Dagger: Combining Benefits of Synchronous and Asynchronous Communication Styles

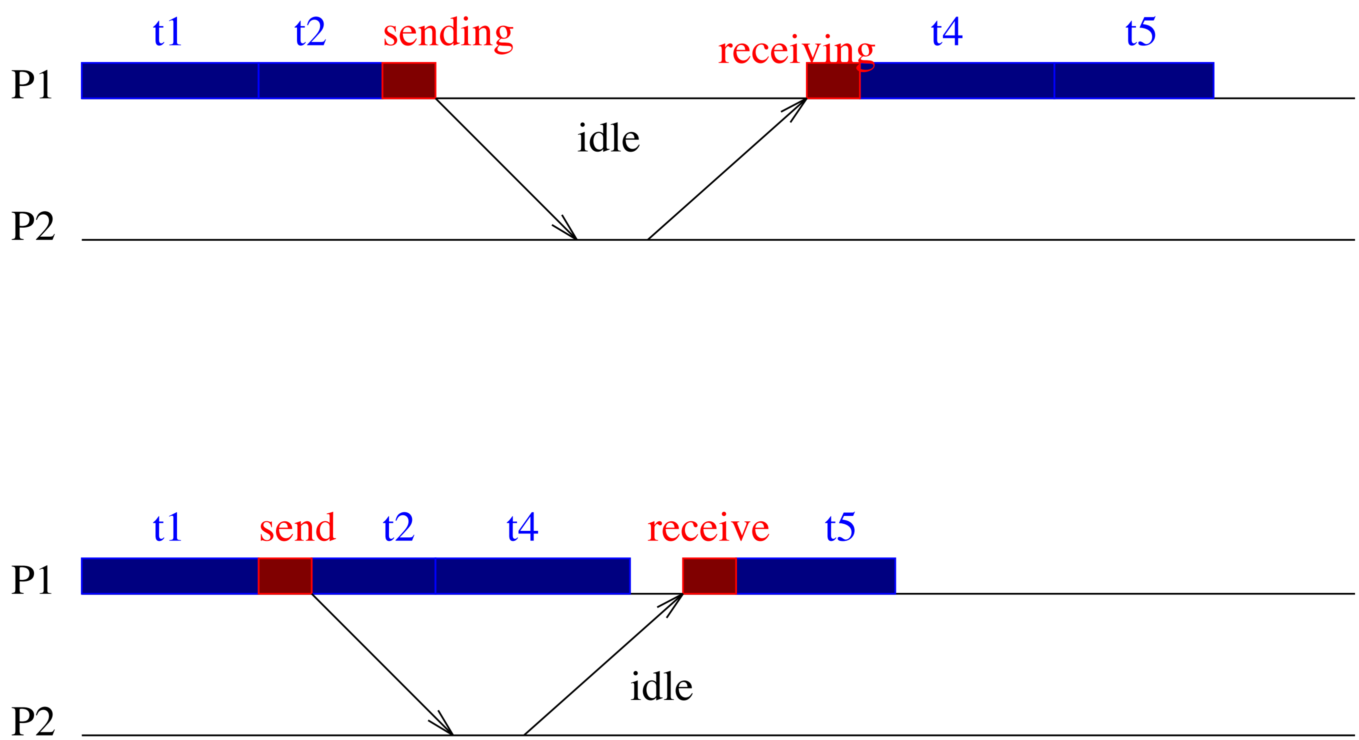
Laxmikant V. Kalé
Attila Gürsoy
Department of Computer Science
University of Illinois, Urbana

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



- However,
  if latencies are unpredictable, SPMD cannot overlap adaptively

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

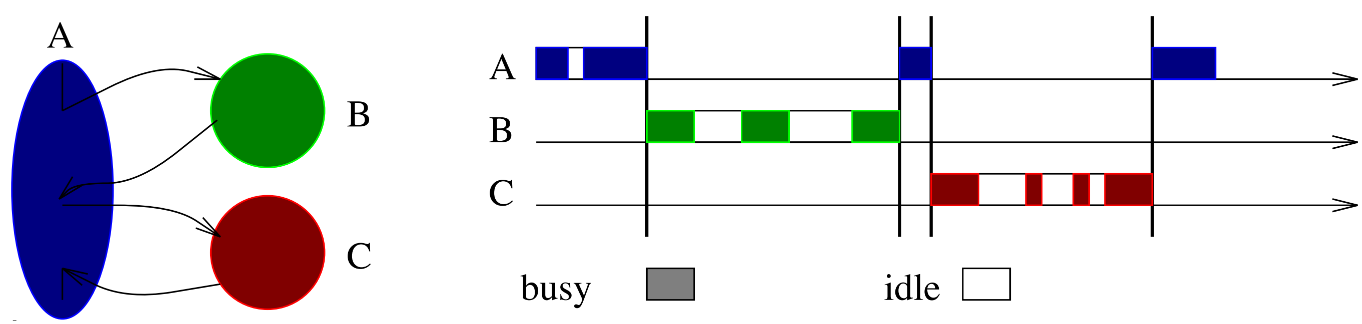
```
recv(tag1,a);
t1 = f(a);
recv(tag2,b);
t2 = f(b);
```

