Parallel Architectures

Chapter 1: Introduction

Lecturer: *Paul H J Kelly*

Autumn 1997

Department of Computing
Imperial College

Room 423
email: phjk@doc.ic.ac.uk

Course web site:
http://www-ala.doc.ic.ac.uk/~phjk/phjk-ParallelArchitectures.html

THE OBJECTIVE

This course should provide the basis for the student to engage in the development of performance-critical computer applications and in conjunction with other courses should provide important background for research and development in advanced computer architectures, compilers and operating systems.

How to enjoy, learn from and pass this course:

- Pre-requisites
- Planned syllabus handout
- Textbook
- Make handwritten notes during lectures
- Tutorial exercises really help
- Assessed coursework really matters
- Ask questions
- Examinations – tutorial exercises, past papers

Lecture notes will largely consist of copies of overhead transparencies. Together with tutorial exercises they form your best guide to the examinable content of the course.

Statistically, students who do not do tutorial exercises *during* the course do very badly.

Textbook

Computer Architecture: A Quantitative Approach **(second edition)**, by David A Patterson and John L Hennessy. Morgan Kaufmann publishers, 1996.

- This is the required text for this course. Credit will be given in coursework assessment and exam marking for evidence that you have read and understood the appropriate sections of the textbook.
- Make sure you get the second edition
- Don't confuse it with their more introductory book, *Computer Organisation and Design*. You need the advanced text.
- The sooner you read the first two chapters the more worthwhile it will be.
- Use the Syllabus handout to guide your reading.

Other background

- To do well on this course you should get to know the textbook pretty well, and develop a broad general knowledge of high-performance computing
- See the course website for pointers to other books and information sources.

COURSE OVERVIEW (Subject to change)

- **Benchmarks, performance metrics**
- **Pipelining, static scheduling**
- **Branch prediction**
- **Dynamic instruction scheduling**
- **Scoreboarding vs Tomasulo**
- **Speculative execution**
- **Multiple issue: superscalar, VLIW**
- **Vector processors**
- **Compiler issues: dependence analysis**
- **Parallel programming**
- **Caches and memory systems**
- **Shared-memory multiprocessors**
- **Interconnection networks**

This chapter

1.1 Introduction

- **Example:**

  Suppose we want to build a 100 Gigaflop computer (i.e. a machine capable of 10^{11} floating point calculations per second on some problem of interest)

  There is a range of possible ways of doing this.

KELLY 1

- Single, straightforward processor taking roughly 5 cycles per FLOP
- Memory system capable of providing two floating-point operands, and storing one, every 5 cycles
- Standard off-the-shelf optimising compiler
- Achieves close to peak performance on most problems
- Clock rate 500GHz, clock cycle time = $1/(5 \times 10^{11}) = 2$ps (picoseconds)

  Requires logic gates and wiring with propagation delays around 0.1-0.2 ps (in this time light travels about 30μm).

KELLY 1: Conclusion

- Attractive since performance is predictable and easily achieved.
- Problem: development cost and delivery date.

(Is it possible to build a CPU with a 2ps clock cycle?)

(Do the signals in a CMOS digital circuit travel at the speed of light?)

(If you did try to build a computer using optical or superconducting technology, you would want to keep it simple - imagine debugging...)

KELLY 2

- One thousand processors
- Each processor is a "commodity" microprocessor (i.e. very cheap), with peak performance 100 MFLOP
- Useful computation requires communication between processors
  - Could exchange 1TByte tapes by Royal Mail (first class)

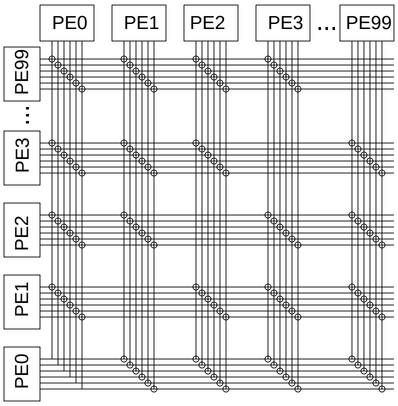
    1TByte/day $\approx$ 10MByte/s

    Any-to-any: everyone could send to same processor at same time, or vice versa (?)

