

An introduction to heterogenous computing

Accelerating numerical codes

Krishna Kumar *¹

August 11, 2015

¹github.com/kks32

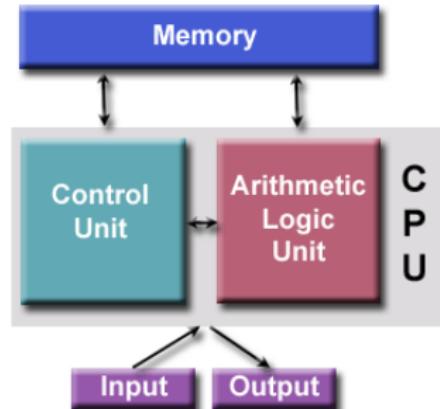
Outline

1 Overview: Architecture

2 Parallel computing

von Neumann Architecture

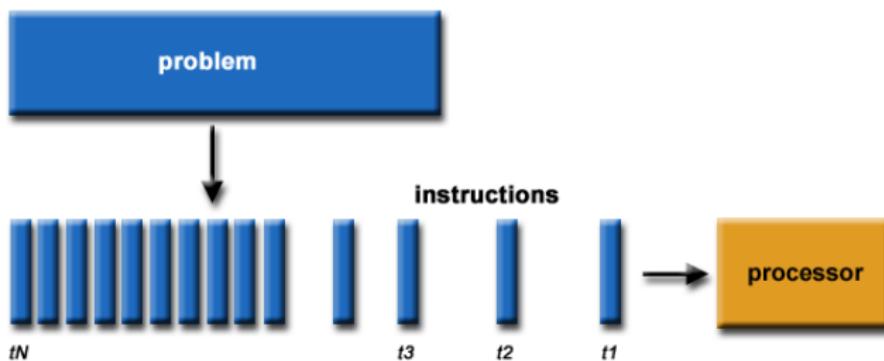
- Hungarian mathematician John von Neumann circa 1940 - the general requirements for an electronic computer.
- “Stored-program computer” - both program instructions and data are kept in electronic memory.
 - *Read/write*, random access memory is used to store both program instructions and data.
 - *Control unit* fetches instructions/data from memory, decodes the instructions and then sequentially coordinates operations to accomplish the programmed task.
 - *Arithmetic Unit* performs basic arithmetic operations
 - *Input/Output* is the interface to the human operator



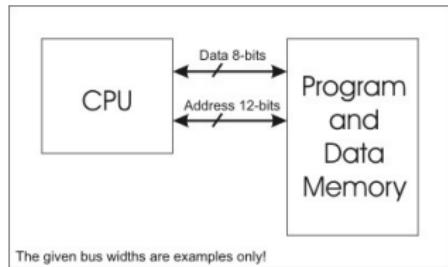
Basic architecture

Serial computing

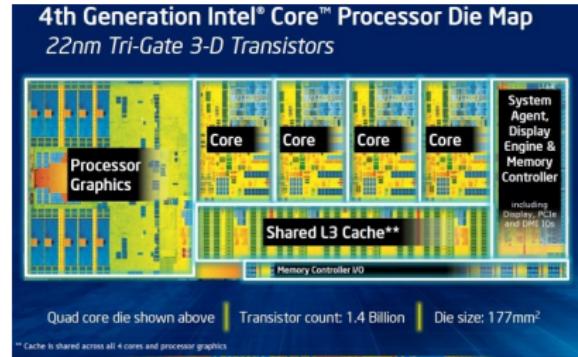
- Traditionally, software has been written for serial computation:
 - A problem is broken into a discrete series of instructions
 - Instructions are executed sequentially one after another
 - Executed on a single processor
 - Only one instruction may execute at any moment in time



Memory model



Ideal memory model:
We write for this architecture



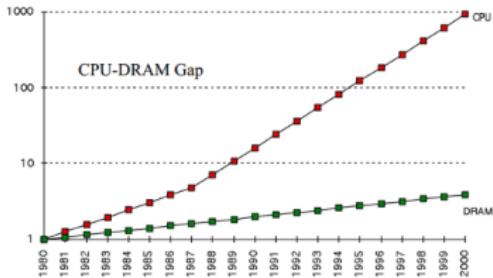
Real memory model: How it actually looks

The underlying assumption is cache coherency!

In a shared memory multiprocessor with a separate cache memory for each processor, it is possible to have many copies of any one instruction operand: one copy in the main memory and one in each cache memory. When one copy of an operand is changed, the other copies of the operand must be changed also. Cache coherency ensures that changes in the values of shared operands are propagated throughout the system in a timely fashion.

