

Parallel programming paradigms and their performances

A high level exploration of the HPC world

George Bosilca
University of Tennessee, Knoxville
Innovative Computer Laboratory

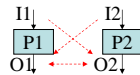
Overview

- Definition of parallel application
- Architectures taxonomy
- Laws managing the parallel domain
- Models in parallel computation
- Examples

Formal definition

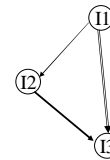
Bernstein
 $\{ I1 \cap O2 = \emptyset \text{ and } I2 \cap O1 = \emptyset \text{ and } O1 \cap O2 = \emptyset \}$
 General case: $P1 \dots Pn$ are parallel if and only if each for each pair Pi, Pj we have $Pi \parallel Pj$.

- 3 limit to the parallel applications:
1. Data dependencies
 2. Flow dependencies
 3. Resources dependencies



Data dependencies

I1: $A = B + C$
 I2: $E = D + A$
 I3: $A = F + G$

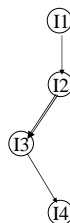


- Dataflow dependency
- Anti-dependency
- Output dependency

How to avoid them?
Which can be avoided ?

Flow dependencies

I1: $A = B + C$
 I2: $\text{if}(A) \{$
 I3: $D = E + F \}$
 I4: $G = D + H$

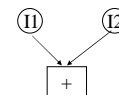


- Dataflow dependency
- Flow dependency

How to avoid ?

Resources dependencies

I1: $A = B + C$
 I2: $G = D + H$



How to avoid ?

Flynn Taxonomy

- Computers classified by instruction delivery mechanism and data stream
- 4 characters code: 2 for instruction stream and 2 for data stream

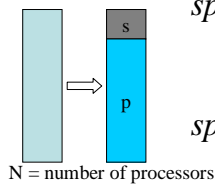
	1 Instruction flow	> 1 Instruction flow
1 data stream	SISD Von Neumann	MISD pipeline
> 1 data stream	SIMD	MIMD

Flynn Taxonomy: Analogy

- SISD: lost people in the desert
- SIMD: rowing
- MISD: pipeline in the car construction chain
- MIMD: airport facility, several desks working at their own pace, synchronizing via a central database.

Amdahl Law

- First law of parallel applications (1967)
- Limit the speedup for all parallel applications



$$speedup = \frac{s + p}{s + \frac{p}{N}}$$

$$speedup = \frac{1}{a + \frac{(1-a)}{N}}$$

Amdahl Law

Speedup is bound by 1/a.

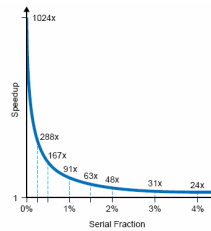


FIGURE 1. Speedup under Amdahl's Law

Amdahl Law

- Bad news for parallel applications
- 2 interesting facts:
 - We should limit the sequential part
 - A parallel computer should be a fast sequential computer to be able to resolve the sequential part quickly
- What about increasing the size of the initial problem ?

Gustafson Law

- Less constraints than the Amdahl law.
- In a parallel program the quantity of data to be processed increase, so the sequential part decrease.

$$\left. \begin{array}{l} t = s + P/n \\ P = a * n \end{array} \right\} speedup = \frac{s + a * n}{s + a}$$

$$a \rightarrow \infty \Rightarrow speedup \rightarrow n$$

Gustafson Law

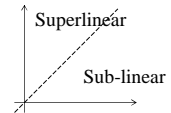
- The limit of Amdahl Law can be transgressed **if** the quantity of data to be processed increase.

$$speedup \leq n + (1 - n)s$$

Rule stating that if the size of most problems is scaled up sufficiently, then any required efficiency can be achieved on any number of processors.

Speedup

- Superlinear speedup ?



Sometimes superlinear speedups can be observed!

- Memory/cache effects
 - More processors typically also provide more memory/cache.
 - Total computation time decreases due to more page/cache hits.
- Search anomalies
 - Parallel search algorithms.
 - Decomposition of search range and/or multiple search strategies.
 - One task may be "lucky" to find result early.

Parallel execution models

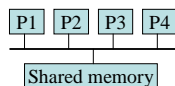
- Amdahl and Gustafson laws define the limits without taking in account the properties of the computer architecture.
- They cannot be used to predict the real performance of any parallel application.
- We should integrate in the same model the architecture of the computer and the architecture of the application.

What are models good for ?

- Abstracting the computer properties
 - Making programming simple
 - Making programs portable ?
- Reflecting essential properties
 - Functionality
 - Costs
- What is the von-Neumann model for parallel architectures ?

Parallel Random Access Machine

- One of the most studied
- World described as a collection of synchronous processors which communicate with a global shared memory unit.



How to represent the architecture

- 2 resources have a major impact on the performances:
 - The couple (processor, memory)
 - The communication network.
- The application should be described using those 2 resources.

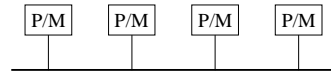
$$T_{app} = T_{comp} + T_{comm}$$

Models

- 2 models are often used.
- They represent the whole system as composed by n identical processors, each of them having his own memory.
- They are interconnected with a predictable network.
- They can realize synchronizations.

Bulk Synchronous Parallel – BSP

- Distributed-memory parallel computer Valiant 1990
- Global vision as a number of processor/memory pairs interconnected by a communication network

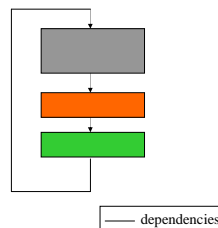


- Each processor can access his own memory without overhead and have a uniform slow access to remote memory

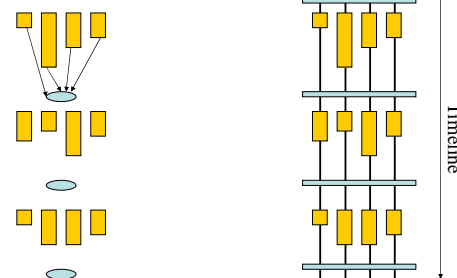
BSP

- Applications composed by Supersteps separated by global synchronizations.
- One superstep include:
 - A computation step
 - A communication step
 - A synchronization step

Synchronization used to insure that all processors complete the computation + communication steps in the same amount of time.



BSP



BSP

$$T_{\text{superstep}} = w + g * h + l$$

Where:

w = max of computation time

g = 1/(network bandwidth)

h = max of number of messages

l = time for the synchronization

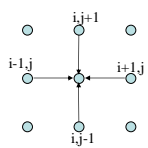
Sketch the communications

BSP

- An algorithm can be described using only **w**, **h** and the *problem size*.
- Collections of algorithms are available depending on the computer characteristics.
 - Small L
 - Small g
- The best algorithm can be selected depending on the computer properties.

BSP - example

- Numerical solution to Laplace's equation

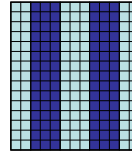
$$U_{i,j}^{n+1} = \frac{1}{4}(U_{i-1,j}^n + U_{i+1,j}^n + U_{i,j-1}^n + U_{i,j+1}^n)$$


```

for j = 1 to jmax
  for i = 1 to imax
    Unew(i,j) = 0.25 * ( U(i-1,j) + U(i+1,j)
                      + U(i,j-1) + U(i,j+1))
  end for
end for
    
```

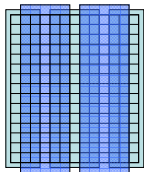
BSP - example

- The approach to make it parallel is by partitioning the data



BSP - example

- The approach to make it parallel is by partitioning the data



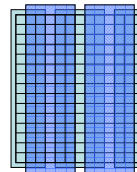
Overlapping the data boundaries allow computation without communication for each superstep

On the communication step each processor update the corresponding columns on the remote processors.

BSP - example

```

for j = 1 to jmax
  for i = 1 to imax
    unew(i,j) = 0.25 * ( U(i-1,j) + U(i+1,j)
                      + U(i,j-1) + U(i,j+1))
  end for
end for
    
```



```

if me not 0 then
  bsp_put( to the left )
endif
if me not NPROCS - 1 then
  bsp_put( to the right )
endif
bsp_sync()
    
```

BSP - example

$$T_{\text{superstep}} = w + g * h + l$$

h = max number of messages
 = l values to the left +
 l values to the right
 = 2 * l (ignoring the inverse communication!)

$$w = 4 * l * l / p^2$$

$$T_{\text{superstep}} = 4 \frac{l^2}{p} + 2 * g * l + l$$

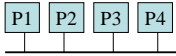
BSP - example

- BSP parameters for a wide variety of architectures has been published.

Machine	s	p	l	g
Origin 2000	101	4	1789	10.24
		32	39057	66.7
Cray T3E	46.7	4	357	1.77
		16	751	1.66
Pentium 10Mbit	61	4	139981	1128.5
		8	826054	2436.3
Pentium II 100Mbit	88	4	27583	39.6
		8	38788	38.7

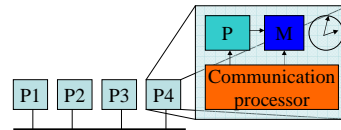
A more sophisticated model LogP

- Tend to be more empirical and network-related.



