

CSE 260 - Introduction to Parallel Computation

Class 3 - Sept 27, 2001

Programming Parallel Computers

Reminder: quizlet next Tuesday

- Vocabulary, concepts, and trends of parallel machines and languages
- 15 minutes – multiple choice and short answers

Possible mini-report

Get rough estimate of how fast Google's computers could execute the Linpack benchmark






Assume 20,000 PC's, equally distributed at 4 sites, 1 GB/sec connection between sites, at each site PC's are in groups of 40; 100 MB/sec between groups, 10 MB/sec to each PC. 128 MB memory/proc.

(Ask me about computation & communication requirements of Linpack benchmark.)

Approaches to Parallel Programming

- Write sequential program, leave it to compiler to parallelize
 - perhaps guided by user's directives (e.g. Cray Fortran)
- Write sequential program that calls parallel subroutines (e.g. matlab)
 - of course, someone else had to write those subroutines
- Data parallel languages
- Message passing libraries or language features
- Shared memory programming (threads)

Programming – Architecture affinities

- Write sequential program, leave it to compiler to parallelize 
- Write sequential program that calls parallel subroutines 
- Data parallel languages 
- Message passing libraries or language features 
- Shared memory programming (threads) 

BUT, it's possible to use any programming style on almost any architecture

Data Parallel Languages

- Single thread of control consisting of parallel operations
 - array operations: $A = B + C$
 - communication to neighbors: $A(0..n-1) = B(1..n)$
 - ,reduce operations: $s = \text{sum}(A)$
- Example: Fortran 90
- Not very successful on MPP's
 - HPF (High Performance Fortran) was a high-profile attempt; hasn't gained much acceptance

Threads and Processes

- Threads: sequences of instructions that have access to a common memory (threads can have private variables too)
 - often described as "light-weight" since you don't need to create new address space or copy data when spawning (creating) or swapping to a different thread
- Processes: sequences of instructions that have separate address spaces
 - spawning (i.e. creating new) processes and swapping between them can take 1000's of cycles.

A common programming style is to have only one thread or process per processor

Separate vs. shared address spaces

P1:

```
x = 1;  
y = 10;  
z = 100;  
printf( "%d" ,x+y+z );
```

P2:

```
z = 200;  
y = 20;  
x = 2;
```

Separate address spaces:

Only possible answer is "111"

Shared address space:

Could be "111", "112", "211", ...

Is "212" possible??

Separate vs. shared address spaces

P1:

```
x = 1;  
y = 10;  
z = 100;  
printf("%d", x+y+z);
```

P2:

```
z = 200;  
y = 20;  
x = 2;
```

Separate address spaces:

Only possible answer is "111"

Shared address space:

Could be "111", "112", "211", ...

Is "212" possible??

Yes, unless there
is sequential consistency

Parallel Languages have ability to...

Move data between processors

- Message-passing: processors communicate by sending messages to one another
- Threads: processors communicate by writing and reading shared variables

Synchronize threads or processes

Let P1 know if P2 has finished with some work

Perhaps allow P1 and P2 to negotiate which one will do some piece of work

Message Passing Communication

Basic message passing primitives

- Send(parameter list)
- Receive(parameter list)

Depending on language, parameters might include pointer to array holding data, length, id of process communicating with, ...

Some Message Passing Libraries

- PVM (Parallel Virtual Machine)
 - Designed for heterogeneous machines (does data conversion; allows processors to be added)
- MPI (Message Passing Interface)
 - Designed for homogeneous MPP; now can be used by heterogeneous platforms too.
 - MPI CH is public domain; vendors have optimized versions
 - Portable; callable from C and Fortran
- KeLP (Kernel Lattice Parallelism)
 - Library for C++
 - Developed by Scott Baden & Steve Fink
 - Targeted for certain classes of scientific computations

Flavors of message passing

- Synchronous routines return when the message transfer has been completed
 - Synchronous send waits until the complete message is accepted by the receiver before proceeding.
 - Or perhaps wait until it is moved to a buffer
 - Anyway, data can be overwritten in next statement
 - Synchronous receive waits until the message it is expecting arrives.

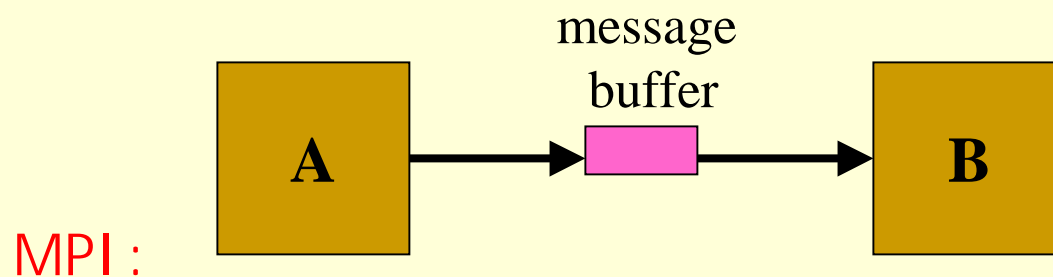
Also called blocking routines

Synchronous message passing accomplishes both data transfer and synchronization.

