Australian National University

COMP4300/8300 Parallel Systems

Shared Memory Parallel Computing & GPU Computing

Prof. John Taylor

School of Computing
Australian National University
Canberra, Australia

April 2024

Logistics

Australian National University

- Ask as many questions as needed during the lectures.
- I will be asking questions to involve you actively in the lecture material.
  - Please do not be afraid to answer questions. My questions will be only trying to stimulate your reasoning.
- Asking or answering questions
  - In-person: I can triage questions/answers
  - Online: I will be do my best to monitor the Ed Discussion Forum
- Careful with usage of Gadi resources
- Lecture material is uploaded on Wattle **before** the live lecture

Assessment

Australian National University

The total workload for this (6 units) course is around 150 hours (around 12 hours per semester week). For semester 1 2024, the following assessment protocol:

Assignments – 50% of final mark

(First Assignment due today @ 23:55 - no pushes after deadline)

- There will be two assignments, each worth 25% of the final course mark (in total 50% of the final course mark).
- These assignments will take the form of programming tasks and/or answering
- practical and theoretical questions relevant to the course content.

Mid-Semester Exam – 10% of final mark – Redeemable (Done)

- 1 hour exam held in week 6, worth 10% of your final course mark.
- If you missed the mid-semester exam for a legitimate reason (e.g. medical) you will be awarded zero and your final course mark will be derived by scaling your final exam mark to represent 50%.

Final Exam – 40% of final mark (Date yet to be announced)

- This will be a 3-hour written exam held at the end of the semester during the normal examination period.
- This exam is worth 40% of the final course mark.

Where we stand

Australian National University

Participants

14
12
10
8
6
4
2
0

0.00 - 1.00
1.00 - 2.00
2.00 - 3.00
3.00 - 4.00
4.00 - 5.00
5.00 - 6.00
6.00 - 7.00
7.00 - 8.00
8.00 - 9.00
9.00 - 10.00
10.00 - 11.00
11.00 - 12.00
12.00 - 13.00
13.00 - 14.00

Mid-Semester Exam Results Mean = 7.54

Australian National University

Shared Memory Parallel Computing & GPU Computing

Contents Covered

Australian National University

A likely table of contents for the second part of the course.

- Motivation for Parallel Computers!
- Shared Memory Parallel Programming, Pthreads
- Thread Synchronization, Implementation of Locks
- Shared Memory Parallel Programming with OpenMP, OpenMP Tasks
- Shared Memory Computer Architecture: Snooping-Based Cache-Coherence
- Shared Memory Computer Architecture: Directory-Based Cache Coherence, Memory Consistency
- Hardware Threading, SIMD (intrinsics and OpenMP)
- GPU Architecture, CUDA (GPU) Programming and Execution Models
- CUDA Memory Hierarchy and Memory Management, Streams and Concurrency
- Performance Assessment with Roofline CPU and GPU
- GPU Program Tuning, Multi-GPU Programming

Review & exam preparation

Any questions about the course?

References

Australian National University

- R. Dennard et al., “Design of Ion- Implanted MOSFETs with Very Small Physical Dimensions,” IEEE J. Solid State Circuits, vol. 9, no. 5, 1974, pp. 256–268.
- M. T. Bohr and I. A. Young, "CMOS Scaling Trends and Beyond," in IEEE Micro, vol. 37, no. 6, pp. 20-29, November/December 2017, doi: 10.1109/MM.2017.4241347.
- M. B. Taylor, "A Landscape of the New Dark Silicon Design Regime," in IEEE Micro, vol. 33, no. 5, pp. 8-19, Sept.-Oct. 2013, doi: 10.1109/MM.2013.90.
- The Free Lunch Is Over. A Fundamental Turn Toward Concurrency in Software. Herb Sutter. http://www.gotw.ca/publications/concurrency-ddj.htm

Why are you taking this course?

Australian National University

Performance Boost	Solve problems significantly faster by harnessing the power of multiple processors or cores.
Scalability Prowess	Tackle large datasets and complex problems with ease by efficiently utilizing multiple computing resources.
Real-World Expertise	Gain the skills to model and analyze real-world systems using the world's fastest supercomputers
Career Advantage	Stay ahead of the curve in an increasingly parallel computing-driven world.
Problem-Solving Prowess	Develop advanced problem-solving skills applicable to various domains.

What is parallelism and why do we want to exploit it?

Australian National University

Advantages	Disadvantages
Increased Performance	Increased Complexity
Improved Scalability	Overhead Costs
Better Resource Utilization	Limited Applicability
Potential for Simulating Real-World Systems	Hardware and Software Requirements

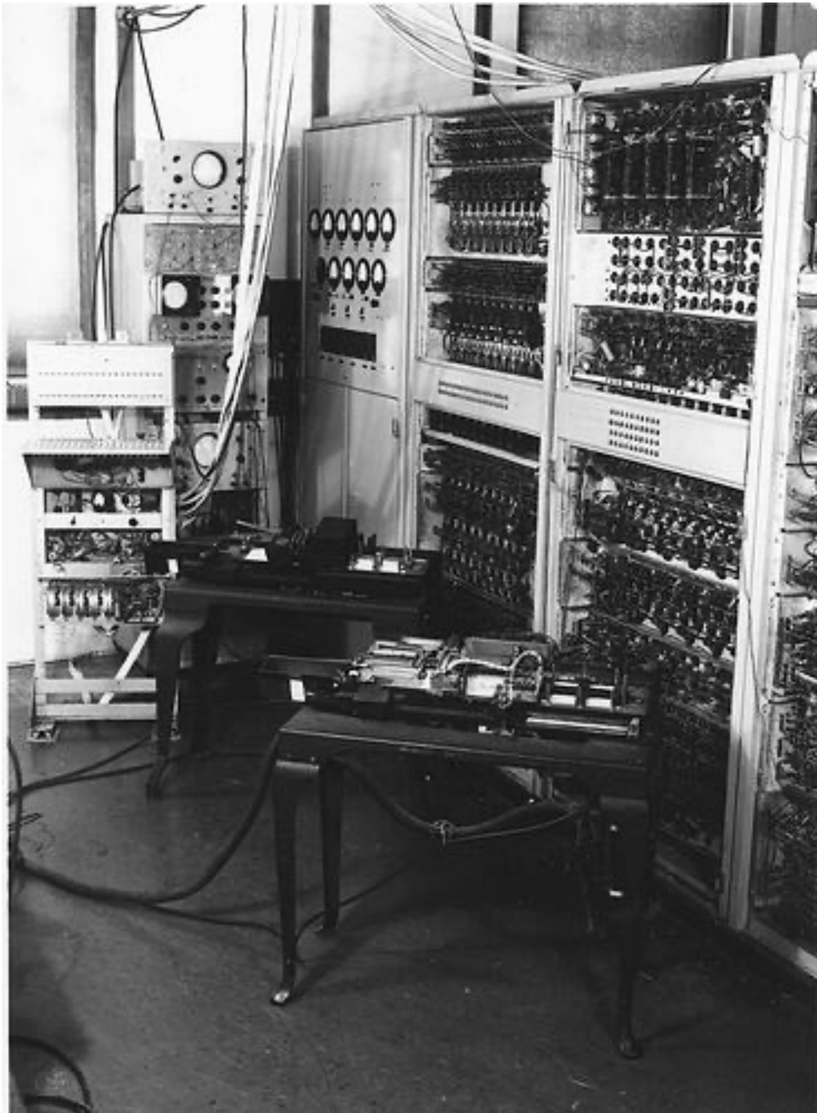
Why parallel computers?

