Dagger: Combining Benefits of Synchronous and Asynchronous Communication Styles

Laxmikant V. Kalé
Attila Gürsoy
Department of Computer Science
University of Illinois, Urbana
Traditional SPMD

- single process per processor
- mostly blocking message passing
  - messages have tags

```
t1 = f()
t2 = f()
send(tag1,t1)
recv(tag1,t3)
t4 = g(t1,t2)
t5 = g(t1,t3)
```
Overlapping Communication Latency in SPMD

![Diagram showing overlapping communication latency between P1 and P2.]
However,
if latencies are unpredictable, SPMD cannot overlap adaptively

```
recv(tag1,a);
recv(tag2,b);
t1 = f(a);
t2 = f(b);
```

```
recv(tag1,a);
t1 = f(a);
recv(tag2,b);
t2 = f(b);
```
Overlapping in SPMD cont.

Also, SPMD cannot

- overlap communication latencies across modules
- overlap idle times across modules
  (due to load imbalances and critical path)
Message Driven Execution

- Many processes per processor
- System maintains a pool of arriving messages
- Processes are activated by the arrival of messages
- Message Scheduling - selection of messages from the pool
  - FIFO
  - Priorities
- Message driven execution overlaps idle times:
  - while a process is waiting, another can take over
  - a single process may wait for multiple messages
Message Driven Execution

- Adaptively overlaps delays within a module
  
  \[
  \begin{align*}
  &\text{recv}(\text{tag1},a); \quad \text{this spmd code can be specified in} \\
  &\text{recv}(\text{tag2},b); \quad \text{message driven style such that t1 or} \\
  &t1 = f(a); \quad \text{t2 is computed first depending on} \\
  &t2 = f(b); \quad \text{which message arrives first}
  \end{align*}
  \]

- Message driven code:

  \[
  \begin{align*}
  &\text{entry tag1 : (message MSG } *a) \{ f(a); \} \\
  &\text{entry tag2 : (message MSG } *b) \{ f(b); \}
  \end{align*}
  \]
Overlapping in Message Driven Execution cont.

- Adaptively overlaps delays across modules not only idle times due to communication latencies but also due to load imbalances and critical path
A Message Driven System - Charm

- dynamic creation of processes (charms)
- dynamic load balancing
- specific information sharing modes
- compositionality and reuse
- runs on distributed and shared memory machines
  - intel iPSC/860, Paragon, CM5, NCUBE/2
  Multimax, Sequent Symmetry
  network of workstations
• Chare definition

```c
char char-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}
}
```

• Basic calls

- CreateChare(charName, entryPoint, msg)
- SendMsg(entryPoint, msg, charID)
Problems with Message Driven Execution

- But,
  - Nondeterministic flow of control
  - Message ordering bugs
  - Need to handle local synchronization with buffers, counters, and flags
Dagger

- expresses dependencies between messages and computations
- a message can trigger a computation if it is expected

Charm

Dagger
Example: Matrix Multiplication Chare

char mult_chare {
int count, *row, *column; ChareIDType charaid;
entry init:  (message MSG *msg) {
    count = 2; MyChareID(&charaid);
    Find(Atable, msg->row_index,recv_row, &charaid,NOWAIT);
    Find(Btable, msg->colm_index,recv_column,&charaid,NOWAIT);
}
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); }
}
Example: Matrix Multiplication Dag-Chare

```
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,...);
        Find(Btable, msg->colm_index,...);
        expect(recv_row); expect(recv_column);
    }
    when recv_row, recv_column :
        { multiply(row->data,column->data) }
}
```
Dag-Chare Definition

```c

dag char template {
    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()
    private function fm()
}
```
Dag-Chare cont.

- Entry Points
  
  ```
  entry entry_name : (message msg_type *msg)
  ```

- Expect Statement
  
  ```
  expect(entry_name)
  ```

- Ready Statement
  
  ```
  ready(cond_var_name)
  ```

- When Blocks
  
  ```
  when \( e_1, \ldots, e_n, c_1, \ldots, c_m \) : when-body
  ```
Expressing Loops in Dagger

```
when north,south,east,west : 
    update(n,s,e,w);
    iteration_count = iteration_count + 1;
    if (iteration_count < ITERATION_LIMIT) {
        send_boundaries();
        expect(north);
        expect(south);
        expect(east);
        expect(west);
    }
```
Problem with the loop example
Extended Language

- Reference Numbers
  - messages has reference numbers
  - a when block instance is activated if reference numbers match

- statements are modified
  - entry entry_name MATCH : (message msg_type *msg)
  - expect(entry_name,reference_number)
  - ready(cond_var_name,reference_number)

- new statements
  - SetRefNumber(msg,reference_number);
Correct Loop Program

```c
when north,south,east,west : 
    update(n,s,e,w);
    iteration_count = iteration_count + 1;
    if (iteration_count < ITERATION_LIMIT) {
        send_boundaries(iteration_count);
        expect(north,iteration_count);
        expect(south,iteration_count);
        expect(east,iteration_count);
        expect(west,iteration_count);
    }
```
Performance Results

Concurrent Reductions

Processor 1:

Processor 2:

Processor p:

reduce

1 2 k

n
Performance Results cont.

Concurrent Reductions on NCUBE/2,
problem size per processor = 4096 words, number of segments = 8
Summary:

- Message driven execution has performance advantages but expresiveness difficulties
- Dagger provides benefits of both

On going work:

- Visual Dagger
- Structured Dagger
- Simulation system for message driven programs
  - difficult without Dagger
    -> simpler flow for a restricted but common case