

# Parallel Programming

## *Options and Design (Part II)*

Nick Maclaren

**nmm1@cam.ac.uk, ext. 34761**

May 2011

# Summary

Topic has been (**slightly artificially**) split into two  
**Simple solutions** are mentioned as they arise

- **Reasons** for parallelism, and **basic design**  
Strengths and **weaknesses** of each approach  
**Thread pools**, **client-server**, **CPU farms** etc.  
Most important models of **HPC parallelism**
- Available parallel **implementations**  
**OpenMP** and other **shared-memory** models  
**PGAS** (**Fortran coarrays**, **UPC** etc.)  
**MPI** and other **message passing** models

# Beyond the Course

Email [scientific-computing@ucs](mailto:scientific-computing@ucs) for advice

[http://www-users.york.ac.uk/~mijp1/teaching/...  
.../4th\\_year\\_HPC/notes.shtml](http://www-users.york.ac.uk/~mijp1/teaching/.../4th_year_HPC/notes.shtml)

[http://www.hector.ac.uk/support/documentation/...  
.../userguide/hectoruser/hectoruser.html](http://www.hector.ac.uk/support/documentation/.../userguide/hectoruser/hectoruser.html)

See “[References and Further Reading](#)”

[http://www.epcc.ed.ac.uk/library/documentation/...  
.../training/](http://www.epcc.ed.ac.uk/library/documentation/.../training/)

# Contents

Some commonly misunderstood **terminology**

**Distributed memory**, **MPI**, etc.

**Shared memory**, **POSIX threads**, **OpenMP** etc.

**Fortran coarrays**, **UPC** etc.

**Memory models** and similar cans of worms

**Kernel scheduling** issues (for sysadmins only)

# SIMD – The Easy Case

**SIMD** means **Single Instruction, Multiple Data**

I.e. a **serial program** runs with **parallel data**

Think of a **vector system** when you say this

This includes **indexed** and **masked** arrays

**Great** advantage: **code** and **debug** just like **serial**  
**Optimisation** of it is well-understood and **automatic**  
Actually **implemented** as simplest case of **SPMD**

It also includes **MMX**, **SSE** and **AltiVec**

- But regard them as part of **serial optimisation**

**GPUs** will be described later

# MIMD

MIMD means Multiple Instruction, Multiple Data

All modern parallel systems are MIMD

And almost all parallel languages and libraries

You can run SIMD designs on MIMD systems

- So the term isn't practically useful nowadays  
Incorrectly used to mean distributed memory

Most MIMD languages are actually SPMD

# SPMD (1)

**SPMD** means **Single Program, Multiple Data**  
I.e. exactly the **same program** runs on all **cores**  
But **programs** are allowed **data-dependent** logic

So each **thread** may execute different **code**  
Can test the **thread identifier** or do it explicitly

```
#pragma omp sections
{
    #pragma omp section
    { ... }
    #pragma omp section
    { ... }
}
```

# SPMD (2)

The more like **SIMD** it is, the **easier** it is  
At least for **coding**, **debugging** and **tuning**

Minimise **thread-specific** code as far as possible  
It isn't a major problem, except for **efficiency**  
It makes **tuning** quite a lot harder

- Watch out for **locking** and **communication**  
Best to leave all **communication** to **compiler**  
**Locking** indicates **SPMD** may be wrong model



# Multiple Programs

This is where each **process** runs **independently**

**MPI** is a widespread and classic example

Term **MPMD** is almost never used, but could be

- But they **usually** run a **single executable**

So **some** people call them **SPMD** models

And, as implied, **some** people don't ...

- This is not really a meaningful debate

There is a **continuum** from **SIMD** onwards ...

to each **process** running **different executables**

# SMP

**SMP** stands for **Shared Memory Processor**  
All **threads** can access all the **memory**

- It does **NOT** guarantee **consistency!**  
We will return to this **minefield** later

It used to mean **Symmetric Multi-Processing**  
That use still occasionally crops up

# NUMA

**NUMA** means **Non-Uniform Memory Architecture**

I.e. not all **memory** is easily **fast** to access

Usually, some sort of **memory/CPU** affinity

All large **SMP** systems are **NUMA** nowadays

Even larger **Intel** and **AMD Opteron** ones are

- **Caches** make even **Intel** look like it

Details are too complicated for this course

# Distributed Memory (1)

This refers to separate CPUs (e.g. clusters)  
Including separate processes on SMP systems

- Each thread is a completely separate process  
No shared memory, real or virtual – see later  
Code and debug each process as if serial
- Communication is by message passing  
That is simply a specialist form of I/O  
Usually a library, but may use language syntax

# Distributed Memory (2)

Can **trace** or **time** the **message passing**  
That is how to do **debugging** and **tuning**  
It is **complicated**, but no harder than **serial**

Some **parallel debuggers** and **tuning tools**  
Mostly for **MPI** – even built-in to **libraries**

- Hardest problem is **data distribution**  
Each **process** owns some of the **data**  
But **other** processes may need to **access** it

# Message Passing

Many **interfaces** used in **commerce**, **Web** etc.  
**CORBA**, **SOAP** etc. – let's ignore them

Some **languages** – **Smalltalk**, **Erlang** etc.  
Few used outside **computer science** research

**MPI** (**Message Passing Interface**) dominates  
A **library** callable from **Fortran** and **C**  
**Bindings** available for **Python**, **Java** etc.

- Essentially all **HPC** work on **clusters** uses **MPI**

# Take a Breather

That has covered most of the **terminology**

Will mention a few high-level **design guidelines**  
About how to **structure** your application

Then will cover what **MPI** can do

Then move onto the **shared memory** morass  
Sorry, but that is very complicated

# Languages

Next slide is a summary of the **language landscape**

**Automatic** parallelisation is the **holy grail**

i.e. where the **compiler** does it all for you

**40** years of research has got **almost nowhere**

**Limited** compiler support for **shared memory**

But you **almost always** have to program it **explicitly**



# Language Landscape

Shared

Memory

Distributed

Automatic

Parallelism

This is largely an open research area;

i.e. we don't know how to do it,

and deliver a useful

language

OpenMP

Cilk

C++

Coarrays/UPC

MPI

Programmed

# Designing for Distribution (1)

A good **rule of thumb** is the following:

- Design for **SIMD** if it makes sense
- Design for **lock-free SPMD** if possible
- Design as **independent processes** otherwise

For **correctness** – order of increasing **difficulty**

**Not** about **performance** – that is different

**Not** about **shared** versus **distributed** memory

- **Performance** may be the **converse**

**There Ain't No Such Thing As A Free Lunch**

# Designing for Distribution (2)

- Next stage is to design the **data distribution**  
**SIMD** is usually easy – just chop into **sections**
- Then work out need for **communication**  
Which **threads** need which **data** and **when**  
Do a back of the envelope efficiency estimate
- If too slow, need to **redesign** distribution  
Often the stage where **SIMD** models rejected

# Designing for Distribution (3)

- Don't skimp on this design process

Data distribution is the key to success

- You may need to use new data structures

And, of course, different algorithms

- Above all, KISS – Keep It Simple and Stupid

Not doing that is the main failure of ScaLAPACK

Most people find it very hard to use and debug

# MPI

This was a **genuinely open** specification process  
Mainly during the second half of the **1990s**

