High Performance Computing
Course Notes 2009-2010

Models of Parallel Programming
Models of Parallel Programming

Different approaches for programming on parallel and distributed computing systems include:

- Dedicated languages designed specifically for parallel computers
- Smart compilers, which automatically parallelise sequential codes
- Data parallelism: multiple processors run the same operation on different elements of a data structure
- Shared memory: processors share a common address space
- Message passing: the memory in each processor has its own address space
Specially Designed Language
Occam

- Occam is a concurrent programming language
- Occam is an executable implementation of Communicating Sequential Processes (CSP) theory
  - CSP: a mathematical theory for describing the interactions of tasks in a concurrent system
  - Can theoretically prove if the program written in Occam is correct
- Occam is specially designed to make full use of the architecture characteristics of the computer system consisting of the transputer (transistor computer) chips, developed by INMOS
  - Transputer is the first microprocessor specially designed for parallel computing
  - A number of transputer chips are wired to form a complete computer system (no bus, RAM or OS)
- In Occam, the processes communicate through channels
  - Channel can be regarded as the message passing mechanism within a computer
Occam

Sequential execution
SEQ
  x := x + 1
  y := x * x

Parallel execution:
PAR
  p()
  q()

Communication between processes:
ProcessA ! Channel_var
ProcessB ? Channel_var
Occam

- The Occam language was not popular
  - Poor portability
  - Transputer chip is very expensive

For more information:

Occam 2 reference manual
  www.wotug.org/occam/documentation/oc21refman.pdf

Occam archive
  http://vl.fmnet.info/occam/

Transputer
Dedicated languages

In general dedicated languages are going to do a better job
1. Designed with the hardware architecture
2. Structure of the language reflects the nature of parallelism

However
1. Niche languages are not generally popular
2. It’s hard to port existing code

Much better to modify and extend an existing language to include parallelism, because
1. Better audience
2. Only need to learn new constructs or API, not a new language
3. Porting is a lot easier
Compiler Approach
Examples

A compiler takes code written in a standard language (e.g. C or Fortran) and automatically compiles it to be run in parallel (e.g. by parallelising loops)

Example 1:
DO I=1, N
   A(I)=A(I)+B(I)
ENDDO

Example 2:
DO I=2, N
   A(I)=A(I-1)+B(I)
ENDDO

Compiling the code:
f90 –O3 –autopar foo.f
Features of the Compiler approach

Can work fairly well for some regular problems

Fully automatic (efficient) parallelisation is difficult, and unlikely to be efficient in general
Assisting Compiler

- Programmers can assist the compiler by writing the code in a way that explicitly expresses the parallelism in the program
  
  - Usually done using directives (pseudo-comments that instruct the compiler where parallelism lies)
  
  - During 1990s *OpenMP* emerged as a common set of directives for implementing various types of parallel execution and synchronization
  
  - Add these to serial code (and ignored if targeting a single processor machine)
  
  - We cannot blame the compiler but ourselves if there is something wrong
Examples of Assisting Compilers

Correct:
C$OMP PARALLEL
  DO I=1, N
    A(I)=A(I)+B(I)
  ENDDO
C$OMP END PARALLEL

Wrong:
C$OMP PARALLEL
  DO I=2, N
    A(I)=A(I-1)+B(I)
  ENDDO
C$OMP END PARALLEL
Compilers

For more information on this subject area see:

- *Further reading in High Performance Compilers for Parallel Computing*, Michael Wolfe

- *Vectorizing C Compilers: how good are they?* Lauren Smith, ACM/IEEE Conference on Supercomputing

- *Parallelizing and Vectorizing Compilers*, Eigenmann and Hoeflinger, (also on course web page)
Data Parallelism
Task parallelism vs. Data parallelism

Task parallelism:

if CPU="a" then
  do task "A"
else if CPU="b" then
  do task "B"
end if

Data parallelism:

d is an one-dimensional array
if CPU="a" then
  low_limit=1; upper_limit=50
else if CPU="b" then
  low_limit=51; upper_limit=100
end if
do i = low_limit , upper_limit
  Task on d(i)
end do
Data Parallelism

- If we are applying the same operation to every element in an array (or list, set, tree, database or whatever) then we may be able to exploit data parallelism.

- F90 and HPF provide the support for Data parallelism.

- F90 and HPF allow scalar operations to be applied to arrays to support the data parallelism.
  - adding two matrixes can be written as

\[
A = B + C \quad ! \ A, \ B, \ C \text{ are arrays}
\]

(matrix B and C must conformal), which is equivalent to

\[
\begin{align*}
& \text{do } i = 1,m \\
& \quad \text{do } j = 1,n \\
& \quad \quad A(i,j) = B(i,j) + C(i,j) \\
& \quad \text{enddo} \\
& \text{enddo}
\end{align*}
\]

- The former expression is called explicit parallel statement while the latter expression called implicit parallel statements (the latter can be parallelised by the compiler if possible).
Data Parallelism

A *data parallel* program is a sequence of explicitly and/or implicitly parallel statements.

The compiler can partition the data into disjoint sub-domains, one per processor.

Data placement is clearly an essential part of data-parallelism, and if you get the *data locality* wrong, you will take a performance hit.

Compilers can achieve data partition, data allocation, data communications.
Data Parallelism

A *data-parallel* programming is higher-level than the message passing-type approach

- programmer does not need to specify communication structures
- this is done by the compiler (inferred from the program and underlying computer architecture)

However,

- not all algorithms can be specified in data parallel terms
- if the program has irregular communication patterns then this will be compiled less efficiently

Examples of data parallel languages include F90 and HPF
Data Distribution in HPF

!HPF$ DISTRIBUTE arrays(parameters)

- Each dimension of an array may be distributed in one of three ways.
  - *: no distribution,
  - block(n): block distribution (default n=N/P),
  - cyclic(n): cyclic distribution (default n=1)

- BLOCK distribution divides the indices in a dimension into contiguous, equal-sized blocks of size $N/P$

- a CYCLIC distribution uses a cyclic fashion to allocate indices

- The optional integer argument to BLOCK and CYCLIC specifies the number of elements in a block.

- Declare abstract processors

  !HPF PROCESSORS Q(4, 8)
This figure shows different parameters are used to specify different distributions of a two-dimensional array of size 8x8 onto four processors. A PROCESSORS directive declares abstract processors.

!HPF PROCESSORS Q(2, 2)