MONASH University
Information Technology

FIT3142 Distributed Computing

Topic 8: Distributed Application Performance Modelling Part 1

Dr Carlo Kopp
Faculty of Information Technology
Monash University

Why Study Distributed Application Performance?

- Unlike applications on standalone systems, Distributed Systems can be extremely sensitive to performance problems, to the extent of making a system unusable;
- Foundation knowledge: Because Distributed Systems are now widely used, understanding their performance is essential to both design and implementation of code;
- Practical skills: When coding distributed applications you will have to design the application with due consideration to performance constraints in the algorithms used;
- Practical skills: If you do not understand performance problems in distributed systems, you might end up producing code that appears to be correct, functions as intended, but is unusable due to performance problems that cannot be fixed as they result from the behaviour of the algorithm used in a distributed system;

What is High Performance / Super Computing?

- The terms "supercomputer" and "supercomputing" are typically used to describe very powerful computing systems employed for scientific, engineering, cryptographic, economics, environmental or other simulation and modelling tasks.
- Contemporary literature mostly uses the term "High Performance Computing" (HPC) instead.
- The term "supercomputing" emerged during late 1960s and early 1970s with the appearance of vector processing architectures.
- It is now also used for parallel computing.

The Parallelism Problem

- **Any individual stored program processor will have performance limitations.**
- **These limitations are determined by internal design features:**
- **How quickly it can execute an instruction.**
- **How quickly it can fetch instructions and load/store operands.**
- **Speedup can be effected by performing operations in *parallel* – i.e. concurrently.**
- **Exploiting opportunities for concurrency is the basis of superscalar/parallel/grid architectures.**

The Data Dependency Problem

- Some computations exhibit a property which is termed “data dependency”.
- A data dependency between two computations arises when one of these computations depends upon the result of the other.
- *Therefore the dependent computation cannot be executed until the computation upon which it depends is completed.*

Data Dependency Levels

- Data dependencies can arise at the instruction/operand level, function or subroutine level, and at the program level.
- The level at which data dependency occurs determines what techniques we can use to exploit parallelism and make the program run faster.
- Programs with frequent data dependencies at all levels do not lend themselves to speedup via parallel techniques.

Performance Gains in Multiprocessing - Amdahl

- A system with N CPUs can do more useful work than a system with a single CPU, but only where the application program being executed can be divided across all N CPUs.
- In general, a program can be functionally divided into computations which are "serial" and computations which are "parallel".
- "Serial" computations have some data dependencies and cannot be parallelised, unlike the "parallel" computations.

Performance Gains in Multiprocessing

- Assume a program has no computations with mutual dependencies.
- If this is true, then running the program on a machine with N CPUs results in an N-fold gain in performance as the workload can be evenly divided across all N CPUs.
- This N-fold gain in performance is termed “linear speedup”.
- *What happens if the program comprises only computations with mutual dependencies?*

Performance Gains in Multiprocessing

- **Assume a program only has computations with mutual dependencies.**
- **Then the program cannot be usefully divided across a pool of N CPUs, and no performance gain can be realised.**
- **Many computational programs fall into this category and are not suitable for multiprocessing systems.**
- ***What happens if a program contains a mix of computations, some of which are "serial" and and some of which are "parallel"?***

Amdahl's Law

- **In 1967 Gene Amdahl formulated "Amdahl's Law" which describes the achievable speedup in a multiprocessing system which is executing a program with a mix of "serial" and "parallel" computations (Amdahl, G.M. *Validity of the single-processor approach to achieving large scale computing capabilities.* In AFIPS Conference Proceedings vol. 30 (Atlantic City, N.J., Apr. 18-20). AFIPS Press, Reston, Va., 1967, pp. 483-485).**

