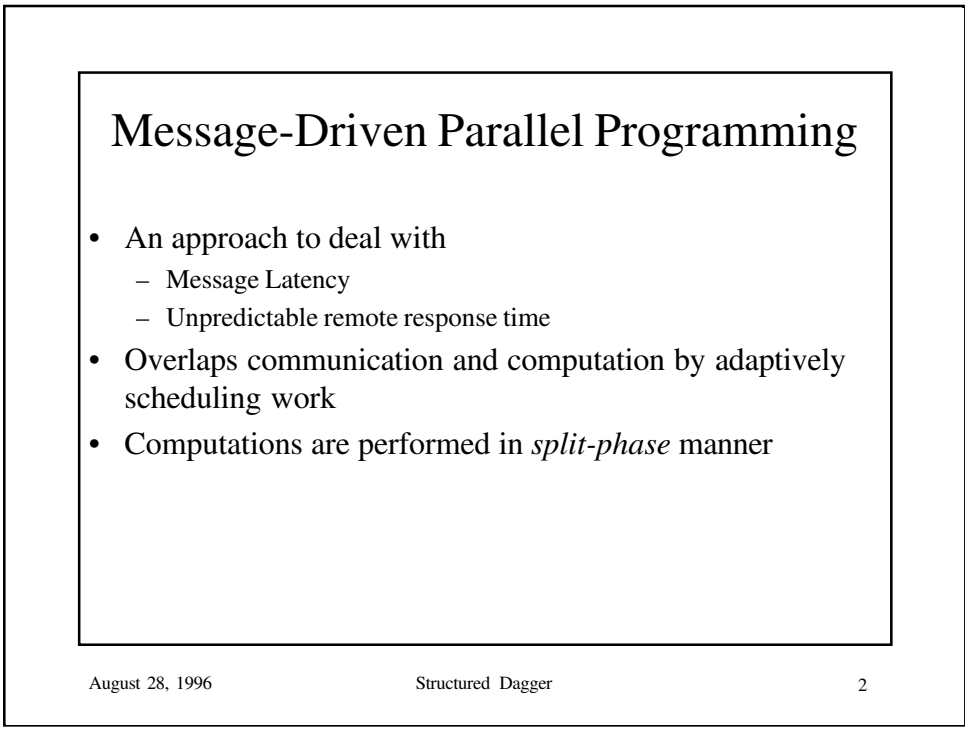
Structured Dagger

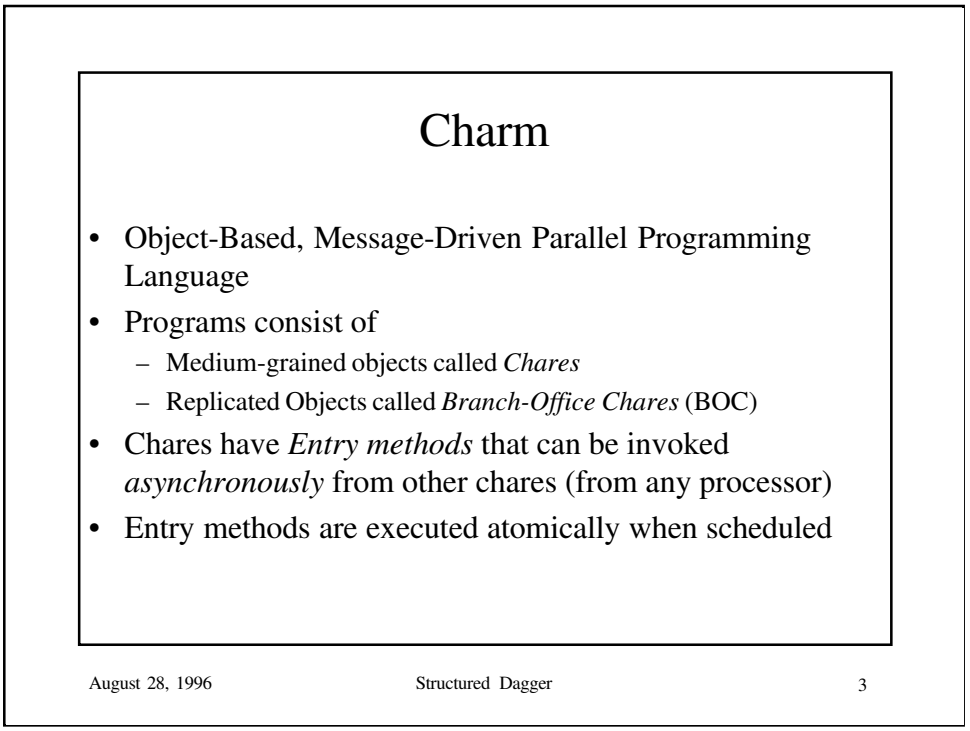
A Coordination Language for Message-Driven Programming