    Overnight delivery

Full cross-bar

- Could connect each processor to every other processor using copper wire



- Number of wires: at least $N(N-1)/2 \approx 500000$
- Mass: several hundred kg

KELLY 2: Conclusion

- Not straightforward to program. Without parallelism, performance only 100 MFLOP.
- With full interconnection, communications can be very fast: data can be exchanged with other processors every few instructions
- If all processors choose to communicate with the same processor all at once, performance is still determined by the performance of one processor.
- We need some cleverer way of connecting the processors, to reduce the amount of wire:
  - need high point-to-point bandwidth
  - need low delivery delay ("latency")
  - need high overall capacity ("bisection bandwidth")

KELLY 3

- One hundred high-performance microprocessors, 1GFLOP peak each
- A more sophisticated interconnection network

Interconnection network topologies

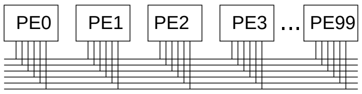


Figure 1: Bus

- Bandwidth
- Contention - blocking, queuing
- Broadcast capability?
- Arbitration/deadlock?
- Ease of expansion

Examples:

- PCI, MBus, SBus, IEEE "FutureBus"
- Multiprocessor PCs, Sunsite, SGI...
- Ethernet

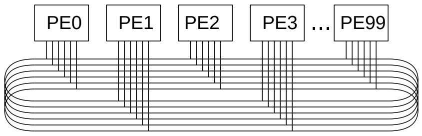


Figure 2: Ring

- Bandwidth? Contention? Broadcast? Ease of expansion?

Examples:

- SCI (IEEE Scalable coherent interconnect)
- Convex Exemplar, Kendall Square KSR-1
- IBM Token Ring (local-area network)

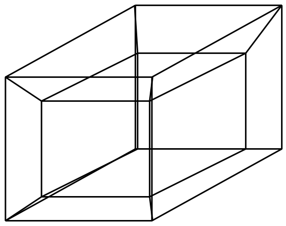


Figure 3: Hypercube

- Bandwidth? Contention? Broadcast? Ease of expansion?

Examples:

- CalTech Cosmic Cube, NCUBE, Intel iPSC
- Variants: 2D & 3D meshes (easier to build)
- Fujitsu AP3000 (2D mesh), Cray T3E (3D mesh)

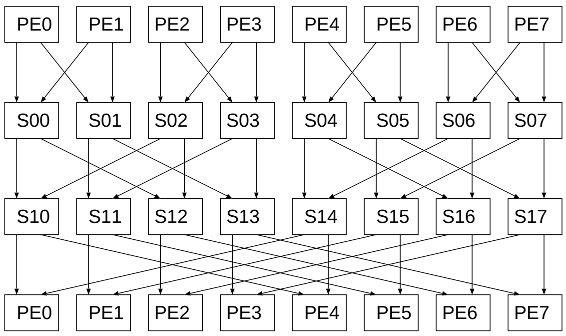


Figure 4: Multistage

- Bandwidth? Contention? Broadcast? Ease of expansion?

Examples:

- IBM RP^3, BBN Butterfly, SB-PRAM, Imperial's ALICE, ICL Goldrush
- ATM switches (wide-area networks)

KELLY 3: Conclusion

- Not straightforward to program:

  Without parallelism performance only 1GFLOP
- Good parallel programs must minimise communication *and* organise communication carefully to avoid bottlenecks.
- Bad parallel programs can run very slowly.