Australian National University

Overcome Performance Limits	Traditional single-core processors have reached physical performance limitations. Parallel computers offer a way to continue increasing computational power.
Tackle Large-Scale Problems	They can manage massive datasets and complex calculations that single-processor systems cannot handle.
Simulate Complex Systems	Parallel computers can accurately model real-world phenomena that involve multiple simultaneous interactions (e.g., weather patterns, biological systems).
Improve Efficiency	Parallel computers often complete computationally heavy tasks much faster than single-processor machines, saving time and resources.
Drive Innovation	The need for parallel computing pushes advancements in hardware, software, and algorithms, expanding computational possibilities.

Uniprocessor Computing Era

Australian National University



CSIR Mk 1 with Hollerith equipment, Sydney 1952
Source: Museums Victoria
Public Domain (Licensed as [Public Domain Mark](#))

Early Beginnings: the 1940s-1950s

Vacuum Tubes and Transistors: The first computers relied on bulky and power-hungry vacuum tubes, later replaced by more efficient transistors.

Mainframes: Large, expensive machines dominated, used primarily by governments, universities, and large corporations.

Uniprocessor Computing Era

DEC PDP-10

https://livingcomputers.org

The Rise of Microprocessors: 1970s-1980s

The Integrated Circuit Revolution:

Microprocessors placed entire CPUs on single chips, dramatically reducing size and cost.

Personal Computers Emerge:

The PC revolution began, making computing accessible to businesses and individuals.

Uniprocessor Computing Era

Australian National University

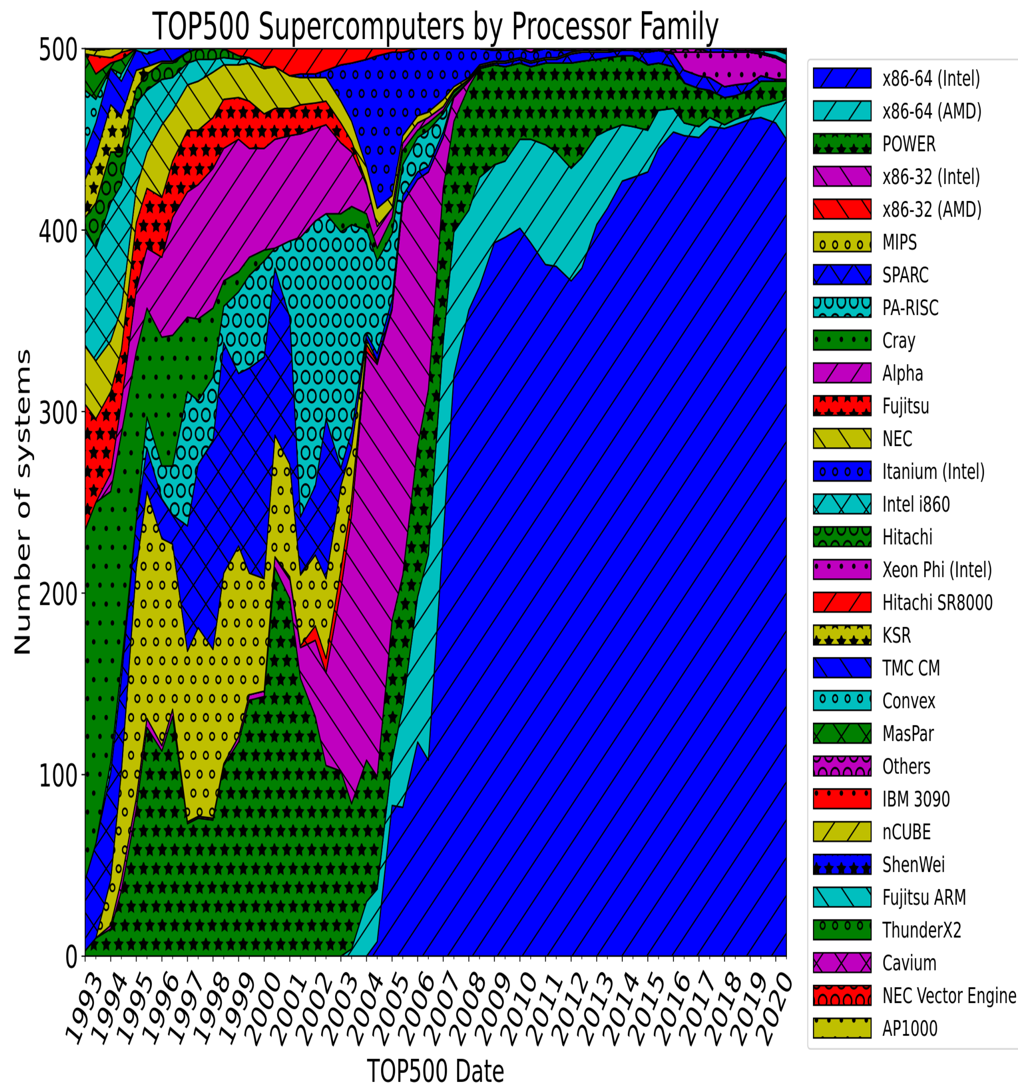
The Quest for Speed: 1980s-2000s

Focus on Clock Speed: Performance gains centered on increasing the rate at which a processor could execute instructions (measured in megahertz and later gigahertz).

Architectural Improvements: Innovations like pipelining and superscalar designs enhanced efficiency, allowing processors to do more per clock cycle.