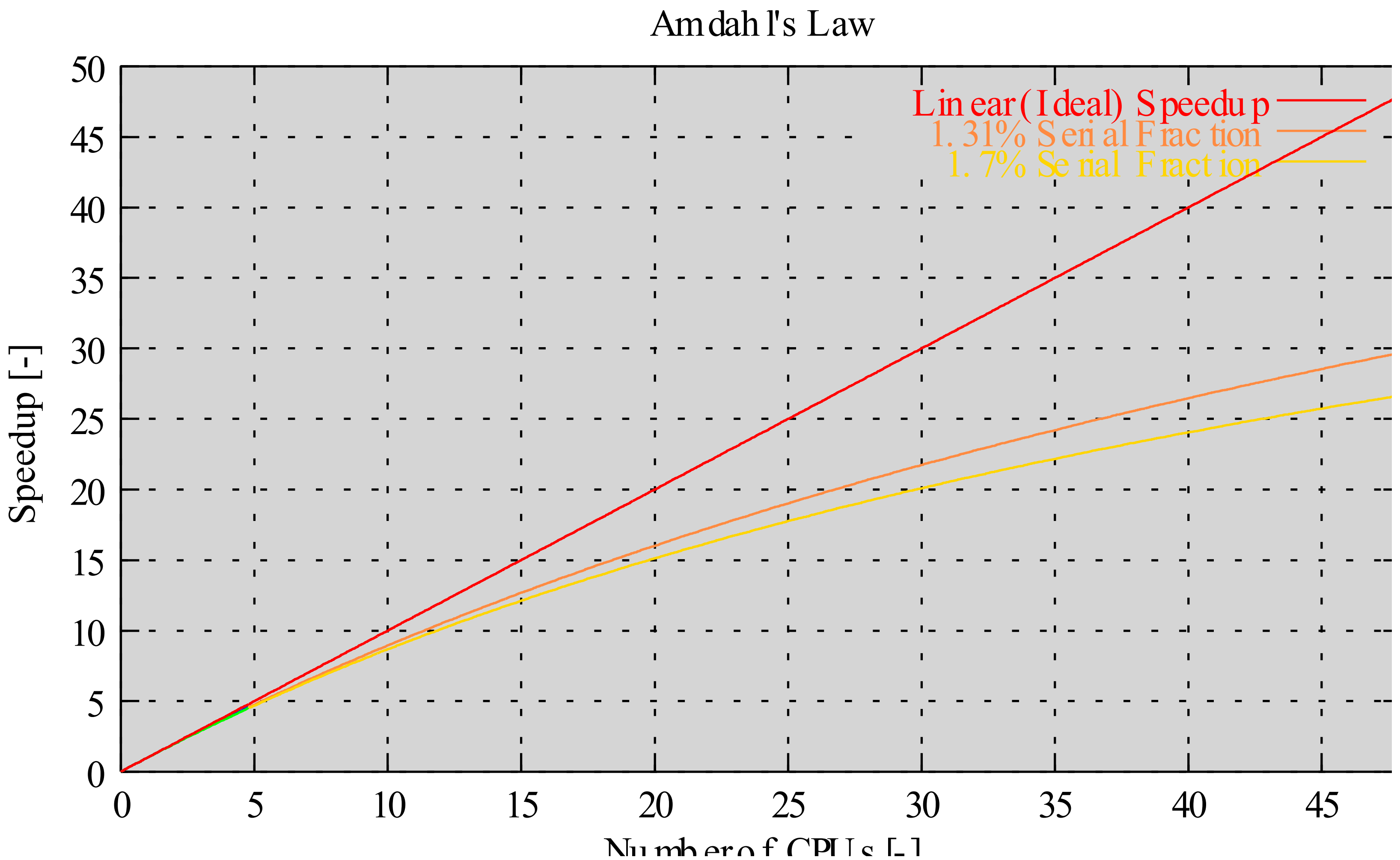
Amdahl's Law

- Assume N is the number of processors, and *s*, *p* are the respective fractions of time spent on the serial and parallel components of a program, where

$$S = \frac{1}{s + \frac{p}{N}} \quad where\ s + p = 1$$

- This expression is known as *Amdahl's Law*, and it shows that the performance gain in a multiple processor system depends strongly upon the behaviour of the program.
- Adding CPUs only makes sense when the program has a large parallel component. Amdahl's Law can also be remapped into a queueing system model.

Amdahl's Law Example (Measured Performance)



