Structured Dagger: A Coordination Language for Message-Driven Programming

Laxmikant V. Kalé and Milind A. Bhandarkar

Department of Computer Science
University of Illinois, Urbana IL
{kale,milind}@cs.uiuc.edu

Abstract. Message-Driven Programming style avoids the use of blocking receives and allows overlap of computation and communication by scheduling processes (or objects) depending on availability of messages. Charm is a parallel programming system that uses message-driven execution to exhibit latency-tolerance. Charm supports objects whose methods can be triggered by remote objects asynchronously. Such asynchronous invocation capability endows Charm programs to tolerate communication latencies in an adaptive manner. However, many parallel object-based applications require the object to coordinate the sequencing of the execution of their methods. Structured Dagger is a coordination language built on top of Charm that supports such applications by facilitating a clear expression of the flow of control within the object without losing the performance benefits of adaptive message-driven execution.

1 Introduction

One of the daunting tasks for parallel programmers is to tolerate message latency and unpredictable delays in remote response. Message-driven style of parallel programming attempts to tolerate such latencies by disallowing any process to block the processor when trying to receive messages and scheduling computation depending upon availability of messages. Message-driven parallel programming languages provide constructs for attaching code blocks to availability of specific messages. (In object-oriented systems these blocks correspond to methods of parallel objects.) These blocks are scheduled for execution by the run-time system when the specified messages arrive. This scheme minimizes the performance impact of communication latency by scheduling a ready process for execution while other processes are waiting for data.

Charm [6] is one of the first object-based portable parallel programming languages that embodies message-driven execution and promotes modularity while exhibiting latency tolerance. The order of execution of processes is determined by the order of messages received. Due to unpredictable delays in remote response times, the messages may arrive in any order and the programmer must deal with all possible message orderings. However, imposing an order on the arrival of messages, as is done in the traditional message-passing systems, tends to make the parallel program inefficient by letting the communication latency affect its performance.

To solve this problem, a coordination language called Dagger [3] was developed on top of the Charm programming system. However, the structure of Dagger programs still does not clearly express the flow of control in certain situations. We propose a new coordination language called Structured Dagger, which reduces the complexity of message-driven objects further by providing constructs to express control flow as a series-parallel graph.

The next section describes Charm language. Sections 3 and 4 discuss Structured Dagger language and its implementation. We discuss related work in section 5 and conclude in section 6.

2 Charm

Charm is a machine independent parallel programming system [6]. Programs written using this system run unchanged on MIMD machines with or without shared memory. The programs are written in C with a few syntactic extensions. Charm currently runs on many distributed and shared memory parallel machines, as well as workstation networks.

Charm programs consist of potentially medium grained objects (*chares*), and a special type of replicated objects, called branch-office chares. Charm supports dynamic creation of chares, by providing dynamic (as well as static) load balancing strategies. Chares interact by sending messages to each other and via specific information sharing modes.

The Charm runtime system is message-driven. It repeatedly selects one of the available messages from a pool of messages, switches to the context of the chare to which it is directed, and initiates execution of the code specified by the message.

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. Local variables of a chare are shared among the chare's entry-points and private functions. Calls are provided to create chares and send messages to existing chares.

A branch office chare (BOC) is a form of chare that is replicated on all processors. An instance of a BOC has a branch chare 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. All the branches of a single BOC instance share a global ID. One can send a message to a specific branch chare of a BOC, on a particular processor, or broadcast it to all its branches. BOC's are useful for some computations such as reduction operations, expressing static load balancing, and SPMD style programs.

