More Tools

Done basic, most useful programming tools:

- MPI
- OpenMP

What else is out there? Languages and compilers?

Parallel computing is significantly more complicated than serial computing.

Places a significant burden on the application developer, as we have seen
- Design of parallel programs
- Implementing parallel programs

Why can’t we automatically parallelise everything?

Desire:
- Programming to be as easy as possible
- Resulting programs to be portable across platforms with modest effort
- The programmer should retain as much control over performance as possible.

Conflicting goals?

More Tools (cont)

Programming system would need to solve three fundamental problems:

- It must find significant parallelism in the application. May involve input from the user but must not force a complete re-write of the code by the user.
- It must overcome performance penalties due to the complex memory hierarchy of modern parallel computers. This could involve significant transformations of the data to increase locality. These two may need to be traded off.
- It must support migration to different architectures with only modest changes. Means that programming interface must be independent of machine, and system must optimise for different machines.

Each component of the system should do what it does best:

- Application developer should concentrate on problem analysis and decomposition at a fairly high abstract level
- System (programming language and compiler) should handle details of mapping the abstract decomposition onto the computing configuration.
- Both should work together to produce a correct and efficient code.
More Tools (cont)

Why can’t all this be fully automatic?
To be acceptable, fully automatic parallelisation must achieve similar performance to hand-coded programs, or else worthless.
Early days (1970s), did a good job of automatic vectorisation.
Required ability to figure out a dependency analysis:

```plaintext
REAL A(1000,1000)
DO J=2,N
  DO I=2,N
    A(I,J) = (A(I,J+1)+2*A(I,J)+A(I,J-1))*0.25
  ENDS DO
END DO
```
J-loop not vectorisable but I-loop is:

```plaintext
REAL A(1000,1000)
DO J=2,N
  A(2:N,J) = (A(2:N,J+1)+2*A(2:N,J)+A(2:N,J-1))*0.25
END DO
```
Re-write and hand-vectorise computationally intensive loops this way.
Not always possible: e.g. reference to \( A(\text{IND}(I)) \) can only be checked at runtime.

More Tools (cont)

Expanding these automatic vectorisation techniques to automatic parallelisation techniques was ok for MIMD programs on shared memory architectures with modest numbers of processors (e.g. Cray C90).
Then came distributed memory machines!

**Additional complexity:**

- How to partition data to the disjoint memories of processors, maximising locality, minimising communication.
- Then has to automatically arrange communication operations.

**PLUS** (even on shared memory machines), parallel regions have to be large enough such that the benefits of concurrency overcome the overheads of the communication.
The latter means you **have to find LARGE parallel regions**.
This implies the tougher problem for the compiler of analysing data dependencies over much larger sections of code -- interprocedural analysis and optimisation.
Tough. Can be done, but long compiler times. Rare problems and rare success on salable machines.

Turn to language-based strategies that use some simple input from the user.
Key to high performance on distributed memory is allocation of data to processors allowing maximum locality, minimum of communication.

Data parallelism is sub-dividing the data so that this is possible.

Design languages which assumes user input to guide this, then automate the rest.

Note this only works for DATA parallel not TASK parallel. Later things (OpenMP) more flexible that HPF.

Early versions: Fortran D, CM Fortran, C*, data-parallel C, PC++

Culmination: High Performance Fortran (HPF). Also HPC++.

HPF is an extended version of Fortran90

Idea: automate most of the details of managing data.

Provides set of directives that user uses to specify data layout.

Compiler turns these into low-level operations to do the communication and synchronisation.

Directives merely provide advice to compiler -- no actual change to program.

How achieved is implementation dependent, hence portable

Directives as structured comments that extend the variable type declarations:

    REAL A(1000,1000)
    !HPF$ DISTRIBUTE A(BLOCK,*)

Distribute data across the processors in contiguous chunks.

Could also be other distribution patterns:

    !HPF$ DISTRIBUTE A(CYCLIC,*)

(round-robin) or

    !HPF$ DISTRIBUTE A(CYCLIC(K),*)

(round-robin with blocks of size K)

(block good for regular, nearest-neighbour problems. Cyclic provides better load balancing but possibly worse communication)

To ensure locality, match data layouts between arrays

    !HPF$ ALIGN B(I,J) WITH A(I,J)

or

    !HPF$ ALIGN B(:) WITH A(:,J)

etc
Data-Parallel Programming: HPF (cont)

```
REAL A(1000,1000), B(1000,1000)
DO J=2,N
  DO I=2,N
    A(I,J) = (A(I,J+1)+2*A(I,J)+A(I,J-1))*0.25 & &
     (B(I+1,J)+2*B(I,J)+B(I-1,J))*0.25
  END DO
END DO

------------------- HPF VERSION ---------------------------------------

REAL A(1000,1000), B(1000,1000)
!HPF$ DISTRIBUTE A(BLOCK,*)
!HPF$ ALIGN B(I,J) WITH A(I,J)
  WITH A(I,J)
!HPF$ INDEPENDENT
DO J=2,N
  !HPF$ INDEPENDENT
  DO I=2,N
    A(I,J) = (A(I,J+1)+2*A(I,J)+A(I,J-1))*0.25 & &
     (B(I+1,J)+2*B(I,J)+B(I-1,J))*0.25
  END DO
END DO
```

Typical implementation would distribute the loop according to the “owner-computes” rule: processor owning the array value on left side of assignment statement is responsible for updating/computing it. All communication done transparently.

