Parallel programming paradigms and their performances

A high level exploration of the HPC world
George Bosilca
University of Tennessee, Knoxville
Innovative Computer Laboratory

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

• Allow overlapping communication computation
• Hiding latencies
• Additional cost for management

• 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
\[ a[i] = c[i] + d[i] \]
\[ b[i] = a[i] \times f[i] \]
Correct sequential semantic: $a[0], b[0], a[1], b[1], \ldots$

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
\[ a[i] = c[i] + d[i] \]
\[ b[i] = a[i] \times f[i] \]
For $i = 0$ to $64$ do
\[ b[i] = a[i] \times f[i] \]
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 $64$

$\begin{align*}
    c[i] &= a[i] + b[i] \\
    \text{LoadV } A, \text{RV1} \\
    \text{LoadV } B, \text{RV2} \\
    \text{AddV } RV1, RV2, RV0 \\
    \text{StoreV } C, RV0
\end{align*}$

Cost:
- sequential $64 \times 4$ cycles
- Vector $63 + 4$ cycles

Vector architecture - example

- Total cost of the vectorized loop:
  \[ T_{\text{init}} + 63 \quad \text{(LoadV A || LoadV B)} \]
  \[ + 4 + 63 \quad \text{(AddV)} \]
  \[ + T_{\text{init}} + 63 \quad \text{(StoreV)} \]
  \[ = 2 \times T_{\text{init}} + 103 \text{ ticks} \]

- Chaining = linking the pipelines together
  \[ T_{\text{init}} \times 4 + T_{\text{init}} + 63 = 2 \times T_{\text{init}} + 67 \]

- How about the memory access?
Vector architecture

- Several banks 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 banks)
- 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 MultiThreaded Architecture
  - 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 (8x88020 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

**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

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
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

ShMeme – 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

ShMeme – 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

Modified, Exclusive, Shared and Invalid

Still intolerable...
ShMem – Snoop protocols

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

Ordering (memory consistency)

<table>
<thead>
<tr>
<th>$P_1$</th>
<th>$P_2$</th>
</tr>
</thead>
</table>

"Assume initial values of $A$ and $B$ are 0!" 

1a) $A = 1$; 
1b) $B = 2$; 
2a) print $B$; 
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

Lamport 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)

<table>
<thead>
<tr>
<th>$P_1$</th>
<th>$P_2$</th>
</tr>
</thead>
</table>

"Assume initial values of $A$ and $B$ are 0!" 

1a) $A = 1$; 
1b) $B = 2$; 
2a) print $B$; 
2b) print $A$;

- 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

Remote Procedure Call

• Sun 1980 (?)
• Hide the underlying network transport
• Mostly synchronous
• Manual data conversion XDR
• RPC reliability?

**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**

```c
struct varintarr {
    int *data;
    int arrlnth; } 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->arrlnth,
        MAXLEN, sizeof(int), xdr_int));
}
```

---

**RPC – Middle Layer**

- **callrpc & registerrpc (UDP)**

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

unsigned long *

nuser(char
    *indata)

{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(I,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.
- \texttt{!HPF$ PROCESSORS}
  
  \begin{tabular}{c}
  \texttt{line(4)} \hspace{1cm} \texttt{square(2,2)} \\
  \end{tabular}

- \texttt{p1 p2 p3 p4}
- \texttt{p1 p2}
- \texttt{p3 p4}

HPF: DISTRIBUTE

- Declare the data distribution on the processor arrangement.
  \texttt{!HPF$ DISTRIBUTE \{array\} \{how-to\} \{ONTO\ \texttt{proc}\}}
  \texttt{!HPF$ DISTRIBUTE \{how-to\} \{ONTO\ \texttt{proc}\} :: \{array\}}
- The distribution \textit{should} be done on each dimension of the array
  - BLOCK, CYCLIC, CYCLOC(k), *
- \texttt{ONTO\ \texttt{specifies\ a\ processor\ arrangement.}}

HPF: DISTRIBUTE

- How-to distribute the data:
- \texttt{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 \(\geq\) N/p)
HPF: DISTRIBUTE

- BLOCK

```
REAL a(4,4)
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO square :: a
```

```
REAL b(7)
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO line :: b
```

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
    
    \[
    \{ \frac{i}{m} \bmod p \}
    \]

    - No specific distribution
    - All of them, overlapping ...

HPF: DISTRIBUTE

- (BLOCK,*)
- (*, CYCLIC)
- (BLOCK, BLOCK)
- (CYCLIC(2), CYCLIC(3))
HPF: DISTRIBUTE

- Communication point of view:
  - BLOCK usually good for local communication
  - Locality of cyclic not easy 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

• !HPFS 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 <what>
    • "**" 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

• !HPFS ALIGN A() WITH D(:,*)
  Equivalent to !HPFS 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

• !HPFS ALIGN A(:,*) WITH D()
  Equivalent to !HPFS ALIGN A(:,j) WITH D()
  Or for every valid j, align A(:,j) with D()

HPF: ALIGN

• Equivalence:
  !HPFS ALIGN A(:,;K,...) WITH B(31:;K+3,20:100:3)
  is equivalent to
  !HPFS 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
Not all notation are equivalent

