Dagger: Combining the benefits of synchronous and asynchronous communication styles

Attila Gursoy
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana IL 61801
phone : (217) 333-5827
email : gursoy@cs.uiuc.edu

Laximikant V. Kalé
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana IL 61801
phone : (217) 244-0094
email : kale@cs.uiuc.edu

Abstract

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 - their performance does not degrade significantly with latency. It also induces compositionality - multiple independently developed modules can be combined correctly without loss of efficiency. However, as the flow of control is not explicit in such programs, they are often difficult to develop and debug. We present a coordination language called Dagger to alleviate this problem. The language 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.

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 also promotes modularity. Consider a situation when two processes belonging two different modules are waiting for messages directed to them. When a message arrives, the runtime system will activate the appropriate process; neither module needs to know about the other module. In contrast, in the blocking-receive paradigm that is widely used currently, this can be accomplished, if at all, at the cost of modularity: The code in a module may issue a *wild-card* receive, but if it receives a message meant only for another module, it must somehow resume the other module, and hand over the message to it. This means that writer of a module issuing a blocking receive must know about all other modules that may be waiting for a message at that moment, and must design mechanisms to recognize and hand over messages belonging to other modules.

Although it imparts these benefits, message-driven execution often 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 complex reasoning about which message-orderings will not arise, which are harmless, and which must be dealt with by buffering, counters, and flags.

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 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 describes Charm which supports a message driven execution model. The Dagger language with examples is discussed in section 3 and 4. The techniques used in implementing Dagger are discussed in section 5, followed by the preliminary performance results in section 6.

2 The Charm Language

Charm 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, Encore Multimax, Sequent Symmetry, ALLIANT FX/8, single-processor UNIX machines, and networks of workstations. It is being ported to a CM-5, Parsytec GC-el, and Alliant FX/2800.

[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.

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 described below.

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.

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. Some of the important Charm system calls are:

`CreateChare(chareName,entryPoint,msg)`
: This call is used to create an instance of a chare named as `chareName`. As all other Charm system calls, `CreateChare` is a non-blocking call, that is, it immediately returns. Eventually as the system creates an instance of chare `chareName`, it starts to execute the `entryPoint` with the message `msg`.

`SendMsg(chareID,entryPoint,msg)`
: This call deposits the message `msg` to be sent to the `entryPoint` of chare instance `chareID`. `chareID` represents an instance of a chare. It is obtained by a system call `MyChareID()`, and it may be passed to other chares in messages.

---

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

A branch office chare (BOC) is a form of chare that is replicated on all processors. An instance of a BOC has a branch on every processor. A BOC definition is similar to a chare definition except

it contains public functions which can be called by other chares on the same processor. BOC's are useful for some computations such as reduction operations (i.e., collecting some information locally on each processor, and then combining it across processors), as well as for expressing static load balancing, and SPMD style programs.

In addition to messages and BOC's, Charm provides other ways in which processes share information. Some of these abstractions that are relevant for this paper are described below:

readonly A readonly variable is initialized at the beginning of a Charm program, and its value can be accessed by `ReadValue` call from any part of the program.

distributed table A distributed table is a set of entries with key and data fields. A number of asynchronous access and update calls are allowed on table entries.

Additional information sharing abstractions supported include monotonic variables, writeonce variables and accumulators. Charm also provides a sophisticated module system that facilitates reuse, and large-scale programming for parallel software. Details about these features can be found in [9].

```
chare mult_chare {
   int         count;
   ChareIDType chareid;
   int         *row,*column;
   entry init        : (message MSG *msg) {
       count = 2;
       MyChareID(&chareid);
       Find(Atable, msg->row_index, recv_row, &chareid,TBL_NEVER_WAIT);
       Find(Btable, msg->column_index,recv_column,&chareid,TBL_NEVER_WAIT);
   }
   entry recv_row    : (message TBL_MSG *msg) {
      row = msg->data;
      if (--count == 0 ) multiply(row,column);
   }
   entry recv_column : (message TBL_MSG *msg) {
      column  = msg->data;
      if (--count == 0) multiply(row,column);
   }
}
```