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

**MPI-1** is basic facilities – all most people use  
Most people use only a **small fraction** of it!

**MPI-2** is **extensions** (other facilities)  
Also includes the **MPI 1.2** update

**MPI-3** has fairly recently been standardised  
**Non-blocking collectives** and **Fortran 90** support

# MPI-1 (1)

- Bindings for Fortran and C

Trivial to use for arrays of basic types

Tricky for Fortran 90 and C++ derived types

- Point-to-point and collective transfers

Latter are where all processes interoperate

Can define process subsets for communication

- Blocking and non-blocking transfers

Not always clear which are more efficient

Issues are beyond scope of this course

# MPI-1 (2)

Two **open source** versions – **MPICH** and **OpenMPI**  
Most vendors have own, inc. **Intel** and **Microsoft**

Wide range of **tuning** and **debugging** tools

Mostly **commercial**, but not all

Or can use built-in **profiling interface**

Easy to use and can help with **debugging**

- Not ideal, but consensus is pretty good

Some **higher-level interfaces** built on top of it

# I/O in MPI

Applies to all **distributed memory** interfaces

No problem with **separate files** for each **process**

Or if they all **read** the same file at once

- Provided that the **file server** is adequate!

Problems occur with **stdin**, **stdout** and **stderr**

Immense variation in ways that is implemented

- Best to do their I/O **only** from **primary process**

Use **MPI** calls to transfer data to and from that



# Some Sordid Details

There may be a **separate thread** for I/O

Or the **master thread** may handle **stdout**

Or each thread may **write directly**, and **lock**

- All can cause severe **scheduling** problems

**stdin** may be **copied** or **shared**

**stdout** and **stderr** may interleave badly

- Causes programs to fail when changing systems

# MPI-2 (1)

- Not all **implementations** support all of this  
Use only the **extensions** you actually **need**
  - Miscellaneous extensions (gaps in **MPI-1**)  
Some of these are **useful**, but rarely **essential**
  - **One-sided** communications  
Some people find these much easier, but I don't  
Explaining the issues is beyond this course
- It used to have **C++** bindings, but no longer does  
But using the **C** bindings in **C++** works, too

# MPI-2 (2)

These **extensions** are less likely to be available

- **Dynamic process** handling – to supersede **PVM**  
Few, if any, people use this, and I don't advise it
- **Parallel I/O** – direct and sequential  
Few people here do **I/O intensive HPC**

# PVM – Parallel Virtual Machine

This was a predecessor of **MPI**

Based around a **cycle stealing** design

But with **inter-processor** communication, too  
**CPUs** could leave and join the **processor pool**

- It's **effectively dead** – thank heavens!  
A pain to **use**, with **unrealistic** assumptions  
A positive **nightmare** to **administer** and **debug**

**MPI-2** includes all of **PVM**'s facilities

# Shared-Memory Terminology

**Atomic** means an action happens or doesn't  
It will never overlap with another **atomic** action

**Locked** means software makes it look **atomic**  
Usually a lot less efficient, and can **deadlock**

A **data race** is when two **non-atomic** actions overlap  
The effect is completely **undefined** – often chaos

**Synchronisation** is coding to prevent **data races**

# Shared Memory (1)

All **threads** have access to all **memory**

- Unfortunately, that isn't exactly right ...

There are three general classes of shared memory

- **Fully shared** within single **process**

As in **POSIX threads** and **OpenMP**

- **Shared memory segments**, **POSIX mmap** etc.

Shared between **processes** on same **system**

- **Virtual shared memory** (rarely called that)

**Cray SHMEM**, **PGAS**, even **BSP** etc. – see later

# Shared Memory (2)

Shared memory has memory model problems  
Will return to that in more detail later

If two threads/processes access same location:

- Either all accesses must be reads
- Or both threads must be synchronised
- Or all accesses must be atomic or locked

Details depend on the interface you are using

- Critical to read specification carefully

# Shared Memory (3)

Updates may not transfer until you synchronise  
But they may, which is deceptive

Memory will synchronise itself automatically

- Now, later, sometime, mañana, faoi dheireadh

So incorrect programs often work – usually  
But may fail, occasionally and unpredictably

Makes it utterly evil investigating data races

- Any diagnostics will often cause them to vanish



# Fully Shared Memory

POSIX/Microsoft threads, Java and OpenMP  
No other interface is used much at present

Plus some computer science research, of course  
There are also a few specialist ones

Most SMP libraries implemented using OpenMP  
See later about the consequences of this

OpenMP is implemented using POSIX threads  
And Microsoft threads when relevant, I assume

# Shared Memory I/O

- **POSIX** is seriously **self-inconsistent**  
I/O is **thread-safe** (2.9.1) but not **atomic** (2.9.7)  
Can **you** guess what that means? I can't

Don't even **think** of relying on **SIGPIPE**

Most other **interfaces** are built on **POSIX**  
Some interfaces may implement I/O like **MPI**  
Warnings on that **may** apply here, too

- Do **all** your I/O from the **initial thread**

# OpenMP (1)

A **language** extension, not just a **library**

Designed by a **closed commercial consortium**

“**Open**” just means **no fee** to use **specification**

Dating from about **1997**, still **active**

<http://www.openmp.org>

Specifications for **Fortran**, **C** and **C++**

Most **compilers** have some **OpenMP** support

- This is the **default** to use for **SMP HPC**

Unfortunately the **specification** is ghastly

# OpenMP (2)

- The **compiler** handles the **synchronisation**  
Covers up problems in underlying **implementation**  
E.g. **ambiguities** in the threading **memory model**
- Mainly **directives** in the form of **comments**  
They indicate what can be run in parallel, and how  
Also a **library** of utility functions

**OpenMP** permits (**not** requires) **autoparallelisation**  
I.e. when the **compiler** inserts the **directives**  
Available in many **Fortran** compilers, rarely in **C**

# OpenMP (3)

- Easiest way of **parallelising** a **serial** program  
Can just modify the areas that take the most time
- Can usually mix **SMP libraries** and **OpenMP**  
Start with calls to **parallel library functions**  
And set **compiler options** for **autoparallelisation**
- Then use **SIMD** or **SPMD** directives  
**Finally**, worry about more **advanced parallelism**

Too good to be true? I am afraid so

# OpenMP (4)

- Inserting **directives** trickier than it **seems**  
Make even a **minor** mistake, and **chaos** ensues
- That is why I advise ‘**pure**’ **SPMD** mode  
No **synchronization**, **locking** or **atomic**  
Will get the best **diagnostics** and other help
- **Debugging** and **tuning** can be nightmares  
It is **MUCH** easier to avoid them in **design**

Too complicated to go into details here

# OpenMP Debugging (1)

- **Aliasing** is when two **variables** overlap  
And the **compiler** hasn't been told that  
Bugs often **show up** only when run in **parallel**
- Must declare **variables** as **shared** or not  
And obviously must declare that correctly!  
Note that **shared objects** need **synchronisation**
- Failure is often **unpredictably incorrect** behaviour
- **Serial debuggers** will **usually** get confused

# OpenMP Debugging (2)

- **Variables** can change **value** ‘for no reason’  
Failures are **critically** time-dependent
- Many **parallel debuggers** get confused  
Especially if you have an **aliasing** bug
- A **debugger** changes a program’s **behaviour**  
Same applies to **diagnostic code** or **output**  
Problems can **change**, **disappear** and **appear**
- Try to avoid ever needing a **debugger**



# OpenMP Tuning (1)

- Unbelievably, **tuning** is **much** worse
- Most compilers will help with **parallel efficiency**  
I.e. **proportion of time** in parallel (**Amdahl's Law**)  
Most users know that from their **initial design!**
- Below that, **hardware performance counters**  
Not easy to use and not available under **Linux**
- The **debugging** remarks also apply to **profiling**

# OpenMP Tuning (2)

- Can also lose a factor of **2+** in **overheads**  
**Manually** analyse the **assembler** for **efficiency**
- Worst problems are **scheduling** glitches  
You have **NO** tools for those!