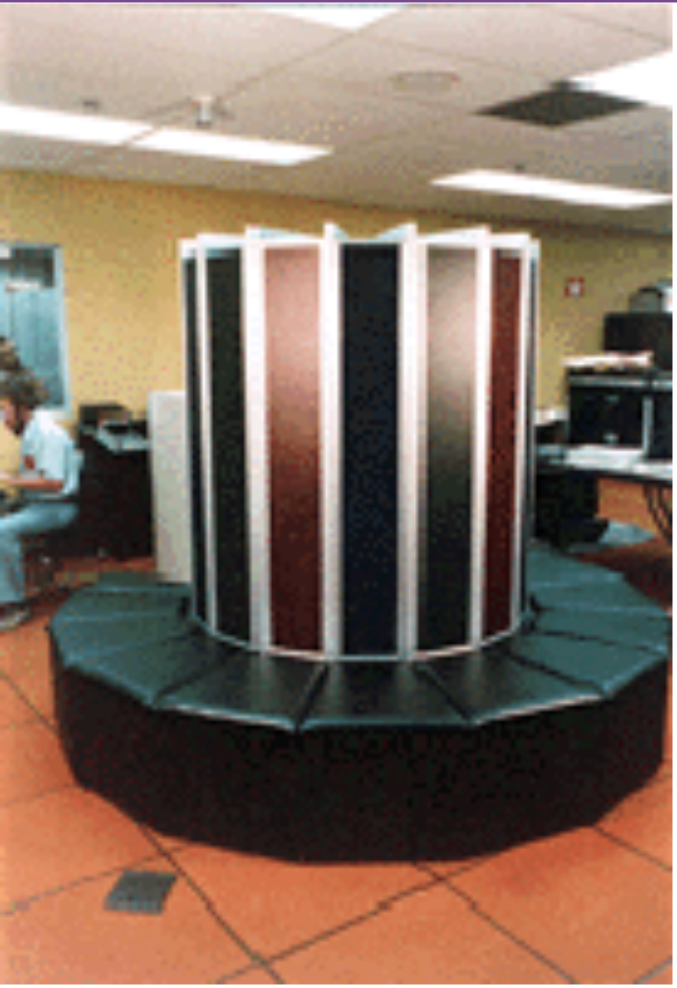
Evolution of Supercomputing

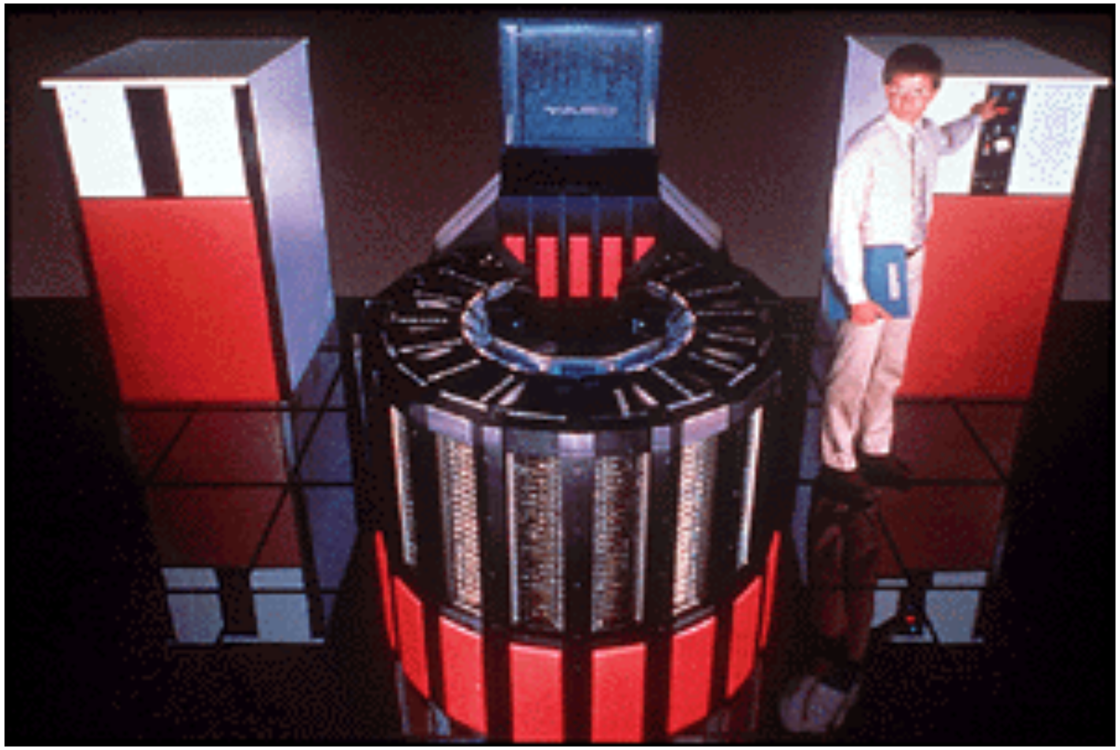
- 1960s – superscalar processing to exploit instruction/operand level parallelism.
- Late 1960s – vector processing to exploit instruction/operand level parallelism.
- 1960 - 1970s – multiprocessors with internal bus connections between CPUs.
- 1990s – clusters with local area networks connecting CPUs.
- 2000s – grids with CPUs distributed across a larger network.