Figure 2: Matrix multiplication chare

2.1 An Example in Charm

Consider an algorithm for matrix multiplication that is dynamically load balanced. Such a formulation may be useful on a machine where different processors operate at different speeds, for example. We assume that the two matrices to be multiplied have been stored in distributed tables. Matrix A is stored as a collection of entries such that each entry is a block of contiguous rows. Similarly, the matrix B is stored as a collection of columns. One of the chares (`mult_chare`) used in implementing such an algorithm is shown in Figure 2. This chare is responsible for multiplying

a block of rows of A, and a block of columns of B. The entry `init` is executed when an instance of the chare is created. The message `msg` contains indices of the row and column blocks that are to be multiplied. First, the chare requests the row and columns from the tables `Atable` and `Btable` (these tables store the matrices A and B) by calling `Find` which is a system call in Charm . In the `Find` call, the row (or column) index, return entry point and the chare instance identifier are supplied. Note that the `Find` call is non-blocking, and it immediately returns. Eventually, the row (and column) data will be sent in a message (of type `TBL_MSG` which is defined in the Charm language) to the entry point `recv_row` (`recv_column`), and these messages may arrive in any order.

The multiplication depends on availability of both rows and columns. Therefore a shared variable, `count`, is used to detect that both messages are available. Initially the `count` is set to 2 (since only two messages are expected). Whenever a message is received, the message is saved, the `count` is decremented by one. If the value of `count` becomes zero, then the multiply function is called. This example has been chosen to be a simple one in order to demonstrate the necessity of counters and buffers. In general, a parallel algorithm may have more interactions leading to the use of many counters, flags, and message buffers, which complicates the program development significantly.

3 Basic Language

In order to reduce the complexity of program development, a coordination language called Dagger has been developed on top of Charm system. In Charm, an entry point is executed when there is a message directed to it. If the computation in that entry point is dependent on the computation in another entry point within the same chare, then the programmer must handle this unexpected message by buffering it, and fetching it whenever the entry point becomes eligible for execution. Dagger hides these details from the programmer by providing ready/expect and when-block constructs which will be discussed in the following section.

3.1 Dag Chare

The Dagger language is defined by augmenting Charm with a special form of chare called a *dag chare*. A dag chare 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. Before describing the Dagger language in detail, let us consider the matrix multiplication example, and show how it looks like in Dagger. Figure 3 shows the matrix multiplication written as a dag chare. Whenever the entries `recv_row` and `recv_column` receive the messages, the multiply function is called with the rows and columns that have been received. The Dagger takes care of the bookkeeping functions such as counters, flags and buffering the messages. Therefore, the resulting code is more readable (and easy to program), while still retaining the benefits of a message driven model.

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

```
dag chare mult_chare{
   entry init        : (message MSG *msg);
   entry recv_row    : (message TBL_MSG *row);
   entry recv_column : (message TBL_MSG *column);
   when init : {
       MyChareID(&chareid);
       Find(Atable, msg->row_index, recv_row, &chareid,TBL_NEVER_WAIT);
       Find(Btable, msg->column_index,recv_column,&chareid,TBL_NEVER_WAIT);
       expect(recv_row);
       expect(recv_column);
   }
   when recv_row, recv_column : { multiply(row->data,column->data) }
}
```

Figure 3: Matrix multiplication 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 function f1() {C_code_1}
    ....
    private function fm() {C_code_m}
}
```

Figure 4: The Dagger chare template

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 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.

3.2 Dag Chare Example

As another example of a Dagger program, we will consider a simple version of a numerical problem - Jacobi relaxation to solve a penta-diagonal linear system (which arises in the solution of partial differential equations). We will present the problem without getting into the details of the

application, and present the dag code for it. This problem involves a 2-dimensional grid of points. The grid is partitioned into rectangular blocks, and each processor is assigned to one block. The basic computation in a processor is to perform some local computation on its own block, exchange some information (boundary values) with four neighbour processors (east,west,north,south), and to carry out a reduction operation across all processors (maximum operation). The computation continues in this manner until the solution is reached. Figure 5 shows a dag BOC to carry out this