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

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

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

this spmd code can be specified in message driven style such that t1 or t2 is computed first depending on which message arrives first

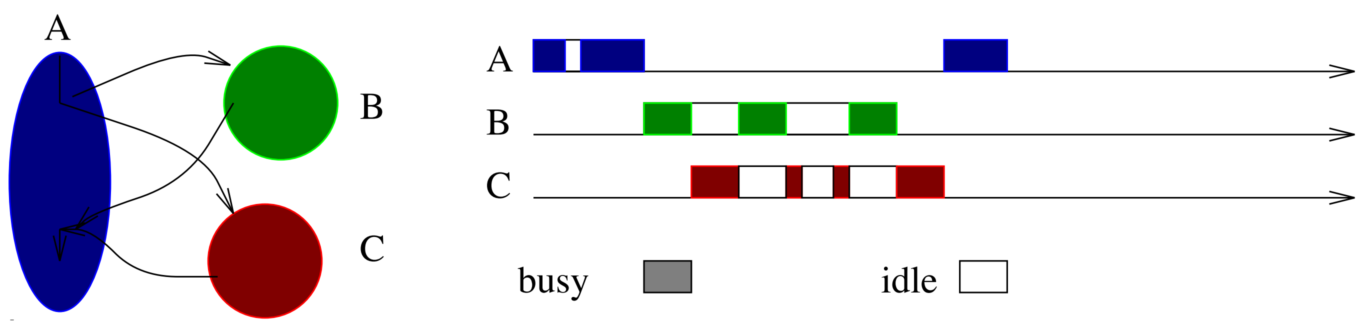
- Message driven code:

```
entry tag1 :  (message MSG *a) { f(a); }
entry tag2 :  (message MSG *b) { f(b); }
```

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 (chares)
- 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

```
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 }
}
```

- Basic calls

  – `CreateChare(chareName,entryPoint,msg)`

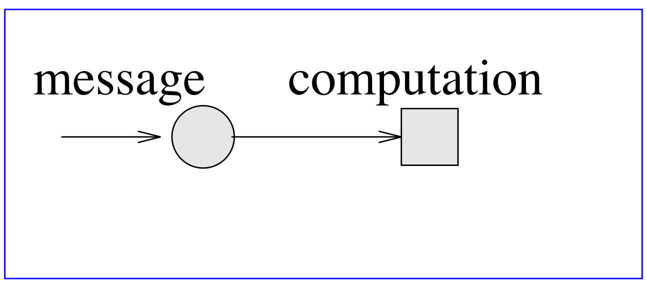
  – `SendMsg(entryPoint,msg,chareID)`

Problems with Message Driven Execution

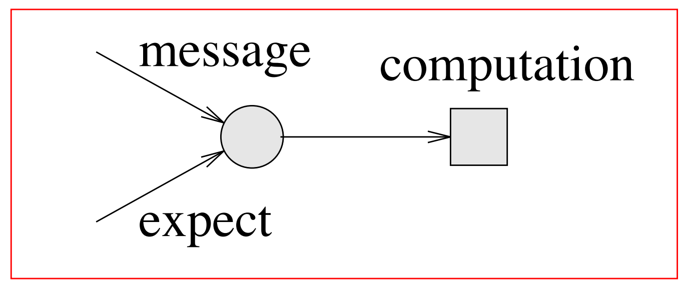
- Significant performance advantages
  (A.Gursoy, Ph.D Thesis 1994)
- But,
  - Nondeterministic flow of control
  - Message ordering bugs
  - Need to handle local synchronization with buffers, counters, and flags