KELLY 4

- Sixteen very sophisticated processors
- Connected by full crossbar to 64 interleaved memory banks
- Each processor has three floating point arithmetic units, each divided into five pipeline stages.
- Under ideal circumstances, all three FPUs are fully busy and can complete a FLOP every cycle
- With a clock rate of 2GHz (cycle time 0.5ns), each processor can achieve 6 GFLOPs
- 16 processors working flat out at 6 GFLOPs each would achieve 96 GFLOPs
- We assume that the memory system can accommodate this

KELLY 4: Conclusion

- Parallelism is being exploited at three levels: pipelining of arithmetic, multiple FPUs, and multiple processors:

Arithmetic pipelining	FP pipeline depth	× 5
Multiple FPUs	Control complexity	× 3
Multiple CPUs	Interconnection	× 16
Total parallelism		× 240

- The memory system parallelism is needed to sustain this performance on applications which don't fit into cache
- For peak performance we must write programs which are optimal at all three levels - but sub-optimal programs can still run at a large fraction of peak.

The moral:

> High performance computing relies on achieving the maximum cost-effective advantage at every level.

1.2 Aspects of design quality

It is rare that raw computational speed is the single goal in computer design.

- **Manufacturing cost**
  - chip cost, chip count
  - wiring, packaging, assembly
  - testing
- **Design Cost**
  - hardware complexity
  - new technology, new tools
  - software: compilers, operating systems
- **Time to market**
  - component delays, tool delays
  - design delays
  - software delays (bugs, evolving standard)
- **Other**
  - maintenance, marketing, defects

1.3 Reporting Performance

Performance is about the time taken to complete a specified task:

- **Execution time**

  The elapsed time between starting a task and finishing it.
- **Faster than**

  For a specified task, machine A is faster than machine B if the execution time for A is smaller than B.

It is often tempting to use *throughput* as a performance measure:

- **Throughput**

  The rate of completion of tasks over an extended period

It is easy to use parallelism to improve throughput. Improving execution time is more challenging. (See Hennessy & Patterson page 18)

MIPS "millions of instructions per second"

"MIPS" is a popular metric for performance - the customer knows what is means, and more is better.

$$\text{MIPS} = \frac{\text{instruction count}}{\text{execution time} \times 10^6}$$

- depends on instruction set
- depends on the program (why?)
- decreasing MIPS can mean increased performance

HOW ?

- Modifying the hardware to introduce powerful instructions (e.g. floating point) reduces the number of instruction executed, but may improve performance.
- If an optimising compiler avoids the need for some instructions, it changes the MIPS rating (increases? or decreases?)

(See Hennessy & Patterson page 44)

1.4 Evaluating Speed-up Mechanisms: limits of parallelism

In balancing cost against performance, it is necessary to focus resources on those areas where maximum benefit can be gained.

This is in contrast to technological improvements which tend to yield a general enhancement. Indeed this is arguably the defining slogan of computer architecture:

> **Make the common case fast, and make the fast case common.**

Evaluating smart ideas

Suppose you invent some clever trick (cache, pipelining, special instruction, parallelism)

1. How good is it?
   - When the trick is used, what speedup over the unenhanced case is achieved?
2. How much good is that?
   - For what fraction of the overall execution time can the trick be employed?

So what's the overall benefit?

AMDAHL'S LAW

Define

- **Fraction$_{\textbf{enhanced}}$**:
  The proportion of the run-time without the enhancement, which will be affected by the enhancement.
- **Speedup$_{\textbf{enhanced}}$**:
  $$\frac{\text{Exec}^{\text{n}} \text{ time of fraction}_{\text{enhanced}} \text{ without enhancement}}{\text{exec}^{\text{n}} \text{ time of fraction}_{\text{enhanced}} \text{ with enhancement}}$$

Then the overall speedup is given by:

$$\frac{\text{execution}-\text{time}_{\text{old}}}{\text{execution}-\text{time}_{\text{new}}}$$

$$= \frac{1}{(1-\text{fraction}_{\text{enhanced}}) + \frac{\text{fraction}_{\text{enhanced}}}{\text{speedup}_{\text{enhanced}}}}$$

AMDAHL'S LAW and PARALLELISM

- In our discussion of the KELLY 3 machine (100 high-performance microprocessors with a sophisticated interconnection network), we argued that 100 processors, each capable of 1GFLOPS, could meet out target of 100 GFLOP.
- Amdahl's law means that since we want

  $$\text{speedup}_{\text{overall}} = 100$$

  but we can at best apply parallelism to get

  $\text{speedup}_{\text{enhanced}} = 100$ (have only 100 processors)