- \!HPFS ALIGN A(:,*) WITH D(·)
- \!HPFS ALIGN A(:,j) WITH D(·)

- \!HPFS ALIGN A(·) WITH D(:,*)
- \!HPFS ALIGN A(·) WITH D(:,j)

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

REAL A(4), B(8)

- \!HPFS ALIGN A(·) WITH B(1:4)
- \!HPFS ALIGN A(·) WITH B(2:8:2)
- \!HPFS ALIGN A(·) WITH B(8:5:-1)
- \!HPFS ALIGN A(j) WITH B(8-2*j+1)

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 an ALIGN target.

```
HPF$ TEMPLATE DISTRIBUTED(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
  !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

\[
\text{FORALL (index-spec-list[,mask-expr]): forall-assignment}
\]

\[
\text{FORALL (I = 1, 10) a(I) = b(I) + c(I+1)}
\]

\[
\text{FORALL (I = 1, 10) a(index(I)) = b(I)}
\]

- Does not support multiple assignments

\[
\text{FORALL (I = 1, 10, a(I) > 1.0) a(I) = 1.0/a(I)}
\]

\[
\text{FORALL (I = 2, 9) a(I) = a(I-1) + a(I+1)}
\]

\[
a(2:9) = a(1:8) + a(3:10)
\]

HPF: FORALL

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

HPF: FORALL

- Multi-statement FORALL

\[
\text{FORALL (index-spec-list[,mask-expr]): forall-body-statements}
\]

\[
\text{END FORALL}
\]

- Equivalent to multiple single FORALL
- Nested: the inner FORALL modifies the active index set.

\[
\text{FORALL (I=1:3, J=1:3, K=1:3)}
\]

\[
\text{FORALL (K=1:3, L=1:J, K+L > J)}
\]

\[
A(I,J,K,L) = J*K + L
\]

\[
\text{END FORALL}
\]

\[
\text{END FORALL}
\]
HPF: FORALL

Index sets for this example:
Outer valid set: \{(1,1),(2,1),(3,1),(1,2)\ldots(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)\ldots(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

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

- Can be used in FORALL statements

```fortran
!HPF$ INDEPENDENT (i=1 : n, j=1 : n, i .NE. j) a(i, j) = f(i, j)
```
- Most of the intrinsic Fortran90 functions

HPF: INDEPENDENT

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

```fortran
!HPF$ INDEPENDENT (, NEW(privat-list),REDUCTION(reduct-list))
```
- DO loop

```fortran
DO i = 1, n
  j(i) = i * i
END DO
```
- FORALL loop

```fortran
FORALL (i=1 : n) j(i) = i * i
```

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

- Support temporary variables which will be local to every virtual processor via `NEW`
  
  ```hpf```
  ```
  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: DO vs. FORALL

![Diagram showing the difference between DO and FORALL constructs in HPF.]

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
  $\text{SUM}\_\text{PREFIX}(A, \text{MASK}=A.LT.6)$ 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

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

for $j = 1$ to $j_{\text{max}}$
for $i = 1$ to $i_{\text{max}}$
  $$U_{\text{new}}(i,j) = 0.25 \ast (U(i-1,j) + U(i+1,j) + U(i,j-1) + U(i,j+1))$$
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)
!HPFS DISTRIBUTE u(BLOCK, BLOCK)
!HPFS 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 gets 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 shifts do not cause contention

HPF: Gauss Elimination

- Linear solver: Given N linear equations in N unknowns $x_i$, 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 $x_{n-1}$, ..., $x_2$, $x_1$
HPF: Gauss Elimination

- Gaussian elimination looks 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 to make a parallel algorithm (except the last stages).

HPF: Gauss Elimination

- How about communications?
  - Pivot search:
    - Reduction among columns
    - 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
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	 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)
!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

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 an 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**

```plaintext
REAL x(nnodes), flux(nedges)
INTEGER edges(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 I = 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
    - \( \text{flux}(i) = \frac{(x(\text{edge}(i,1)) - x(\text{edge}(i,2)))/2}{x(\text{edge}(i,1)) + x(\text{edge}(i,2)) + \text{flux}(i)} \)
    - Not independent as sometimes \( \text{edges}(i,1) = \text{edges}(i,2) \)
    - 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

![Irregular mesh relaxation with BAD data distribution]

HPF: Irregular mesh relaxation

- Good data distribution

![Irregular mesh relaxation with Good data distribution]

HPF: Irregular mesh relaxation

REAL a(node), f(node)
INTEGER edge[5]
DIMENSION permute_node[node], permute_edge[edge]
HPF SCATTER(a, node[i]) WITH f(node)
HPF ALLOC permute_node[i] WITH f(node)
HPF ALLOC permute_edge[i] WITH f(node)
CALL proc_index(n, node, permute_node)
proc_index(node, node, permute_node)
FORALL (i = edge(i)) -> permute_node[permute_edge(i)]
END_ALL (i = edge(i)) -> permute_node[permute_edge(i)]
err = 0.0
DO WHILE (err > tol) 
  f(node) = proc_index(n, node, node, node, node, node)
  err = SUM (|f(node) - f(node)|) / |
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 is it? 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?