Software Evolution: Operating systems and applications became more sophisticated, demanding faster hardware.

Uniprocessor Computing Era



Hitting the Wall: Early 2000s

Limits of Clock Speed: Increasing clock speeds ran into physical limitations due to heat dissipation, power consumption, and manufacturing hurdles.

The Need for a New Approach: The relentless focus on clock speed ultimately reached its limits, necessitating a shift towards parallel computing architectures to continue the performance growth trajectory.

The end of Dennard scaling (2002-2004)

Australian National University

With feature size below $K < 65$nm (currently 4nm)

$$P = QfCV^2 + VI_{\text{leakage}}{}^{1} \tag{4}$$

The leakage current grows exponentially with the voltage, as we decrease feature size K,

$$f = Kf_0, \qquad Q = K^2 Q_0, \qquad P_k = K^2 P_0 \tag{5}$$

To keep the same power envelope, large number of transistors are switched off (dark silicon effect), operated at lower frequencies (dim silicon effect) or organized in different ways.

[1] This is formally the correct equation, however for feature size $\geq$ 65 nm $I_{\text{leakage}} \sim 0$

The end of the uniprocessor era

Australian National University

Intel CPU Trends
(sources: Intel, Wikipedia, K. Olukotun)

Dual-Core Itanium 2
Pentium 4
Pentium
386

10,000,000
1,000,000
100,000
10,000
1,000
100
10
1
0

1970 1975 1980 1985 1990 1995 2000 2005 2010

- Transistors (000)
- Clock Speed (MHz)
- Power (W)
- Perf/Clock (ILP)

Clock rate capped (< 4 GHz) by power constraints

Instruction Level Parallelism (ILP), superscalarity and pipelining, saturated

Power saturated (limitations in heat removal)

The role of power consumption

Australian National University

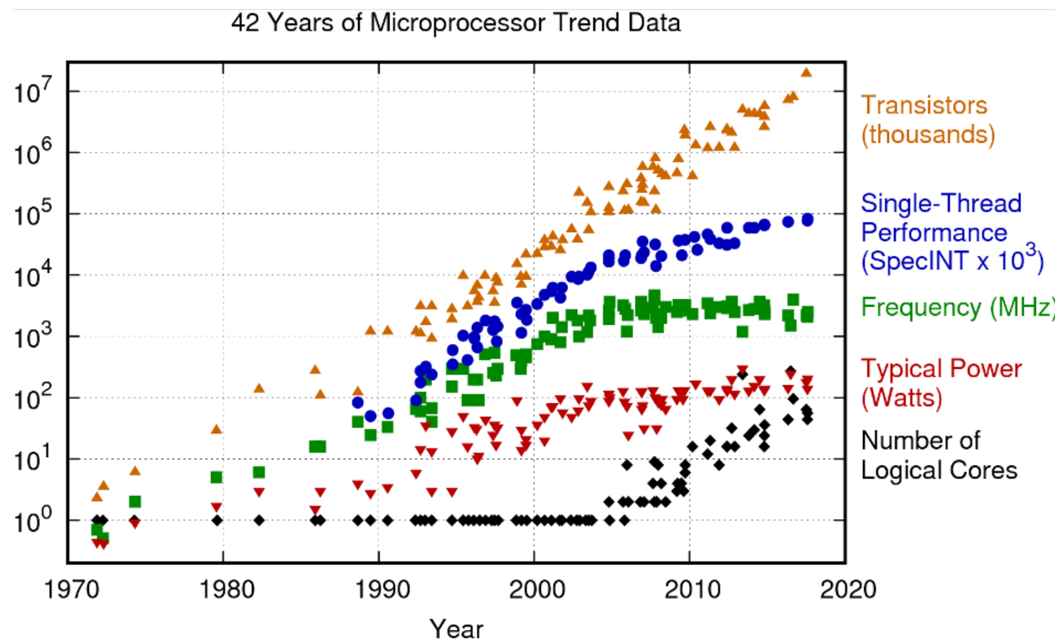
Why do we care so much about power?

- shifted from desktop to our smart phone
- democratization of HPC: e.g. ultrascale cloud computing (costs > 50˜ power)
- IoT
- physical limitations! Micro-processors are at their thermal limit.

Energy efficiency is the limitation

The multicore era

The only way to increase performance is to use parallel Computers (e.g. multi-core) efficiently.



- This is very different from the golden years of ILP where hardware architects did all the work for us.
- Programmers are now forced to bear the burden of finding and exploiting parallelism.
- This is also an exciting era of opportunities for computational scientists: new algorithms and efficient implementations make a difference on what is achievable in computing.