```
dag branchoffice jacobi {
   ChareNumType mycid;
   PeNumType    neighbour[4];
   CONDVAR      SEND;
   entry init : (message MSGINIT *msg);
   entry NORTH: (message BOUNDARY *north);
   entry SOUTH: (message BOUNDARY *south);
   entry WEST : (message BOUNDARY *west);
   entry EAST : (message BOUNDARY *east);
   entry CONVERGENCE : (message CONV_MSG *convergence);
   when init : { MyChareID(&mycid);initialize(); ready(SEND);}
   when SEND : {  BOUNDARY *m;
                  for each direction in (NORTH,SOUTH,WEST,EAST)
                    m = copy_boundary(direction);
                    SendMsgBranch(entry_no[direction],m,neighbour[direction]);
                    expect(entry_no[direction]);
              }
   when NORTH,SOUTH,WEST,EAST : {
       update(north,south,west,east);
       reduction(my_convergence(),CONVERGENCE,&mycid);
       expect(CONVERGENCE);
   }
   when CONVERGENCE : {if (convergence->done)
                           print_result()
                       else
                           ready(SEND); }
}
```

Figure 5: Jacobi relaxation dag BOC

computation. The four entry points (`NORTH,SOUTH,EAST,WEST`) receive the boundary values from neighbouring processors. The entry point `CONVERGENCE` receives the result of the global reduction operation. When an instance of the Jacobi BOC is created, the init entry point is executed first. The code at this entry point carries out some initializations such as the determination of the identity of the neighbour processors. After initialization, the condition variable `SEND` is set by the `ready` statement. This enables the when-block which depends on the variable `SEND`. In this when-block, boundary values are sent to neighbour processors (`copy_boundary` creates a message, and boundary values are copied into messages, and `SendMsg` Charm call initiates the transfer). In addition to this, an `expect` statement is executed for each direction because the processor is ready to process boundary values from neighbours. After receiving all the boundary values, the local data is updated and the reduction operation is initiated. The function reduction is a call to another dag chare which is not explained here. It suffices to note that a local convergence data is collected by `my_convergence()`, and sent to the reduction chare, which is responsible for determining the global convergence. The reduction call is non-blocking, and eventually the reduction dag sends the result to the entry point `CONVERGENCE`. After initiating the reduction,

an `expect` statement is issued to state that the computation is ready to process the reduction result. When the reduction result arrives, the code checks if global convergence has been reached. If not, it reactivates the when-block which sends the new boundary values, and the computation continues. The `expect` and `ready` statements serve an important purpose - imposing an order on the processing of messages. Without `expect/ready` construct (i.e., implicitly every message is expected), one of the processors might receive the boundary messages from its neighbours before the reduction message, and it may start next iteration. This violates the dependences in the algorithm. The Dagger performs the necessary synchronization by keeping counters and flags, and buffering unexpected messages to guarantee the correct order of execution.

4 Extended Language

The Dagger Language as defined in section 3 was kept simple for ease of exposition. Supporting full generality of parallel programming acquires two extensions embodied in Dagger, which are motivated and described below.

4.1 Reference Numbers

The `expect` statement imposes an order on the execution of messages. This is sufficient for a simple dag computation. However, 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.

First, we will illustrate the problem with a simple example, and then describe the solution provided by Dagger. Consider the Jacobi relaxation program of Figure 5 without any global convergence tests. Each processor receives boundaries from its neighbours, computes new values