In addition to messages and BOC's, Charm provides other ways in which objects share information. The information sharing abstractions supported include readonly variables, monotonic variables, writeonce variables, accumulators and distributed tables. 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, *row, *col;
  ChareIDType chareid;
  entry init: (message MSG *msg) {
    count = 2; MyChareID(&chareid);
    Find(Atable, msg->row_index, recv_row, &chareid,NOWAIT);
    Find(Btable, msg->col_index,recv_col,&chareid,NOWAIT);}
  entry recv_row: (message TBL_MSG *msg) {
    row = msg->data; if (--count == 0 ) multiply(row,col);}
  entry recv_col:(message TBL_MSG *msg){
    col = msg->data; if (--count == 0) multiply(row,col);}
}
```

Fig. 1. Matrix multiplication chare

An example in Charm. Consider an algorithm for matrix multiplication that is dynamically load balanced. 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 1. 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 supported by the distributed tables mechanism in Charm. Note that the `Find` call is non-blocking, and it immediately returns. Eventually, the row (and column) data will be sent in a message 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. The dependence (i.e. the flow of control within mult_chare) should therefore be enforced using mechanisms such as counters and message buffers. Here, a chare-private variable, `count`, is initially set to 2, and is decremented with arrival of each message. When `count` becomes zero, the stored messages are fetched and multiplication is performed. 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 Structured Dagger : The Language

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

Structured Entry-Methods: The Structured Dagger language is defined by augmenting Charm with structured entry-methods, which specify pieces of computations (*when-blocks*) and dependences among computations and messages. A when-block is guarded by dependences that must be satisfied before it can be scheduled for execution. These dependences include arrival of messages or completion of other constructs. Before describing the Structured Dagger language in detail, let us consider the matrix multiplication example once again, and show how it can be coded in Structured Dagger. Figure 2 shows the matrix

```
chare mult_chare {
  structentry init : (message MSG *msg){
    atomic {
      Find(Atable, msg->row_index,...);
      Find(Btable, msg->col_index,...); }
    when recv_row(TBL_MSG *row), recv_col(TBL_MSG *col) {
      atomic{ multiply(row->data,col->data) }}
  }
}
```

Fig. 2. Matrix multiplication

multiplication written using Structured Dagger. Whenever the entries `recv_row` and `recv_column` receive messages, the `multiply` function is called with the rows and columns that have been received. Structured Dagger takes care of the bookkeeping functions such as incrementing counters, flags and buffering the messages. Therefore, the resulting code is more readable (and easy to program), and retains the benefits of the message-driven model.

When-Blocks: When-blocks specify dependence between computation and message arrival at an entry-point. In general, a when-block may specify its dependence on more than one entry-point. When all constituent entry-points receive messages, computation corresponding to the when-block may be triggered.

When-blocks combined with the ordering constructs are adequate for specifying computations where concurrent phases of the same computations may not co-exist. However, in many practical problems, such as Jacobi Relaxation in numerical methods, many phases of the same computation may be running concurrently. Since Charm does not ensure in-order delivery of messages by the underlying communication mechanism, messages intended for different phases of computations may get mixed if they arrive out-of-order and as a result, the computation will go wrong.

In order to handle this problem, Structured Dagger provides *reference numbers* attached to messages to distinguish between messages belonging to different phases of computation. A when-block optionally specifies the reference numbers for the messages triggering its constituent entry-points. Messages that belong to the same phase of the computation are given specific reference numbers by the user. Structured Dagger matches the messages with those reference numbers to activate a when-block.

Atomic Construct: The `atomic` construct is a wrapper around C statements and specifies that no Structured Dagger constructs appear inside it. Since no structured construct exists inside an `atomic` construct, it does not contain code executed depending on the arrival of remote messages and is therefore executed atomically.

Ordering Constructs: Receiving a message at an entry-point is not sufficient to trigger a computation. The computation must be in a state where it is ready to process the message. Even if all the entry-points specified in a when-block have received messages, the computation specified in the when-block is not triggered until other constructs occurring previously in the program order may not have completed. The program order may be specified in Structured Dagger using the ordering constructs, `seq` and `overlap`.

The `seq` construct is written as `seq{construct-list}` and ensures that each of the constructs in the list is enabled only after its predecessor completes. Note that, `seq` construct is not the same as `atomic` construct because it may contain other Structured Dagger constructs. The `seq` construct completes when the last of its component constructs reaches completion.

The `overlap` construct enables all its component constructs concurrently and can execute these constructs in any order. Actual execution of these component constructs may be dependent on arrival of messages that trigger them. An `overlap` construct reaches its completion only after each of its component constructs has completed.

Conditional and Looping Constructs: In many situations, one may need to conditionally enable the Structured Dagger constructs, or to iterate over a set of constructs. Since `atomic` construct cannot include any Structured Dagger constructs, the C statements such as `if`, `while`, and `for` cannot be used for

this purpose. Therefore, Structured Dagger provides the equivalent constructs. If more than one component constructs appear inside such a construct, thet are implicitly enclosed by a `seq` construct. The constructs supported include:
if (*condition*) {construct-list} **else** {construct-list}
while (*condition*) {construct-list}
for (*stmt; condition; stmt*) {construct-list}
forall (var=*const,const,const*) {construct-list}
A `forall` construct enables its component constructs for the entire iteration space as opposed to the `while` and `for` constructs, which enable their component constructs for each element of the iteration space in strict sequence.