Australian
National
University

COMP4300/8300
Parallel Systems

Shared Memory Parallel Computing

Prof. John Taylor

School of Computing
Australian National University
Canberra, Australia

April 2024

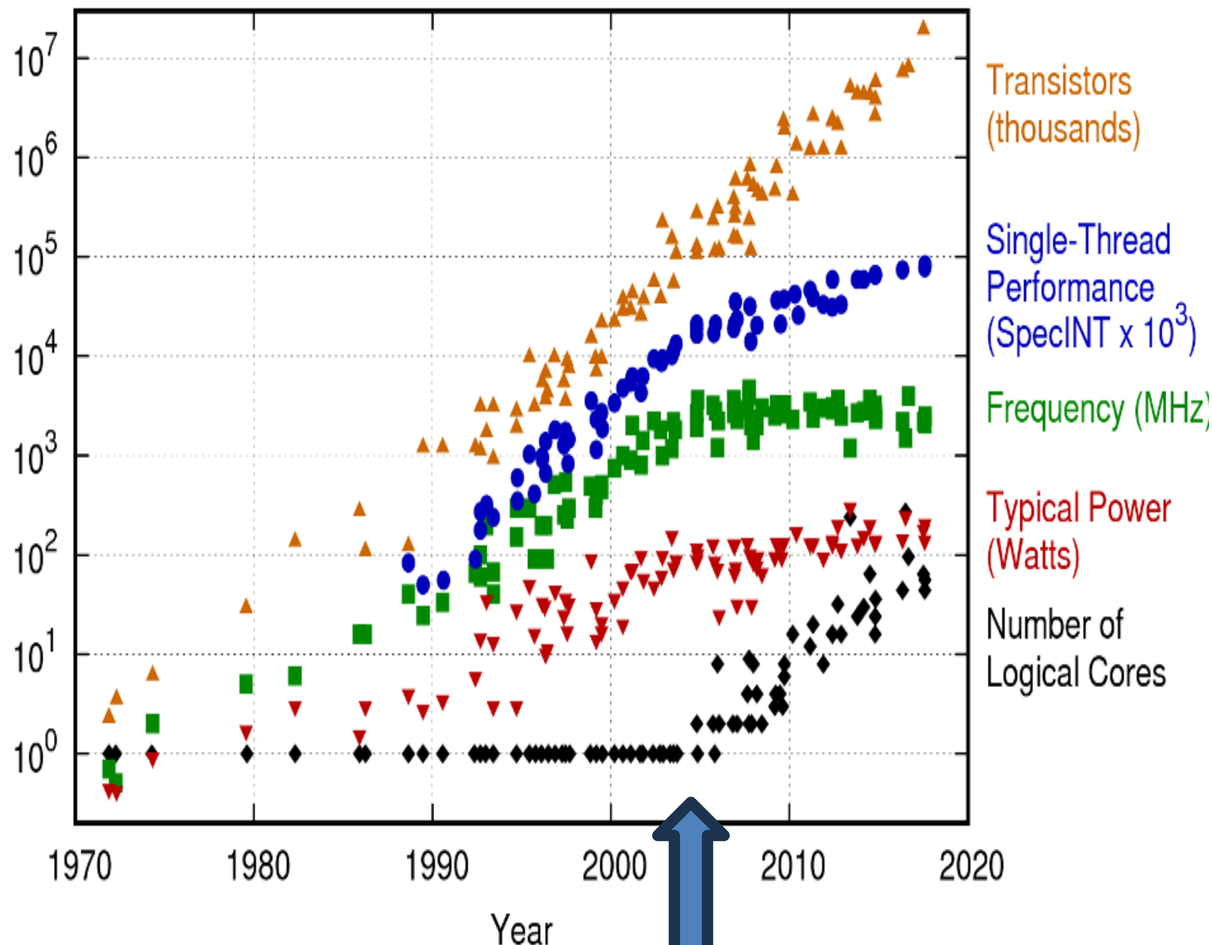
References

Australian National University

- The end of the road for general purpose processors & the future of computing. John Hennessy. https://www.hanahaus.com/blog-1/2019/1/3/newport-beach-a-fascinating-location-mpsfz-rwrbk
- The Free Lunch Is Over. A Fundamental Turn Toward Concurrency in Software. Herb Sutter. http://www.gotw.ca/publications/concurrency-ddj.htm
- Chapter 12 from Computer Systems A Programmer's Perspective, Third Edition, Randal E. Bryant and David R. O'Hallaron, Pearson Education Heg USA, ISBN 9781292101767.
- Programming with POSIX Threads, David R. Butenhof, Addison-Wesley Professional, ISBN-13： 978-0201633924.

Parallel Computers & Programming Models, Pthreads

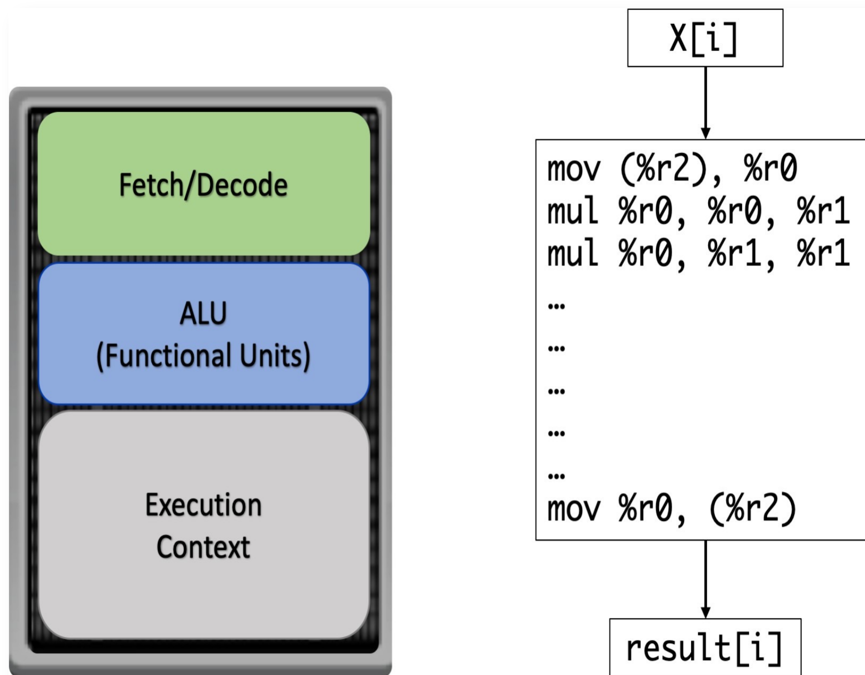
The end of the uniprocessor era



- Clock rates are now capped (< 4 GHz) by power constraints
- ILP (superscalarity and pipelining) is saturated
- Power usage saturated

The performance free lunch is over!

Limitations of ILP



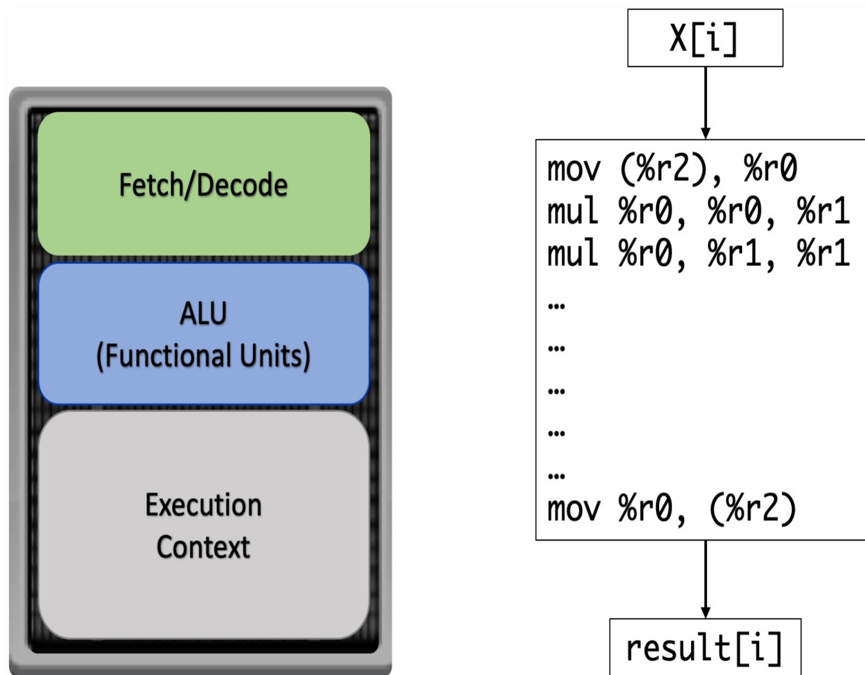
What is Instruction Level Parallelism (ILP)?