```
when NORTH,SOUTH,EAST,WEST:{
    update();
    if (iteration_count++ < limit)
       for each direction {
         m = copy_boundary(direction);
         SendMsgBranch(entry_no[direction],m,neighbour[direction]);
         expect(entry_no[direction]);
       }
}
```

Figure 6: Jacobi relaxation without reference numbers

and sends the new boundaries to the neighbours. This computation is repeated a fixed number of times. In Figure 6, the Dagger code is listed for that computation. Each processor is executing the same dag code. Depending on the machines and its operating system, it is possible that

messages belonging to different iterations may arrive out of order, or may be delayed for some reason. A scenario where the computation goes wrong is illustrated in Figure 7. The processors i and j are exchanging messages and doing some local computations. The message sent by the processor i in the second iteration is delayed. When the processor i receives a message from j in the third iteration, it performs the local computation and sends the message belonging to the fourth iteration. The processor j receives the message which belongs to the fourth iteration before the one which belongs to the third iteration.

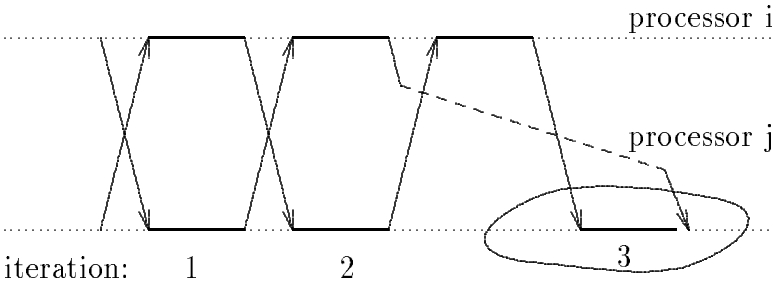

Figure 7: Out of order messages

In order to handle this problem, the messages belonging to different phases of the computation must be distinguished. To accomplish this, we modify the language to include *reference numbers*. Each message may have a reference number. The messages which belong to the same phase of the computation are given the same reference number explicitly by the user. Then, the Dagger matches the messages with the same reference number to activate a when-block (condition variables may have reference numbers too). In other words, an instance of a when-block is scheduled for execution only if the dependence list is satisfied with the availability of messages and condition variables with matching reference numbers. The `expect` and `ready` statements are modified to support reference numbers as follows:

```
expect(entry_name,reference_number)
ready(cond_var_name,reference_number)
```

The reference number of messages are accessed and set by the function calls provided by the system: `GetRefNumber(msg)`, and `SetRefNumber(msg)`. In Figure 8, a fragment of the version of the Jacobi relaxation which uses reference numbers to solve the problem discussed above is listed.

4.2 Entry Points with multiple messages

Another extension to the Dagger deals with entry points which receive multiple messages. This situation arises when the number of messages that a when-block depends on is known only at run time, or differs from processor to processor. As an example, we will program a concurrent reduction operation. In a reduction operation, some data values from every processor are collected, and the result of the reduction operation over the collected data is distributed to all the processors

```
when NORTH,SOUTH,EAST,WEST : {
   update();
   if (++loop_count < limit)
      for each direction {
           m = copy_boundary(direction);
           SetRefNum(m,loop_count);
           SendMsgBranch(entry_no[direction],m,neighbour[direction]);
           expect(entry_no[direction],loop_count);
      }
}
```

Figure 8: Jacobi relaxation with reference numbers

(for example, a global sum operation). An efficient way of implementing the reduction is to utilize a spanning tree of processors. Each processor in the tree collects and reduces the data from its children and passes its partial result to the parent. When the root receives all partial results from its children, the final result is broadcasted. We can express this computation in the basic Dagger language by a when statement which is guarded by the entry points declared for each child. However, the number of children is not fixed for all processors even with a fixed number of processors; also ,as the number of processors changes, the spanning tree changes. To solve this problem, the language is extended by allowing entry points to receive multiple messages. In Figure 9, a dag chare for this example is shown. The entry declaration:

```
entry collect[n] MATCH : (message MSG *subresult[]);
```

associates a variable `n`, with the entry collect. This variable is initialized to a user specified value at the beginning (in `init` entry). The entry point `collect` now expects `n` messages stored in aqm array of message pointers called `subresult` in order to trigger a when-block. In the example, the variable `n` is assigned to the number of children, and the when-block "`when collect ..`" is activated only after receiving `n` messages at the entry point `collect`, and an `expect` for the entry `collect` has been executed. It is possible to access each of these messages (if an `expect` has been executed already) individually immediately after it is received, by using the keyword `ANY` as follows:

```
when collect[ANY] : { ... }
```

The reduction example handles multiple reduction requests concurrently. In order to achieve this, reference number mechanism is used. Client dags make their request by calling the public function `deposit` of the reduction dag. They associate a unique number with their request. This number is used by the reduction dag as a reference number of the messages traversing the spanning tree so that messages belonging to different requests will not be mixed. When a reduction is completed the result is sent back to the requester with the same reference number.

