performed for each partition and the reductions are independent of each other.

Two versions of this computation was implemented: one in C with library function `nrmaxn` provided by the system to perform the reductions, the second one in Dagger language. The system call `nrmaxn` is a blocking call. Every processor calls this function and is blocked until the reduction is completed. In the Dagger version, the program initiates a reduction operation, then it continues with the next available piece of computation. The reductions are carried out concurrently through a spanning tree. Figure 4-(a) shows the elapsed time of these two programs. As the number of processors increases, the blocking-receive version takes more time, because the cost of reduction operation is in the order of $\log p$ where p is the number of processors. The Dagger version tolerates this by overlapping the computation and communication and yields scalable performance.

6. SIMULATOR

A trace driven simulation model has been developed to study the impact of communication latency on the performance of message driven algorithms.

Execution traces from a particular run of a parallel algorithm is not sufficient to simulate behavior of message driven algorithms. The difficulty is that if messages arrive in a different order during the simulation than the one in the real execution, then, the traces are no longer valid. However, we want to simulate the behavior of the parallel algorithm under changing communication latencies. At this point, Dagger helps. A program expressed in Dagger can be simulated correctly irrespective of runtime message ordering as long as a simple requirement is met [2]. The Dagger provides the dependence information among messages and computations. The real execution trace contains information about the execution of when-blocks, message-send, expect, and ready statements.

The communication latency is modelled by the following parameters: software send overhead (attributed to cpu - no overlapping), software receive overhead (attributed to cpu - no overlapping), and network latency which is the time between the header of message injected into network and the tail exits the network. In addition to communication latency, dagger message handling overhead (matching and queueing messages, scheduling when-blocks) is included in simulation time. The details of the simulation and the abstract parallel machine model can be found in [7].

[2] [For correct simulation] a variable defined in one when-block must not be used or defined in an incomparable block.

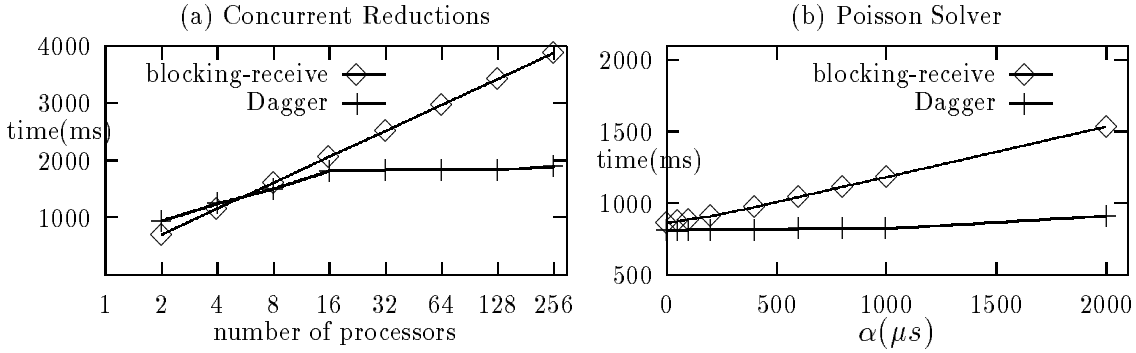

Figure 4: Performance of some Dagger Programs

The simulator will be used to study the impact of latency on the performance of parallel algorithms, and to develop latency tolerant algorithms. Some preliminary results of the simulator are shown in Figure 4-(b). The figure depicts the simulated performance of the inner part of a 3D fluid flow problem as a function of the network latency (α which is defined in Section 2). The program solves multiple 2D Poisson equations with iterative techniques which are independent of each other. The blocking-receive version follows a fixed sequence to solve these equations, i.e., first equation, then the second and so on. The order in the Dagger version, however, depends on the message arrival order. As shown in the figure, the performance of blocking-receive version increases linearly as the network latency increases. The Dagger version tolerates the increase in the latency as well as any computational delay occurring in other processors (A processor might be waiting for a reply from another one which might be busy to reply immediately).

7. RELATED WORK

The original Actor model as described in [1] is purely message driven. The issue of synchronization within an actor was addressed in [10] which proposed the *enable set* construct. The enable construct is analogous to our expect statement. However, there is no analogue of a when-block viz. a computation block that can be executed only when a specific group of messages have arrived. A more recent paper [3] supports much more complex model which subsumes synchronization of multiple actors depending on message sets. It should be noted that Dagger/Charm provides a programming model that differs from Actors in many ways. The discussion above only focuses on how they deal with message driven execution. Recent work on Active messages [4] also deals with message driven execution and split phase transactions. The split-C language based on this employs polling for arrival of messages. However the TAM compiler built on Active messages has some similarities to Dagger. As messages always enable the corresponding threads of an activation frame, there appears to be no way of buffering unexpected messages. Counters and flags for synchronizing on arrival of multiple messages are explicitly maintained. However, TAM is meant as the back end for a data flow compiler as opposed to a language meant application programmer. So these inconveniences may not be of much consequences. Macro data flow [5] approaches share with us the objective of message driven execution and local synchronization. However, much of the past work in this area has aimed at special purpose hardware. Our experience with using Dagger as back-end for a compiler for a data parallel language [9] indicates that the dagger might provide more convenient intermediate language than macro data flow.

8. CONCLUSION

The communication latency and other delays in remote responses are a major source of inefficiency of parallel computations. Therefore dealing this latency is essential in parallel computing. We presented a coordination language called Dagger which allows to tolerate the latencies incurred in parallel computations. Dagger combines the efficiency of message driven execution with the conceptual simplicity of blocking-receives. Programming in blocking-receive paradigm is easier since it imposes a strict synchronization. However, it allows communication latency to impact performance significantly. Dagger allows users to express dependencies among messages and computations. The send-expect mechanism together with when-blocks which are guarded by the availability of messages allow expression of parallelism with ease at the same time retaining message driven execution. The Dagger has been implemented on top of Charm which supports message driven portable parallel programming on MIMD machines.

9. REFERENCES

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

[2] W.Dally, and et al. "The J-Machine: A Fine-Grain Concurrent Computer", In *IFIP Congress*, 1989.

[3] S.Frolund, G.Agha, "Activation of Concurrent Objects by Message Sets", Internal Report, University of Illinois at Urbana-Champaign.

[4] T.von Eicken, D.E.Culler, S.C.Goldstein, K.E. Schauser, "Active Messages: a Mechanism for Integrated Communication and Computation", *Proceedings of the 19th Int'l Symposium on Computer Architecture*, Australia, May 1992, pp256-266.

[5] A.S.Grimshaw, *Mentat : An Object Oriented Macro Data Flow System*, UIUCDCS-R-88-1440, Ph.D Thesis, University of Illinois at Urbana-Champaign, June 1988.

[6] A.Gursoy, L.V.Kale, "Dagger: combining the benefits of synchronous and asynchronous communication styles", Report 93-3, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, March 1993.

[7] A.Gursoy, L.V.Kale, "Simulating message driven programs",Report 93-9, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, July 1993.

[8] L.V.Kale, "The Chare Kernel parallel programming language and system", *Proceedings of the International Conference on Parallel Processing*, Vol II, Aug 1990, pp17-25.

[9] E.Kornkven, "Overlapping Computation and Communication in an Implementation of A Data Parallel Language", Report 92-4, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, Oct 1992.

[10] C.Tomlinson, V.Singh, "Inheritance and Synchronization with Enabled-Sets", ACM OOPSLA 1989 , pp103-112.

[11] The CHARM(3.0) programming language manual, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, 1992.