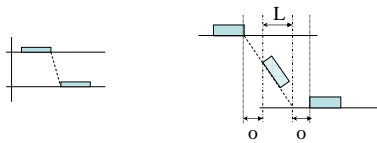
A more sophisticated model LogP

- Tend to be more empirical and network-related.



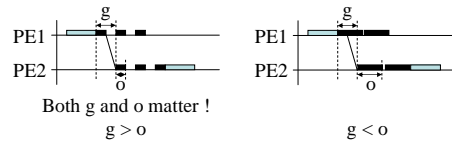
LogP

- Decompose the communications in 3 elements:
 - Latency : small message cross the network
 - overhead : lost time in communication



LogP

- Decompose the communications in 3 elements:
 - Latency : small message cross the network
 - overhead : lost time in communication
 - gap : between 2 consecutive messages
- And P the number of processors.



LogP

- The total time for a message to go from the processor A to the processor B is:
 $L + 2 * o$
- There is no model for the application
- We can describe the application using the same approach as for BSP: supersteps

$$T_{\text{superstep}} = w + h * (L + 2o) + l$$

LogP

- The P parameter does not interfere in the superstep computation ?
- When the number of processors is not fixed:
 - The time of the computation change $w(p)$
 - The number of messages change $h(p)$
 - The synchronization time change $l(p)$

LogP

- Allow/encourage the usage of general techniques of designing algorithms for distributed memory machines: exploiting locality, reducing communication complexity and overlapping communication and computation.
- Balanced communication to avoid overloading the processors.

LogP

- Interesting concept : idea of finite capacity of the network. Any attempt to transit more than a certain amount of data will stall the processor.
- This model does not address the issue of message size, even the worst is the assumption of all messages are of "small" size.
- Does not address the global capacity of the network.

Design a LogP program

- Execution time is the time of the **slowest** process
- Implications for algorithms:
 - Balance computation
 - Balance communications
 Are only sub-goals !
- Remember the capacity constraint $\left\lceil \frac{L}{g} \right\rceil$

LogP Machines

Maschine	L	o	g	P
CM-5	6	2.2	4	512
Meiko CS-2	8.6	1.7	$14.2 + 0.03x$	64
Power Explorer	$21 - 0.82x$	$70 + x$	$115 + 1.43x$	8
Para-Station	$50 - 0.10x$	$3 + 0.112x$	$3 + 0.119x$	4
IBM SP-2	$13 - 0.005x$	$8 + 0.008x$	$10 + 0.01x$	128
IBM SP-2	$17 - 0.005x$	$8 + 0.008x$	$10 + 0.01x$	256

Improving LogP

- First model to break the synchrony of parallel execution
- LogGP : augments the LogP model with a linear model for long messages
- LogGPC model extends the LogGP model to include contention analysis using queuing model on the k -ary n -cubes network
- LogPQ model augments the LogP model on the stalling issue of the network constraint by adding buffer queues in the communication lines.

The CCM model

- Collective Computing Model transform the BSP superstep framework to support high-level programming models as MPI and PVM.
- Remove the requirement of global synchronization between supersteps, but combines the message exchanges and synchronization properties into the execution of a collective communication.
- Prediction quality usually high.

How to predict the performances ?

George Bosilca

How to predict the performances?

- Using one of the models ...
- Lack of parameters to represent the whole architecture
- Parallel Architecture = Computer Architecture + Communication architecture

Toward improving performance

- First questions : did I choose the right programming model ? Did it match the target architecture ?
- Improving the code of each parallel unit
- Decreasing the number of communications and/or their cost
- Decreasing the cost of management

What kind of parallel architecture

- Shared memory or distributed memory
- What kind of network ? Which topology ?
- What tools can we use ?
- How the user level programs interact with the hardware ?

Asynchronous vs. synchronous

- | | |
|---|--|
| <ul style="list-style-type: none">• Allow overlapping communication computation• Hiding latencies• Additional cost for management | <ul style="list-style-type: none">• No overlapping• No additional cost• All latencies included in the final time |
|---|--|

Architectures

- Vector architecture
- Multi flow architectures
- Shared memory
- Distributed memory

Vector architecture

- Specialized on computation on arrays
- One instruction can be applied to several data from the same arrays (loops)
- Load/Store through vector registers

```
For i = 0 to 64 do
  a[i] = c[i] + d[i]
  b[i] = a[i] * f[i]
```

Correct sequential semantic: a[0], b[0], a[1], b[1], ...

Vector architecture

- Specialized on computation on arrays
- One instruction can be applied to several data from the same arrays (loops)
- Load/Store through vector registers

```
For i = 0 to 64 do
  a[i] = c[i] + d[i]
  b[i] = a[i] * f[i]
```



```
For i = 0 to 64 do
  a[i] = c[i] + d[i]
For I = 0 to 64 do
  b[i] = a[i] * f[i]
```

a[0], a[1], a[2], ..., a[63], b[0], b[1], b[64]

Vector architecture

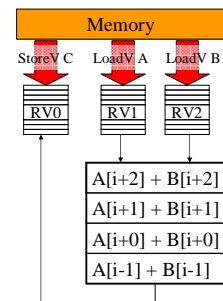
- Vector operation = pipeline
- Operation applied directly on the *vector registers*
- Instruction with strong semantics: one instruction applied on the whole vector register

Vector architecture - example

```
For i = 0 to 64 do
  c[i] = a[i] + b[i]
```

```
LoadV A, RV1
LoadV B, RV2
AddV RV1, RV2, RV0
StoreV C, RV0
```

Cost:
 -sequential 64 * 4 cycles
 -Vector 63 + 4 cycles



Vector architecture - example

- Total cost of the vectorized loop:

$$\begin{aligned} &T_{\text{init}} + 63 \quad (\text{LoadV A} \parallel \text{LoadV B}) \\ &+ 4 + 63 \quad (\text{AddV}) \\ &+ T_{\text{init}} + 63 \quad (\text{StoreV}) \\ &= 2 * T_{\text{init}} + 193 \text{ ticks} \end{aligned}$$

- Chaining = linking the pipelines together

$$T_{\text{init}} + 4 + T_{\text{init}} + 63 = 2 * T_{\text{init}} + 67$$

- How about the memory access ?

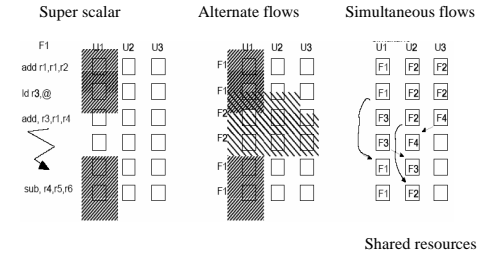
Vector architecture

- Several bancs to sustain the high bandwidth
- Components "state of the art" from the technology point of view
- First vector processor: Cray1 (12 vector units + chain MAC)
- Vector multiprocessor: CrayT90 32 procs (1024 memory bancs)
- Vendors:SGI/Cray, Fujitsu, NEC, Hitachi

Multiflow architecture

- Hyper-Threading it's a new idea ?
- Basic idea: **do something else while waiting for memory latency** or **how to deal with cache misses and data dependencies**
- When to switch ?
 - On every load operation
 - On cache miss
 - On every instruction (no cache locality)
 - On instruction block
- How to switch ?
 - Context switch too expensive : thread approach

Multi flow architectures

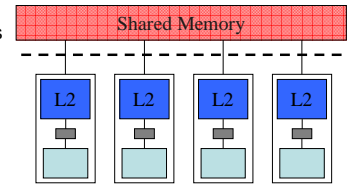


Multiflow architecture

- TERA **M**ulti**T**hreaded **A**rchitecture
 - Heavily alternate multi threaded
 - No caches (direct access to the memory)
 - Change the flow after each load
 - One memory access ~ 100 cycles
 - 16 protection domains (register, status, CP) sharing 128 threads **by processor** ...
- Up to 256 processors !!!

Shared Memory

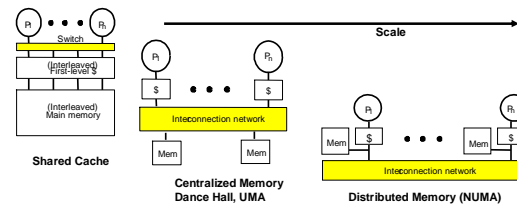
- Each processor have his own cache (one or several levels)
- They can access the whole shared memory
- How about consistency ? How can a data be on several processors in same time.



