

Parallelism and Programming Basic Models and Architectures

Nick Maclaren

University of Cambridge

`nmm1@cam.ac.uk`

The Cambridge-Cranfield
High Performance Computing Facility

<http://www.hpcf.cam.ac.uk>

20th June 2005

Technical Disclaimers

This is from the HPC viewpoint
Maximise calculations/real-second
I.e. an earlier date for the answer

Not from the Web server or database
POSIX threads used for asynchronicity
Plus some independence of failure

Will attempt to explain issues
Only reliable prediction is change
Current approaches still specialist

Haven't yet studied Fortress in detail

Why Does this Matter?

Applications have models, too

Use systems not fight them

Not all combinations make sense

Becomes feasible to code and debug

Fewer system/compiler problems

And much more efficient!

Little conflict with portability

Program for a parallelism model

Impractical to program for all models

About to see a paradigm shift

Niagara, Rock, IBM/Sony Cell, etc.

Intel hasn't responded, but will

Stratospheric Overview

This viewpoint not only valid one

Two correlated dimensions

One dimension is communication

Vector is almost synonym of SIMD

Then implicit data sharing
I.e. automatically shared memory

Then explicit data sharing
I.e. using synchronisation primitives

And lastly message passing

Another is interaction methodology

Parallel operations (SIMD)

Master/slave restrictive but easy

Start with lockstep (SPMD) operation

And then there are dataflow models

OpenMP/Fortress etc. are hybrids

Vector (SIMD) Approaches

Single Instruction, Multiple Data

True vector systems are vanishing

Cray, NEC and Fujitsu are last vendors

Easy and efficient for vector codes

Hitachi S-3600 delivered 75% of peak

SSE-3, AltiVec etc. are widespread

3-D rotations, complex operations, etc.
Optimised libraries, just as on SPARC

Not enough to affect programming model
Some compound operations run faster

Write clean code, set flag and ignore

ICL DAP, BBN Butterfly another approach

Many CPUs running same code, lockstep

May return, sometime, as 'active memory'

Implicit Data Sharing

Essentially the POSIX threads model
Thread data automatically kept in step
A near-total disaster for HPC

Shared data must not be optimised
Problem is compiler can't analyse code
In C/C++/Java most data can be shared
Better in Fortran, but still not good

Not soluble in existing languages
30 years of research has got nowhere
Can argue is theoretically insoluble
Problem is analysis of aliasing

An Example of the Horrors

```
extern int a; extern double b, c;
```

Thread A:

```
while (1) {while (a == 0) ; b = 2.0*c; a = 0;}
```

Thread B:

```
while (1) {while (a == 1) ; c = 0.5*b; a = 1;}
```

Real code is less simple, but similar

Java has problem, but says is user error

Fortress draft is unclear about this

Explicit Data Sharing

Essentially OpenMP and Java model

Data assignment is explicit

Multiple reader OR single writer
OR scatter and reduce

Explicit synchronisation points

Compiler has a chance to optimise

Aliasing is worst debugging problem

Fortran/Java bad, C/C++ much worse

Simple debugging and tuning is easy

No effective tools for hard problems

Theoretically, compilers could do better

Message Passing

This is the MPI and PVM model

Data not shared between threads

Explicit data communication only

Only possible communication on clusters

Can use shared memory for performance

TLBs, false sharing etc. problems on SMP

Gigabit Ethernet now fairly good

17 μ sec and 100 MB/sec on Opteron

1–2 μ sec and 500+ MB/sec for \$\$\$

Single-sided communication a red herring

Usually need handshake to say data arrived

Designing Applications

Choose model before designing code

KISS – Keep It Simple, Stupid

Constraints help to get things working

Master and slave model is the simplest

Used for setup, I/O, termination

Also farmable codes (e.g. Monte-Carlo)

Doing your own thing is fiendish

Non-alias synchronisation order problems

Deadlocks, wrong answers, livelocks ...

Better use either lockstep or dataflow

Any structured design model is OK

Orthogonal to shared memory or messages

Lockstep (SPMD) Operation

Single Program, Multiple Data

Essentially the BSP model

Also used by MPI collectives

Single program doesn't mean much

```
switch (tid) { ... }
```

But communication is lockstep

All threads compute with local data

Then all threads synchronise data

Fairly easy to code, debug and tune

More efficient than might appear

Adding barriers often saves time!