Laxmikant V. Kalé and Milind A. Bhandarkar
Parallel Programming Laboratory
Department of Computer Science,
University of Illinois, Urbana IL
{kale,milind}@cs.uiuc.edu



Message latency is one of the main performance degradation factors in parallel programs. One of the major approaches used to reduce the impact of message latency on the performance of parallel programs is to overlap communication and computation. That is, scheduling useful work while the process is waiting for response from remote processor. But the remote response time is often unpredictable because of various factors, some of them beyond programmer's control. This prompts for adaptive scheduling of work based on occurrence of certain events, such as availability of messages.

Message driven programming languages provide constructs to attach code blocks to be executed depending upon availability of messages. Computations are performed in split-phase manner: A request for remote response is posted and the code attached to availability of remote response is executed when the remote response arrives in the form of a message. In the meanwhile, other computations are scheduled.



Charm, developed in our group over the last 10 years, was one of the first object-based message-driven parallel programming languages. Charm programs consist of medium-grained objects (called chares) and replicated objects (called branch-office chares). Each chare consists of local state variables and several entry-methods that can be remotely invoked by sending messages to the chare. This method invocation is asynchronous, because the remote chares do not wait for method invocation and return immediately after sending a message. Branch office chares is similar to an array of objects such where one instance is placed on each processor. Branch office chares are used in Charm for reductions, load-balancing and expressing SPMD programs.

Charm (Contd.)

- Scheduler

```
while(there are messages in the queue) {
  remove message from queue
  context switch to object to which the message is directed
  execute method specified in the message
}
```

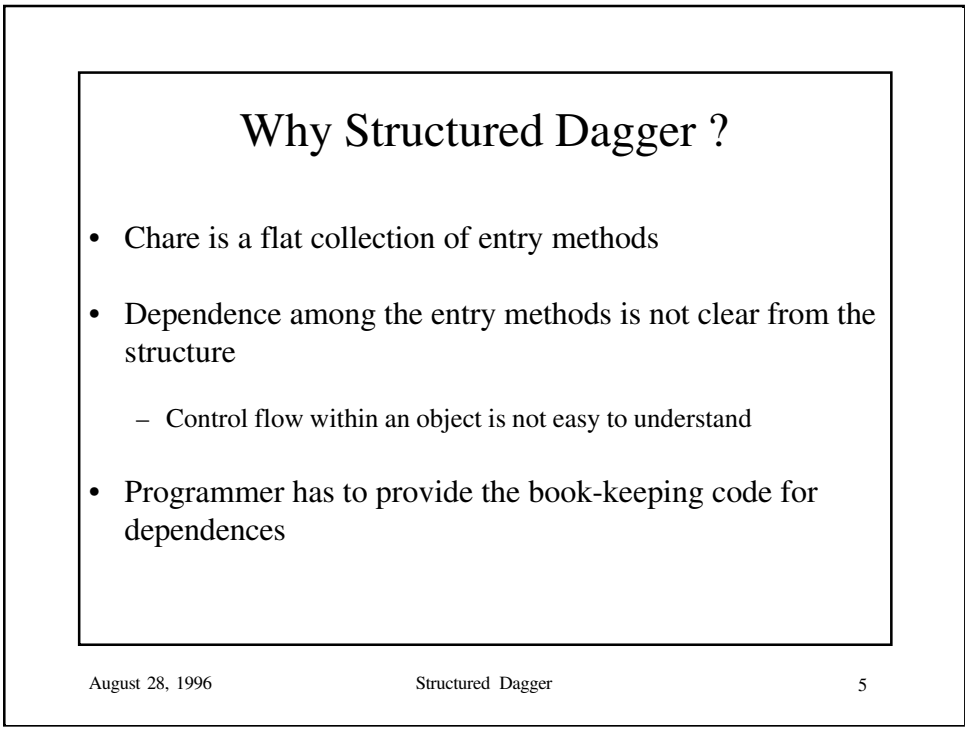
- Dynamic Creation of Chares
  - `CreateChare(ChareName@EntryName,Message,..)`
- Asynchronous method invocation
  - `SendMessage(ChareID, Entry, Message)`
- Specific Information Sharing Abstractions
  - readonly, writeonce variables, accumulators, distributed tables

August 28, 1996 Structured Dagger 4

At the heart of a Charm program lies a scheduler that maintains a pool of available messages. It repeatedly picks up a message from that pool , switches the context to the chare at which the message is directed and invokes the method specified in the message.