Note, easy writing generates complicated underlying optimised object code! :

```
REAL A(10000)
!HPF$ DISTRIBUTE A(BLOCK)
X=0.0
DO I=1,100000
  X=X+A(I)
END DO
```

Simple HPF code but compiler must know to make a parallel reduction efficient! Help compiler out:

```
REAL A(10000)
!HPF$ DISTRIBUTE A(BLOCK)
X=0.0
!HPF$ INDEPENDENT, REDUCTION(X)
DO I=1,100000
  X=X+A(I)
END DO

OR

REAL A(10000)
!HPF$ DISTRIBUTE A(BLOCK)
X=SUM(A)
```
Data-Parallel Programming: HPF (cont)

HPF has support for data parallelism

Notice data layout is set at the declaration points and therefore spans procedural extents and is not specifiable per operation.

Does not allow task parallelism

⇒ Open MP better!

✓ Data parallel equivalent statements: parallel do, do reduction etc

✓ PLUS task parallel constructs e.g. sections

HPF also had drawback that is obviously designed for regular grids so finite-volume codes etc SOL. Rectified somewhat in HPF 2.0 that allows irregular distributions.

No public domain HPF compilers! BUT there are some commercial ones, notably the Portland Group’s compiler (pgf)

Also NO SUPPORT FOR I/O!

More information:

http://hpff.rice.edu


SPMD Programming: Co-Array Fortran

HPF almost made it to the Fortran standard but did not. Some concepts from HPF remain in the Fortran standard.

Co-Array Fortran is to be incorporated in the next release of the Fortran standard (2008? 2010?)

Previously known as F--

CAF assumes that multiple copies of the program called images execute asynchronously.

Data objects are replicated in all images and may have different values in the different images.

Main concept:

Arrays that are declared with an additional co-dimension specified in square brackets [ ] become a co-array that is accessible by all the images.

Extent of the co-dimension is the number of images

e.g.

REAL X(10)[*], Y(4,4)[*]

REAL (1000/NUM IMAGES(),1000)[*]
Co-array can have more than one co-dimension to reflect multi-dimensional processor layouts but must have same size on all images; can be allocatable:

```fortran
REAL, ALLOCATABLE :: X(12)[4,3]
```

Programs use the co-dimension as they would any other dimension. This allows simple expressions to communicate data between images. E.g.

```fortran
PARAMETER (MY_N = 1000/NUM_IMAGES())
REAL A(0:MY_N+1)[*]
ME = THIS_IMAGE()
IF (ME > 1) THEN
  A(0)[ME] = A(MY_N)[ME-1]
ENDIF
IF (ME < NUM_IMAGES()) THEN
  A(MY_N+1)[ME]=A(1)[ME+1]
ENDIF
```

Intrinsic data transfer. Notice only ONE processor needs to make the call => one-sided communication (not send and receive).

Local and remote variable copies:

```fortran
X = Y[P] => "GET"
REAL, DIMENSION(N)[*] :: X,Y
X(:)=Y(:)[P]
Y[P] = X => "PUT"
Y(:) = X => "BROADCAST X"
Z(:) = Y[1] => "GATHER"
X=SUM(A(:)[1]) => Global reduction operations
```

Synchronisation is NOT automatic in CAF. Need to explicitly

```fortran
CALL SYNC_ALL() CALL SYNC_TEAM() CALL_SYNC_MEMORY()
```

Force one image at a time (cf. OpenMP):

```fortran
CALL START_CRITICAL
CALL END_CRITICAL
```
A super-cool thing with CAF: Provision for parallel I/O!

Normally assume each images writes to separate files on separate I/O units

However, extensions allow several images to be connected to the same file attached to the same unit in each image:

```fortran
REAL A(8)!
INQUIRE(IOLENGTH=LA) A
IF (THIS_IMAGE().EQ.1) THEN
  OPEN( UNIT=11, FILE='fort.11', STATUS='NEW', ACTION='WRITE',
  & FORM='UNFORMATTED', ACCESS='DIRECT', RECL=LA*NUM_IMAGES())
  WRITE(UNIT=11, REC=1) A(:)
  CLOSE(UNIT=11)
ENDIF
OPEN( UNIT=21, FILE='fort.21', STATUS='NEW', ACTION='WRITE',
  & FORM='UNFORMATTED', ACCESS='DIRECT', RECL=LA, &
  TEAM=(/ (I, I=1, NUM_IMAGES()) /))
WRITE(UNIT=21, REC=THIS_IMAGE()) A
CLOSE(UNIT=21, TEAM=(/ (I, I=1, NUM_IMAGES()) /))
```

Co-array A is written identically to units 11 and 21. Unit 11 is open on the first image only, and the co-array is written as one long record. Since A is a co-array, the communication necessary to access remote image data occurs as part of the write statement. Unit 21 is open on all images and each image writes its local piece of the co-array to the file. Since each image is writing to a different record, there is no need for `sync_file` and the writes can occur simultaneously. Both the OPEN and CLOSE have implied synchronization, so the WRITE on one image cannot overlap the OPEN or CLOSE on another image.

Advantages:
- Provide high-level operations for common stuff
- BUT allow access to low-level manipulations too.
- Very small extension to actual language (not directives)
- Hardware implementation simple IF allow one-sided communication :)

Disadvantages:
- In the formative stages.
- How portable?
- Compiler support for optimisation but dependence analysis difficult since SPMD mode is basically asynchronous.

Notice can mimic CAF using OpenMP where arrays have an extra dimension.

Can pretty easily translate between CAF and OpenMP and there exists translators to do thins.
Which one to use?

Comparison

**MPI** : extremely flexible, all communication is explicit, everything in the hands of the application developer => hard work!

**OpenMP** : directive-based, small changes to sequential code, great for quick and dirty, limited to shared-memory. (Cluster OpenMP???)

**HPF** : use OpenMP unless want similar functionality on distributed memory machine and have a data parallel problem.

**Co-Array Fortran** : elegant, simple, may be the way of the future, but yet to be proven (VHS vs Betamax???)