Dataflow Models

Very old models, sadly neglected
Execution units, with data piped through

Can run as soon as all inputs ready
Must run before any dependent units

Some designs statistically analysable

Others have detectable deadlock

Natural model for many applications

Personal prejudice: I think that way

Petri nets are very closely related

Difference is mostly terminological

Petri nets have a lot of theory to use

Enough flexibility to hang yourself

Enough discipline to help debugging

Variants of Dataflow Models

One old, good form is ‘sea of threads’

Split program into many tiny units

N servers run any executable unit

Leads to provably good efficiency

Trivial to do for functional languages

Fortress seems to use this model
Smalltalk is a message-based variant

Streaming modules is also good

Think of looped Unix pipes on steroids

Issue is avoiding avoidable deadlock

Unix/POSIX/UDP/TCP have serious “gotchas”

Mainframe/ancient (MVS/VMS) systems better

But can really fly when implemented right

‘Transparent’ Models

HPF, OpenMP, Fortress etc. models

Compiler distributes data and threads

Coder describes intent, not actions

Fairly easy to code, if kept simple

Debuggability needs discipline

Tuning can be easy, or a nightmare

Problem is that it hides system issues

Cache line sharing etc. main problem

Housekeeping overheads can be serious

Advanced tools are poor or absent

Question: are these problems soluble?

Answer: nobody has solved them yet

Where Will We Be in 2010?

MPI will hold position at high end

People will build libraries on top

Mostly master/slave, lockstep or dataflow

Question is 4- to 16-way boxes
Plus Niagara, Rock, Cell ...

Need simple, efficient, flexible model

Fortran/C/C++/Java not good bases

Hundreds of attempts – none worked

OpenMP most successful so far

Will Fortress make a breakthrough?

Will Smalltalk make a recovery?

Or Petri nets or other dataflow?

Possibly something as yet unknown

May you live in interesting times

References

Bulk Synchronous Parallel (BSP)

<http://www.bsp-worldwide.org>

The Fortress Language Specification

research.sun.com/projects/plrg/fortress0618.pdf

Message Passing Interface (MPI)

<http://www-unix.mcs.anl.gov/mpi>

<http://www.mpi-forum.org>

<http://www.open-mpi.org>

OpenMP

<http://www.openmp.org>

Parallel Virtual Machine (PVM)

http://www.csm.ornl.gov/pvm/pvm_home.html

Petri Nets (description)

http://en.wikipedia.org/wiki/Petri_net

Some random Smalltalk pages

<http://www.gnu.org/software/smalltalk/smalltalk.html>

<http://www.squeak.org>

Shared Memory Parallelism (SMP)

Following foils were drafted, but not used

Background on communication techniques

Included in case they are interesting

Shared Memory Parallelism (SMP)

Problem is register/memory coherence

Optimisation uses registers, preloading etc.

But must not break ordering assumptions

Needs compiler to do this, not library

Needs cache-coherent SMP hardware

Incoherent memory used to be common

Only for optimising messages and pipes

May return – but not as programming model

Register coherent systems have been built

Seriously unscalable – not coming back

Shared Memory Systems

Multiple cores and CPUs very similar

Shared cache and TLBs at some level

Today, SGI leads at 1024 – Sun is at 144

Workstations will be 4- to 8-way by 2007

32- or 64-way on Niagara etc.

Cache line sharing one main problem

TLBs, false sharing etc. also problems

Memory bandwidth other main problem

Hyperthreading (SMT) not interesting

Shares CPU pipelines, not optimisable

Almost always slower for HPC

Why Use SMP?

Can incrementally tune serial code

Faster primitives (e.g. BLAS)

Vector code is easy to convert

Not easy to tune, but feasible

Can share file reading automatically

Data transfer is constant time in size

Can simplify higher-level models

Dataflow, BSP and many others

Useful for ‘sea of threads’ (see above)

Often easier to manage (one machine)

4-way plus on workstations from 2007

Why Use Message Passing?

Can run on commodity clusters

Also runs very well on SMP systems

And sometimes on workstation networks

Advanced debugging and tuning is easier

Can instrument communication calls

For both tuning and checking

Fairly easy to analyse theoretically

Can scale almost indefinitely

1024 is normal, 1048576 is feasible

Will continue to dominate leading-edge

Hardware and System Issues

Binding threads to CPUs is often good

For lockstep, not sea of threads

Often only for processes, not threads

Don't forget to bind memory as well

But can rarely control physical location

Randomisation better for shared memory?

Often need to spin on communication

Only because sleeping invokes kernel

Can get 100% CPU during communication

Universal modern hardware/system misdesigns

Thread scheduling should be unprivileged

Should be able to request memory layout

Tuning as much trouble as designing code

Can be worse for shared memory codes