Charm provides functions to create chares dynamically and to send messages to existing chares. In addition to the messages and chares, Charm provide specific ways to share information among chares. Some of these are: readonly variables, monotonics, writeonce variables, accumulators, distributed tables.

Charm has a sophisticated module system that facilitates software reuse in large-scale parallel programming.



A Charm program is only a flat collection of dependences between messages and computations. It is the programmer's responsibility to honor dependences between computations belonging to the same chare. In addition, the dependence structure is not readily visible to the reader of the Charm code. And therefore it is difficult to write and debug programs where there are a large number of such dependences. Next slide presents a simple example, which will make this clear.

Example : Matrix Multiplication

```
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);}
  }
}
```


This simple example program implements a parallel matrix multiplication algorithm. Each chare is responsible for computing one element of the result. The matrices are stored in distributed table abstraction supported by Charm . Upon initialization, the entry method **init** issues asynchronous requests to get a row and a column from the table. It also specifies the entry method in the chare which will be invoked upon arrival of those data. The computation of the result depends on arrival of both messages. These messages may arrive in any order because Charm does not put any restrictions of the message ordering. Therefore, the programmer is responsible for handling all possible orderings of messages by using counters and message buffers. (This example is deliberately chosen to be simple. Programming overhead increases exponentially when there are more dependences.) Structured Dagger was designed to automate these book-keeping tasks while maintaining the message-driven nature of the program.

Structured Entry Methods

- Structured Dagger augments Chare definitions with *Structured Entry Methods*
- Invocation similar to the ordinary entry methods (using `SendMsg()`)
- Only structured entry methods can contain Structured Dagger constructs.

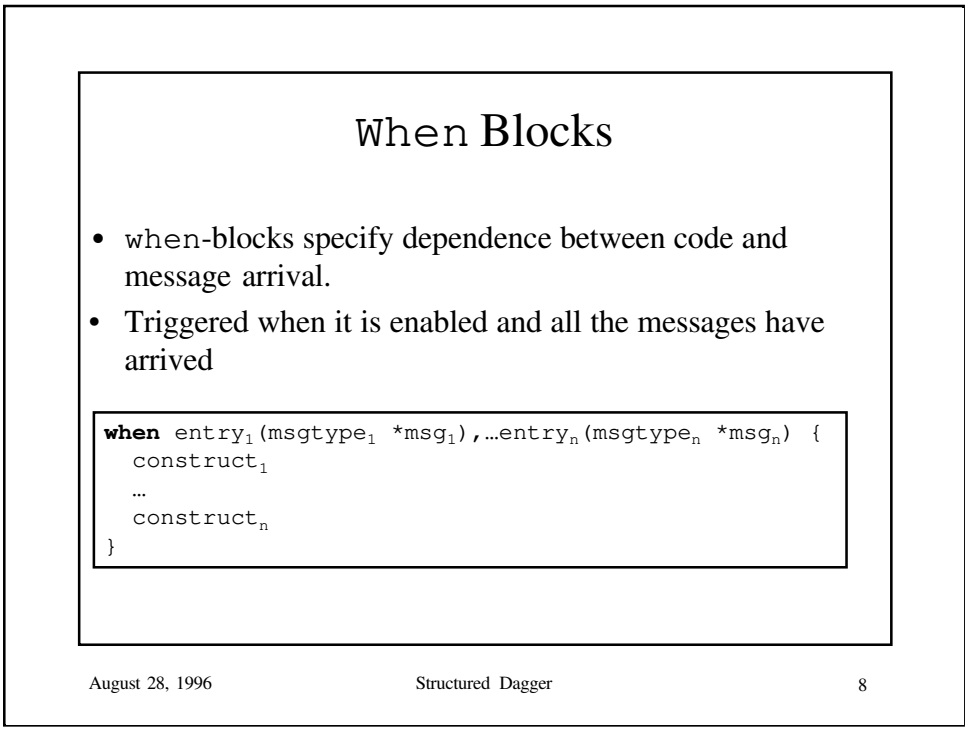
```
structentry entry_name: (MsgType *msg) {
  construct1
  …
  constructn
}
```