Most people who try tuning **OpenMP** retire hurt  
[ I have succeeded, but not often ]

- Same applies to **POSIX threads**, incidentally  
and **Microsoft** and **Java** ones ...

# C++ Threads (1)

- Latest **C++ 2011** standard defines **threading**  
The design is OK, but **compilers** don't support it yet
- **Ordinary** memory accesses must not conflict  
Roughly, you must **write-once** or **read many**
- **Atomic** memory accesses impose **synchronisation**  
Default is **sequentially consistent** – so not scalable  
Beyond that is definitely for **experts only**

**Standard Template Library (STL)** is still **serial**  
It currently has only **low-level** facilities

## C++ Threads (2)

Inherited C facilities are full of **gotchas**

- Often **not behave** as you expect or simply **not work**  
Includes all I/O – so use from only one thread

Some bad ideas – e.g. **cross-thread exceptions**  
Issues too complicated for this course

- Using it will be **harder** than using **OpenMP**  
The C standard is copying it – if anyone cares!
- I really **don't recommend** it for most people  
Except for **small, encapsulated** primitives

# POSIX/Microsoft/Java Threads

- Using **threads** like **processes** usually works  
I.e. **minimal**, **explicit** thread communication  
Precisely how most **threaded applications** work

- Beyond that, is task for **real experts** only  
Morass of **conflicting**, **misleading** specifications  
With more **gotchas** than you believe **possible**

Mention some of the issues, usually in passing  
Details are too complicated for this course

- Please ask for help if you need it here

# Java Threads

The first version was a failure, and was redesigned  
Reports of the merits of the second one are mixed

[http://java.sun.com/docs/books/tutorial/...  
.../essential/concurrency/](http://java.sun.com/docs/books/tutorial/.../essential/concurrency/)

**Essential** to read sections **Thread Interference**  
and **Memory Consistency Errors**

Also applies to **POSIX** and **Microsoft**, of course

- Users of those should read those sections, too

# POSIX Threads

C90 and C99 are entirely serial languages  
Legal C optimisations break POSIX threads

Neither C nor POSIX defines a memory model  
Reasons why one is essential are covered later

No way of synchronising non-memory effects  
Not just I/O, but signals, process state etc.  
Even simple ones like clock() values and locales

[http://www.opengroup.org/onlinepubs/...  
.../009695399/toc.htm](http://www.opengroup.org/onlinepubs/.../009695399/toc.htm)

# Microsoft Threads

I failed to find a suitable **Web** reference  
Finding the **API** was easy enough

But I failed to find a proper **specification**  
Or even a decent **tutorial**, like **Java's**

Searching **threads "memory model"**  
From **\*.microsoft.com** had 167 hits, but ...

I have reason to think it is currently **in flux**  
Yes, I do mean that it is about to change



# Others

If you are really interested, try:

[http://en.wikipedia.org/wiki/Concurrent\\_computing/](http://en.wikipedia.org/wiki/Concurrent_computing/)

The following look potentially useful for scientists:  
Both have been used for production code

**Cilk** – possibly useful for **irregular problems**

Based on **C**, and needs **disciplined** coding

**Extended language** now supported by **Intel** compilers

**GNU Ada** – a largely **checked** language

Reported to be safest **open source** compiler

# GPUs

Extending GPUs to use for HPC

Will describe current leader (NVIDIA Tesla)

Hundreds of cores, usable in SPMD fashion

Cores are grouped into SIMD sections

Can be expensive to synchronise and share data

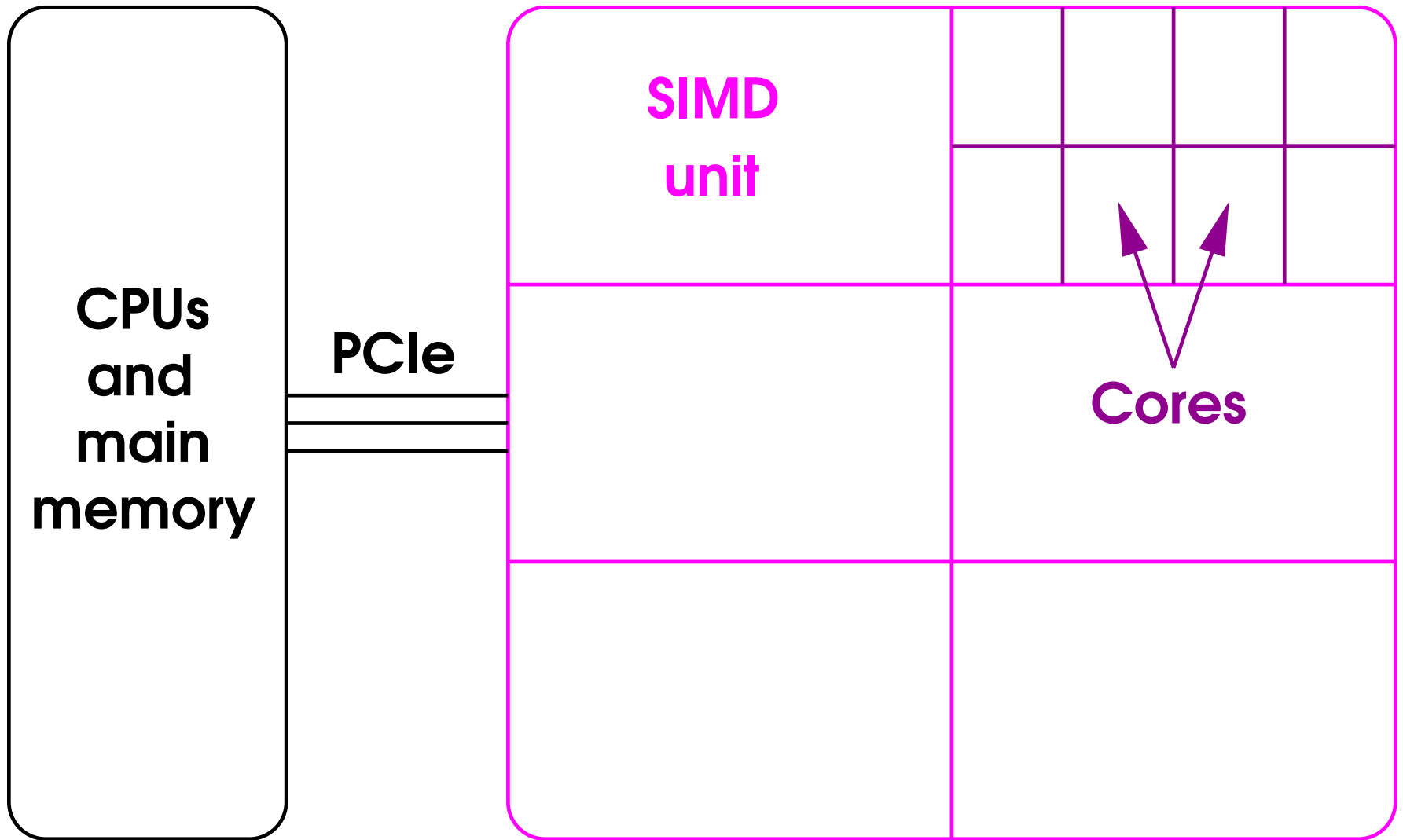
Can be 50–100 times as fast as CPUs

- Only for some applications, after tuning

And that is only for single precision code

NVIDIA Fermi should be better in this respect

# NVIDIA GPU Design



# CUDA, OpenCL, OpenAcc

CUDA, OpenCL based on extended C99/C++  
Some Fortran prototypes are available for OpenAcc