```
BranchOffice Harlow_Welch{
//chare-local variables declarations
structentry init:(MSGINIT *msg){
  seq {
    atomic { initialize();for(i=0; i<Z; i++) convdone[i] = FALSE; }
    forall(i=0,Z-1,1){
      while(!convdone[i]){
        atomic { for (dir=0; dir<4; dir++){
                   m[i][dir] = copy_boundary(i,dir);
                   SendMsgBranch(entry_no[dir],m[i][dir],nbr[i][dir]);}}
        when North(Bdry *n),South(Bdry *s),East(Bdry *e),West(Bdry *w){
          atomic { update(i, n, s, e, w);
                   reduction(my_conv, i, Converge, &mycid);}}
        when[i] Converge(Conv *c) {atomic{convdone[i] = c->done;}}
      }
      atomic { print_results(); }
    }
  }
}}
```

Fig. 3. Harlow-Welch Program

Example Program: We present an example Structured Dagger program that implements the Harlow-Welch scheme in Computational Fluid Dynamics. The control flow is expressed in Figure 3. Each iteration in this scheme consists of communicating the boundary elements with neighbors in the 2-D grid followed by a global reduction to check whether the scheme has converged. (The reduction is carried out asynchronously by a separate object and is not shown here.) This is done concurrently for all the planes and each of the planes could converge independently of each other.

4 Structured Dagger : Implementation

Structure Dagger is implemented on top of Charm as a translator and a runtime library. The Structured Dagger translator transforms the program to an equivalent Charm program. The translation consists of splitting a structured entry-point into a number of Charm entry-methods and chare-private functions, inserting counters and flags to specify dependences between different component constructs of the structured entry-point.

For each construct, the translator generates code for enabling the construct and for the completion of the construct. Code generated for completion of the construct contains code to free the message buffers occupied by the messages arrived during its execution as well as to enable the constructs that may be dependent on its completion.

The runtime library maintains one message queue. Whenever any when-block is enabled, it checks for the messages intended for its component entries. If all of these are available, it enables its component constructs and if possible executes them (In particular, atomic constructs do not have dependence on message arrival, therefore the code generated for when-block executes code in those constructs.) The entry-method generated corresponding to each of the entries within when-blocks contains code to buffer the message, set the appropriate flags and awaken any when-blocks that may be waiting. By doing a careful analysis of this dependence, the translator avoids periodic checking for all enabled when-blocks and only the when-block that may be waiting for a particular entry-point is enabled.

For assessing the performance impact of our translation scheme, we ran a simple program on a single node of CM-5. This program created two objects, supplying one of them with a *seed* message, which then started sending messages and receiving responses from the other object in a loop for a specified number of times. We compared the performance of our Structured Dagger program with a Charm program and also with a multi-threaded program written using thread-objects in Converse [7]. The results for 10000 round-trip messages (each of size 4 bytes) are in table 1. As can be seen from these results, Structured Dagger program does not add significant overhead to the native Charm code, while it reduces the program complexity. Even for a simple program such as this, the cost of context-switching in a multi-threaded program is very high, which justifies our use of message-driven execution in Structured Dagger.

Table 1. Performance Results

Program	Charm	Multi-Threaded	Structured Dagger
Time(seconds)	1.390	5.654	1.890

5 Related Work

Dagger [3] is an earlier attempt to build a coordination language on top of Charm. The concept and structure of when-blocks in Structured Daggeris borrowed from Dagger. Dagger makes it easy to program a more general class of control flow graphs than Structured Dagger, using when-blocks, `expect` and `ready` statements, and condition variables. A when-block specifies dependences as a list of entries and condition variables with their associated reference numbers. A Dagger program tells the run-time system that it is at a stage to process a message by issuing an `expect` statement. If the arrived message is not expected, it is buffered for later retrieval. A condition variable is used to signal the end of a when-block with a `ready` statement. Thus control-dependences among when-blocks belonging to the same chare can be expressed using condition variables. However, the structure of Dagger programs is not as perspicuous as Structured Dagger because a Dagger program is a flat collection of when-blocks. Structured Dagger adds on to Dagger the ability to express dependences between the when-blocks with a cleaner structure, while sacrificing the generality that Dagger provides.

