Outline

- Parallel Computers & Computing
- Designing Parallel Algorithms
- Understanding Performance
- Parallel Programming Tools
- **Application Examples**
- Summary and Conclusions

Tool Descriptions and Application Examples

- FM: Ring pipeline
  - Functional decomposition
- CC++: Hartree-Fock quantum chemistry
  - Access to distributed data structures
- MPI: Diverse examples
  - Point-to-point and collective communication
- HPF: Linear algebra
  - Choice of data distribution

Fortran M

- A small set of extensions to Fortran 77 supporting explicit parallelism
- Extensions are modeled on Fortran concepts
  - Process = subroutine
  - Communication = file I/O
  - Resource management = processor array
- Extensions are easy to learn and use
- Public-domain compiler

FM Language Features

- Concurrency and communication
  - Processes, ports, and channels
- Resource management
  - Virtual computers and process placement
- Determinism and nondeterminism
  - Explicitly nondeterministic constructs
- Argument passing

FM Keywords

```
PROCESS                 SEND
PROCESS COMMON          RECEIVE
                        ENDCHANNEL
INPORT                  PROBE
OUTPORT                 MOVEPORT
CHANNEL
MERGER                  INTENT

PROCESSES               PROCESSORS
ENDPROCESSES            LOCATION
PROCESSDO               SUBMACHINE
PROCESSCALL
```

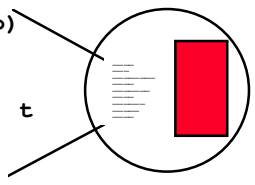
Building Block: The Process

- Encapsulates code & data
- Can encapsulate concurrency & communication
- Ports define interface to other processes

INPORT (*data-type-list*) *name-list*

OUTPORT (*data-type-list*) *name-list*

```
PROCESS atmosphere(inp, outp)
INPORT  (. . .) inp
OUTPORT (. . .) outp
PROCESS COMMON /state/ u, v, t
. . .
end
```



Concurrency

- **Parallel blocks** used to represent concurrency
  - Use for concurrency and/or latency hiding
  - Simplifies reasoning about programs
- Semantics are independent of the execution environment (e.g., number of processors)
  - Concurrent execution is *fair* and *interleaved*
  - Block terminates when its statements terminate

Parallel Blocks

- Process block: a set of processes + at most one subroutine
- Process do loop: enumerated set of processes

Process Block :

```
PROCESSES
  PROCESSCALL process1
  ...
  PROCESSCALL processN
ENDPROCESSES
```

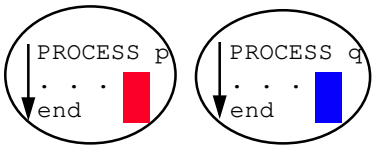
Process Do Loop :

```
PROCESSDO i = 1,n
  PROCESSCALL process
ENDPROCESSDO
```

Process Creation

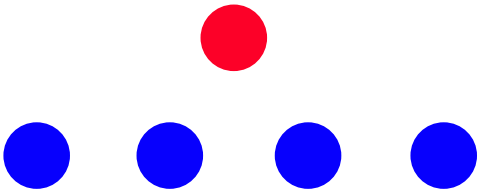
- A parallel block suspends until offspring processes complete execution
- Offspring & parent do not share data

```
program main
...
PROCESSES
   PROCESSCALL p
   PROCESSCALL q
ENDPROCESSES
...
end
```



Process Creation Example

```
PROCESSES
   PROCESSCALL master
   PROCESSDO i = 1,4
      PROCESSCALL worker(i)
   ENDPROCESSDO
ENDPROCESSES
```



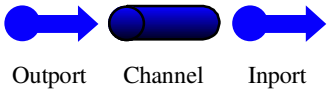
Communication: Port Variables

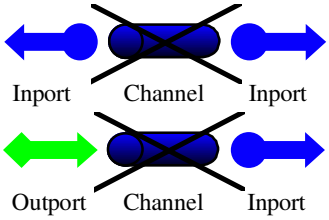
- Ports are unidirectional and typed

```
OUTPORT (integer, real) source
INPORT  (integer, real) sink
```

- Ports are connected by channels

```
CHANNEL(OUT=source, IN=sink)
```





Operations on Ports

- SEND and RECEIVE statements add data to and take data out of a channel

```
SEND (PORT=source) i, x
RECEIVE (PORT=sink) j, y
```

- Attempting to receive from an empty channel causes the statement to block

- ENDCHANNEL terminates channel

Communication Operations

`SEND(out-port) a1,...,ak`

- Send (a1, . . ., ak) on `out-port`
- Asynchronous (non-blocking)