Disadvantage: processor is idle while waiting
(assuming there is one process per processor)

Nonblocking message passing

- Nonblocking send returns whether or not the message has been received
 - If receiving processor not ready, message may be move to a message buffer
 - Message buffer used to hold messages being sent by A prior to being accepted by receive in B



- routines that use a message buffer and return after their local actions complete are blocking (even though message transfer may not be complete)
- Routines that return immediately are non-blocking

Programming Models which sound like architecture models, but aren't:

- Single Program Multiple Data (SPMD)
 - processors run the *same* program but not necessarily in lock step.
 - very popular and scalable programming style
 - Programming model used in MPI
- Multiple Program Multiple Data (MPMD)
 - different processors run different programs
 - PVM distribution has some simple examples

Synchronization in shared memory

Barriers: No thread can proceed past barrier until all threads have reached it.

Full-empty bit:

```
P1: while (b==1){ };  
    x = ...;  
    b = 1;
```

```
P2: while (b==0) { };  
    ... = x;  
    b = 0;
```

"spin" until **b==1**

Suitable for coordinating two threads

Locks (or mutex = mutual exclusion):

```
lock *l = alloc_and_init();  
acquire(l);  
    ...access data...  
release(l);
```

Only one thread can be in this "critical section" at a time

Some shared memory languages

- BSP (Bulk Synchronous Processes)
 - Only provides barrier synchronization
 - If multiple threads write to same variable in a given phase, result is unpredictable
 - Popular in Europe
- Posix Threads (pthreads)
 - Portable, relatively low level
- OpenMP
 - Higher level standard, good for scientific programming

What base languages are used
for high performance?

Fortran

- “I don't know what the parallel programming language of the next decade will be, but I know that it will be named Fortran.”
 - F77 (sequential)
 - F90 (includes some data parallelism)
 - Fortran-D (SPMD; has directive for distributing arrays to grid of processors)
 - HPF (High Performance Fortran, outgrowth of F90 and Fortran-D)
- Almost all the programs that run for zillions of hours at SDSC are written in Fortran
- But Livermore develops many C programs

What's so great about Fortran

- Compilers can produce good code
 - alias analysis
 - 2D arrays
- Legacy code: Many important programs written in Fortran
- Scientists are comfortable with it

What's so terrible about Fortran

- Missing language features
 - structures
 - memory management (until fairly recently)
- Dusty decks: Many important programs written in obscure Fortran that no one understands.
- Computer scientists don't know it.

Fortran vs. C

```
DO I = 1, N
  A(I) = B(I)
ENDDO
```



```
CL.8:
L4A    gr0=b(gr5,4)
L4A    gr6=b(gr5,8)
L4A    gr7=b(gr5,12)
L4AU   gr8,gr5=b(gr5,16)
ST4A   a(gr4,8)=gr6
ST4A   a(gr4,4)=gr0
ST4A   a(gr4,12)=gr7
ST4U   gr4,a(gr4,16)=gr8
BCT    ctr=CL.8,
```

```
for (i=0; i<N; i++) {
  b[i] = a[i];
}
```



```
CL.6:
ST4U   gr4, (*)int(gr4,4)=gr24
L4AU   gr24,gr3=(*)int(gr3,4)
BCT    ctr=CL.6,
```

Fortran vs. C - what's going on??

- C prevents compiler from unrolling code
 - A feature, not a bug!
 - User may want `b[0]` and `a[1]` to be same location
 - tricky way to set `a[n] = ... = a[1] = a[0]`
- Most C compilers don't try to prove variables can't be aliased
 - `a` and `b` were `malloc`-ed in this example
- Fortran doesn't allow arrays to be aliased
 - Unless explicit, e.g. via **EQUIVALENCE**

Fortran vs. C - does it matter??

Yes - Fortran is 1.1 cycle per load-store
- C code is 2.2 cycle per load-store

(Experiments run on Blue Horizon)

No - you could get the "Fortran" object

code from:

```
for (i=0; i<N-3; i+=4) {
    b0 = a[i+0];    b1 = a[i+1];
    b2 = a[i+2];    b3 = a[i+3];
    b[i+0] = b0;    b[i+1] = b1;
    b[i+2] = b2;    b[i+3] = b3;
}
for ( ; i<N; i++) {
    b[i] = a[i];
}
```

Other languages

- C
 - Gaining popularity
- C++
 - not very portable; lots of overhead
- Java
 - Many language features make high performance difficult, but IBM's Jalapeno and other efforts are credible
- ZPL (Snyder, UW), Titanium (Yelick, UCB), ...
 - Computational scientists are reluctant to risk their career on flaky creations by computer scientists.

Granularity

Embarrassingly Parallel Very little communication
(e.g. just at beginning and end.)

Coarse-grained programs perform lots of
computation between communication steps.

Fine-grained ones communicate frequently.

Suggests definition:

Granularity = Average compute time between communications
= compute time / number of communications

Virtual Shared Memory

- VSM gives shared memory programming on distributed address space computers
- Proprietary VSM models provided by HP/Convex and Cray (T3D&E) on their physically-distributed MPP's.
- TreadMarks provides virtual shared memory environment for network of workstations
- HPF and other languages give shared memory programming environment on MPP's.

Message Passing on SMP's

- Many shared address space and multi-tiered machines provide MPI implementations.
- Communication has much lower latency than MPI on distributed address space machines.
- Enhances portability.
- Great for multi-tiered machines
 - It's easier to program a multi-tiered machine using only MPI rather than MPI+threads
 - And (so far, at least) performance is better!