August 28, 1996 Structured Dagger 7

Structured Dagger augments Charm by including structured entry methods inside the definition of a Chare. A structured entry method can be invoked similar to Charm entry methods (that is, by sending a message to that chare specifying the entry method). However, the structured entry methods could contain structured dagger constructs and are translated into a number of internal entry methods, and private functions that take care of the book-keeping tasks.

Each structured dagger construct could have component constructs that depend on the outer construct. Each construct has enabling and completion semantics. As mentioned earlier, the execution of entry code depends not only on the availability of message directed to it, but also on the enabling of the construct which contains that piece of code.

Over the next few slides, we present structured dagger constructs.



When-Blocks specify dependences between computations and message arrival at entry points. A when-block dependent on only one message at an entry point is equivalent to a Charm entry point. However, the ability of the when-block to have multiple such "guards" gives the programmer a convenient notation to express dependences between multiple message-arrivals.

A when-block is triggered (that is, the code inside it is executed) when it is enabled and all the messages on which it is dependent have arrived.

Once the when-block is triggered, it enables the first of its component constructs.

Reference Numbers

- `when`-blocks are extended to allow concurrent phases of the same computation
- Messages have a *reference number* attached to them
- Reference numbers are *matched* for triggering the `when`-block

```
when[ref_num] entry1(type1 *msg1),…entryn(typen *msgn) {
   construct1
   …
   constructn
 }
```

August 28, 1996 Structured Dagger 9

A combination of When-blocks and the ordering constructs described in the next slides is adequate for expressing a class of algorithms where concurrent phases of computations do not co-exist. However, some algorithms in numerical methods and many other disciplines do not belong to this class. For example, in Jacobi relaxation or Harlow-welch algorithm, the computations are to be performed on multiple data samples concurrently inside a single chare. In the ordinary when-blocks presented earlier, the messages directed at different data samples may get mixed while resolving dependences. Therefore we augment the messages in Charm by including a “reference number” inside them. Their reference numbers are matched while determining whether the dependences for when-blocks are satisfied or not.

`atomic` Construct

- Contains C code
- Does not contain any Structured Dagger constructs
- (Therefore) Executed atomically

```
atomic {
   C statements
}
```


Atomic construct is a wrapper around C (Charm) statements. No structured dagger constructs can appear inside an atomic construct. Therefore the code inside this construct executes atomically.

Ordering Constructs : `seq`

- Each of the component constructs in the `seq` list is enabled in sequence
- Not the same as the `atomic` construct
- Reaches completion when the last construct in the list completes

```
seq {
  construct1
  …
  constructn
}
```

August 28, 1996 Structured Dagger 11

A **seq** construct enables each of its component constructs sequentially. That is , **Construct$_i$** is enabled only after **Construct$_{i-1}$** has reached completion. **Seq** is not the same as **atomic**, because **Seq** will contain structured dagger constructs which will typically be executed non-atomically. For example, **seq** might contain a **when**-block, which will be triggered after all its dependences are satisfied. **seq** construct reaches completion when the last of its component constructs have reached completion.

Ordering Constructs : `overlap`

- Enables all the component constructs simultaneously
- Component constructs execute concurrently on a single processor within the same object
- Reaches completion when all the component constructs complete

```
overlap {
  construct1
  …
  constructn
}
```

August 28, 1996 Structured Dagger 12

This is perhaps the most important construct in structured dagger and gives structured dagger the ability to do adaptive scheduling of constructs. **Overlap** construct enables all its component constructs simultaneously. Any of them could be chosen for execution depending upon whether the dependence of component constructs has been satisfied or not. This construct reaches its completion when all the component constructs have reached their completion. Note that **overlap** executes its component constructs concurrently, not parallelly. That is, all the component constructs of an **overlap** construct are executed on a single processor but execution may be interleaved in a disciplined fashion.