```
message { int value; } MSG;
dag BranchOffice reduction {
    int      parent,penum,is_root,is_a_leaf,n;
    entry init       : (message MSG *initmsg);
    entry collect[n] MATCH : (message MSG *subresult[]);
    entry recvresult MATCH : (message MSG *result);
    when init : {
        penum  = McMyPeNum();
        /* initialize the multi-message entry collect */
        n      = McNumSpanTreeChildren(McMyPeNum());
        parent = McSpanTreeParent(penum);
        is_root = (parent == -1) ? TRUE : FALSE;
        is_a_leaf = (n==0) ? TRUE: FALSE;
    }
    /* When all the results received, propagate or broadcast the result */
    when collect : {
        MSG *msg;
        int refnum;
        refnum = GetRefNum(subresult[0]);
        msg = (MSG *) CkAllocMsg(MSG);
        SetRefNumber(msg,refnum);
        msg->value = combine(subresult) + get_myvalue(refnum);
        if (is_root)
            BroadcastMsgBranch(recvresult,msg);
        else
           SendMsgBranch(collect,msg,parent);
        expect(recvresult,refnum);
    }
    when recvresult : {
        send_result(result);
    }
    public deposit(v,refnum,return_entry,return_cid)
    int v,refnum;
    EntryNumType return_entry;
    ChareIDType  return_cid;
    {
        save_myvalue(v,refnum,return_entry,return_cid);/* save my contribution*/
        if  (is_a_leaf) {
             MSGUP *msg;
             msg = (MSGUP *) CkAllocMsg(MSGUP);
             msg->value = get_myvalue(refnum);
             SetRefNumber(msg,refnum);
             SendMsgBranch(collect,msg,parent);
             expect(recvresult,refnum);
          }
        else expect(collect,refnum);
    }
}
```

Figure 9: A reduction dag BOC illustrating multiple message entries

5 Implementation

The Dagger has been implemented on top of the Charm system. The implementation is composed of two parts: translation of a dag chare and run time management.

The Charm translator has been extended to transform dag chares into chares. The translation of a dag chare is illustrated in Figure 10. Each entry declaration in the dag chare is converted to an entry point. All the when-blocks become private functions of the chare. In addition to these, some control data structures are defined as local data of the chare and are manipulated by the Dagger run-time. These data structures include *three queues*: message queue, waiting queue and ready queue. Message queue keeps the messages until they are consumed by the when-blocks. The waiting queue contains the instances of activated when-blocks that are waiting for some messages or `expect/ready` statements. The