Shared memory

- Allow fine grain resources sharing
- Communications are implicit in load/store on shared addresses
- Synchronization is performed by operations on shared addresses

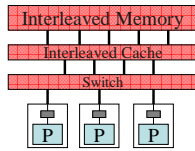
Shared memory - ShMem



- Uniform Memory Access
- Non Uniform Memory Access
- Cache Coherent – Non Uniform Memory Access
- Cache Only Memory Access

ShMem – Shared Cache

- Alliant FX-8 (8x68020 512KB); Encore & Sequent (2xN32032)
- Advantages
 - Identical to uni processor systems
 - Only one copy of any cached block
 - Smaller storage size
 - Fine-grain sharing
 - Potential for positive interference
 - One proc prefetch data for another
 - Can share data within a line without "ping-pong"
 - No false sharing for long data



ShMem – Shared Cache

- Drawbacks
 - Sharing cache bandwidth between processors
 - Increase latencies for ALL accesses
 - Potential for negative interference
 - One proc flush data for another

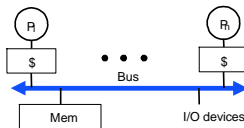
Many L2 caches are shared today

ShMem – Bus based approach

- Cheap, usual components => dominate the market
- Attractive as servers and convenience parallel computers
 - Fine grain resource sharing
 - Uniform access using Load/Store
 - Automatic data movement and coherent cache replication
 - Cheap and powerful extension

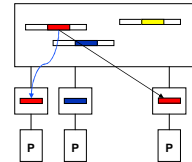
- **Sequential access**

Normal uni processor mechanism to access data

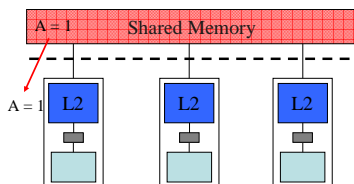


ShMem - caches

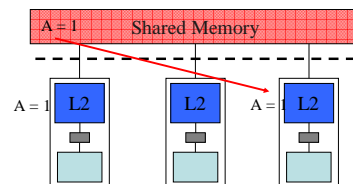
- Caches become critical
 - Reduce average latency (replication closer to proc)
 - Reduce average bandwidth
 - Manage consistency
- Data goes from producer to consumer to memory
- Many processors can share the memory efficiently
- Concomitant read accesses to the same location



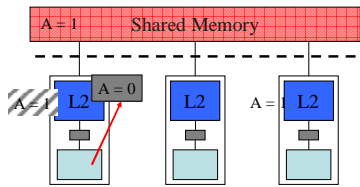
ShMem – cache coherence example



ShMem – cache coherence example

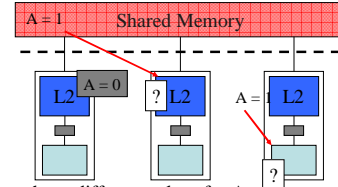


ShMem – cache coherence example



Processors have different values for A

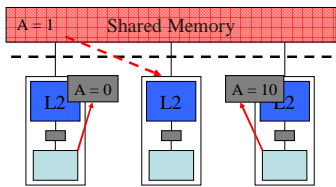
ShMem – cache coherence example



Processors have different values for A
Write back caches depend on the happenstance of which caches flushes (?)

Intolerable from the programmer point of view

ShMeme – cache coherence example2



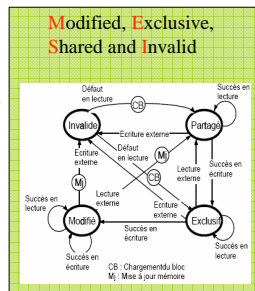
Still intolerable

ShMem – Caches and coherence

- Caches play an important role in all cases
 - Reduce average access time
 - Reduce bandwidth on shared interconnection
- Private caches create a coherence problem
 - Copies of the same data on several caches
 - A write may not become visible to other procs
- Solutions
 - Another memory organization
 - Detect and take actions to avoid this problem

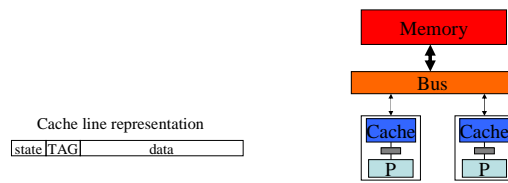
ShMem – Cache coherence protocols

- 2 main categories:
 - Invalidation
 - Any write preceded by a block invalidation for all others processors
 - Broadcast (diffusion)
 - Before any write all caches containing the same data will be invalidated



ShMem – Snoop protocols

- Snooping (or monitoring) the bus
- set of states
- state-transition diagram
- actions



Ordering (memory consistency)

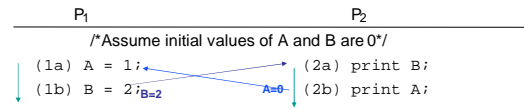
P ₁	P ₂
/*Assume initial values of A and B are 0*/	
(1a) A = 1;	(2a) print B;
(1b) B = 2;	(2b) print A;

- What's the intuition?
- Whatever it is, we need an ordering model for clear semantics
 - across different locations as well
 - so programmers can reason about what results are possible

Lampert give the definition of a multi processor sequentially consistent:

- The result of all executions is the same as the sequential atomic execution of each instruction
- The operations of each processor appear in the sequential order as specified by the program.

Ordering (memory consistency)



- What matters is order in which operations *appear to execute*, not the chronological order of events
- Possible outcomes for (A,B): (0,0), (1,0), (1,2)
- What about (0,2) ?
 - program order => 1a->1b and 2a->2b
 - A = 0 implies 2b->1a, which implies 2a->1b
 - B = 2 implies 1b->2a, which leads to a contradiction
- What is actual execution 1b->1a->2b->2a ?
 - appears just like 1a->1b->2a->2b as visible from results
 - actual execution 1b->2a->2b->1a is not

ShMem – Cache & Directories

- Centralized
 - Keep the state and the tag of each block of data for all caches
 - For each memory access the controller check the tag and state of all blocks
- Distributed
 - Each processor keep a directory for the data in his cache
 - Update this data depending on the information on the bus.
- Strongly depend on the interconnection network. (broadcast)

POSIX Threads & RPC: 2 parallel programming models

George Bosilca
bosilca@cs.utk.edu

Process vs. Thread

- A process is a collection of virtual memory space, code, data, and system resources.
- A thread (lightweight process) is code that is to be serially executed within a process.
- A process can have several threads.

Threads executing the same block of code maintain separate stacks. Each thread in a process shares that process's global variables and resources.

Possible to create more efficient applications ?

Process vs. Thread

- Multithreaded applications must avoid two threading problems: deadlocks and races.
- A deadlock occurs when each thread is waiting for the other to do something.
- A race condition occurs when one thread finishes before another on which it depends, causing the former to use a bogus value because the latter has not yet supplied a valid one.

The key is synchronization

- Synchronization = gaining access to a shared resource.
- Synchronization REQUIRE cooperation.

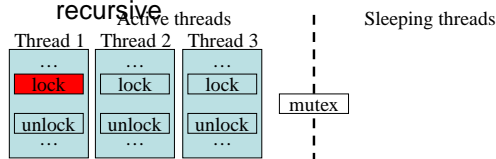
POSIX Thread

- What's POSIX ?
 - Widely used UNIX specification
 - Most of the UNIX flavor operating systems

POSIX is the Portable Operating System Interface, the open operating interface standard accepted world-wide. It is produced by IEEE and recognized by ISO and ANSI.

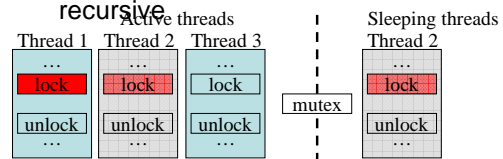
Mutual exclusion

- Simple lock primitive with 2 states: lock and unlock
- Only one thread can lock the mutex.
- Several politics: FIFO, random, recursive



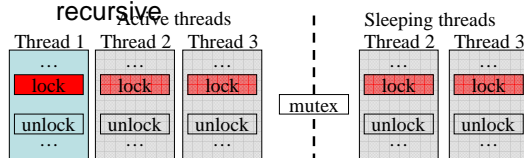
Mutual exclusion

- Simple lock primitive with 2 states: lock and unlock
- Only one thread can lock the mutex.
- Several politics: FIFO, random, recursive



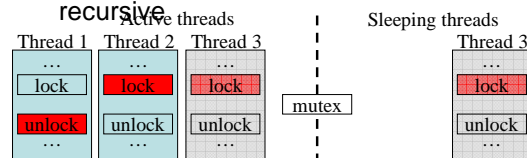
Mutual exclusion

- Simple lock primitive with 2 states: lock and unlock
- Only one thread can lock the mutex.
- Several politics: FIFO, random, recursive



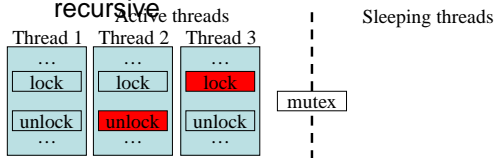
Mutual exclusion

- Simple lock primitive with 2 states: lock and unlock
- Only one thread can lock the mutex.
- Several politics: FIFO, random, recursive



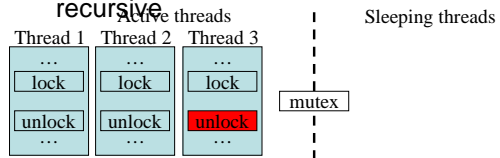
Mutual exclusion

- Simple lock primitive with 2 states: lock and unlock
- Only one thread can lock the mutex.
- Several politics: FIFO, random, recursive



Mutual exclusion

- Simple lock primitive with 2 states: lock and unlock
- Only one thread can lock the mutex.
- Several politics: FIFO, random, recursive



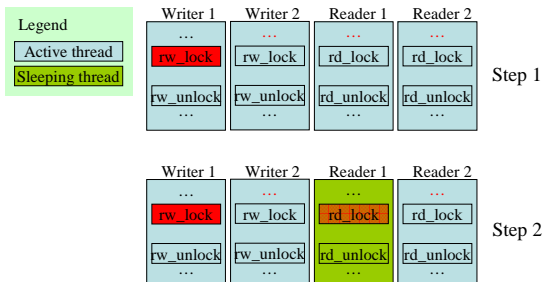
Mutual exclusion

- Spin vs. sleep ?
- What's the desired lock grain ?
 - Fine grain – spin mutex
 - Coarse grain – sleep mutex
- Spin mutex: use CPU cycles and increase the memory bandwidth, but when the mutex is unlock the thread continue his execution immediately.