`ENDCHANNEL(out-port)`

- Send end-of-sequence (EOS) on `out-port`

`RECEIVE(in-port[,END=label]) a1,...,ak`

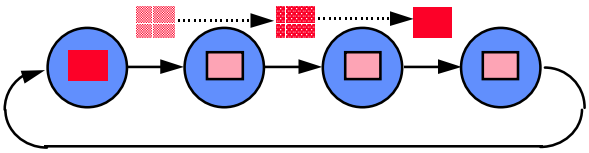
- Receive into (a1, . . ., ak) from `in-port`
- Synchronous (blocking)
- Branch to label if EOS

Example: Pairwise Interactions

- Compute interactions between N objects

$$F_i = \sum_{j=0}^{N-1} I_{ij}$$

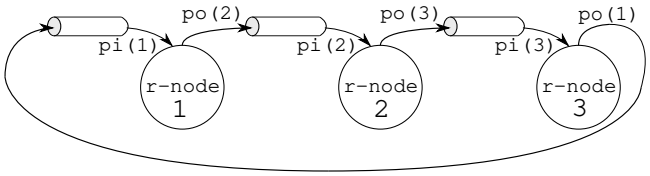
- *N* tasks, one per object, connected in a ring
- Objects circulate, tasks compute interactions



DBPP215

FM Pairwise Interactions: Task & Channel Setup

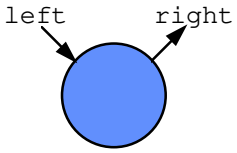
```
program ring
inport (real x(3)) pi(P)
outport (real x(3)) po(P)
channel(in=pi(1:P-1),out=po(2:P))
channel(in=pi(P),out=po(1))
processdo i=1,P
   processcall rnode(i,P,pi(i),po(i))
endprocessdo
end
```



DBPP215

FM Pairwise Interactions: Task Definition

```
process rnode(me,np,left,right)
inport  (real x(3)) left
outport (real x(3)) right
real state(3), forces(3), msg(3)
call initstate(me, state)
call copy(state, msg)
call zero(forces)
do i=1,np-1
   send(right) msg
   receive(left) msg
   call updateforces(msg, state, forces)
enddo
end
```



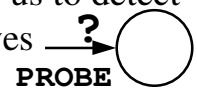
DBPP225

Determinism

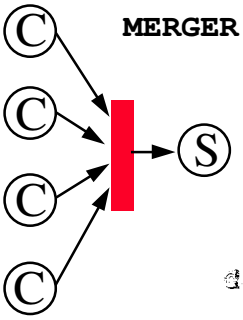
- Guaranteed w/o nondeterministic constructs
- At most one sender & one receiver/channel
  - This is true when a channel is created
  - No assignment to port variables
  - No passing port variables to multiple processes
  - Port variable invalidated when communicated
- Two processes cannot modify same variable
  - Ensured even when argument passing is used

Nondeterminism

- Nondeterminism is introduced by using special statements
  - `PROBE` allows us to detect pending receives



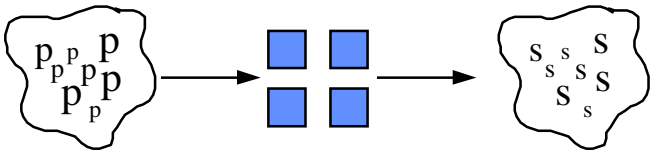
  - A `MERGER` allows us to define many-to-one communication structures



DBPP218

Example: Parameter Study

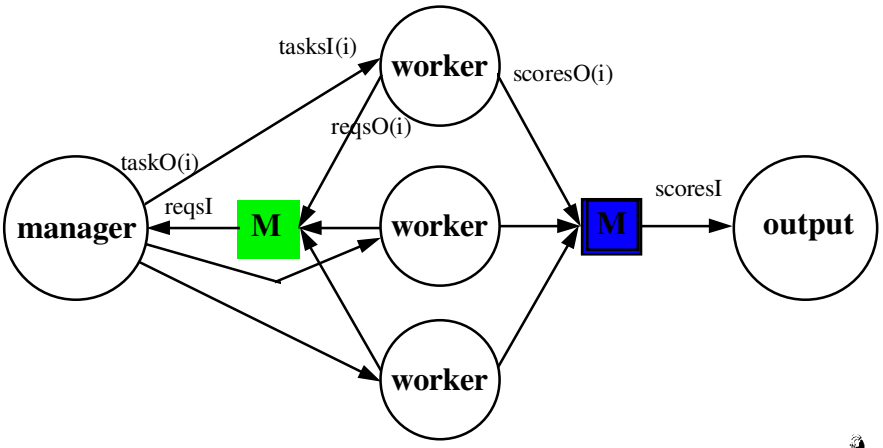
- Read multiple "problem specifications" from file, and solve; write results to a file
- Problem solution time is problem dependent
- Many more problems than processors



Parameter Study

- Use manager/worker structure
  - Single manager reads problems from disk
  - Multiple workers repeatedly request and solve problems
  - Single outputer writes results to disk