- ILP refers to a microarchitectural design principle that focuses on executing multiple instructions simultaneously within a single processor.

- The goal is to increase throughput (the number of instructions completed in a given time) without solely relying on increasing the clock speed of the processor.

Limitations of ILP

Australian National University



Significance of ILP

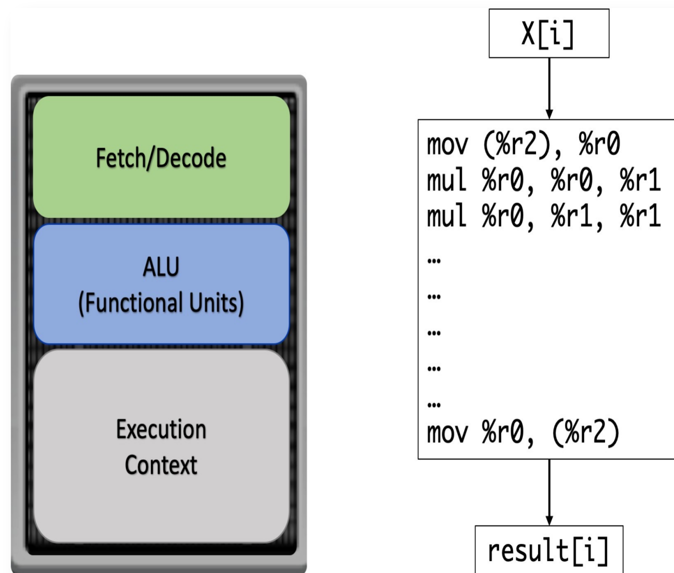
- **Performance Benefits:** ILP was essential in driving performance gains at a time when increasing CPU clock speeds was hitting physical limitations.

- **Foundation for Innovation:** Techniques developed for exploiting ILP form the basis of modern processor designs using multi-core and vector processing architectures.

Limitations of ILP

Australian National University

Techniques to Implement ILP:



Pipelining: Breaking down an instruction's execution into smaller stages (e.g., fetch, decode, execute, write back). This assembly-line-like approach allows multiple instructions to be in different stages of processing simultaneously. Pipelines can have 15+ stages.
Superscalar Execution: Having multiple execution units of the same type within a processor. This allows the processor to execute multiple similar instructions (e.g., multiple additions or multiplications) in parallel.
Out-of-Order Execution: Modern processors dynamically analyze instruction sequences, identify independent instructions, and re-order their execution if needed to increase parallelism and avoid stalls.
Branch Prediction: Speculatively executing instructions following a branch before the branch outcome is determined. This helps keep the pipeline filled even with conditional jumps.

Branches

Definition: Branches are instructions in code that cause the program flow to deviate from a linear sequence. This means that instead of executing the next instruction in order, the processor jumps to a different location in the code.

Types:

Conditional Branches: The decision to branch depends on a condition (e.g., if-else statements).

Unconditional Branches: The program always jumps to a different location (e.g., function calls, goto statements).

Code Example (Conditional Branch):

```
int x = 10;
if (x > 5) {
   // Code to execute if x is greater than 5
   printf("x is greater than 5\n");
} else {
   // Code to execute if x is less than or equal to 5
   printf("x is less than or equal to 5\n");
}
```

Code Example (Unconditional Branch):

```
void my_function() {
   // Some code here
}

int main() {
   // ... some other code
   my_function(); // Unconditional branch - jumps to my_function
   // ... code continues here after my_function returns
}
```

Australian National University

Memory Aliasing

Definition: Memory aliasing occurs when the same memory location can be accessed through multiple different names (pointers, variables, etc.).

Why it matters: This can lead to unexpected behavior if you modify the memory location through one name and then read its value through another name because the compiler might make assumptions about values not changing.

Code Example:

```
int *p = malloc(sizeof(int)); // Allocate memory
*p = 5;                       // Set the value at the memory location

int &q = *p;  // q is now an alias for the same memory location

*p = 10;      // Changes the value through the pointer p

printf("%d\n", q); // Will print 10, since q refers to the same location
```

How Branches and Memory Aliasing Interact

Optimization Challenges: Memory aliasing can make it difficult for compilers to apply certain performance optimizations. This is because the compiler cannot always be sure whether two pointers refer to the same memory location, leading to potentially conservative optimizations.

Branch Prediction Impact: Branches, especially conditional ones, introduce complexity in predicting which code path will be taken. Incorrect predictions can lead to performance penalties. Memory aliasing can further complicate this prediction process for a compiler.

Code Example:

The Challenge: If ptr1 and ptr2 happen to point to the same memory location (i.e., they are aliases), the compiler might assume that their values are independent. This could prevent optimizations like keeping the value of *ptr2 in a register, assuming it won't change unexpectedly.

```
int should_optimize = 0; // Some condition that may change

void some_function(int *ptr1, int *ptr2) {
  if (should_optimize) {
     *ptr1 = *ptr2 + 5;
  }
}
```

How to manage Aliasing

Minimize Aliasing: When feasible, try to limit situations where multiple pointers reference the same memory. When a compiler sees multiple pointers in your code, it has to assume that they might point to the same memory location (aliasing). This conservative approach is needed for correctness but can prevent these optimizations.

Compiler Directives: Some languages provide keywords or compiler flags (e.g., restrict in C) to give compilers hints about aliasing, allowing for better optimization opportunities. Using restrict incorrectly can lead to undefined behavior if you break your promise about aliasing. Use wisely!