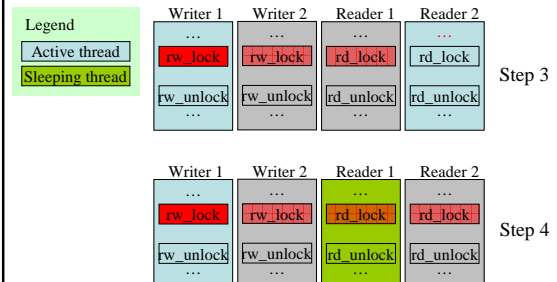
Shared/Exclusive Locks

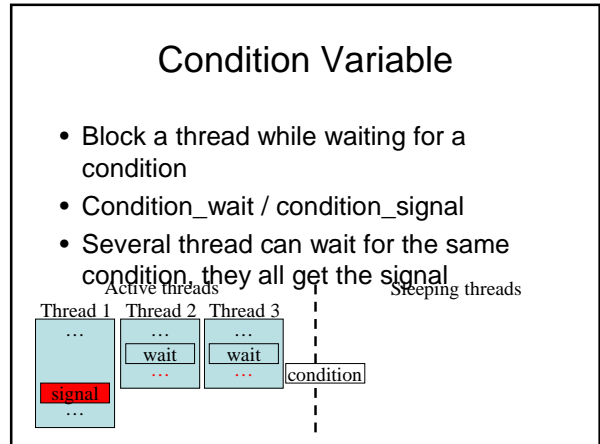
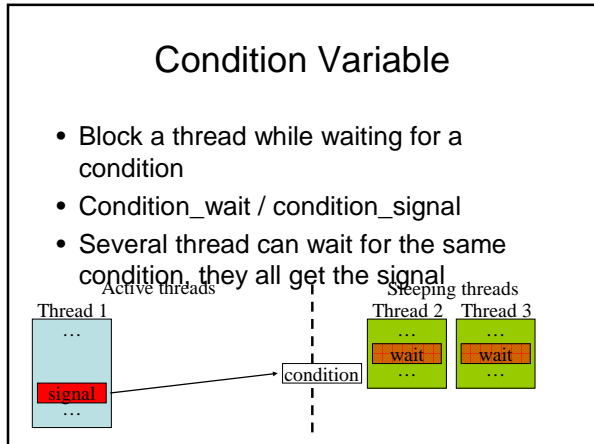
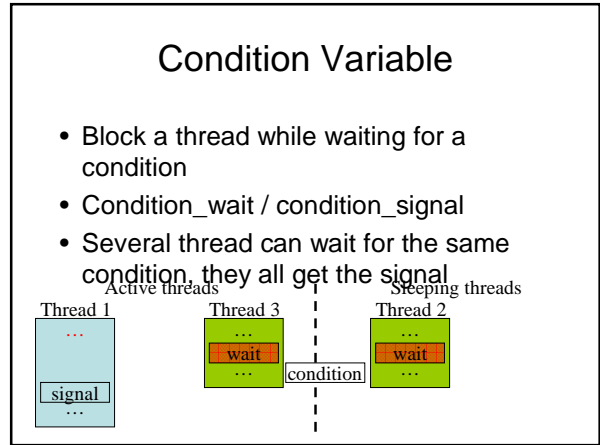
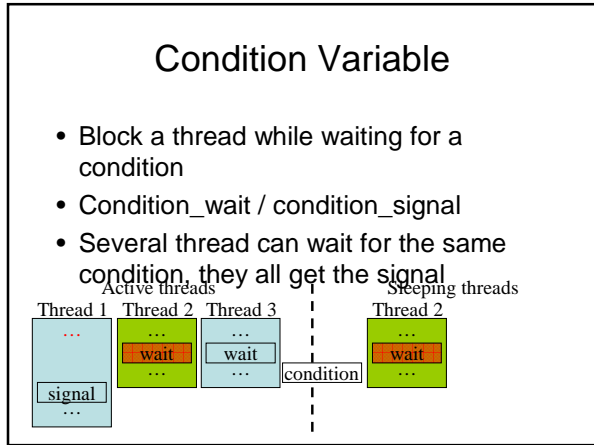
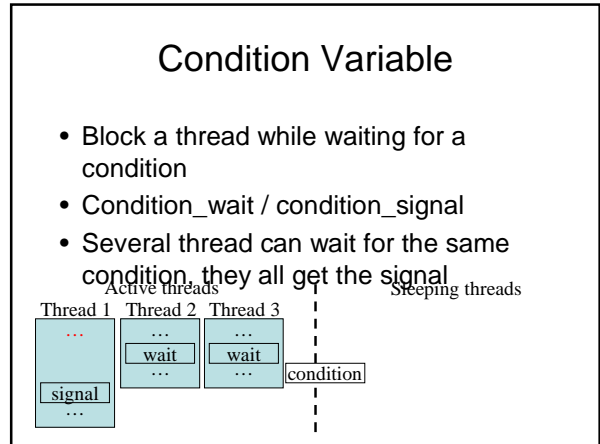
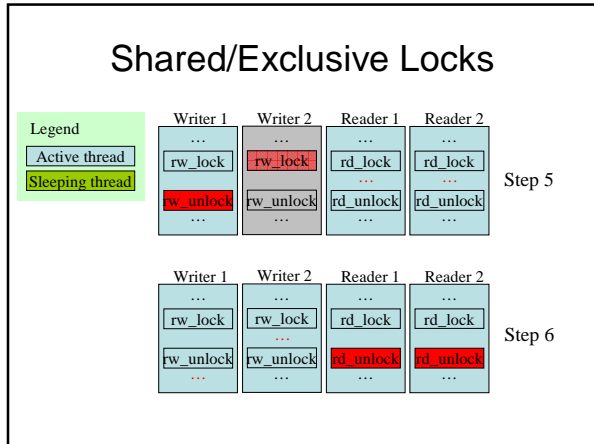
- **ReadWrite Mutual exclusion**
- Extension used by the reader/writer model
- 4 states: write_lock, write_unlock, read_lock and read_unlock.
- multiple threads may hold a shared lock simultaneously, but only one thread may hold an exclusive lock.
- if one thread holds an exclusive lock, no threads may hold a shared lock.

Shared/Exclusive Locks



Shared/Exclusive Locks

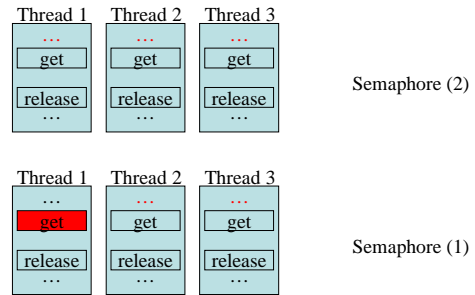




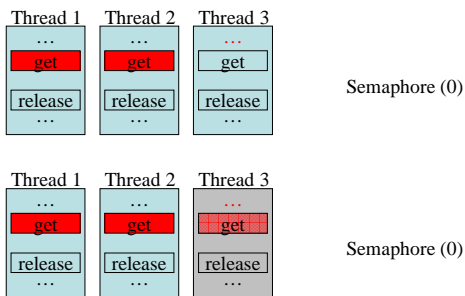
Semaphores

- simple counting mutexes
- The semaphore can be hold by as many threads as the initial value of the semaphore.
- When a thread get the semaphore it decrease the internal value by 1.
- When a thread release the semaphore it increase the internal value by 1.

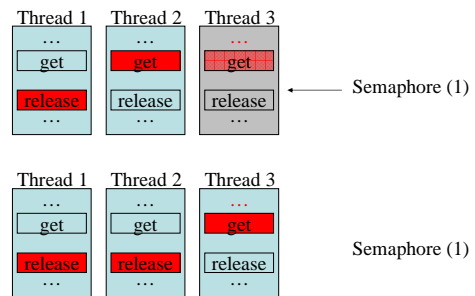
Semaphores



Semaphores



Semaphores

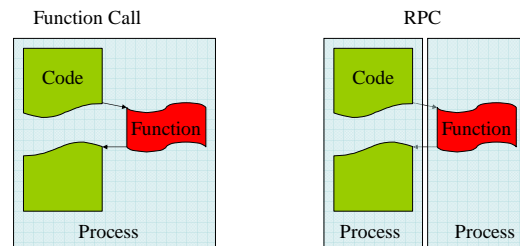


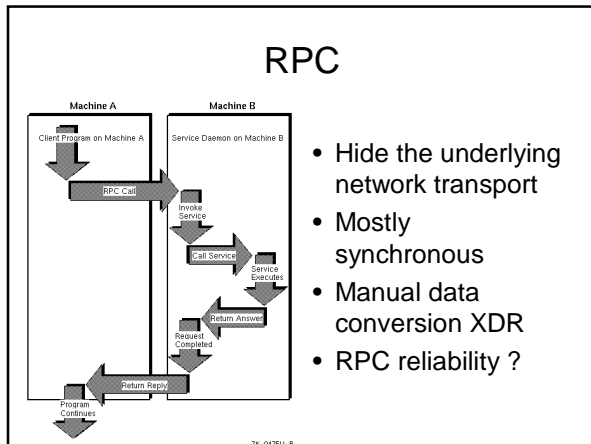
Atomic instruction

- Is any operation that a CPU can perform such that all results will be made visible to each CPU at the same time and whose operation is safe from interference by other CPUs
 - TestAndSet
 - CompareAndSwap
 - DoubleCompareAndSwap
 - Atomic increment
 - Atomic decrement

Remote Procedure Call

- Sun 1980 (?)





