High Performance Computing

Course Notes 2008-2009

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

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

Can work fairly well for some regular problems
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, database or whatever) then we can 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:

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

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

    ```
    do i = 1,m
      do j = 1,n
        A(i,j) = B(i,j) + C(i,j)
      enddo
    enddo
    ```

- 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.
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 global communication patterns then this will be compiled less efficiently

Examples of data parallel languages include F90 and HPF
Data Alignment

!HPF$ ALIGN source_array WITH target_array

ALIGN A(I) WITH B(I)

ALIGN A(I) WITH B(I+2)

ALIGN C(I) WITH B(2*I)

ALIGN D(I,J) WITH E(J,I)

ALIGN D(i,*) WITH A(i)

ALIGN A(i) WITH D(*,i)
Data Distribution in High Performance Fortran

!HPF$ DISTRIBUTE arrays(parameters)

This figure shows different parameters are used to specify different distributions of a two-dimensional array of size 8x8 onto four processors.
High Performance Computing
Course Notes 2008-2009

Shared Memory Parallel Programming
Techniques

- Multiprocessing
- User space multithreading
- Operating system-supported (or kernel) multithreading
- Distributed Shared Memory
Multiprocessing
Features of Processes

A process is associated with a particular program

A process has its own state information

A process has its separate address space
- fork() is used to create a child process

- The child process is exactly the same as the parent except the returned value of fork()

- Use parent and child to do different tasks
Load a new program after fork

main () {
    int pid, ret;
    pid=fork(); /*generate a child process*/
    if(pid==0) {
        /*run by the new process*/
        ret=execl("/bin/ls", ";"); /*load a new program*/
    } else {
        /*run by the parent process*/
        perform whatever operation
        ret=wait(&status); /* wait the child process exit */
    }
}

Computer Science, University of Warwick
How a new process is created

Before fork
- Parent process: Global Data
- Child process: Global Data

Executing
- Parent process: Code
- Child process: Code

Stack
- Parent process: Stack
- Child process: Stack

During fork
- Global Data is copied to Child process

After fork
- Parent process: Global Data
- Child process: Global Data

Executing
- Parent process: Code
- Child process: Code

Stack
- Parent process: Stack
- Child process: Stack
Scheduling Processes

→ When to switch the processes (timing for scheduling)
  - time slice runs out
  - System call
  - trap

→ Overhead of switching processes is relatively high, have to save and load the following information
  - Three segments
  - Open File descriptors
  - Signal handler table
  - program counter
Each entry in the process table (in kernel space) contains the following:

- Process ID
- Parent process ID
- Real and effective user and group IDs
- State
- Pending signals
- Code segment
- Data segment (static data)
- Stack segment (temporary data)
- User area
- Signal handler table
- Open file descriptors
- Recent CPU usage
- Hardware register contents (unless running)
- Page table