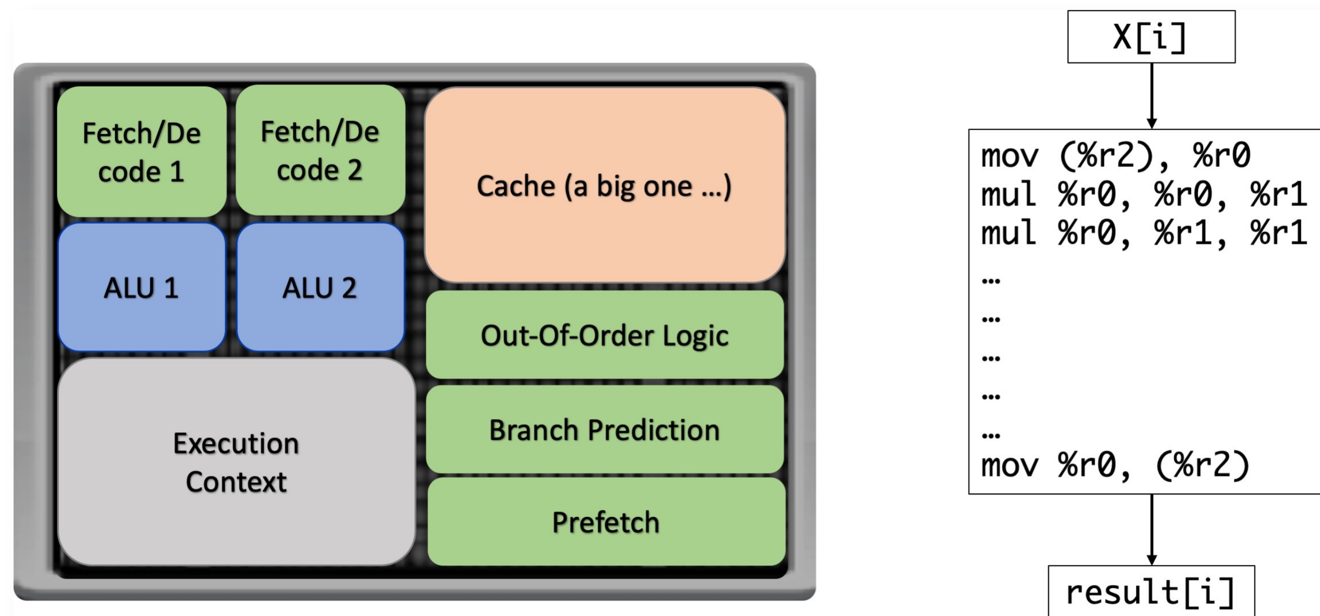
Careful with Optimization: Be aware that aggressive optimizations based on assumptions about memory aliasing might lead to unexpected behavior if those assumptions are violated in your program.

Code Example:

```
void my_function(int * restrict p, int * restrict q, int n) {
   for (int i = 0; i < n; i++) {
      p[i] = p[i] + q[i];
   }
}
```

Limitations of ILP

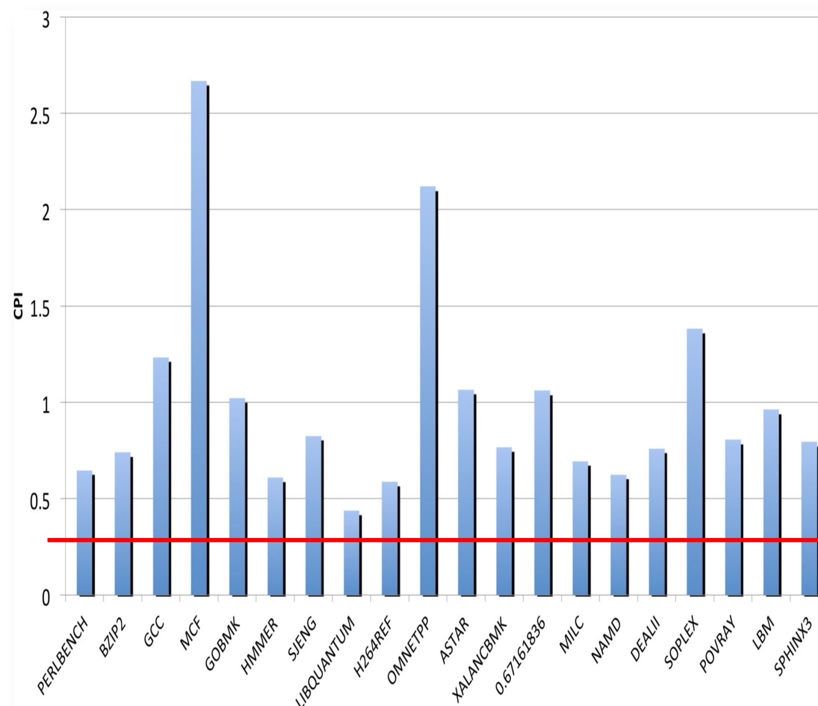
Large numbers of transistors are used for logic that help a single instruction stream run faster.



This had diminishing returns!

Limitations of ILP

Australian National University



Actual clock cycles per instruction (CPI) on Intel i7. Theoretical (in red) is 0.25.

Actual clock cycles per instruction (CPI)

In reality, the actual CPI is almost always higher than the theoretical best case due to various factors:

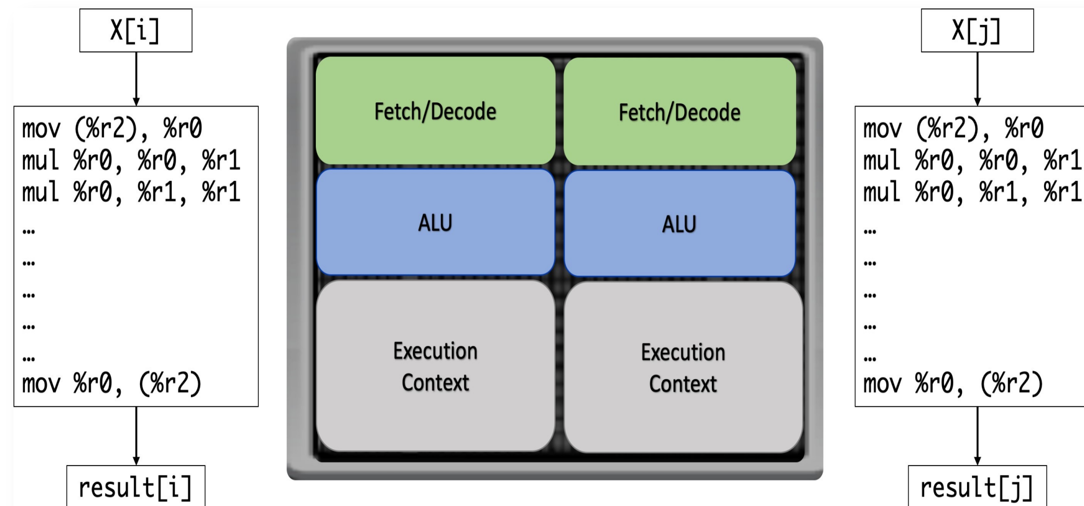
Instruction Complexity: Not all instructions take a single cycle. Complex operations (e.g., multiplication, floating-point operations) generally take more cycles.