- ## RPC - reliability
- Again about reliability ...
 - If and only if the user program is reliable
 - Example:
 - On UDP (non reliable protocol) RPC retransmitted after timeout. When reply is received, the application infers that the RPC has been executed **at least 1 times**.
 - On TCP (reliable protocol) reply = only one execution, but no reply doesn't means no execution ... Still need timeout to handle server crashes.

- ## RPC – selecting network protocol
- UDP (non reliable) if :
 - procedures are idempotent (no side effects for multiple executions).
 - The size of arguments or results is less than the RPC packet size (8K)
 - The server should handle hundred clients.
 - TCP (reliable) if :
 - The application needs a reliable underlying transport
 - The procedures are non-idempotent
 - The size of either the arguments or the results exceeds 8K bytes

- ## RPC – eXternal Data Representation
- Network standard representation
 - Machine-independent description and encoding of data
 - Both sides involved:
 - Machine format to XDR = **serializing**
 - XDR to machine format = **deserializing**
 - Handle arbitrary data structures

RPC - XDR

- *XDR Protocol Specification: RFC 1014*

```

struct varintarr {
    int *data;
    int arrInth; } arr;

callrpc(hostname,
        PROGNUM, VERSNUM, PROCNUM,
        xdr_varintarr, &arr...);

xdr_varintarr(xdrsp, arrp)
XDR *xdrsp;
struct varintarr *arrp;
{
    return (xdr_array(xdrsp, &arrp->data,
                    &arrp->arrInth,
                    MAXLEN, sizeof(int), xdr_int));
}
  
```

RPC – Middle Layer

- callrpc & registerrpc (UDP)

```

Client
-----
if (stat = callrpc(argv[1],
    RUSERSPROG, RUSERSVERS, RUSERSPROC_NUM,
    xdr_void, 0, xdr_u_long, &nusers) != 0) { /* report error */}

Server
-----
unsigned long * nuser(char* indata)
{ /* do something useful */
  return some_unsigned_long; }
registerrpc(RUSERSPROG, RUSERSVERS, RUSERSPROC_NUM,
  nuser,
  xdr_void, xdr_u_long);
svc_run(); /* Never returns */
  
```

RPC – Lower Layer

- It enables you to use TCP as the underlying transport instead of UDP, without restriction on the data size.
- It enables you to allocate and free memory explicitly while serializing or deserializing with XDR routines.
- It enables authentication on either the client or server side, through credential verification

Data-parallel languages

George Bosilca
bosilca@cs.utk.edu

Data-parallelism

- Abstract, machine independent model of parallelism.
 - Fine-grain parallel operations, such as element-wise operations on array
 - Shared data in large, global arrays with mapping "hints"
 - Implicit synchronization between operations
 - Partially explicit communication from operation definitions
- Advantages:
 - Global operations conceptually simple
 - Easy to program
- Disadvantages
 - Difficult to find "perfect" compilers

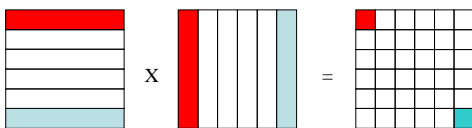
Properties of parallelism

- **Determinacy**: is the behavior of a program repeatable ?
- **Compositionality**: can independently created subprograms be combined ?
- **Expressiveness**: can all sort of parallelism be expressed ?
- **Implementability**: can a compiler generate efficient code for a variety of architectures ?

Matrix Multiply (easy)

- Simple sequential algorithm.

$$C = A \times B \Rightarrow C_{i,j} = \sum_k A_{i,k} B_{k,j}$$



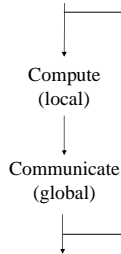
All multiplications can be done in parallel !!

Parallelizing Compilers

- After 20 years of intensive research:
 - Limited success in parallelism detection
 - Instruction-level parallelism at the basic-block level can be detected
 - Parallelism in nested-loops with arrays with simple index expression can be discovered
 - Analysis technique such as data dependence analysis, pointer analysis, flow sensitive analysis, abstract interpretation, still can not be applied across procedure boundaries
- Result: instead of training compiler to parallelize code, people have been trained to write parallel algorithms.

Data parallel Programming model

- All data structures distributed across a grid of virtual processors.
- The owner processor computes the data elements assigned to it.
- Global communication primitives allow processors to exchange data.
- Implicit global barrier after each communication
- All processors execute the same program !



Data parallel Languages

- Work distributed between
- The programmer (concentrate on solution):
 - High level structure and concept
 - Aggregate operations on large data structures
 - Data in global array with mapping information
- The compiler (map conceptual “massive” parallelism to the physical “finite” machine):
 - Complete the details
 - Distribute data guided by the user hints
 - Optimize computations and communications

HPF details

- Analysis
 - Traditional dataflow and dependence analysis
 - Data mapping analysis
- Computation partitioning
 - Use data mapping to create locality
 - Transform the code to enhance this locality
- Communication
 - Move data if data mapping and computation partitioning don't agree
 - Minimize/package communications (!)
- Code generation
 - All others optimizations

What compilation means for programmers

- Help analysis with assertions
 - ALIGN and DISTRIBUTE
 - INDEPENDENT
- Distribute arrays dimensions that exhibit parallelism
 - Conflicts require complex compilers, REDISTRIBUTE or new algorithm
- Consider communication patterns
 - BLOCK generally good for local stencils and fully-filled arrays
 - CYCLIC and CYCLIC(k) generally good for load-balancing and triangular loops

High Performance Fortran

- Defined by the High Performance Fortran Forum (HPFF) as a portable language for data-parallel computation
- History:
 - Proposed at SuperComputing 91 and HPFF Kickoff Meeting
 - Final draft of HPF, version 1.0 June 1993
 - New meeting 1994 & 1995 to make corrections, define further requirements.
- Influences:
 - Industry: C*, MasPar Fortran, Connection Machine Fortran
 - Academia: Fortran D, Vienna Fortran, ADAPT

HPF Features

- Data-parallel oriented
- Based on Fortran 90
- It's more a compiler specification
- Few language extensions (not anymore)
 - FORALL and PURE
- Compiler directives:
 - INDEPENDENT, ALIGN, DISTRIBUTE
- Data alignment and Distribution left to the compiler
- Miscellaneous Support Operations (HPF library)
- Nothing about:
 - I/O, Explicit message passing, Irregular applications, Non-data parallelism

HPF: few words about performance

- Highly dependent on the compiler and the nature of the code
- Close to MPI performance FOR regular problems with simple subscript expression (at least some compilers).
- Research continue (?) on task-parallelism and irregular problems.
 - Branch-and-bound (tree algorithm)
 - Subscript : array dependent (a(index(l,1))

HPF: Data mapping

- **THE** feature of HPF
- Everything is done using the compiler directive HPF\$
- User hints at 2 levels:
 - Distribute the data on the processors (DISTRIBUTE)
 - Create relationship between arrays (ALIGN)

HPF: Data mapping

- Goals:
 - Create locality: a processor should have direct access to all data it needs.
 - Avoiding contention: Data which are written in parallel should reside on different processors.

HPF: PROCESSORS

- Abstract a processors arrangement
- Declaration of virtual processors
 - The compiler will map these virtual processors to the physical one.
- !HPF\$ PROCESSORS

line(4)

square(2,2)



HPF: DISTRIBUTE

- Declare the data distribution on the processor arrangement.


```
!HPF$ DISTRIBUTE [array] [how-to] [ONTO proc]
!HPF$ DISTRIBUTE [how-to] [ONTO proc] :: [array]
```
- The distribution **should** be done on each dimension of the array
 - BLOCK, CYCLIC, CYCLOC(k), *
- ONTO specifies a processor arrangement.

HPF: DISTRIBUTE

- How-to distribute the data:
- BLOCK[size]
 - Equal size blocks of consecutive elements distributed among processors
 - N/p on each processor
 - Maybe less on the last one
 - No wrap around (i.e. size >= N/p)

HPF: DISTRIBUTE

- BLOCK**

REAL a(4,4)
 !HPFS DISTRIBUTE (BLOCK,BLOCK) ONTO square :: a

a square

REAL b(7)
 !HPFS DISTRIBUTE (BLOCK,BLOCK) ONTO line :: b

b line

HPF: DISTRIBUTE

- CYCLIC [blocksize]**
 - Blocks with equal number of elements are cyclically distributed among the processors.
 - Default blocksize is 1.
 - For cyclic(m) element with index I will be mapped on processor

$$I + \left(\frac{i}{m} \text{ mod } p \right)$$
- ***
 - No specific distribution
 - All of them, overlapping ...