  It follows that

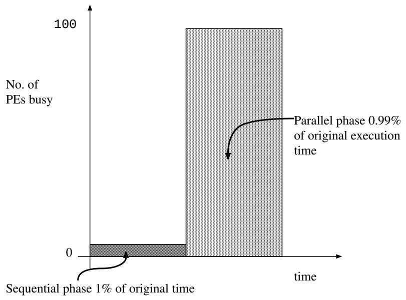
  $$\text{fraction}_{\text{enhanced}} = 100\%$$

- To get the full benefit of parallelism, the entire program must be parallelised

- Suppose instead that while 99% of the sequential execution time can be improved using parallelism, these remains 1% which cannot be parallelised (why might this happen?)
- Recall Amdahl's law: overall speedup is given by

$$\frac{1}{(1-\text{fraction}_{\text{enhanced}}) + \frac{\text{fraction}_{\text{enhanced}}}{\text{speedup}_{\text{enhanced}}}}$$
$$= \frac{1}{1-0.99+0.99/100} = \frac{1}{0.0199}$$
$$= 50$$

- Now, the dominant phase of the computation is the sequential 1% :



PERFORMANCE

- Power consumption, heat dissipation
- Physical size, ruggedness
- Noise generation, immunity
- Compatibility, reliability, availability
- Performance
  - On important applications
  - Peak
  - On standard benchmarks
  - Predictability
  - Ease of programming
  - Code and data size
  - Responsiveness
  - Throughput

Power, size, noise, reliability, cost (?) easy to measure

1.5 Performance Metrics

The performance of a computer system is more difficult :

- Performance is multi-dimensional; floating point performance, integer performance, and input-output (e.g. disk) performance are all orthogonal.
- Performance depends on details of the program, often in very subtle ways.
- Performance depends heavily on the compiler (which may change long after the architecture has been designed)
- Performance can often be improved by manual recoding (especially in parallel programs).

BENCHMARKING IDEA 1

Predict the effectiveness of a general purpose machine using a small collection of inner loops which you think are typical of your applications

Examples:

- The Livermore Loops
- LINPACK
- McCalpin's STREAM benchmark (?)

Problems:

- Real applications may have very different innermost loops
- Future applications may differ from applications studied at purchase time
- Real applications may not be dominated by loops

BENCHMARKING IDEA 2

Predict the effectiveness of a general purpose machine using a specially-constructed benchmark program, designed to be representative of a class of applications.

Examples:

- The Gibson mix ("synthetic benchmarks")
- Whetstone
- Dhrystone

Problems:

- There's no such thing as an 'average' program - typical programs rely heavily on same aspect of performance
- Real programs are bigger than toy benchmarks
- Details of the compiler can severely distort comparisons by optimising parts of the program away

- Heavy reliance on synthetic benchmarks encourages cheating - hacks in the architecture or compiler which improve performance on benchmarks but not real programs
- New architectural techniques can make old attempts to synthesise "typical" programs obsolete

BENCHMARKING IDEA 3

Avoid argument by testing using the actual workload for which the machine will be used.

Problems:

- Details of the program to be solved and the algorithms used to solve it, are sure to evolve with time. Will this expose weakness in the architecture?
- What about machines which are supposed to be "general-purpose"?

Examples:

- The IBM GF11 supercomputer designed specially to solve programs in quantum chromodynamics.
- Embedded systems, eg signal processing

Advantages:

- At least your machine should be good for something!
- You can cheat as much as you like - so much the better!

BENCHMARKING COMPROMISE

Use a set of several real applications, covering a range of application areas and program structure.

Problems:

- Applications are still not average programs, so results can still not reliably be extrapolated.
- It is misleading to combine performance on different programs since it is unclear how they should be weighed.
- Compiler technology can still make a big difference.
- Still fails to account for the effort needed in manual performance tuning for many parallel architectures

Examples:

- The SPEC suites
- The Perfect Club suites