Memory Access: If an instruction needs to load data from memory (especially slower levels of cache or RAM), it will introduce stalls, increasing CPI.

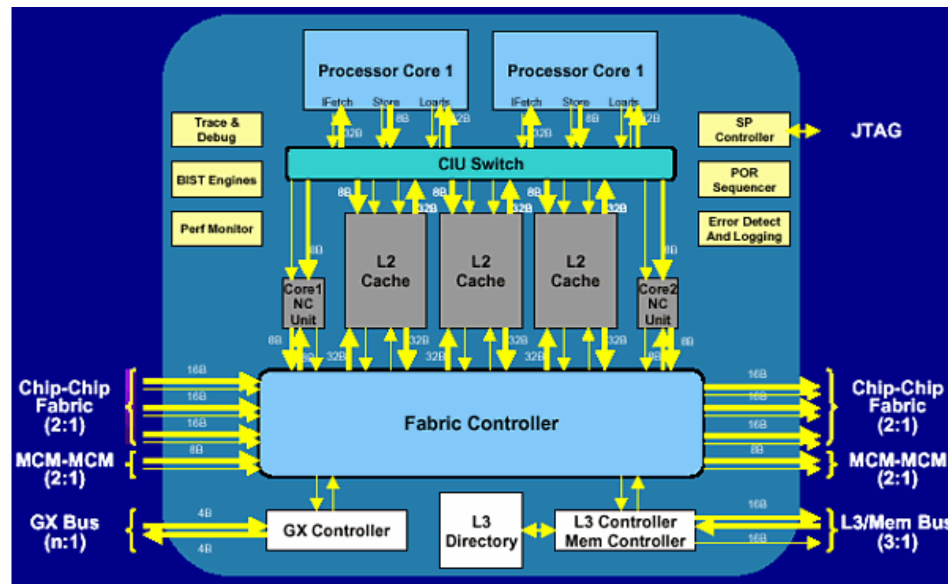
Branch Misprediction: If the processor incorrectly predicts which way a branch will go, it might need to flush the pipeline and start over, leading to wasted cycles and a higher CPI.

Resource Contention: If multiple instructions need to access the same execution units simultaneously, stalls can occur, increasing the CPI.

The multicore era



Rather than use transistors to increase the sophistication of single instruction stream, use additional transistors to add more cores on one die.



IBM Power4 – first multiprocessor (2001)
http://ixbtlabs.com/articles/ibmpower4/

2 cores, each one slightly slower (e.g. 0.75) than my original processor. Potential speedup 1.5x.

Problem: what happens if we run our program on this new processor?

The multicore era

Australian National University

Parallelism as the Norm

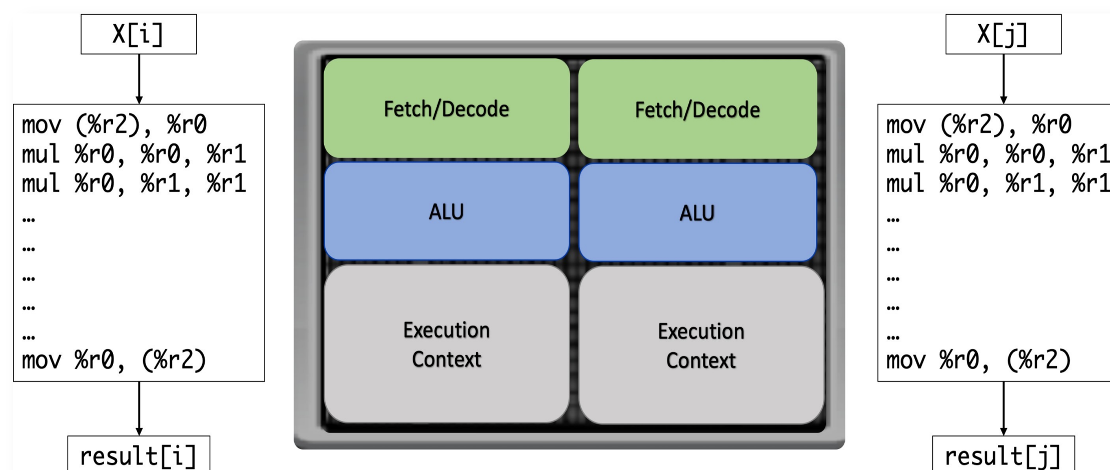
Multi-core computers have fundamentally changed the computing landscape.

Beyond Single Threads: Before multi-core processors, most software was written sequentially. Multi-cores shifted the focus to writing code that can take advantage of multiple cores simultaneously.

New Programming Challenges

Programmers now need to think about concurrency, synchronization, and how to effectively divide work among multiple cores.

Software Complexity: Programming for parallelism often requires specialized design, algorithms, and tools, adding complexity to development.
Amdahl's Law: Not all problems are easily parallelizable. Amdahl's Law recognizes limitations on speedup based on the inherently sequential portions of code.



The multi-core era

Australian National University

Performance Breakthroughs

Beyond Clock Speed Limits: Increasing clock speeds hit diminishing returns. Multi-core processors enabled continued performance growth by adding more cores instead of solely relying on faster single cores.

Scalability: Problems can be scaled across cores, often leading to significant speedups.

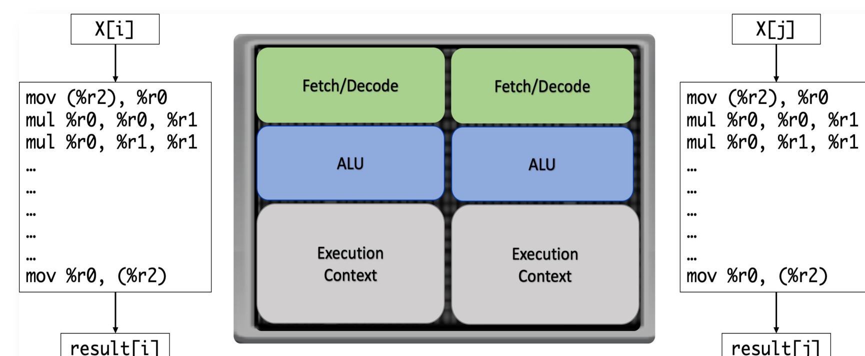
Improved User Experience: Even when a single program isn't fully parallelizable, users noticed better responsiveness because multiple programs can run in parallel on different cores.