HPF: DISTRIBUTE

(BLOCK,*)

(BLOCK, BLOCK)

(*, CYCLIC)

(CYCLIC(2), CYCLIC(3))

ONTO line

ONTO square

HPF: DISTRIBUTE

(BLOCK,*)

(BLOCK, BLOCK)

(*, CYCLIC)

(CYCLIC(2), CYCLIC(3))

ONTO line

ONTO square

HPF: DISTRIBUTE

(BLOCK,*)

(BLOCK, BLOCK)

(*, CYCLIC)

(CYCLIC(2), CYCLIC(3))

ONTO line

ONTO square

HPF: DISTRIBUTE

(BLOCK,*)

(BLOCK, BLOCK)

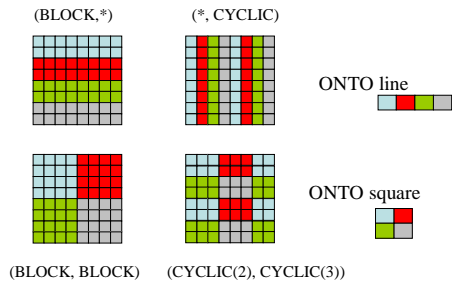
(*, CYCLIC)

(CYCLIC(2), CYCLIC(3))

ONTO line

ONTO square

HPF: DISTRIBUTE



HPF: DISTRIBUTE

- Communication point of view:
 - BLOCK usually good for local communication
 - Locality of cyclic not ease to see
 - CYCLIC(m) has advantages of both
- Problems:
 - Access with stride very expensive
 - Data redistribution can be very expensive
 - DYNAMIC, REDISTRIBUTE

HPF: ALIGN

- Specify that some objects have to be mapped in the same way as certain other object.
 - Computation will be more efficient if data are mapped on the same virtual processor.
- Only the <target> can have a DISTRIBUTE
- Definitions:
 - Dummy variable: scalar variable used locally to distribute the data with values depending on the valid initial axis values
 - Subscript-triplet : similar to an implicit loop in FORTRAN
 - I:E:S values starting from I until E with a step of S
 - “**” is a kind of “dummy variable that cannot interfere in subsequent distribution.

HPF: ALIGN

- !HPF\$ ALIGN <what> <source-list> WITH <target><subscript-list>
 - <what> and <target> can be any dimensional array
 - <source-list> can be:
 - “:” axis will be spread out across the matching axis of <whom>
 - “**” axis is collapsed: position across this axis make no difference in determining the corresponding position.
 - “dummy variable”: scalar range over all valid index values
 - <subscript-list> can be:
 - “:” or “**” or integer value or linear (affine) expression (depending on dummy variables) or subscript-triple or “dummy variable”

HPF: ALIGN

- !HPF\$ ALIGN A(:) WITH D(:,*)
- Equivalent to !HPF\$ ALIGN A(:) WITH D(:,j)
- Or for every valid j align A(:) with D(:,j)
- Or a copy of A is aligned with every column of D
- !HPF\$ ALIGN A(:,*) WITH D(:)
- Equivalent to !HPF\$ ALIGN A(:,j) WITH D(:)
- Or for every valid j, align A(:,j) with D(:)

HPF: ALIGN

- Equivalence:
 - !HPF\$ ALIGN A(:,*,K,::,*) WITH B(31::, K+3,20:100:3)
 - is equivalent to
 - !HPF\$ ALIGN A(I,J,K,L,M,N) WITH B(I-LBOUND(A,1)+31, L-LBOUND(A,4)+LBOUND(B,2), K+3, (M-LBOUND(A,5))*3 + 20)
 - with the conditions:
 - SIZE(A,1) .EQ. UBOUND(B,1)-31
 - SIZE(A,4) .EQ. SIZE(B,2)
 - SIZE(A,5) .EQ. (100-20+3)/3

HPF: ALIGN

- Not all notation are equivalent

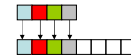
```
!HPF$ ALIGN A(:,*) WITH D(:)
!HPF$ ALIGN A(:,j) WITH D(:)
```

```
!HPF$ ALIGN A(:) WITH D(:,*)
!HPF$ ALIGN A(:) WITH D(:,j)
```

- Only a variable appearing in the source-list is understood to be a align-dummy

HPF: ALIGN

- REAL A(4), B(8)
- !HPF\$ ALIGN A(:) WITH B(1:4)



- !HPF\$ ALIGN A(:) WITH B(2:8:2)



- !HPF\$ ALIGN A(:) WITH B(8:5:-1)



- !HPF\$ ALIGN A(j) WITH B(8-2*j+1)



HPF: ALIGN

- “*” :

- In source-list = collapsing

- Multiple elements of source array will be aligned with a single element of the target

- In subscript-list = replication

- A single element of source array will be copied and aligned to multiple element of the target.
- BUT update at runtime require a **global communication**

HPF: TEMPLATE

- Allow to define a virtual array that can be distributed and used as a ALIGN target.

```
!HPF$ TEMPLATE DISTRIBUTE(BLOCK, BLOCK) :: EARTH(N,N)
REAL, DIMENSION(N,N) :: NW NE SW SE
!HPF$ ALIGN NW(I,J) WITH EARTH(I,J)
!HPF$ ALIGN NE(I,J) WITH EARTH(I,J+1)
!HPF$ ALIGN SW(I,J) WITH EARTH(I+1,J)
!HPF$ ALIGN SE(I,J) WITH EARTH(I+1,J+1)
```

HPF: data alignment

- How about the procedure call ?
- Some functions require different data alignments
- A subprogram can INHERIT data distributions from the calling routine


```
!HPF$ INHERIT array
```

HPF: data alignment

- Explicit interface: subprogram define his own data alignments in the interface block
- Variables present in this interface will be copied into temporary variables for the duration of the subprogram prior the entry in the subprogram
- They will be copied back in the original distribution on return from the subprogram

This approach might generate communication

HPF: data alignment

- Prescriptive: describe the mapping of the dummy argument.
!HPF\$ DISTRIBUTE A(BLOCK, CYCLIC)
 - If the actual argument does not have this mapping the compiler will generate the code to correctly remap the data
 - Information available at compile time
- Descriptive: weak assertion by the programmer that there is no need to remap
 - Compiler check and generate warning
!HPF\$ DISTRIBUTE A *(BLOCK, CYCLIC)
- Transcriptive: no specified mapping
 - The caller pass the mapping information at runtime
 - !HPF\$ DISTRIBUTE A* [ONTO *]

HPF: data alignment

!HPF\$ DYNAMIC array

- Such arrays can/will be realigned or redistributed at runtime
- !HPF\$ REALIGN <what> <source-list> WITH <target><subscript-list>
- Redistribute <what> **and all** arrays aligned to it !!!

HPF: high level parallelism

- FORALL: tightly-coupled parallel execution based on the structure of the index space
- PURE: procedure without side effects (to be used with FORALL)
- INDEPENDENT: assertion that iterations do not interfere with each other.

HPF: FORALL

- Has the semantic of array assignment but it's more clear and concise
- More general array regions
- More general access patterns

FORALL (index-spec-list[,mask-expr]) forall-assignment

```
FORALL ( I = 1,10) a(I) = b(I) + c(I+1)
FORALL ( I = 1, 10) a(index(I)) = b(I)
```

- Does not support multiple assignments

```
FORALL ( I = 1, 10, a(I) > 1.0) a(I) = 1.0/a(I)
FORALL ( I = 2, 9) a(I) = a(I-1) + a(I+1)
a(2:9) = a(1:8) + a(3,10)
```

HPF: FORALL

- The execution of a single statement FORALL has four steps:
- Compute the valid index set.
 - Based on *index-spec-list*.
- Compute the active index set.
 - Based on *mask-expr*.
- Compute the right hand side.
 - Any order - perhaps parallel!
- Assign to the left hand side.
 - Any order - perhaps parallel!

HPF: FORALL

- Multi-statement FORALL
- ```
FORALL (index-spec-list[,mask-expr])
 forall-body-statements
END FORALL
```
- Equivalent to multiple single FORALL
  - Nested: the inner FORALL modifies the active index set.

```
FORALL (I=1:3, J=1:3, I>J)
 FORALL (K=1:3, L=1:J, K+L > J)
 A(I,J,K,L) = J*K + L
 END FORALL
END FORALL
```

## HPF: FORALL

Index sets for this example:

Outer valid set :  $\{(1,1),(2,1),(3,1),(1,2)\dots(3,3)\}$

Outer active set :  $\{(2,1),(3,1),(3,2)\}$

Inner valid set:

$\{(2,1,1,1),(2,1,2,1),(2,1,3,1)\dots(3,2,2,2),(3,2,3,1),(3,2,3,2)\}$

Inner active set:

$\{(2,1,2,1),(2,1,3,1),(3,1,3,1),(3,2,2,2),(3,2,3,1),(3,2,3,2)\}$

## HPF: Pure attribute

- is guaranteed side-effect free
- returns a value and does not modify global data or pointer associations or perform input/output

```
PURE INTEGER FUNCTION f(x, y)
 IMPLICIT NONE
 INTEGER, INTENT(IN) :: x, y
 f = x*x + y*y + 2*x*y
END FUNCTION
```

- Can be used in FORALL statements
- Most of the intrinsic Fortran90 functions

FORALL (i=1 : n, j=1 : n, i.NE. j) a(i, j) = f(i, j)

## HPF: INDEPENDENT

- no iteration of the loop statement to which it applies affects any other iteration

!HPF\$ INDEPENDENT [, NEW(privat-list), REDUCTION(reduct-list)]

- DO loop

iterations or assignments can be performed in any order

- FORALL loop

the whole right-hand side does not have to be evaluated before assignment to the left-hand side can begin

```
!HPF$ INDEPENDENT
DO i = 1, n
 j(i) = i * i
END DO
```

```
!HPF$ INDEPENDENT
FORALL (i=1 : n) j(i) = i * i
```

## HPF: INDEPENDENT

- Support temporary variables which will be local to every virtual processor via NEW

!HPF\$ INDEPENDENT, NEW(tmp)

```
DO i=1,n
 tmp = SUM(B(i,:))
 A = tmp*tmp
END DO
```

- The temporary variable is uninitialized on each loop !