Early Superscalar Examples

- CDC 6600 supercomputer - 10 (4)-way superscalar (1963)
- IBM 360/91 mainframe - 3-way superscalar (1967).
- Cray-1 supercomputer - 13-way superscalar (1975)
- NB: due to the high cost of adding additional pipeline and ALU hardware, superscalar techniques were only used in the largest and most expensive machines of the period.

Cray 1 & 2 Supercomputers





Recent Examples - Superscalar

- Intel Pentium-I - 2-way superscalar (1993).
- Sun SuperSPARC/Viking - 3-way superscalar (1993).
- Intel Pentium-Pro/Pentium-II/III - 5 way superscalar (1996-1999).
- AMD Athlon/K7 - 9-way superscalar (1999).
  - NB: the increasing number of transistors on microprocessor dies allowed by the early nineties the incorporation of superscalar techniques.
  - 1960s supercomputer ~ 1990s desktop.
- Intel Pentium IV/Athlons – Multicore 2 x; 4 x (2008)
- Post 2005 shift to parallel processing + superscalar

Vector Processing Techniques

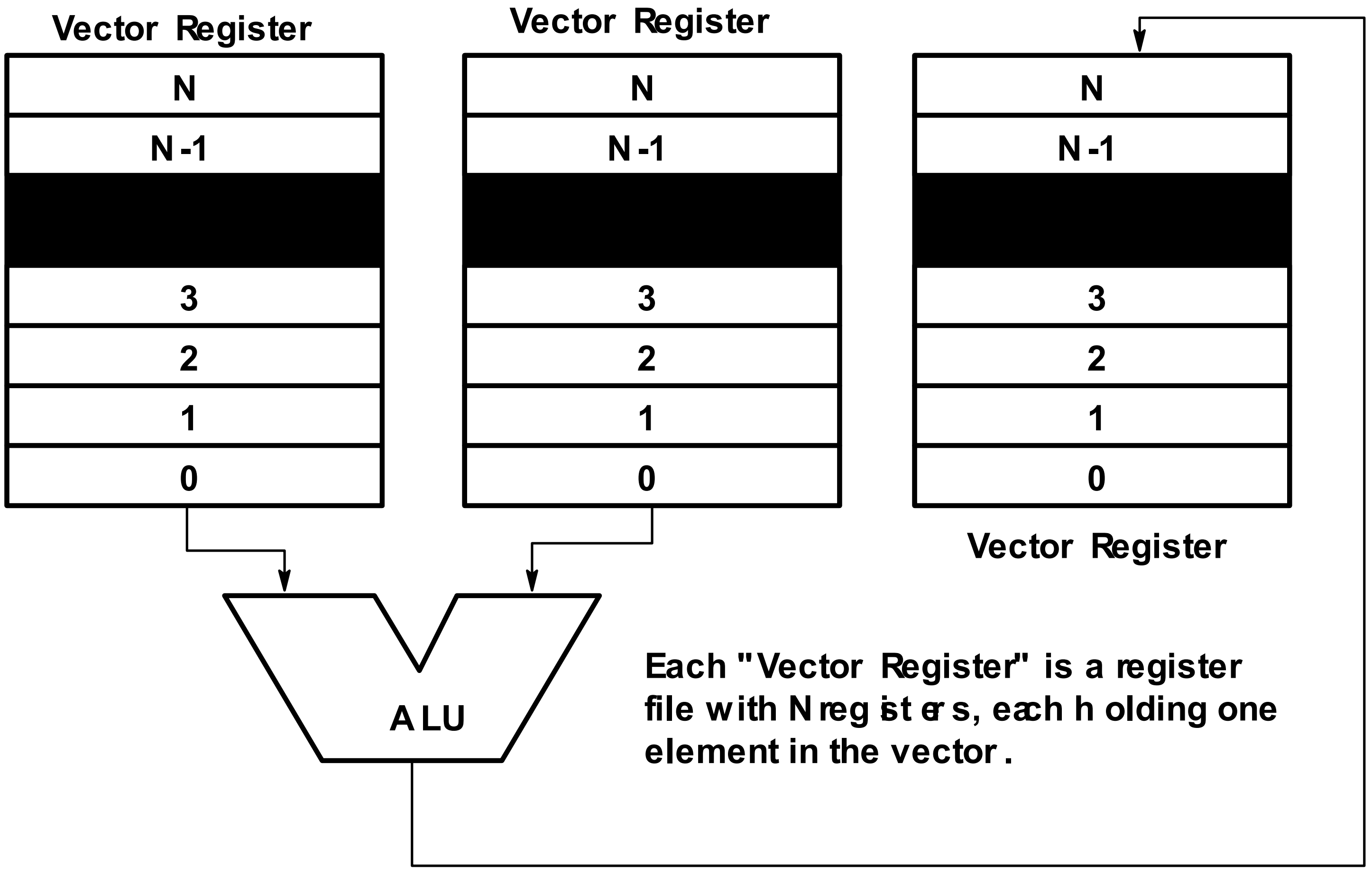
- Typical processors, pipelined or superscalar, do not perform well on vector arithmetic since they do not exploit the properties of vectors.
- Vector arithmetic involves the repeated execution of the same operation on consecutive elements of a vector.
- Conventional processors incur an opcode fetch, operand load and decode delay for every element in the vector - very inefficient