CC++ [4] is an object-parallel language that bears some similarities to Structured Dagger. CC++ is a thread-based system. A computation consists of one or more processor objects each with its own address space. Objects within these processor objects can be accessed by remote objects using global pointers. Within individual processor objects, new threads can be spawned using the structured constructs *par*, and *parfor*, and the unstructured construct *spawn*, which creates a new parallel thread. Multiple threads created by these statements may be executed by different processors, or interleaved on the same processor, and they may share variables.

The *par* and *parfor* constructs of CC++ are analogous to the *overlap*, and *forall* constructs in Structured Dagger. However, they are different in a fundamental sense: two statements in a *par* construct may actually be executed in parallel by two different processors, whereas two constructs in an *overlap* statement are always executed by the same processor. Also they can interleave only in a disciplined fashion: only entire when-blocks can be interleaved, based on the arrival of messages, and not the individual C statements.

The most important difference between Structured Dagger and CC++ (and other systems such as Chant [5]) has to do with threads. Using threads creates a flexibility, but at a cost: thread context switches are more expensive than message-driven invocations of methods in Charm or Structured Dagger(as illustrated in fig. 1); also, threads waste memory: creating hundreds or thousands of threads, each with its own stack, may not be possible, whereas a large number of parallel objects can easily be created without reaching memory limits.

ABC++ [2] is a thread-based object-parallel language. Both synchronous and asynchronous remote method invocations are allowed. There is a single thread associated with each parallel object. This thread receives messages corresponding to method invocations and decides when and whether to invoke methods. Primitives are provided to selectively enable execution of individual methods.

Unlike Structured Dagger, no direct expression of control flow across method invocations is possible.

The *enable set* construct [8] addresses the issue of synchronization within *Actors* [1]. 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. The ordering constructs in Structured Dagger achieve this in a cleaner manner. Also, 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.

6 Conclusion

We presented a coordination language called Structured Dagger which is a notation for specifying intra-process control dependences in message-driven programs. This language combines efficiency of message-driven execution with the explicitness of control specification. Structured Dagger allows easy expression of dependences among messages and computations and also among computations within the same object using when-blocks similar to Dagger and various structured constructs. Structured Dagger has been developed on top of Charm and is portable across many MIMD machines, with or without shared memory.

References

1. G.Agha, *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press. 1986.
2. E. Arjomandi et. al., "ABC++: Concurrency by inheritance in C++", *IBM Systems Journal*, Vol 34, No. 1, 1995.
3. A.Gursoy, *Message Driven Execution and its Impact on the Performance of CFD and other Applications*, Ph.D Thesis, University of Illinois at Urbana-Champaign, Jan 1993.
4. K.Mani Chandy and C. Kesselman, "Compositional C++: Compositional Parallel Programming", *Technical Report no. Caltech-CS-TR-92-13*, Department of Computer Science, California Institute of Technology, 1992.
5. M. Hainer, D. Cronk and P. Mehrotra, "On the Design of Chant: A Talking Threads Package", *Proceedings of Supercomputing '94*, Nov 1994.
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. L.V.Kale et.al., "Converse: An Interoperable Framework for Parallel Programming", *Submitted to International Parallel Processing Symposium, 1996.*
8. C.Tomlinson, V.Singh, "Inheritance and Synchronization with Enabled-Sets", ACM OOPSLA 1989 , pp103-112.
9. The CHARM(4.0) programming language manual, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, 1993.

This article was processed using the LaTeX macro package with LLNCS style