## HPF: INDEPENDENT

- Support global variables update using commutative global functions via REDUCTION

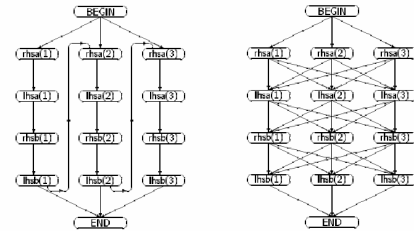
```
Z = 5.
!HPF$ INDEPENDENT, REDUCTION(Z)
DO i = 1, 10
 Z = Z + i
ENDDO
```

- The reduction function should be commutative & associative (as MPI reduction functions): +, -, \*, /, OR, AND, IOR, IAND, MIN, MAX
- Some operations can be used together like (+,-) and (\*,/)

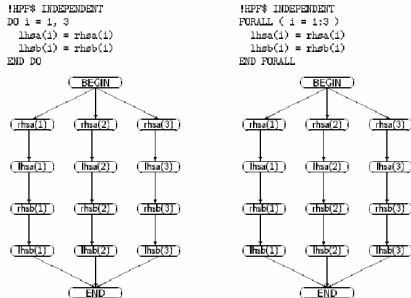
## HPF: DO vs. FORALL

```
DO i = 1, 3
 lhas(i) = rhas(i)
 lhas(1) = rhas(1)
END DO
```

```
FORALL (i = 1:3)
 lhas(i) = rhas(i)
 lhas(1) = rhas(1)
END FORALL
```



## HPF: DO vs. FORALL



## HPF: Intrinsic Functions

- System inquiry: NUMBER\_OF\_PROCESSOR, PROCESSOR\_SHAPE
- Mapping inquiry:
- Bit manipulation: LEADZ, POPCNT, POPPAR
- Array Reduction: IALL, IANY, IPARITY, PARITY
- Array sorting functions: SORT\_UP, SORT\_DOWN, GRADE\_UP, GRADE\_DOWN (index sort depending on the array value at index)

## HPF: Intrinsic Functions

- Array Combining: XXX\_SCATTER(ARRAY, BASE, INDEX1, ..., INDEXN, MASK)
  - Where XXX: ALL, ANY, COPY, COUNT, IALL, IANY, MAXVAL, MINVAL, PRODUCT & SUM
  - An arbitrary subset of the elements of an array can be combined to produce an element of the result.
- Example:
  - if A = [10 20 30 40 -10],
  - B = [1 2 3 4] and
  - IND = [3 2 2 1 1] then
  - SUM\_SCATTER(A, B, IND, MASK=A.GT.0) is
  - [41 52 13 4]

## HPF: Intrinsic Functions

- Array prefix and suffix
    - Each element of the result is a function of the elements that precede it or that follow it.
- Example:  
 If A = [3 5 -2 -1 7 4 8] then  
 SUM\_PREFIX(A, MASK=A.LT.6) is  
 [3 8 6 5 5 9 9]

## HPF through classical examples

- Jacobi
- Gauss Elimination
- Conjugate Gradient\*
- Irregular mesh relaxation

## HPF: Jacobi example

- Numerical solution to Laplace's equation

$$U_{i,j}^{n+1} = \frac{1}{4}(U_{i-1,j}^n + U_{i+1,j}^n + U_{i,j-1}^n + U_{i,j+1}^n)$$

```

for j = 1 to jmax
 for i = 1 to imax
 Unew(i,j) = 0.25 * (U(i-1,j) + U(i+1,j)
 + U(i,j-1) + U(i,j+1))
 end for
end for

```

Plenty of data parallelism

## HPF: Jacobi example

- Element updates:
  - 4 elements from the previous step
  - Generate static communication (compile time)
- Convergence test
  - Use all values from the last step
  - Global communication
    - Reduction operation
    - Efficient operation (encapsulated in HPF library).

## HPF: Jacobi example

- Element updates require 4 neighbors
  - CYCLIC : worst case as all elements will be exchanged between processors
  - BLOCK : less communications as some data will reside on the same processor
  - (BLOCK, \*) : move  $U(I-1,J)$  &  $U(I+1,J)$  for all J
  - (BLOCK, BLOCK) : move
    - $U(ILOW-1,J)$  &  $U(IHIGH+1,J)$  for all J
    - $U(I, JLOW-1)$  &  $U(I, JHIGH+1)$  for all I
- Convergence require the array reduction
  - Any distribution is OK: static, structured communications.

(BLOCK,\*) high latency computers or small problem size  
 (BLOCK,BLOCK) low latency computers

## HPF: Jacobi example

```

REAL u(0:nx), unew(0:nx)
!HPF$ DISTRIBUTE u(BLOCK, BLOCK)
!HPF$ ALIGN (,:) WITH u(:,) :: unew
DO WHILE (err .GT. epsilon)
 FORALL (i=1:nx-1, j = 1:nx-1) &
 unew(i,j) = (u(i-1,j) + u(i+1,j) + u(i,j-1) + u(i,j+1)) / 4
 Err = MAXVAL_SCATTER(ABS(unew-u))
 u = unew
ENDDO

```

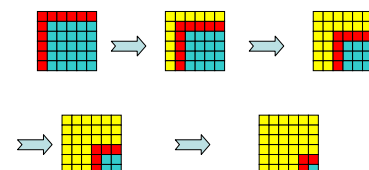
## HPF: Jacobi example

- The compiler get the easy job
  - Allocate data on processors based on DISTRIBUTE
  - Apply owner-compute rule based on left-hand side
  - Detect SHIFT communications from dependencies analysis
  - Identify MAXVAL as a global reduction function
  - Place all communications outside the parallel loop (optimization)
  - Number processors so that shift do not cause contention

## HPF: Gauss Elimination

- Linear solver: Given N linear equations in N unknown  $x_i$ , find all  $x_i$  which satisfy the equations.
- Algorithm:
  - Use eq.1 to eliminate  $x_1$  from equations (2..N)
  - Use eq.2 to eliminate  $x_2$  from equations (3..N)
  - Use eq.N to compute  $x_n$
  - Backward compute  $x_{n-1}, \dots, x_2, x_1$

## HPF: Gauss Elimination



■ Not used     
 ■ Pivot     
 ■ Updated elements

## HPF: Gauss Elimination

- Gaussian elimination look like a sequential algorithm:
  - Need whole column k to find pivot row
  - Need the whole column k and pivot row to perform pivoting
- Each step in the process it's composed by many smaller operations: they can be updated independently
- Still enough data parallelism do make a parallel algorithm (except the last stages).

## HPF: Gauss Elimination

- How about communications ?
  - Pivot search:
    - Reduction among column
    - Static, global communication
  - Element updates:
    - Each element from itself, pivot column and row
    - Static, global communication (broadcast)

## HPF: Gauss Elimination

- Pivot requires a 1-D reduction
  - Distribute rows => parallel with communications
  - Distribute columns => parallel without communications
- Element updates require old value, value from pivot and value from column
  - Distribute rows => parallel + broadcast pivot column
  - Distribute column => parallel + broadcast pivot row
- Which distribution ?
  - BLOCK : processors drop out of computation
  - CYCLIC : work stays distributed until the end, each time all processors have less work to do.
- Best distribution:
  - (\*,CYCLIC) if broadcast > pivoting one column
  - (CYCLIC,\*) if broadcast < one column, synchronous communications
  - (CYCLIC, CYCLIC) if broadcast < one column, overlapped communications

## HPF: Gauss Elimination

```
REAL A(n,n), tmp(n)
!HPF$ DISTRIBUTE a(CYCLIC, CYCLIC)
!HPF$ ALIGN tmp(i) with a(*,i)
```



```
DO k = 1, n - 1
```

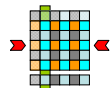
```
 ipivot = MAXLOC(ABS(A(k:n, k))) + k - 1
```

```
 tmp(k:n) = a(ipivot, k:n)
 a(ipivot, k:n) = a(k, k:n)
 a(k, k:n) = tmp(k:n)
```

```
 FORALL (i=k+1:n, j=k+1:n) &
 a(i,j) = a(i,j) - a(i,k) / tmp(k) * tmp(j)
 ENDDO
```