ready queue is a list of when-block instances which are eligible for execution. When a message is received, the corresponding entry point is invoked by Charm. This entry point, (which is produced by the Dagger translator), contains the code to buffer the message and to check if any instance of a dependent when-block is eligible for execution. A when-block instance in the waiting queue has a counter. This counter is initialized to the number of entries and conditions in the dependence list of the when-block. The counter is decremented by the availability of expected messages, and/or execution of `ready` statements. Whenever the counter reaches zero, the when-block instance is put into ready queue. At the end of each entry-point code, the function `process_ready_list` is called to execute the when-block instances in the ready queue before returning the control to the Charm runtime.

6 Preliminary Performance Results

In this section, the preliminary performance results of the Dagger will be presented. First, we will look at the overhead introduced by the Dagger with respect to Charm level. This overhead is a result of buffering messages, and queueing and matching operations. We do not expect the higher level of abstraction and more expressiveness provided by Dagger to come free. However, We expect that the overhead to be insignificant, and the overall performance of the Dagger to be better than the performance of blocking-receive model.

Table 1 depicts the overhead introduced by the Dagger. The Jacobi method, described in Section 3.2, was implemented in Charm and Dagger. The table shows the elapsed time for different computation/communication ratios. As expected the overhead becomes insignificant as the grainsize increases. In addition, there are opportunities to reduce the overhead of the Dagger in the Dagger runtime system.

The main motivation behind the Dagger is to retain the ability to overlap computation and communication. In order to demonstrate this feature, a concurrent reduction program was programmed in C with blocking-receive and in Dagger on NCUBE/2. Each processor has an array which is partitioned into blocks of size 512 words. A reduction (max) operation is performed for each partition. The reductions are independent of each other. The blocking-receive version calls the NCUBE/2 system library function `nrmaxn` to perform the reduction. This call is a blocking call, and the processor waits for the result of the reduction. In the Dagger version, the program initiates reduction operation by calling `deposit` function of the reduction dag which is shown in Figure 9, then it continues with the next available piece of computation. Figure 11 shows the