Programming is reported to be fairly easy  
Rules for sharing memory are trickiest part

- Tuning is where the problems arise  
Can be anywhere from easy and fiendish  
Critically dependent on details of application
- Don't forget CPU  $\leftrightarrow$  GPU transfer time

# Precision Issues

Graphics is numerically very **undemanding**

**Double precision** is **very** much slower

- But most **scientific codes critically** need it!

**Watch out!**

There are some **techniques** to help with this

- Dating from the **1950s** to **1970s**

Most known to (some of) the **GPU-using** community

Look for books on **numerical programming** of that date

Or ask one of the people who was active in that area

# Shared Memory Segments (1)

A way of sharing **memory** between **processes**  
**Almost** always on a single **SMP** system

**POSIX mmap**, '**SysV shmem**' (**POSIX shmat**) etc.  
Surprising, **NOT** most variants of **Cray SHMEM**

- Best regarded as **communication mechanisms**  
Several are actually **memory-mapped I/O**

# Shared Memory Segments (2)

Synchronisation may need to be in both processes  
Or just in sender or just in receiver

POSIX §4.10 is seriously ambiguous – do both

Once transferred, can use just as ordinary memory

- Don't update if in use by another process

It is your responsibility to obey constraints

Often used to implement other interfaces

Including message-passing ones, like MPI!

# Virtual Shared Memory

This is **SPMD** with **separate processes**  
Possibly even running on different **systems**

You program it a bit like **true shared memory**

- But **synchronisation** is **mandatory**

Getting that **wrong** causes **almost all** bugs

But **some systems** **may** transfer automatically

- You can't **rely** on **transfers** being **queued**

Except, with the **right options**, for **BSP**



# Cray SHMEM and Variants

- This is actually **one-sided** communication  
I.e. **one** process calls **put** or **get**  
All **threads** must call **barrier** to synchronise
- May assume same **local** and **remote** addresses  
Or may need to call a **mapping function**

Use them much like **shared memory segments**

- Be **sure** to check **correct** documentation  
A zillion **variants**, even just on **Cray** systems

# BSP (1)

Stands for **Bulk Synchronous Parallel**

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

- **Similar**, but much **simpler** and **cleaner**  
Designed by **Hoare**'s group at **Oxford**
- **Series of computation steps and barriers**  
**All communication** is done at the **barriers**  
**All threads** are involved (i.e. no **subsets**)
- Failures like **deadlock** cannot occur  
Considerably simplifies **debugging** and **tuning**

## BSP (2)

It is used much like Cray SHMEM

I.e. it uses `put`, `get`, `map_address` calls

The data are queued for transfer

- Advantages derive from its restrictions  
Flexibility has its cost (TANSTAAFL)

Only a few people have used it at Cambridge

I haven't, but it is so simple I know I could!

- Consider using its design, at least  
Please tell me of any experiences with it

# PGAS (1)

May stand for **Partitioned Global Address Space**  
Or **Parallel Global Array Storage**, or ...

[ Don't confuse with **Parallel Genetic Algorithms** ]

- **Shared arrays** are spread across threads

Each **block** is owned by exactly one **thread**

Typically, **scalars** are owned by **thread zero**

- All **threads** can access all **memory**

The **compiler** handles the **data transfer**

All **accesses** **must** be **synchronised**

# PGAS (2)

- Usually provided using **language extensions**  
Claimed by **believers** to be much **easier to use**

Very **trendy** at present – the current **bandwagon**  
Mostly used to publish research papers on it

- Little used outside **ASCI**  
**ASCI = Accelerated SuperComputer Initiative**  
USA government's '**Son of Star Wars**' project

Some people in **Cambridge** share codes with **ASCI**

# HPF

HPF stands for High Performance Fortran  
First attempt at a standard for parallel Fortran  
<http://hpff.rice.edu/>

Originated about 1992, never really took off  
Superseded by OpenMP by about 2000

- You may still find some across code written in it
  - Some compilers still have HPF options
- Of course, they don't always actually work ...
- It's dead, a zombie, a late parrot – don't use it

# Fortran Coarrays (1)

Being driven by **Cray/ASCI/DoD**

Some of it available on **Cray** systems since **1998**

- In latest **Fortran 2008** standard (**2011**)

I am not a great fan of it, but was closely involved

<ftp://ftp.nag.co.uk/sc22wg5/...>  
[.../N1801-N1850/N1824.pdf](ftp://ftp.nag.co.uk/sc22wg5/.../N1801-N1850/N1824.pdf)  
<http://www.co-array.org/>

Starting to appear in compilers

# Fortran Coarrays (2)

**Cray** and **Intel** have released, **IBM** will soon  
Most other compilers will follow in due course

**g95** supports **some** of the proposal  
Only an **intermittently active** project, though

The **Rice U. open-source** project is a bit dubious  
<http://www.hipersoft.rice.edu/caf/>

**gfortran** is currently adding syntax checking  
Starting to work on a simple MPI implementation



# Fortran Coarrays (3)

Threads are called **images**

Code looks like the following:

```
real, dimension(1000)[*] :: x,y  
x(:) = y(:)[q]
```

The **square brackets** index the **thread**

Indicate you are **copying** across **threads**

On the **owning image**, you can omit them:

```
x(:) = y[9](:)+y(:)
```

# UPC (1)

Unified Parallel C – a C99 extension

A lot of activity, mainly in USA CS depts

Started in 1999 – open-source compiler since 2003

Lots of predictions of future successes

Little evidence of actual use, even in ASCI

Specification even more ambiguous than C99

However, it does define its memory model

<http://upc.gwu.edu/> and <http://upc.lbl.gov/>

# UPC (2)

Lots of scope for making obscure errors

The very thought of debugging it makes me blench

Syntax is just like C99 arrays – e.g. `a[n][k]`

Semantics and constraints are not like arrays

The '`[n]`' indexes the thread for shared objects

Uses of `a[n][k]` undefined if not on thread '`n`'

Must call library functions to copy between threads

- However, I do NOT recommend using it  
Often recommended on basis of no experience

# Language Support

Fortran does very well, for a **serial** language  
Why **OpenMP Fortran** is the leader in this area

**C** was mentioned earlier, under **POSIX**

**C++** and **Java** were described earlier

**C#** is probably following **Java**

# Memory Models (1)

Shared memory **seems** simple, but isn't  
'Obvious orderings' often fail to hold  
Parallel time is very like **relativistic time**

Too complicated (and **evil**) to cover in this course  
Suitable key phrases to look up include:

Data Races / Race Conditions  
Sequential Consistency  
Strong and Weak Memory Models  
Dekker's Algorithm

# Main Consistency Problem

**Thread 1**

**A = 1**

**print B**

**Thread 2**

**B = 1**

**print A**

Now did **A** get set first or did **B** ?