Manager-Worker Structure



Manager-Worker Structure

```
INPORT (real) taskI(NW)      }
OUTPORT(real) taskO(NW)      }   Channels used to communicate tasks
                                 from manager to workers
INPORT (integer) reqI        }
OUTPORT(integer) reqO(NW)    }   Merger used to communicate requests
                                 from workers to manager
INPORT (real) scoreI         }
OUTPORT(real) scoreO(NW)     }   Merger used to communicate output
                                 from workers tooutput

CHANNEL(in=taskI(:), out=taskO(:))
MERGER(in=reqI, out=reqO(:))
MERGER(in=scoreI, out=scoreO(:))
PROCESSES
   call manager(reqI, taskO)
   PROCESSDO i=1,NW
      PROCESSCALL worker(i, reqO(i), taskI(i), scoreO(i))
   ENDPROCESSDO
   PROCESSCALL output(scoreI)
ENDPROCESSES
```

DBPP220

Worker Process

```
      PROCESS worker(myid, reqs, tasks, score)
      OUTPORT(integer) reqs
      INPORT (real) tasks
      OUTPORT(real) score

      best = 0.0
      do while(.true.)
         SEND(reqs) myid
         RECEIVE(tasks, end=10) task
         val = compute(task)
         if(val .gt. best) best = val
      enddo
   10 continue
      SEND(score) best
      end
```

DBPP220

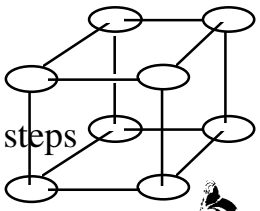
Manager Process

```
PROCESS manager(reqs, tasks)
INPORT (integer) reqs
OUTPORT(real) tasks(NW)

do i = 1,NTASKS
   RECEIVE(reqs) id
   SEND(tasks(id)) nexttask(i)
enddo
do i = 1,NTASKS
   ENDCHANNEL(tasks(i))
enddo
end
```

Example: Reduction using Butterfly

- Goal: Reduce P numbers, distributed one per process, and distribute reduced value to each process
- Algorithm: Connect processes in butterfly (hypercube) structure, and:
  - initialize reduced value
  - perform log$P$ send-receive-reduce steps

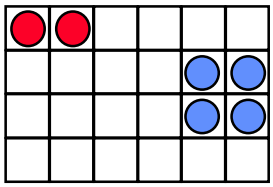


Reduction

```
INPORT (real) inh(logP, P)
OUTPORT(real) outh(logP, P)
do i=1,P
   do j=1,logP
      nbor = 1 + xor(i-1, 2**(j-1))
      CHANNEL(in=inh(j,i), out=outh(j,nbor))
   enddo
enddo
PROCESSDO i=1,P
   PROCESSCALL reducenode(inh(1,i), outh(1,i))
ENDPROCESSDO
END
```

```
PROCESS reducenode(inh, outh)
INPORT (real) inh(logP)
OUTPORT(real) outh(logP)
work = localvalue()
do i=1,logP
   SEND(outh(i)) work
   RECEIVE(inh(i)) newval
   work = reduce(work,newval)
enddo
```

Resource Management



- Control resource allocation by associating processes with processors.
- Mapping constructs specify location within an N-dimensional processor array.
- Mapping effects performance only, not semantics

Mapping Constructs

`PROCESSORS(d1,...,dk)`

- Declaration: Size of a processor array.
- Analagous to an array `DIMENSION` statement.

`processcall p(...)LOCATION(i1,...,im)`

- Annotation: Specifies location within processor array
- Analagous to array access `[x = a(i,j)]`

Note use of F90 array section

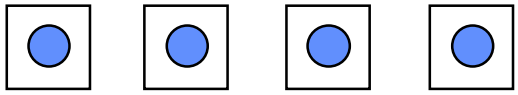
`processcall q(...)SUBMACHINE(m:n,...,r:s)`

- Annotation: Specifies processor array section.
- [C.f. `call p(a(m:n,...,r:s))`]

Finite Difference Mapping: I

- Map processes to an NP-processor array.

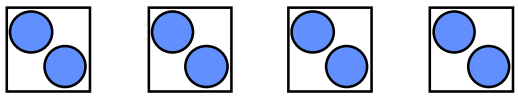
```
PROCESSORS(NP)
.
.
PROCESSDO i = 1,NP
    PROCESSCALL subdomain(Li(i), Lo(i), Ri(i), Ro(i))
$                          LOCATION(i)
ENDPROCESSDO
```



Finite Difference Mapping: II

- Use a cylic mapping for load balancing

```
PROCESSORS(NP/2)
.
.
PROCESSDO i = 1,NP
    processcall subdomain(Li(i), Lo(i), Ri(i), Ro(i))
$                          LOCATION(mod(i,NP/2)+1)
ENDPROCESSDO
```



FM Summary

- A small set of extensions to Fortran supporting explicit parallelism
- Tasks for locality and modularity
- Parallel blocks for concurrency
- Channels for communication

Notes