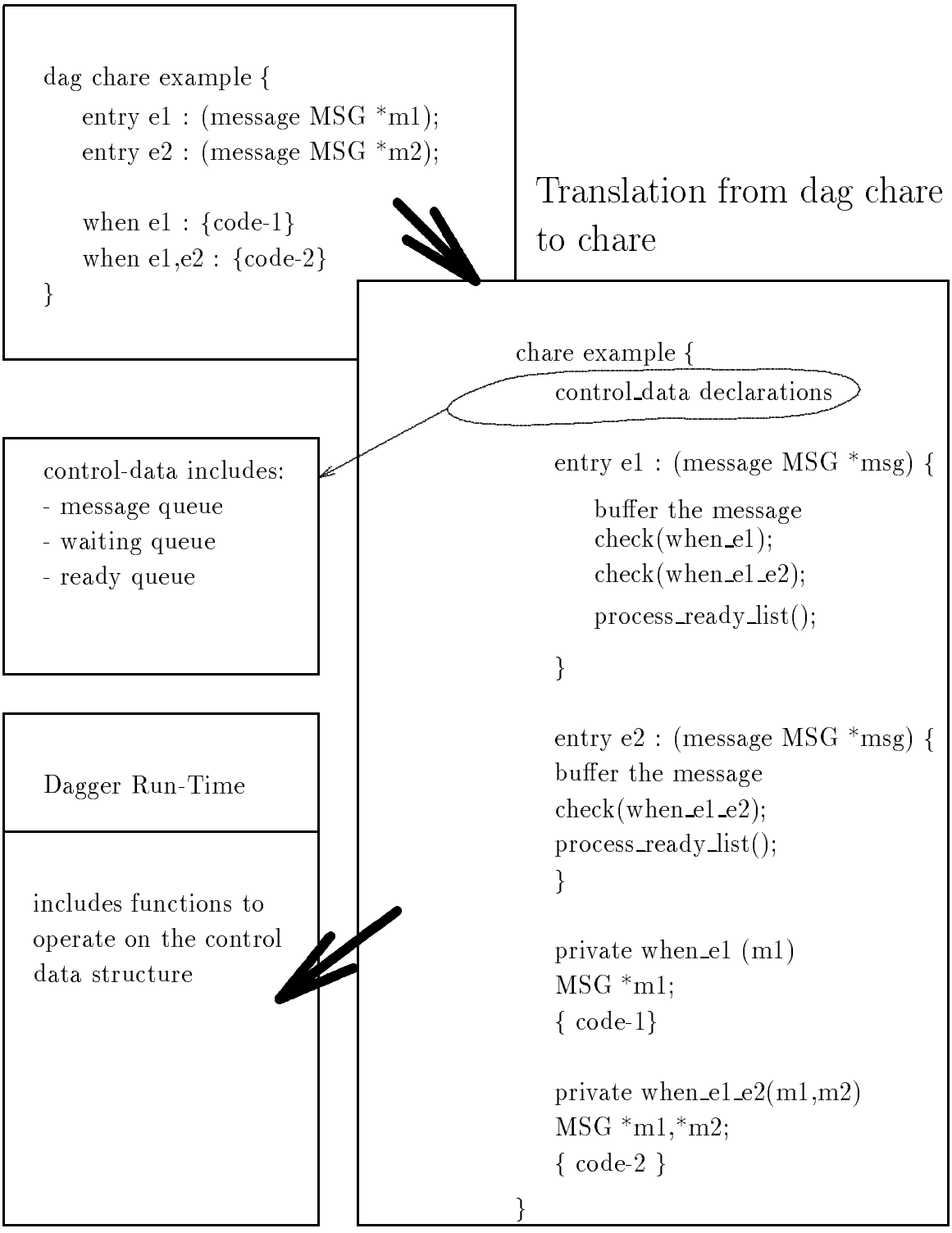

Figure 10: Translation of a dag chare

machine	Sequent Symmetry			NCUBE		
grid size	16x16	32x32	64x64	16x16	32x32	64x64
Dagger version (sec.)	3.10	21.28	159.72	1.39	4.95	21.69
Charm version (sec.)	2.93	20.83	158.1	1.26	4.61	20.95
overhead(percent)	5.8	2.16	1.02	10.7	7.4	3.5

Table 1: Dagger Overhead

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 (the initial increase up to 16 processors is due to the maximum branching factor of the spanning tree employed by Charm which increases from 1 to 4 and stops at 4 beyond 16 processors).

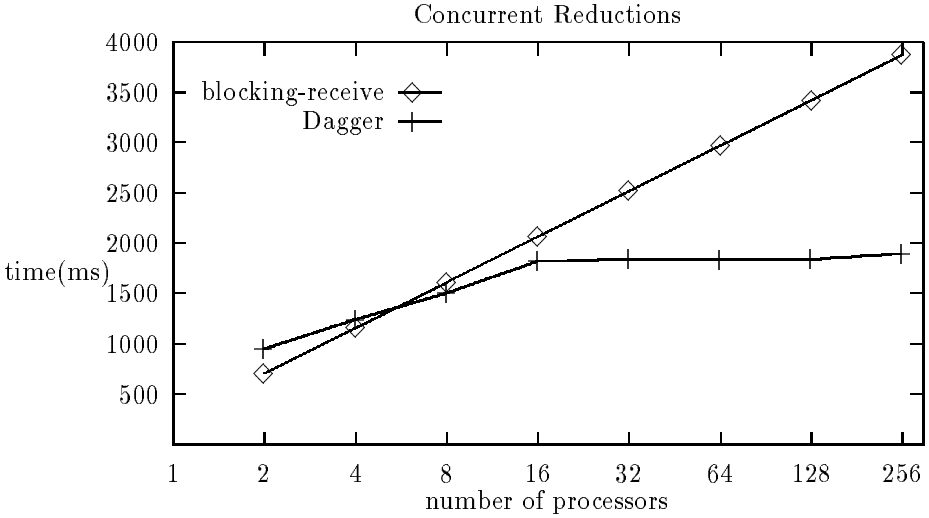

Figure 11: Overlapping Communication and Computation

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 [8] which proposed the *enable set* construct. Using this, one may specify which messages may be processed in the new state. Any other messages that are received by an actor are buffered until the current enable set includes them. Thus,this 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 that special purpose hardware. Also, again, these approaches are often meant to be used as a back-end for compilers. Thus the inconvenience of maintaining the counters and buffers explicitly is not considered significant. These approaches thus are comparable to Charm itself rather than Dagger. Our experience with using Dagger as back-end for a compiler for a data parallel language [7] indicates that the dagger might provide more convenient intermediate language than macro data flow.

8 Conclusion

We presented a coordination language called Dagger which 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 exaggerates the communication latency. 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 allows expression of parallelism with ease at the same time retaining message driven execution.

The Dagger has been developed on top of Charm which supports message driven portable parallel programming on MIMD machines. As future work, the runtime of the Dagger will be tuned to decrease the overhead. Secondly, a visual editor for the Dagger is under development. The visual interface will allow users to express and edit the computations graphically.

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] 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.

[7] 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.

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

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