Vector Processing

- How can we optimise an architecture for vector arithmetic ?
- We can use an instruction set for vector operations: e.g. *Add vector operands of size N, V3 and V2, and put the result into V1.*
- `ADD N, V1, V2, V3`
- A vector operand may be a pointer to an array in memory, or a register.

Vector Processing

- Using vector operations we remove the overheads of N-1 opcode fetches, decodes and operand loads.
- Because the same operation is repeated N times, we can operate a pipeline for N consecutive cycles with no stalls, which is very efficient.
- By using multiple execution units, we can achieve stall free superscalar operation, which is also very efficient.

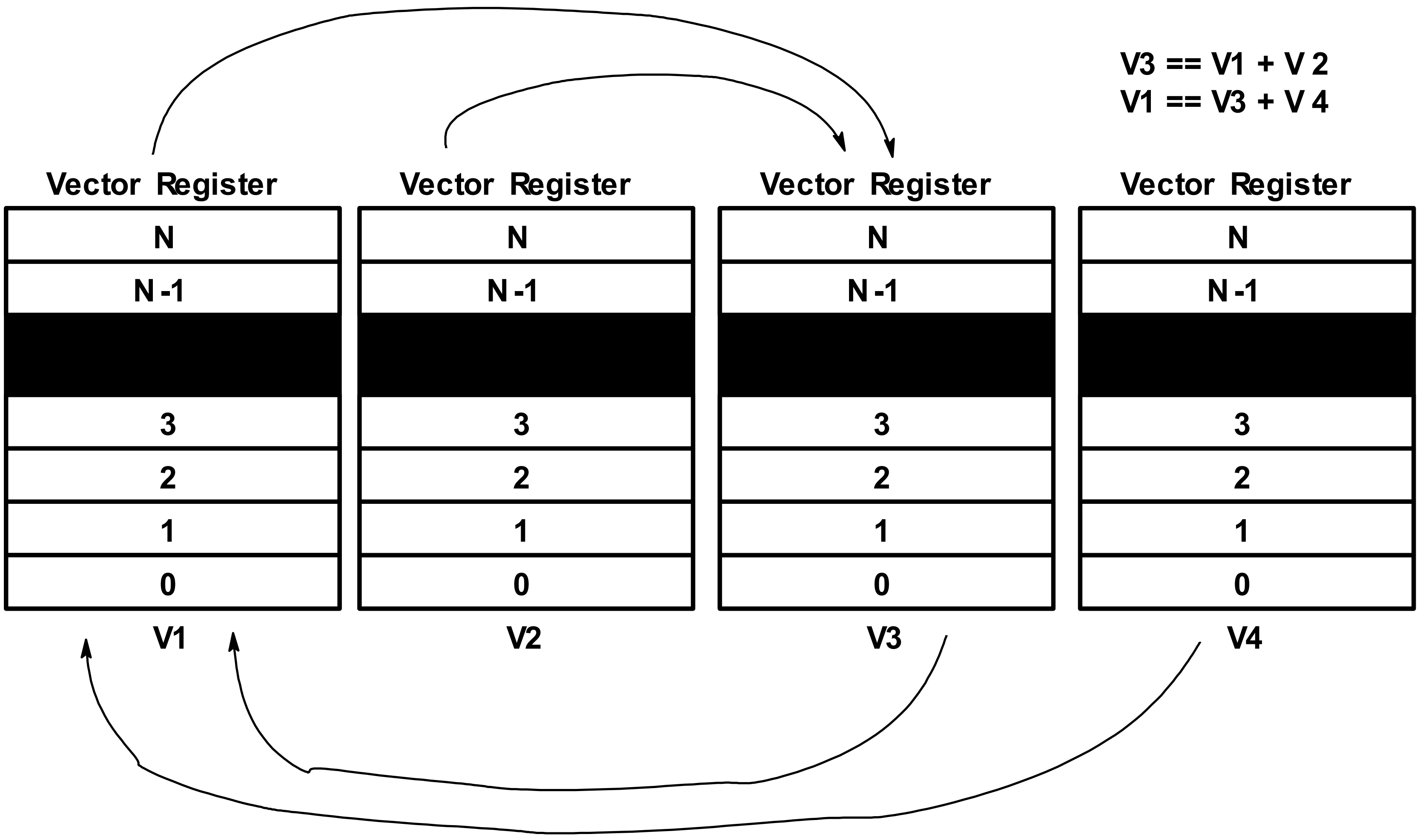


Each "Vector Register" is a register file with N registers, each holding one element in the vector.

