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-Driven Parallel Programming

- An approach to deal with
  - Message Latency
  - Unpredictable remote response time
- Overlaps communication and computation by adaptively scheduling work
- Computations are performed in *split-phase* manner

Charm

- Object-Based, Message-Driven Parallel Programming Language
- Programs consist of
  - Medium-grained objects called *Chares*
  - Replicated Objects called *Branch-Office Chares* (BOC)
- Chares have *Entry methods* that can be invoked *asynchronously* from other chares (from any processor)
- Entry methods are executed atomically when scheduled

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

Why Structured Dagger ?

- Chare is a flat collection of entry methods
- Dependence among the entry methods is not clear from the structure
  - Control flow within an object is not easy to understand
- Programmer has to provide the book-keeping code for dependences

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);}
  }
}
```

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) {
  construct_1
  …
  construct_n
}
```

`When` Blocks

- `when`-blocks specify dependence between code and message arrival.
- Triggered when it is enabled and all the messages have arrived

```
when entry1(msgtype1 *msg1),…entryn(msgtypen *msgn) {
  construct1
  …
  constructn
}
```

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
 }
```

`atomic` Construct

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

```
atomic {
   C statements
}
```

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
}
```

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
}
```

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
}
```

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
}
```

Looping Constructs : `forall`

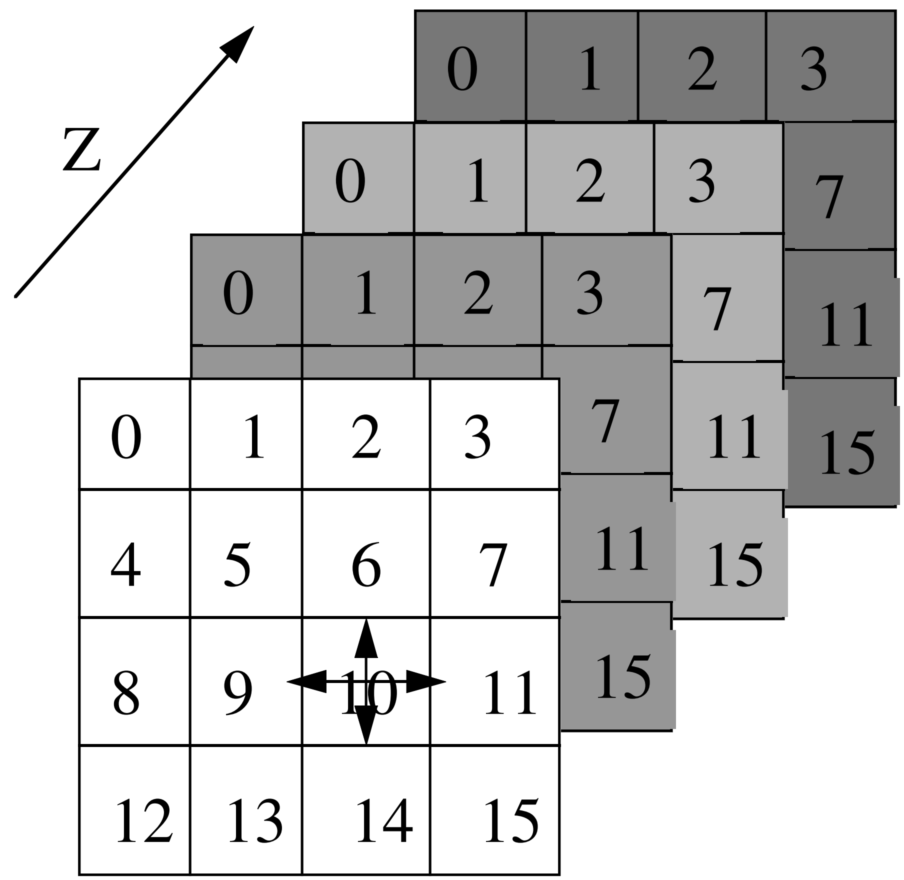
- Enables all the iterations of the loop simultaneously
- Concurrent phases of the loop body co-exist

```
forall (var=first, last, stride) {
  construct_1
  …
  construct_n
}
```

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); }
    }
}
```

Example: Harlow-Welch



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;}}
        }
      }
    }
  }}
```

Implementation

- Translator generates Charm code for
  - Enabling component constructs
  - Completion of constructs
  - Message buffering
  - Counters/flags for book-keeping in `when`-blocks
- Runtime system provides
  - Efficient message retrieval based on reference numbers
- Portable across several shared and distributed memory machines

Structured Dagger and Threads

- Complementary approaches to overlapping communication and computation
- Control-flow graph is not apparent from the program structure in thread-based languages
- Programmer has to provide code for handling dependences within a process (object)
- Threads have more context switching overhead
- However, threads are useful in many applications

Preliminary Performance Results

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

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