## HPF: Gauss Elimination

```
REAL A(n,n), tmp(n)
!HPF$ DISTRIBUTE a(CYCLIC, CYCLIC)
!HPF$ ALIGN tmp(i) with a(*,i)
```



```
DO k = 1, n - 1
```

```
 ipivot = MAXLOC(ABS(A(k:n, k))) + k - 1
```

```
 tmp(k:n) = a(ipivot, k:n)
 a(ipivot, k:n) = a(k, k:n)
 a(k, k:n) = tmp(k:n)
```

```
 FORALL (i=k+1:n, j=k+1:n) &
 a(i,j) = a(i,j) - a(i,k) / tmp(k) * tmp(j)
 ENDDO
```

Select the pivot  
Static global reduction

## HPF: Gauss Elimination

```
REAL A(n,n), tmp(n)
!HPF$ DISTRIBUTE a(CYCLIC, CYCLIC)
!HPF$ ALIGN tmp(i) with a(*,i)
```



```
DO k = 1, n - 1
```

```
 ipivot = MAXLOC(ABS(A(k:n, k))) + k - 1
```

```
 tmp(k:n) = a(ipivot, k:n)
 a(ipivot, k:n) = a(k, k:n)
 a(k, k:n) = tmp(k:n)
```

```
 FORALL (i=k+1:n, j=k+1:n) &
 a(i,j) = a(i,j) - a(i,k) / tmp(k) * tmp(j)
 ENDDO
```

Select the pivot  
Static global reduction  
Swap the rows  
Local copies

## HPF: Gauss Elimination

```
REAL A(n,n), tmp(n)
!HPFS DISTRIBUTE a(CYCLIC, CYCLIC)
!HPFS ALIGN tmp(i) with a(*,i)
```

```
DO k = 1, n - 1
```

```
 ipivot = MAXLOC(ABS(A(k:n, k))) + k - 1
```

```
 tmp(k:n) = a(ipivot, k:n)
 a(ipivot, k:n) = a(k, k:n)
 a(k, k:n) = tmp(k:n)
```

```
 FORALL (i=k+1:n, j=k+1:n) &
 a(i,j) = a(i,j) - a(i,k) / tmp(k) * tmp(j)
 ENDDO
```

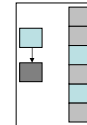
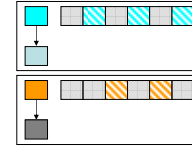


Select the pivot  
Static global reduction

Swap the rows  
Local copies

Broadcast pivot row  
and column (?)

## HPF: Gauss Elimination



## HPF: Gauss Elimination

```
REAL A(n,n), tmp(n)
!HPFS DISTRIBUTE a(CYCLIC, CYCLIC)
!HPFS ALIGN tmp(i) with a(*,i)
```

```
DO k = 1, n - 1
```

```
 ipivot = MAXLOC(ABS(A(k:n, k))) + k - 1
```

```
 tmp(k:n) = a(ipivot, k:n)
 a(ipivot, k:n) = a(k, k:n)
 a(k, k:n) = tmp(k:n)
```

```
 FORALL (i=k+1:n, j=k+1:n) &
 a(i,j) = a(i,j) - a(i,k) / tmp(k) * tmp(j)
 ENDDO
```



Select the pivot  
Static global reduction

Swap the rows  
Local copies

Broadcast pivot row  
and column

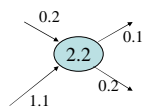
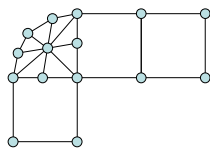
Update the sub-matrix

## HPF: Gauss Elimination

- More difficult for the compiler
  - Allocate portion of array on each processor based on DISTRIBUTE
  - Owner-compute applied based on left-hand side
  - Detect communications from dependencies & intrinsic
  - Transform the program:
    - Reorder computation to always pre-compute the next pivot column
    - Rearrange the communications to pipeline the updates
    - Broadcast values asynchronously
    - Net result: a 2x speedup
  - Use standard numbering for processor

## HPF: Irregular mesh relaxation

- Given a irregular mesh of values
- Update each value using its neighbors in the mesh
- Airflow simulation, car crash ...
- Algorithm
  - Store the mesh as a list of edges
  - Process all edges in parallel
    - Compute contribution of the edge
    - Add to the endpoint, subtracts from the other



## HPF: Irregular mesh relaxation

```
REAL x(nnodes), flux(nedges)
INTEGER iedges(nedges,2)
err = tol * 1e6
DO WHILE(err .GT. Tol)
 DO i=1, nedges
 flux(e) = (x(iedges(e,1)) - x(iedges(e,2)))/2
 err = err + flux(i) * flux(i)
 ENDDO
 DO l = 1, nedges
 x(iedges(i,1)) = x(iedges(i,1)) - flux(i)
 x(iedges(i,2)) = x(iedges(i,2)) + flux(i)
 ENDDO
 err = err / nedges
ENDDO
```

## HPF: Irregular mesh relaxation

- Each iteration use all data computed in the previous step and the edge array.
  - No parallelism at all
  - Instead use data-parallel edge and node updates
    - $\text{flux}(i) = (x(\text{iedge}(i,1)) - x(\text{iedge}(i,2)))/2$
    - $x(\text{iedges}(i,1)) = x(\text{iedges}(i,1)) - \text{flux}(i)$
    - $x(\text{iedges}(i,2)) = x(\text{iedges}(i,2)) + \text{flux}(i)$
    - Not independent as sometimes  $\text{iedges}(i,1) = \text{iedges}(i,2)$
    - But we can use a SUM\_SCATTER
  - Communication needed in both stages
    - Between edges and nodes to compute flux
    - Edge-node and node-node to compute x
    - All communication is static, local with respect to grid, but unstructured with respect to array indices

## HPF: Irregular mesh relaxation

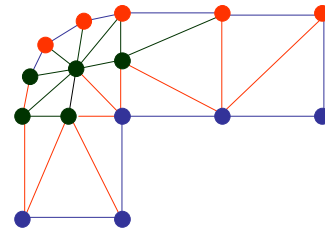
- Computing edge values require edge list and node values
  - Distribute edges => parallel, no communication for edges
  - Replicate edges => sequential or broadcast edge values
  - Distribute nodes => move "shared" endpoints
  - Replicate nodes => no movement of endpoints
- Updating node values require edge list, edge values and node values
  - Distribute edges => parallel, no communications
  - Replicate edges => sequential, no communication
  - Distribute nodes => move "shared" endpoints
  - Replicate nodes => move all endpoints
- The bottom line
  - Always distribute edges
  - Distribute nodes unless the problem is very small

## HPF: Irregular mesh relaxation

- Computation is static, and over full array with respect to edges
  - No load balancing issues
- Access to node array are "nearest neighbor" in the mesh
  - Not reflected in the index order
  - Does not favor either BLOCK or CYCLIC
- To minimize communications, edge and node distribution must fit the topology
  - Difficult (impossible) with HPF regular distributions
  - HPF 2.0 indirect distributions may be better, but require careful construction
- Idea: order the nodes and edges to keep "close" entities together and then use BLOCK

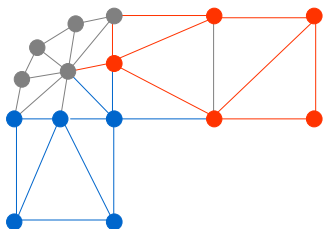
## HPF: Irregular mesh relaxation

- BAD data distribution



## HPF: Irregular mesh relaxation

- Good data distribution



## HPF: Irregular mesh relaxation

```

REAL x(nnode), flux(nedge)
INTEGER iedge(nedge,2)
INTEGER permute_node(nnode), permute_edge(nedge)
!HPF$ DISTRIBUTE x(BLOCK)
!HPF$ DISTRIBUTE flux(BLOCK)
!HPF$ ALIGN iedge(i,*) WITH flux(i)
!HPF$ ALIGN permute_edge(i) WITH flux(i)
!HPF$ ALIGN permute_node(i) WITH x(i)
CALL renumber_nodes(iedge, permute_node)
x(permute_node(:)) = x
FORALL (i=1:nedge) iedge(i,:) = permute_node(iedge(i,:))
permute_edge = GRADE_UP(iedge(:,1))
FORALL (i=1:nedge) iedge(i,:) = iedge(permute_edge(i),:)
err = tol * ied6
DO WHILE (err > tol)
 flux=(x(iedge(1:nedge,1))-x(iedge(1:nedge,2)))/2
 x=SUM_SCATTER(-flux(1:nedge),x,iedge(1:nedge,1))
 x=SUM_SCATTER(flux(1:nedge),x,iedge(1:nedge,2))
 err = SUM(flux*flux) / nedge
END DO

```

## HPF: Irregular mesh relaxation

- Challenging ...
  - Indexing of array will be difficult
  - How to apply the owner-compute rule ?
- Key technique: inspector-executor communication
  - First time the code is executed, generate a table of required communication at runtime (inspector)
    - How big it is ? How we can efficiently distribute the table to all processors ?
  - Use this table to manage unstructured communication until the communication pattern change (executor)
    - How do we know that the pattern change ?