Chaining

- The technique of 'chaining' is frequently used in vector processing architectures.
- Because the loading of an N element vector into a vector register requires N consecutive loads from memory, it is slow.
- By arranging the use of the vector registers properly, in a 3 operand vector architecture, and using at least 4 vector registers, it is possible to load one vector operand while two others are being read into the EU and written into another.

V3 == V1 + V 2
V1 == V3 + V 4



Exploiting Vector Hardware

- Vector processors perform exceptionally well on problems involving matrix arithmetic and vector operations.
- Can such hardware be exploited otherwise to improve performance ?
- Consider a loop with M identical passes, in which no dependencies exist between like scalar operands used in each pass of the loop.

Parallelising Loops

- A suitable compiler can rearrange the loop as a series of vector operations, in which each element of the vector operand is a scalar operand belonging to a pass in the loop.
- Therefore a vector processor can deliver high performance both in vector arithmetic, and scalar arithmetic, if loops can be parallelised.
- Compiler support is vital for performance; the compiler must perform tasks like arranging registers in chaining, and parallelising loops.

Vector Processing Examples

- Cray series vector processors - Cray 1, Cray 2, Cray X-MP, Cray Y-MP - all provide support for long vectors (~ 64 elements), and are aimed at supercomputing applications.
- Convex 'Crayettes' - long vectors.
- Motorola MPC7400 PowerPC G4 - provides hardware for short vectors (4 x 32-bit) and is optimised for graphics, signal processing and multimedia operations.

Multi Processing

- The alternative technique used for parallel processing is the use of multiple CPUs, which share a common main bus, main memory and I/O hardware.
- A multiprocessor allows completely separate programs to be run on each of the CPUs.
- Assymetrical MP is where the operating system is run on one CPU, and user programs on the others.
- Symmetrical multiprocessing is where both the operating system and user programs run on any CPU.

Issues in Multiprocessing

- *Bus throughput* can limit the number of CPUs in the system. If the bus cannot handle the volume of accesses to memory, CPUs will stall and performance is then lost.
- *Cache coherency* with memory is vital to ensure that CPUs do not modify data, unbeknownst to other CPUs.
- If two CPUs each hold copies of the same location in memory within their caches, a mechanism must exist to prevent them from overwriting each others' results.

Cache Coherency

- Numerous techniques exist for maintaining cache coherency.
- *Snooping techniques* use intelligent caches which broadcast any writes to all other caches in the system. Each cache then updates itself to reflect the change.

Parallel Processing - GPGPU Concepts

- GPGPUs evolved from GPUs, GPUs evolved from “Graphics Accelerators”, which in turn evolved from “Frame Buffers”;
- Frame buffers used a dual port memory which was accessed “upstream” by a CPU which rendered graphical images in the frame buffers, which were pixel-by-pixel read by a RAMDAC chip to generate VGA, SVGA, XGA or proprietary analogue video for display on a Cathode Ray Tube;
- The analogue interface has been replaced by DVI/HDMI and the CRT by LCD, plasma, LCD/LED, LED or OLED displays;
- Early accelerators added dedicated hardware to perform common graphics operations within the frame buffer memory, with the CPU programming the accelerator with specific commands to execute;
- Modern GPGPUs follow the same model, driven by a CPU.

GPGPU Concept

- A GPGPU is a direct extension of the "traditional" GPU concept;
- Changes specific to GPGPUs are:
- ✓ programmable pipeline stages;
- ✓ higher precision integer arithmetic units;
- ✓ higher precision floating point arithmetic units;

- A GPGPU is thus able to perform types of computation not unique to graphics processing, such as scientific computations;
- *GPGPUs are still part of a "master-slave" SIMD model, where a general purpose CPU sets up the GPGPU cores to perform their computations.*