Dagger

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



Charm



Dagger

Example: Matrix Multiplication Chare

```
chare mult_chare {
int count, *row, *column; ChareIDType chareid;
entry init:  (message MSG *msg) {
   count = 2; MyChareID(&chareid);
   Find(Atable, msg->row_index,recv_row, &chareid,NOWAIT);
   Find(Btable, msg->colm_index,recv_column,&chareid,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

```
dag chare 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)`

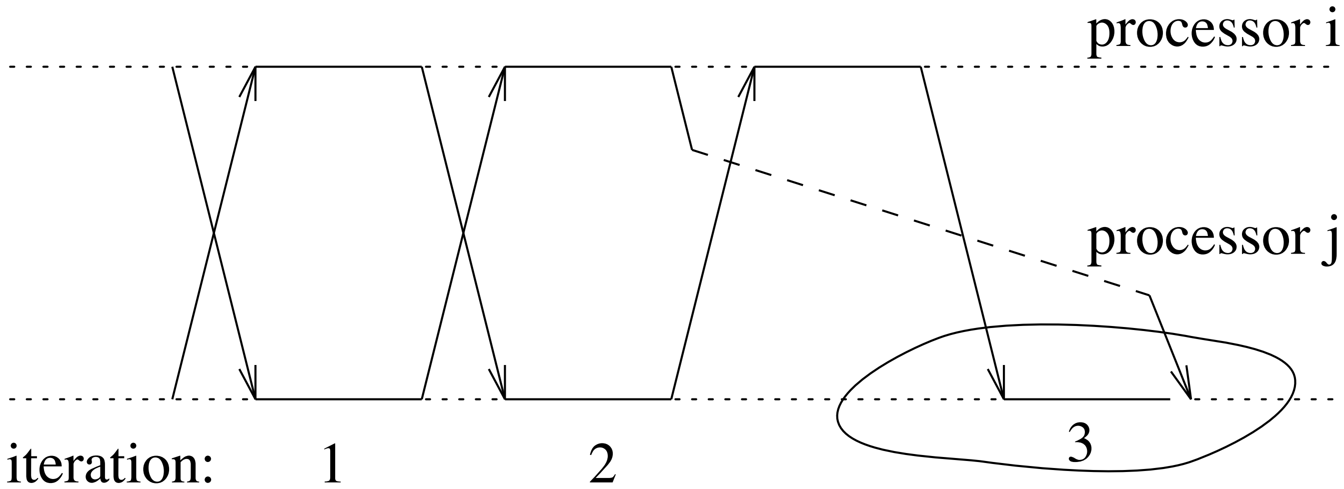
- When Blocks

  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);`

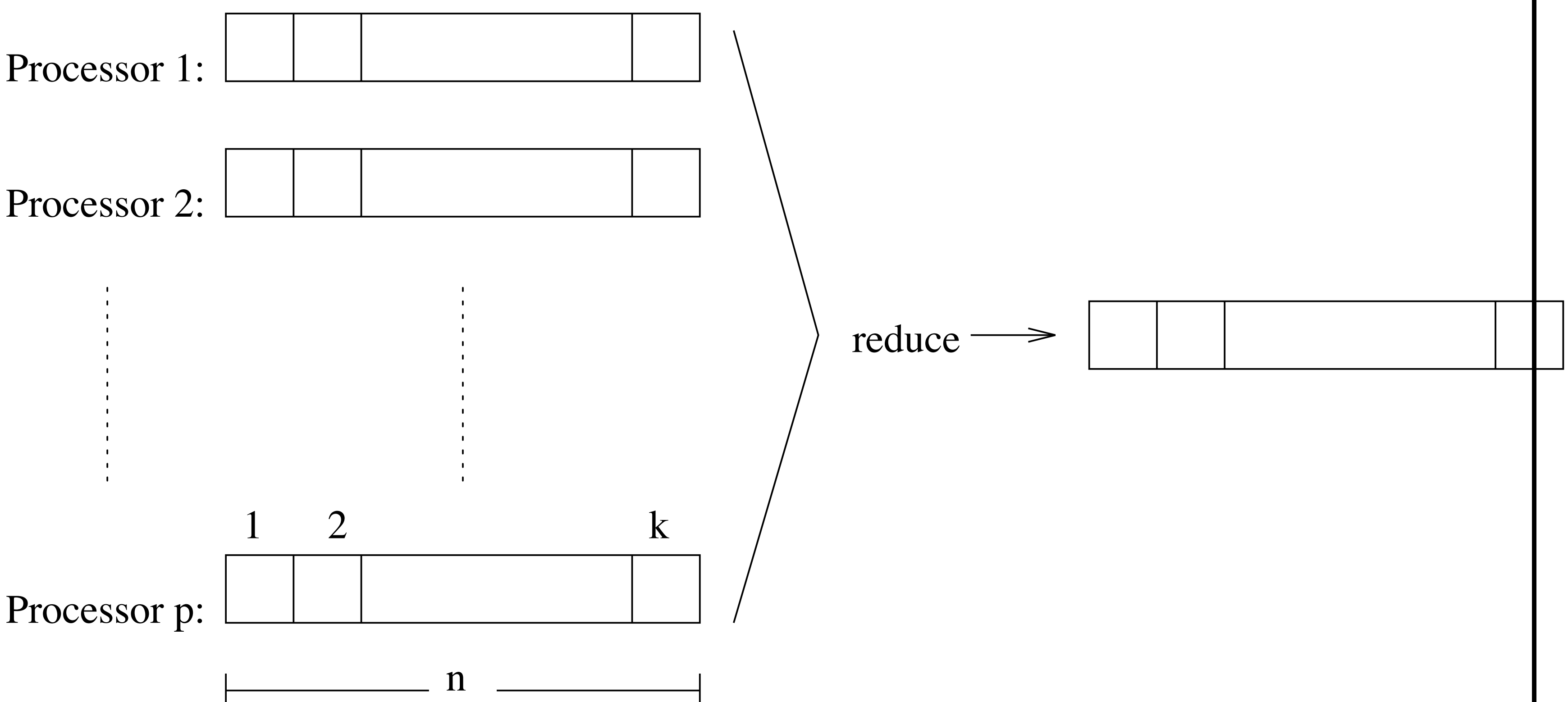
- `GetRefNumber(msg);`

Correct Loop Program