Background Tasks: Operating systems can more smoothly manage background tasks, updates, and system services without heavily impacting foreground applications.

Computationally Intensive Workloads: Multi-core processors enabled demanding applications in scientific computing, machine learning, video editing, 3D rendering, and simulations that were previously not practical on consumer computers (HPC).

Real-Time Performance: Tasks requiring real-time execution and handling multiple inputs at once benefited greatly.

Targeted Power Usage: Not every task needs all cores at full power. This allows for dynamic power management and energy savings.



Using Multiple Cores

Concurrency

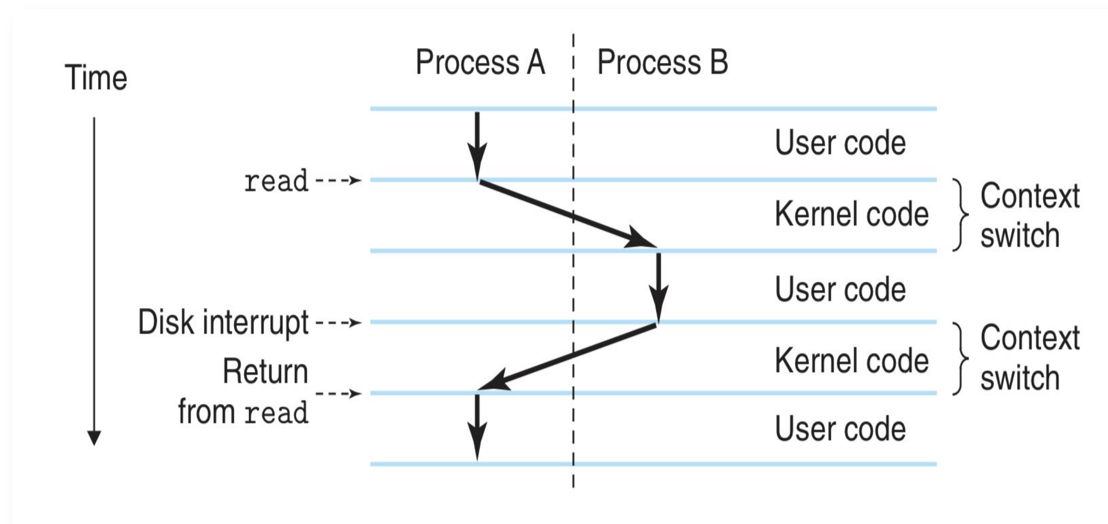
A UNIX *process* is the Operating System's abstraction for a running program.

Multiprocessing is the ability of the OS to run more than one process *concurrently*.

Concurrency is about the structure of a program and how it handles multiple tasks. A concurrent program has parts that can make progress seemingly at the same time or in an interleaved fashion.

Focus: Concurrency is about logical multitasking – managing multiple things in progress at the same time.

Single Core Example: Think of a web server handling multiple requests at the same time ie *concurrently*. It might not be processing them simultaneously, but it switches between requests quickly, giving the illusion of parallel activity.



Parallelism

Definition: Parallelism is about the physical execution of a program. A parallel program has parts that literally run at the same time by using multiple processing units (e.g., cores, processors).

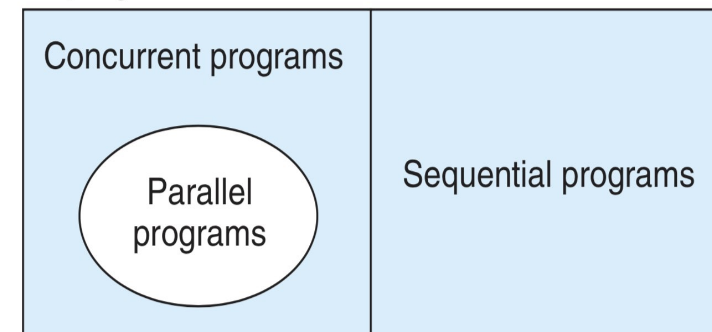
Parallelism is about using multiple processing resources to speed up computational tasks by dividing the work.

Multi-Core Example: A complex image processing task that splits an image into sections and processes each section simultaneously on different cores of a multi-core processor.

Key Points

All parallel programs are concurrent: If a program is running on multiple cores at once, it's also concurrent.

Not all concurrent programs are parallel: A single-core system can achieve concurrency through time-slicing and context switching between tasks.



Australian National University

Example: The UNIX Operating system

The OS is ultimately responsible to map processes efficiently to the hardware, e.g. map different processes to different cores, if available.

Each process is assigned a process identifier (PID) that is unique across the entire system (try using the "ps" or "top" programs).

Processes usually correspond to completely different programs with their own set of instructions, global data, stack and heap (a different virtual address space).

UNIX processes can, however, communicate using pipe(), socket() and various OS-supported shared-memory areas.

Code example: Getting Process Information

```
#include <unistd.h>
#include <stdio.h>

int main() {
   pid_t pid = getpid();  // Get current process ID
   pid_t ppid = getppid(); // Get parent process ID

   printf("Process ID: %d\n", pid);
   printf("Parent Process ID: %d\n", ppid);
   return 0;
}
```

Operating-System-Supported Multiprocessing

Australian National University

Create a new UNIX process

Any UNIX process can spawn new processes (called “children”) using the UNIX fork() utility.

fork() creates a clone of the parent program replicating the code, global variables and stack.

In order to use fork() you have to #include <unistd.h>

Parent and child: The only difference between the parent and the child is that the latter returns from fork() with 0, while the parent returns with the PID of the child

The execution of the parent/child resumes from the fork() call

Code Example

```
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>

int main() {
   pid_t pid = fork();

   if (pid == 0) {
      // Child process code
   } else if (pid > 0) {
      // Parent process code
      wait(NULL); // Wait for child to finish
   } else {
      // Fork failed
      perror("fork");
   }

   return 0;
}
```

Australian National University

When to use fork()

It is possible to use basic UNIX utilities like fork(), socket() and pipe() and shared memory segments to generate a parallel program using multiple cores, however…

Context switching: fork() is a "heavy weight" operation. fork() incurs overhead due to the operating system's scheduler

Memory Overhead: taking substantial time to make an identical copy of the parent process, e.g. it may have to replicate a process that has many GB of memory.

Modern programming often offers better multithreading solutions for fine-grained parallelism.

fork() is, however, used in a number of message passing protocols whereby multiple copies of a program are created once at the beginning of execution and then communicate via a series of messages.

Message passing makes it complicated and potentially inefficient for flows to communicate, especially if they share large amounts of data.