**0** – i.e. **A** did **0** – i.e. **B** did

**Intel x86** allows that – yes, really

So do **Sparc** and **POWER**

# Another Consistency Problem

**Thread 1**

**A = 1**

**Thread 2**

**B = 1**

**Thread 3**

**X = A**

**Y = B**

**print X, Y**

Now, did **A**  
get set first  
or did **B** ?

**Thread 4**

**Y = B**

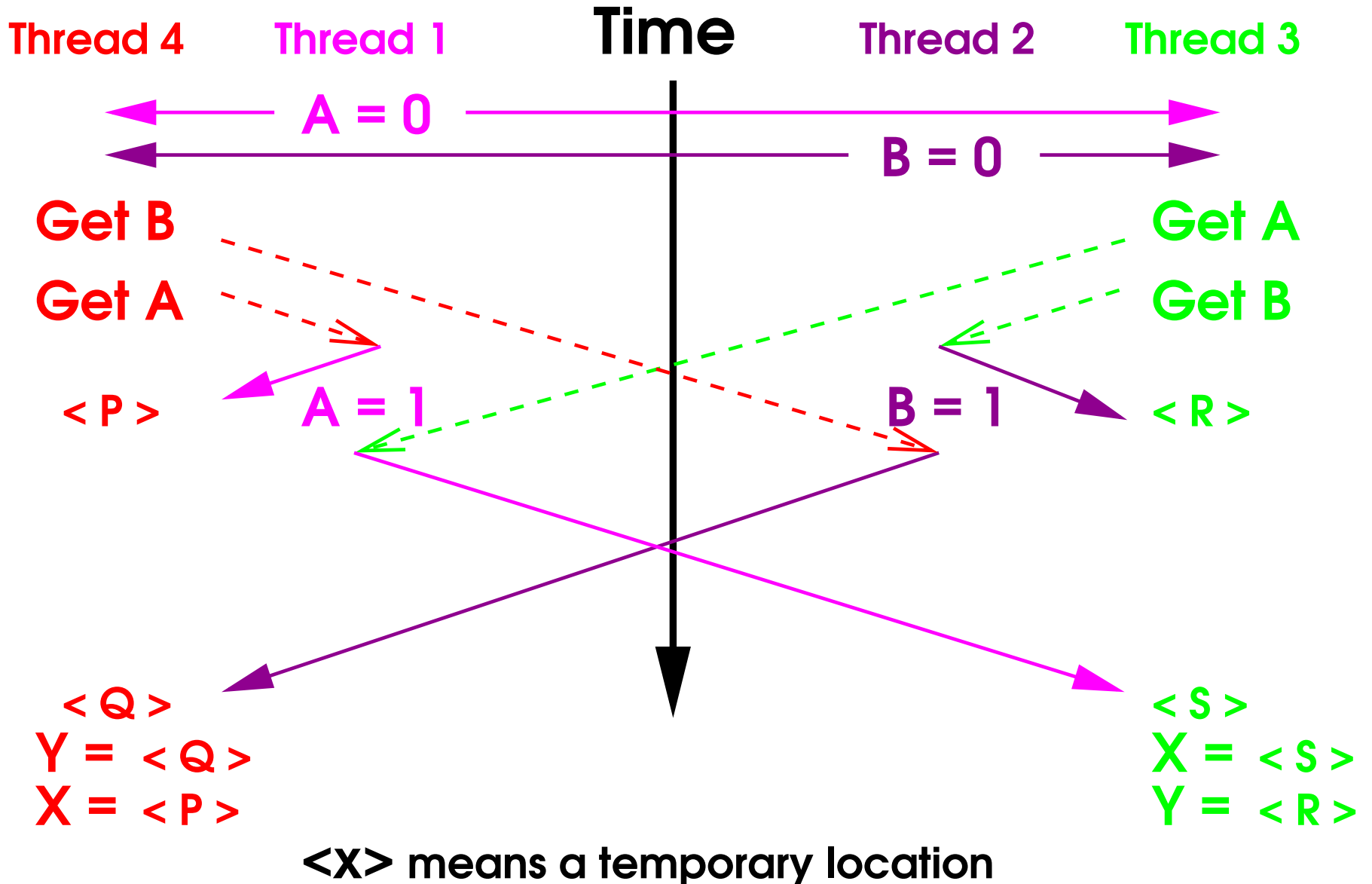
**X = A**

**print X, Y**

**1 0** – i.e. **A** did

**0 1** – i.e. **B** did

# How That Happens





## Memory Models (2)

- Easiest to use **language** that prevents problems  
No current one does that **automatically** and **safely**  
Some can **help**, if you code in a **disciplined** way  
Reasons for **OpenMP+Fortran** and **SIMD** design
- Next easiest solution is **explicit synchronisation**  
Don't assume **data transfer** is **automatic**  
This is model used by **Cray SHMEM** and **BSP**
- Beyond that, **KISS – Keep It Simple and Stupid**  
Check you don't **assume** more than is **specified**  
Even **Hoare** regards this as **tricky** and **deceptive**

# For Masochists Only

[http://www.cl.cam.ac.uk/~pes20/...  
.../weakmemory/index.html](http://www.cl.cam.ac.uk/~pes20/.../weakmemory/index.html)

Intel(R) 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1, 8.2 Memory Ordering

[http://developer.intel.com/products/...  
.../processor/manuals/index.htm](http://developer.intel.com/products/.../processor/manuals/index.htm)

Follow the **guidelines** here, and can ignore them

- Start to be **clever** and you had **better** study them

# Valid Memory Accesses

- Virtual shared memory is the **easy** one  
All you need to do is to **synchronise** correctly  
Trivial in **theory**, not so **easy** in practice
- True shared and segments are tricky  
You **don't** always need to **synchronise**  
But **when do you** and **when don't you**?

In **theory**, can **synchronise** like **virtual**

- That is how I recommend most people to do it  
**OpenMP** without **locks** does it automatically

# Atomicity (1)

Thread 1: <type> A; A = <value1>; A = <value2>;  
Thread 2: <type> B; B = A;

Will B get either <value1> or <value2>?