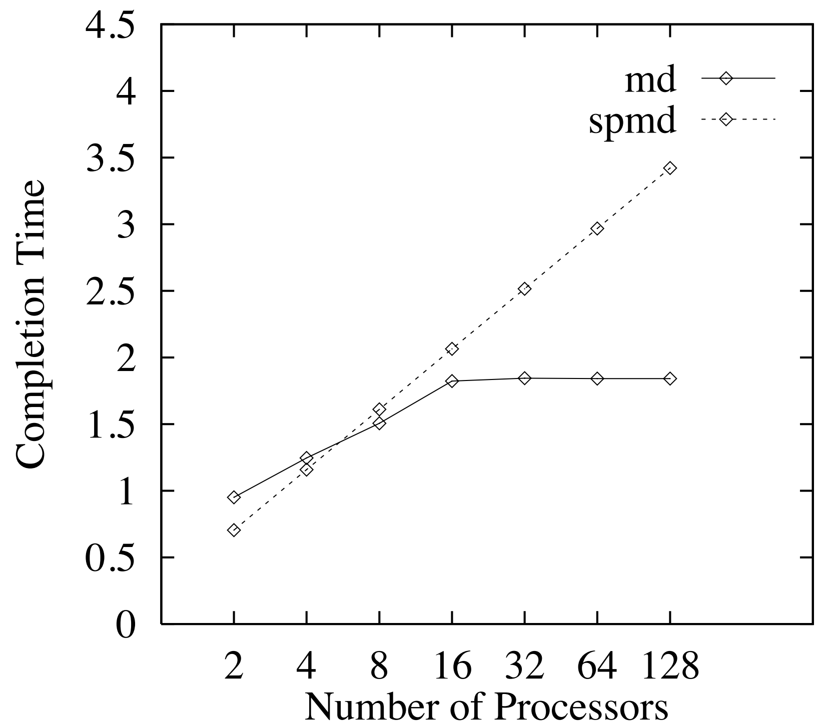
```
when north,south,east,west :  {
   update(n,s,e,w);
   iteration_count = iteration_count + 1;
   if (iteration_count < ITERATION_LIMIT) {
         send_boundaries(iteration_count);
         expect(north,iteration_count);
         expect(south,iteration_count);
         expect(east,iteration_count);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

Performance Results

Concurrent Reductions



Performance Results cont.



Concurrent Reductions on NCUBE/2,

problem size per processor = 4096 words, number of segments = 8

Summary:

- Message driven executin 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
    $\rightarrow$ simpler flow for a restricted but common case