Conditional Construct

- Component constructs are chosen to be enabled based on condition
- Reaches completion when the selected component constructs complete

```
if (condition) {
  construct1
  …
  constructn
} else {
  construct1
  …
  constructn
}
```

August 28, 1996 Structured Dagger 13

Conditional construct is similar in syntax to the **if** statement in C. But the one cannot include structured dagger constructs inside the C **if** statements, because the C statements appear only inside atomic constructs. The semantics of the conditional construct is similar to the C **if** statement. If the component constructs include only atomic constructs, they are executed immediately. If they contain when-blocks, they are enabled and executed based on the availability of messages.

Looping Constructs : `while` and `for`

- Component constructs are enabled in loop based on loop condition

```
while(condition) {
   construct1
   …
   constructn
}
for(C-stmt; condition; C-stmt) {
   construct1
   …
   constructn
}
```


While and For constructs are similar in syntax and semantics to the **while** and **for** statements in C. However, similar to conditional construct, the component constructs of these constructs are only enabled and the actual execution depends on resolving the dependences. The component construct list inside these statements behaves as if it is enclosed by the **seq** construct.

Looping Constructs : `forall`

- Enables all the iterations of the loop simultaneously
- Concurrent phases of the loop body co-exist

```
forall (var=first, last, stride) {
  construct1
  …
  constructn
}
```

August 28, 1996 Structured Dagger 15

A **for** construct described in last slide enables the component constructs sequentially for each iteration. That is, unless the I^{th} iteration is complete, $(I+1)^{th}$ iteration cannot start. In the **forall** construct, however, all the iterations are enabled simultaneously. That is, all the iterations of the **forall** loop could execute concurrently. The construct reaches completion when all its iterations complete.

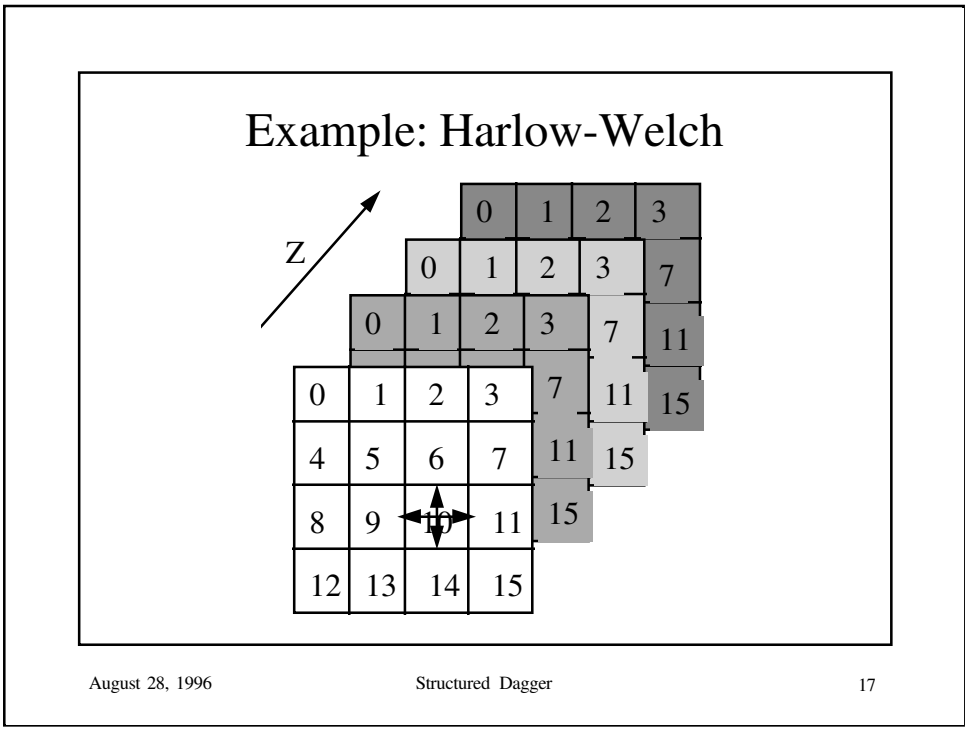
Example : Matrix Multiplication

```
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); }
    }
}
```

August 28, 1996 Structured Dagger 16

Lets consider the simple example of a matrix multiplication chare that we coded in Charm earlier and see how it can be coded using structured dagger. As you can see the book-keeping jobs (maintaining the counter and buffering the message) are now taken care by the structured dagger runtime and makes even this simple program more readable. [Next, we provide a real-life example of a program that implements Harlow-Welch scheme in CFD.]