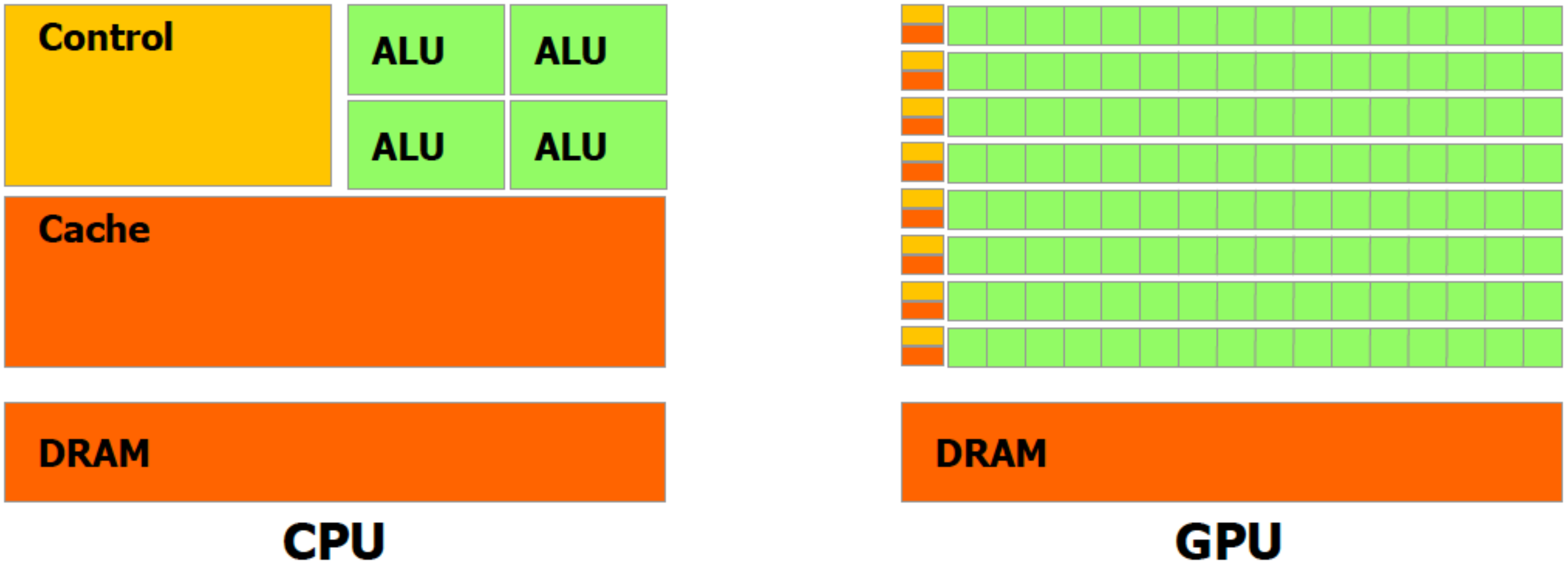
GPGPU Memory Bandwidth

- **A major performance limitation in current processors is the disparity in bandwidth to main memory versus bandwidth to much smaller internal register storage;**
- **In the time taken to read/write main memory, dozens of instructions can be executed – providing the operands are in registers or cache;**
- **GPUs employ dedicated very high speed memory arrays for graphics computations;**
- **A conventional main memory system will have a memory bandwidth typically of Gigabytes/sec up to tens of Gigabytes/sec (AMD HT/HTX, Intel QPI) – currently ~25 GB/s;**
- ***A GPU/GPGPU internal memory subsystem will have a memory bandwidth of more than 100 Gigabytes/sec (NVIDIA Tesla @ 148 GB/s; AMD Radeon 5870 @ 153.6 GB/sec) – sixfold better.***

GPGPU/GPU versus CPU

Processor	GPGPU/GPU	CPU
Number of Cores	Hundreds	2 - 12
Core complexity	Low	High
Memory bandwidth	~6:1	1:1
Vector operations	Yes	Limited or none
ILP Exploitation	Software scheduling	Control unit
Optimisation	Graphics/Scientific	General purpose
Integration	Master/Slave	Symmetrical MP
Architecture	Data Parallel / Stream	Conventional
Software	CUDA / API	Any

GPGPU/GPU versus CPU (NVIDIA)



Limitations in GPGPU Integration

- Most adaptations for GPGPU programming *expose* the GPGPU hardware to the programmer;
- This presents two problems:
- ✓ Programmers often do not understand hardware performance well and make mistakes, causing bugs or performance loss;
- ✓ Application portability can be compromised, as changing GPGPU hardware vendors and/or APIs requires deep alterations to the code;
- Traditional "vectorising" Fortran compilers would *hide* the hardware from the programmer to minimise the required programmer skills and maximise application portability;
- *Performance gains from using GPGPUs can be excellent, but rely upon the programmer understanding the GPGPU design and GPGPU API library calls.*