- Don't bet on it – it's not that simple

Probably OK, IF <type> is scalar and aligned  
But depends on compiler and hardware  
Structures (e.g. complex) are not scalar

- Best to use explicitly atomic operations  
These are language and compiler dependent

# Atomicity (2)

Killer is heterogeneous accesses of any form

- Don't mix explicitly atomic and any other
- Don't mix RDMA transfers and CPU access
- Don't even mix scalar and vector accesses  
SSE probably works, now, but might not
- Don't trust atomic to synchronise

# Cache Line Sharing

```
int A[N];  
Thread i: A[i] = <value_i>;
```

- That can be **Bad News** in critical code  
Leads to **cache thrashing** and dire **performance**

Each **thread's data** should be well **separated**  
**Cache lines** are **32–256** bytes long

- Don't bother for **occasional accesses**  
The code **works** – it just runs **very slowly**

# Kernel Scheduling

Following slides are rather esoteric  
Why **low-level** parallel **tuning** is not easy

They cover points that are normally ignored  
But are a **common** cause of inefficiency

- Try to **avoid** them, not to **solve** them  
Some hints given of how to do that

# Kernel Scheduler Problems

- In both shared memory and message passing  
Both on SMP systems and separate CPUs

Investigation can be simple to impossible

MPI on separate CPUs can be simple

Shared memory and SMP systems are nasty

Often need kernel tools (e.g. Solaris dtrace)

Very often tricky for programmer to fix

- May need to tweak system configuration



# Gang Scheduling (1)

Almost all HPC models assume gang scheduling

Details are system-dependent and complicated

- Principles are generic and very important

- Ideal is that each thread ‘owns’ a core

I.e. that the core is always available, immediately

Even a slight delay can have major knock-on effects

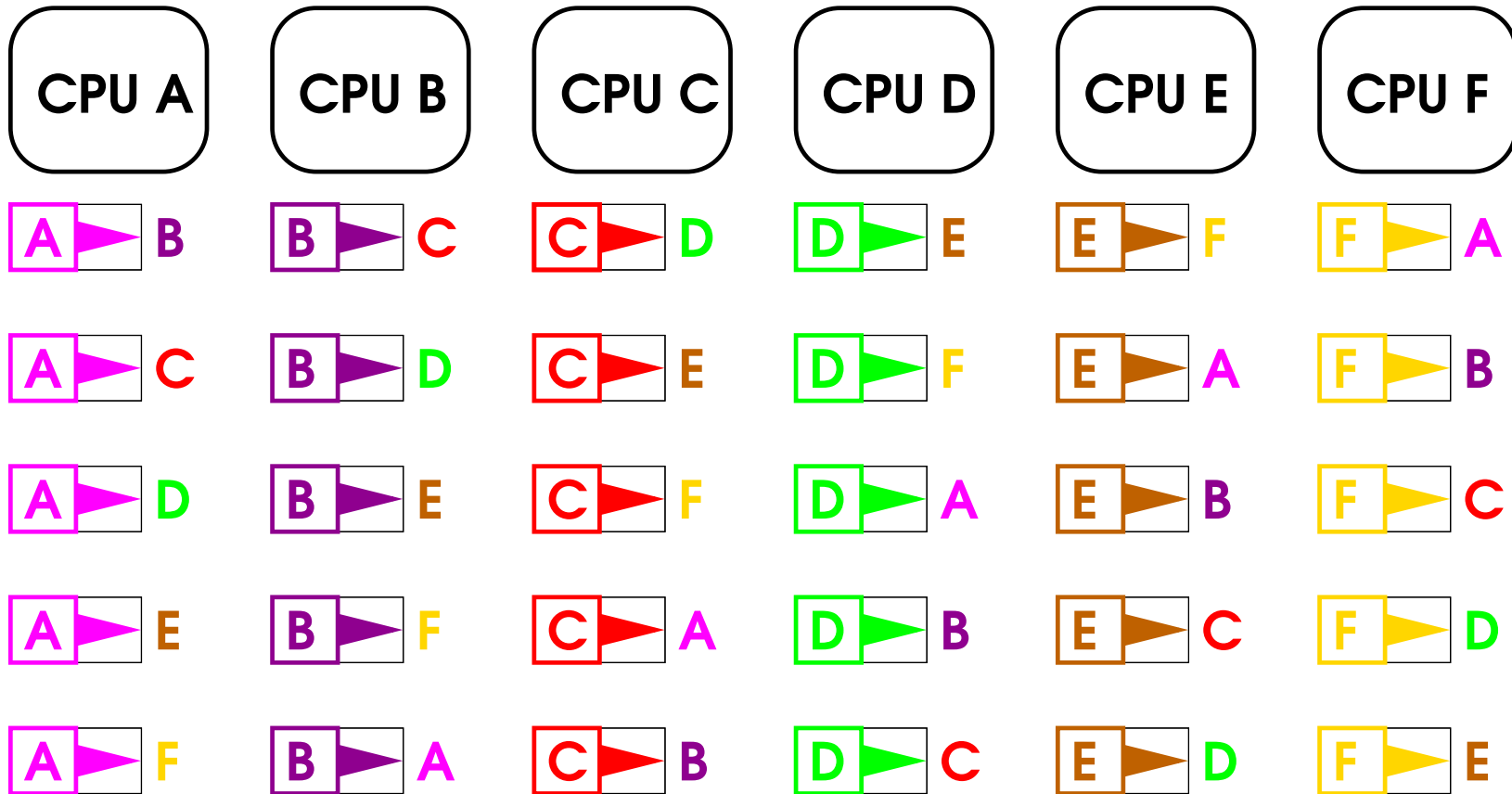
Most system tuning is ensuring  $\text{threads} \leq \text{cores}$

- Don’t make that impossible when programming

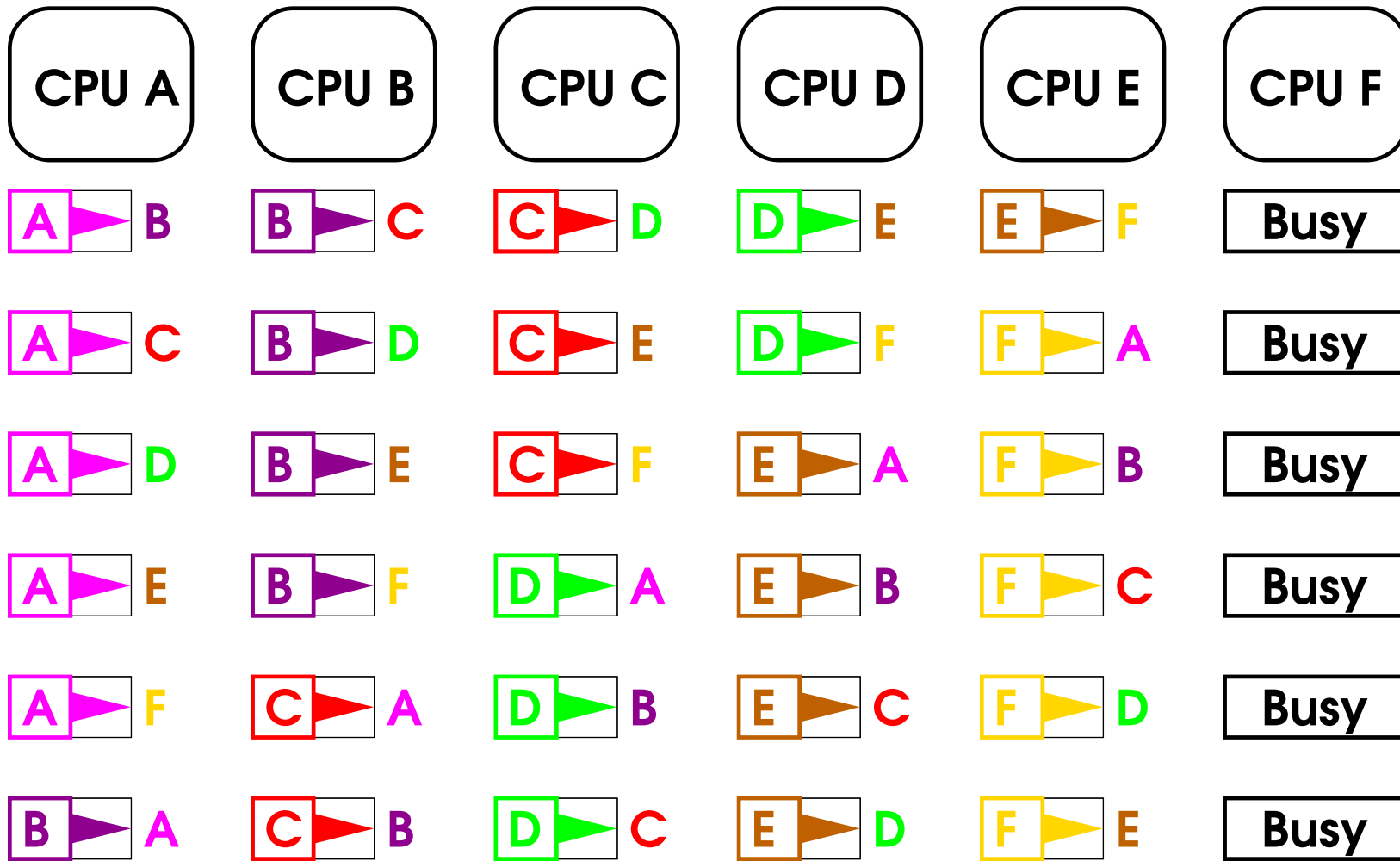
# Gang Scheduling (2)