Next two slides are optional



This figure example shows the communication in Harlow-Welch algorithm. The computational domain consists of a 3-D data-space, divided into multiple planes. A few planes are assigned to each branch office chare. Each plane is divided into sections each assigned to a different processors. The communication occurs only within a plane across processors. Each of these planes could converge at different rate depending on the data and therefore adaptive scheduling is necessary. The structured dagger program shown in next slide achieves that while maintaining readability of the code.

[Optional]

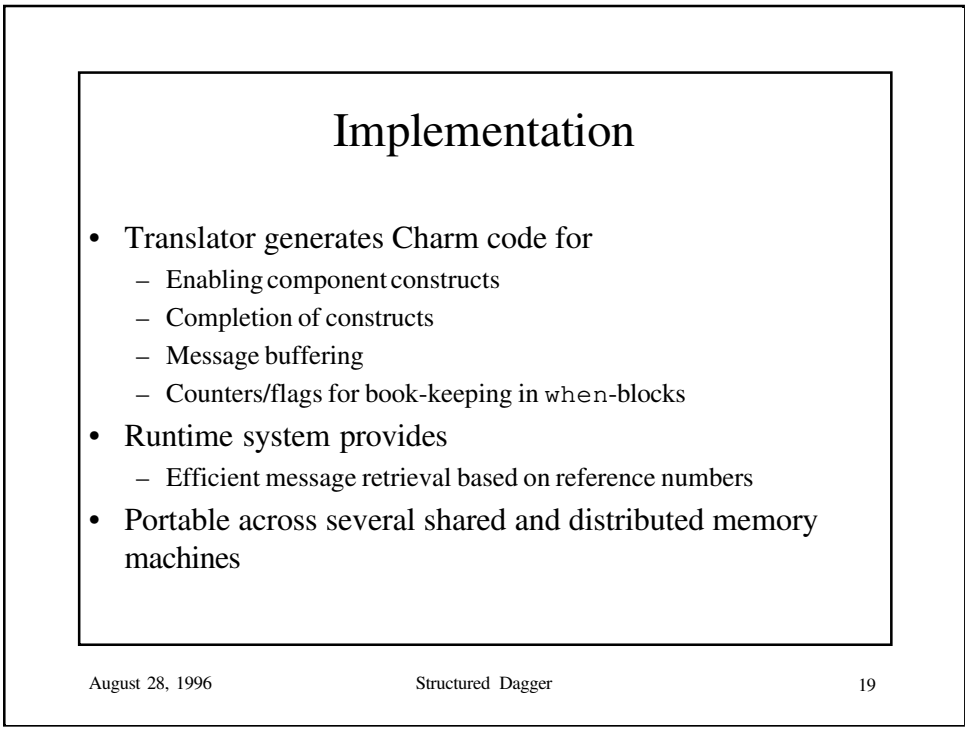
Example : Harlow-Welch

```
BranchOffice Harlow_Welch {
  // chare-local variable declarations
  structentry init: (message 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[I] 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;}}
        }
      }
    }
  }}
```

August 28, 1996 Structured Dagger 18

Each Chare in this program initializes its data, and initiates computation for each plane concurrently. Each section of a plane sends the boundary elements to neighboring sections of the same plane and waits for the neighbors' boundaries. Once all the boundaries arrive, it updates its data and checks for convergence using an asynchronous reduction in Charm. Here reference numbers are used in messages and when-blocks to ensure that the communication happens across sections of the same plane.