Problems in GPGPU Integration

- Scalability for GPGPU hosted applications is excellent, where the number of cores $N <= N_{GPGPU}$ i.e. the number of cores on the GPGPU chip;
- This is because of the very high memory bandwidth to the private memory attached to the GPGPU chip;
- If the required number of cores is higher, memory bandwidth between the GPGPU modules is then much lower than memory bandwidth to the private memory attached to the GPGPU chip;
- GPGPU solutions are therefore very sensitive to the data parallelism in the problem;
- In systems with multiple GPGPUs, the main bus connecting the global system main memory and GPGPUs becomes a critical system performance bottleneck.

Grids == Distributed Multiprocessors

- Grids, which involve the networking of hosts across large distances, and clusters which involve local networking of hosts, represent in architectural terms “distributed multiprocessors”.
- A Grid is subject to the same fundamental performance constraints imposed by Amdahl’s Law for multiprocessors.
- In a Grid environment, data dependencies result in network communications which increase the total time required to complete a serial component.
- Network performance is therefore critical to grids.

Amdahl's Law in Grid Computing

- Consider Amdahl's Law:

$$S = \frac{1}{s + \frac{p}{N}}$$

- The variable *p* reflects the fraction of time processing data with no dependencies, which can be parallelised. *p* is "parallel" time.
- The variable *s* reflects the fraction of time processing data with dependencies, which cannot be parallelised. *s* is "serial" time.

Amdahl's Law in Grid Computing

- In a grid environment, the fraction of time consumed performing "serial" computation comprises the time consumed by the CPUs doing the work, and the time consumed communicating, when hosts working on data with dependencies must transmit the results of communitations.
- Therefore:

$$s = s_{comp} + s_{network}$$

- Where $s_{network}$ is the time for these results to be encapsulated in a message, sent across the network, and extracted for use at the destination.

Amdahl's Law in Grid Computing

- This results in *Amdahl's Law for Grids*:

$$S = \frac{1}{s_{comp} + s_{network} + \frac{p}{N}}$$

- What is important is that the network can impact speedup performance no differently than the fraction of time expended on computations with dependencies.
- If the network provides extremely high capacity and thus short delays to transmit, then $s_{comp} >> s_{network}$ and the grid behaves no differently from a conventional "bussed" multiprocessor.

The Impact of Network Performance

- The time it takes a message to travel across a network depends on a number of factors:

1) Processing delays on the middleware libraries and protocols stack at the transmitting end;
2) Cumulative queueing delays in routers along the transmission route through the network;
3) Propagation delay through cables in the network, proportional to total cable length;
4) Processing delays on the middleware libraries and protocols stack at the receiving end.

- Only 3) is constant in time, the remainder being time variant, and depend on network congestion.

The Impact of Network Performance

- The result of Amdahl's Law and network delays is that the performance of a grid improves with decreasing frequency of messages required, decreasing size of messages required, and decreasing network load.
- If an application is to be run on a grid, it is therefore important to understand what data dependencies exist in the algorithm.
- For many problems, there will be choices in the "granularity" of the computation, i.e. the size of the "chunks" distributed across the grid.
- Increasing N also increases $s_{network}$ costing speed.

Grid Application Performance

- Distributed gaming and military simulation applications – well suited to grids since typically runtime messages are short and relatively infrequent comprising state information.
- Parametric processing applications – often well suited since runtime messaging may be both infrequent and short. Where dependencies exist between parameter sets, care must be taken.
- Mesh based computations, eg finite element models, require careful analysis to determine the best partitioning of the problem, to maximise speedup given some $s_{network}$

References/Reading

- http://www.csse.monash.edu.au/~carlo/SYSTEMS/Vector-CPU-0600.htm
- http://www.csse.monash.edu.au/~carlo/SYSTEMS/Moores-Law-0700.htm
- http://www.csse.monash.edu.au/~carlo/SYSTEMS/GHz-CPU-Performance-0801.htm
- http://www.csse.monash.edu.au/~carlo/SYSTEMS/Backplanes-vs-Cables-1100.htm
- http://www.csse.monash.edu.au/~carlo/SYSTEMS/Cluster-Practical-1299.htm
- http://hint.byu.edu/documentation/Gus/AmdahlsLaw/Amdahls.html
- **Browne J. C. -** Internet/Grid Computing -Fall 2002 - What is Performance for Internet/Grid Computation?