- Avoid running other **daemons** or **interaction**  
Effect can be to reduce **effective number** of CPUs
- Remember there may be extra **controlling thread**  
Describing details is way beyond this course
- Don't want **threads** to wander between **cores**  
Their **active data** has to be copied between **caches**  
Can be caused by **too few CPUs** for **threads**

# Optimal all-to-all

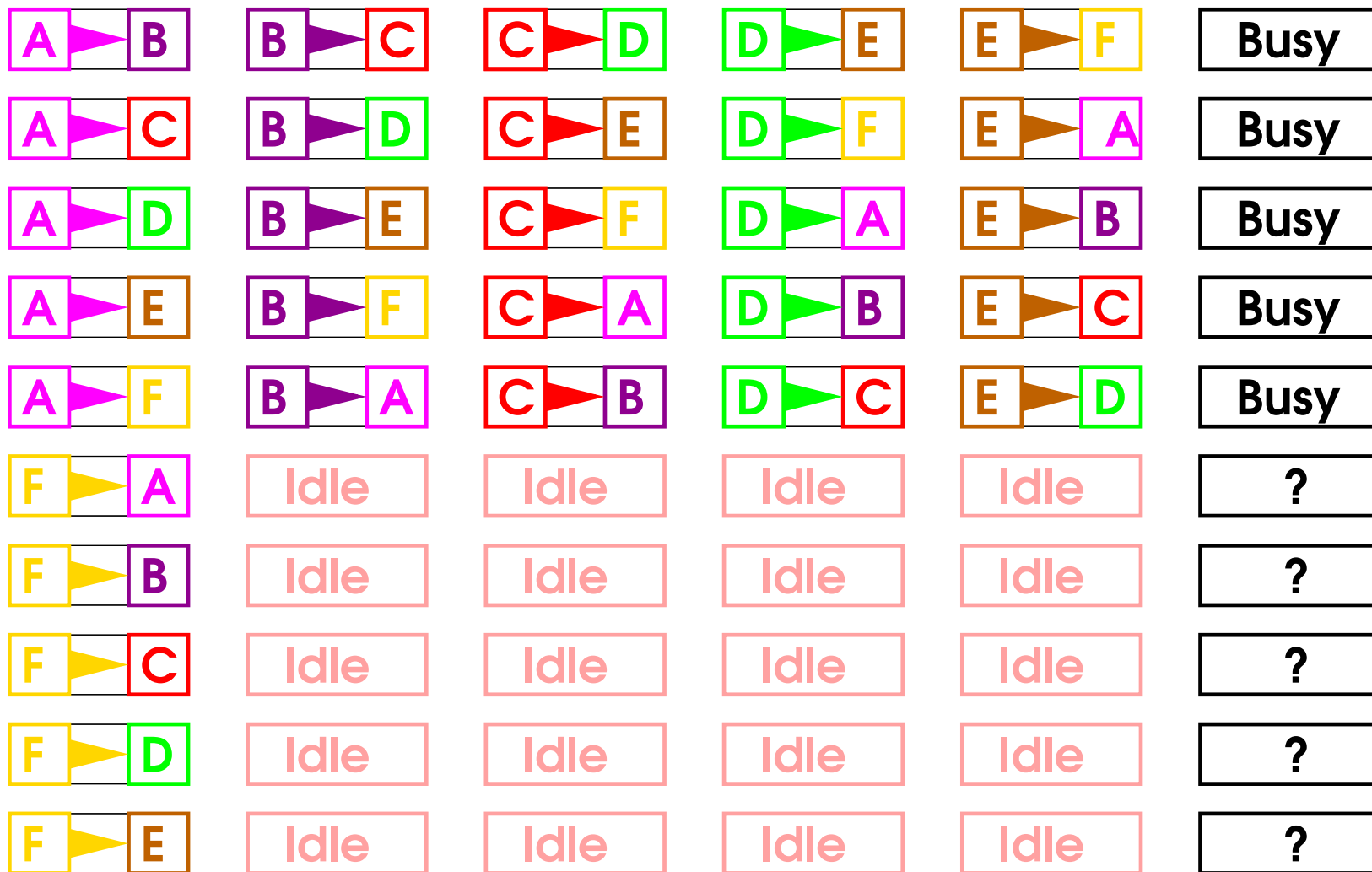


# Near-optimal all-to-all



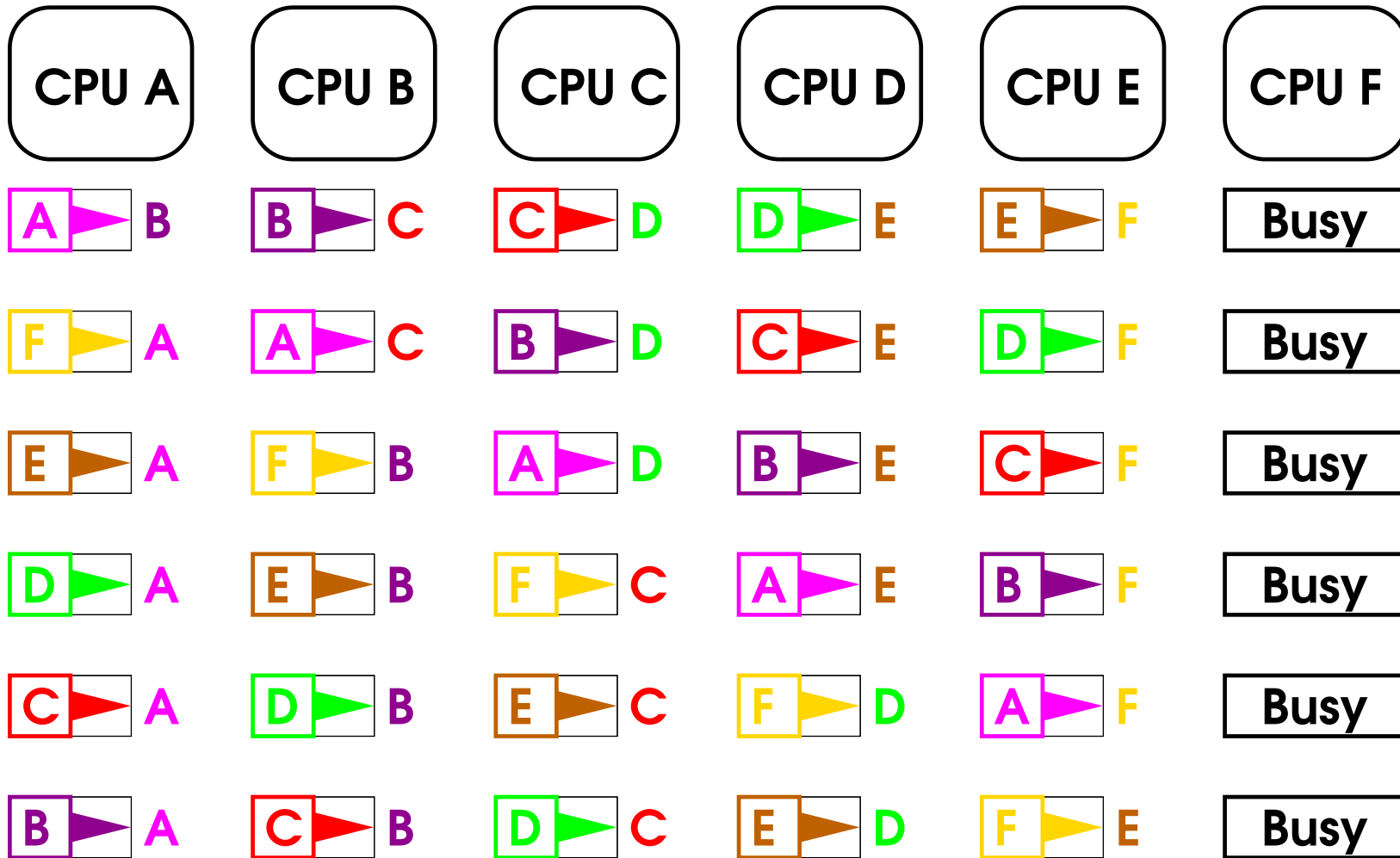
**Only 17% slower + 1 thread swap / core**

# Near-pessimal all-to-all



**This is 100% slower (a factor of two)**

# Near-pessimal all-to-all



**This has 5 thread swaps / core**

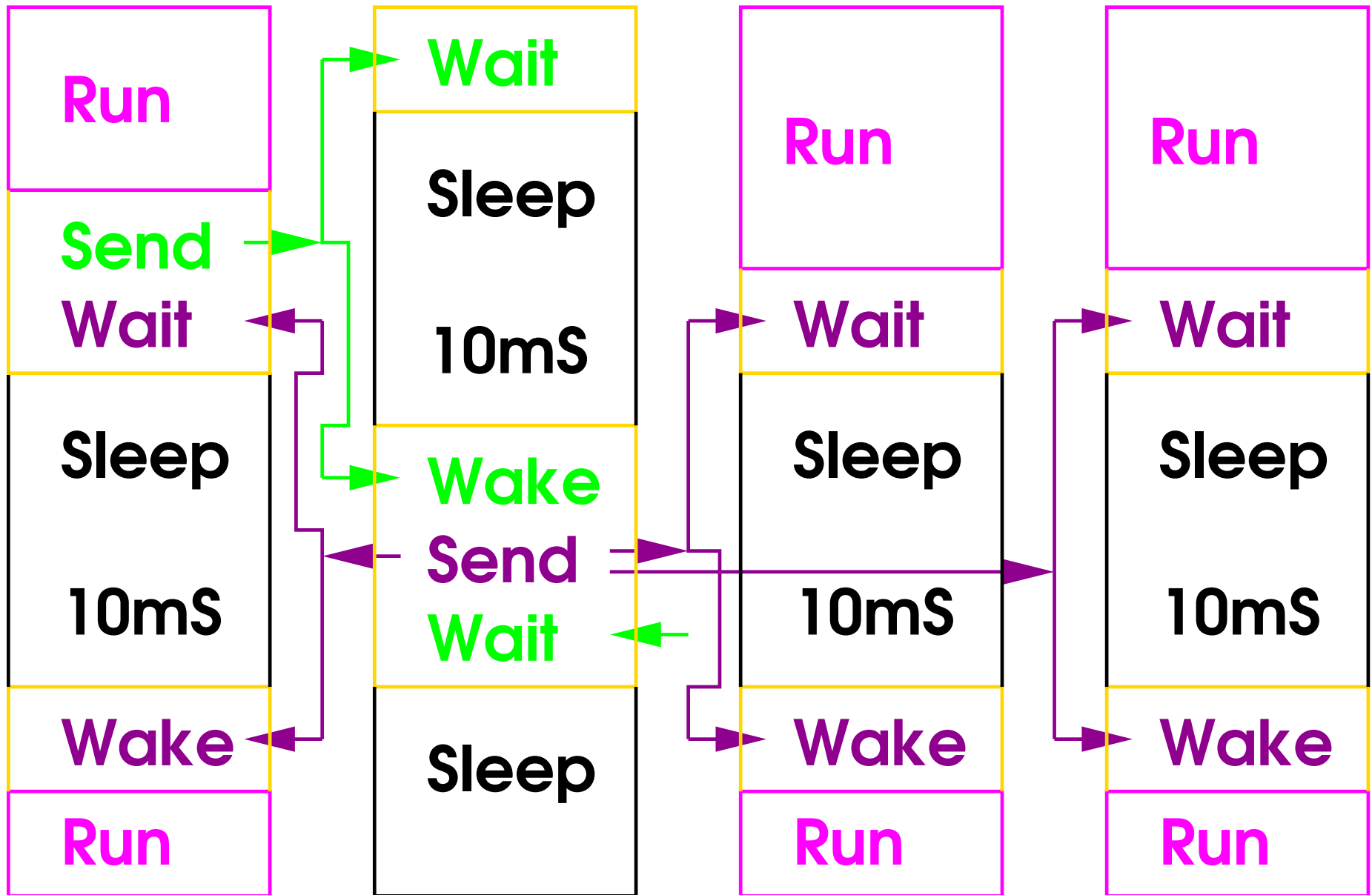
# Scheduler Glitches (1)

Threads being delayed can be far worse  
It may drop the library into 'slow' mode  
This uses sleeping rather than spinning

Most kernel schedulers have a 10 mS cycle  
Will often change thread state once per cycle  
This can degrade to a cycle of scheduler delays

- Ask for help if you have trouble here  
Solution is to avoid starting the cycle

# Scheduler Glitches (2)





# Scheduler Glitches (3)

- Most **I/O calls** trigger **long sleeps**  
And quite a few other **'waiting'** system calls
- Don't include them in **performance-critical** code  
One **thread/process** can hold up others
- Other **solutions** involve **system configuration**  
Search terms include **spin loops**, **nanosleep**  
Too complicated for course – ask offline