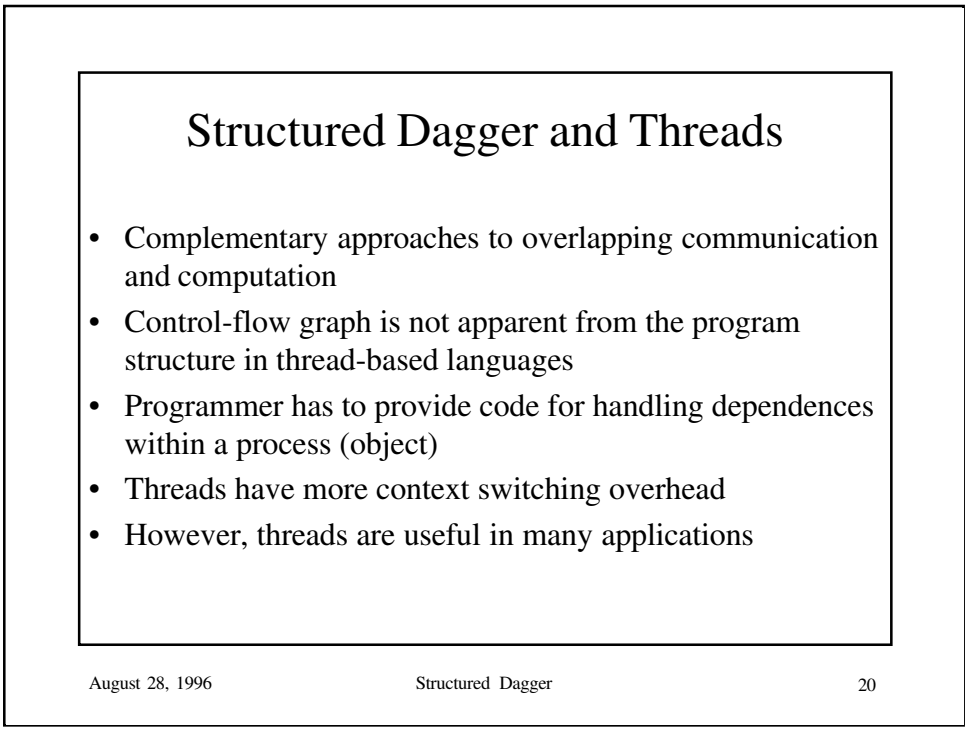
[Optional]



Structured Dagger is implemented as a translator and a runtime library on top of Charm. The translator generates code for enabling and completion of each construct. Code for completion of a construct also generates code to free the resources and to enable other constructs that may be waiting for the completion of this construct.

The runtime library maintains a message queue. Whenever a when-block is enabled, the generated code uses functions in the runtime library to check if the dependencies are satisfied and whether the when-block could be triggered.

The translator does a thorough analysis of the dependence to avoid periodically checking the list of enabled when blocks.



Structured Dagger's approach of message-driven execution is complementary to the approach of thread-based languages. However, threads are more difficult to use than structured dagger because, the control flow is not apparent from the program structure. The transfer of control within a process is buried deep down in the thread's code. If certain code depends on multiple messages to arrive, multiple threads could be forked to receive those messages concurrently. However, programmer has to write code for handling these dependences and message buffering. Also, as we show in our performance results, there is a significant context switching overhead in thread-based languages.

Preliminary Performance Results

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

August 28, 1996 Structured Dagger 21

We ran three simple programs that demonstrate the amount of overhead. These programs were written in Charm, structured dagger and a thread package. These figures show the time taken for 10000 round-trip messages for a ping-pong program. This program creates two objects and provides one with a seed message to start the ping-pong rolling. In a multi-threaded program, a thread in each object waits for a message. And sends back the received message. We have used the thread-package from the runtime system of Charm, called Converse. As one can see from these figures, the context-switching overhead in multi-threaded program is much higher than scheduling of objects in Charm or in Structured Dagger.. Also, the overhead added by structured dagger generated code, is not significantly high considering the tiny grainsize in this program. This experiment was carried out on a single node of CM5.

Related Work

- Dagger
  - Coordination language for Charm
  - Charm is augmented with *dagchare*, a special type of chare
  - when-blocks are guarded by entry methods and condition variables
  - dagchare is a flat collection of when-blocks
  - Can express any control-flow graph
  - Control-flow not visible from the structure of program
- CC++
  - *par* and *parfor* constructs (similar to `overlap` and `forall`)
  - Arbitrary interleaving of C++ statements
  - Thread-based

Dagger was an earlier attempt at building a coordination language on top of Charm. It uses when-blocks to specify dependences between code execution and message arrival. It also uses condition variables to specify dependences within the chare. This use of condition variables makes it possible to build arbitrary control flow graphs in dagger programs. However, the program is still a flat collection of when-blocks. The control flow is not apparent from the structure of program. Structured Dagger is better at expressing algorithms where the control flow graph is a series parallel graph.

CC++ is based on threads and has constructs **par** and **parfor** similar to **overlap** and **forall**. However, there is an important difference. The statements in **par** construct can be executed in parallel by multiple processors whereas that is not true in **overlap** construct. Interleaving in **overlap** construct is disciplined. Entire when-blocks are interleaved, and not the individual C statements. Also, as we have shown in earlier slides, thread context switching is much costlier than method-invocation.