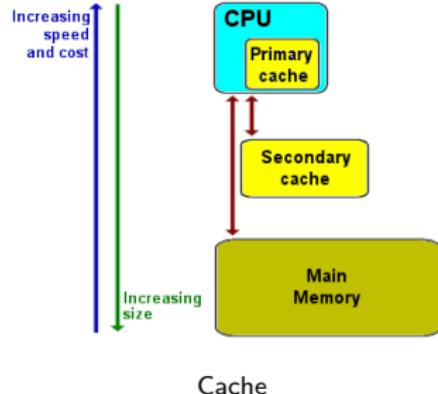
What are caches

- Processor vs Memory Performance



1980: no cache in microprocessor;
1995 2-level cache

CPU vs DRAM



- CPU caches are small pools of memory that store information the CPU is most likely to need next.
- A cache miss means the CPU has to go scampering off to find the data elsewhere. This is where the L2 cache comes into play while it's slower, it's also much larger.
- If data can't be found in the L2 cache, the CPU continues down the chain to L3 (typically still on-die), then L4 (if it exists) and main memory (DRAM).

Does your compiler execute the program you wrote?

No, absolutely not! Compiler most often says “you didn’t intend to write that. I have a better idea...”

- *Sequential consistency*: Executing the program you wrote.
“the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.” - Lesslie Lamport
- *Compiler optimisation*
- *Processor execution*
- *Cache coherency*
- Chip / compiler design annoyingly helpful:
 - It can be expensive to exactly execute what you wrote
 - Often they rather do something else, that's faster

Does your compiler execute the program you wrote?

```
// Your code
1 for (i = 0; i < rows; ++i) {
2     for (j = 0; j < cols; ++j) {
3         a[j*rows+i]+=42;
4     }
5 }
6 }
```

```
// Compiler optimised version
1 for (j = 0; j < cols; ++j) {
2     for (i = 0; i < rows; ++i) {
3         a[j*rows+i]+=42;
4     }
5 }
6 }
```

The CPU will expect a sequential operation. Iterating through each row of data is faster than going through each column. Almost always, a 2D matrix is stored as a 1D linear array.

Stack v Heap

- **Stack:** Stores local data, return addresses, used for parameter passing. e.g., int i;
- typically stored in the “low” addresses of memory and fills upward toward its upper limit.
- faster, but smaller in size. Last In First Out.
- **Heap:** You would use the heap if you don't know exactly how much data you will need at runtime or if you need to allocate a lot of data.
*ClassObj * obj = new ClassObj{0}; ... delete obj;
auto obj = std::make_shared<ClassObj>();*
- Stored at the “top” of the address space and grows towards the stack.
- Slower but larger in size
- Use local variables (stack) when you can. Use dynamic allocation (heap) when you have to.



Stack v Heap

Outline

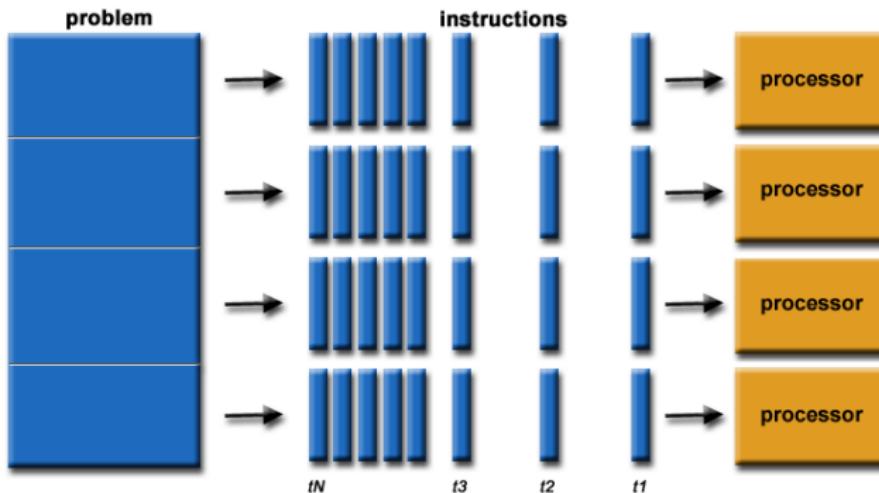
1 Overview: Architecture

2 Parallel computing

What is parallel computing?

In the simplest sense, parallel computing is the simultaneous use of multiple compute resources to solve a computational problem:

- A problem is broken into discrete parts that can be solved concurrently
- Each part is further broken down to a series of instructions
- Instructions from each part execute simultaneously on different processors
- An overall control/coordination mechanism is employed



Concurrency v Parallelism

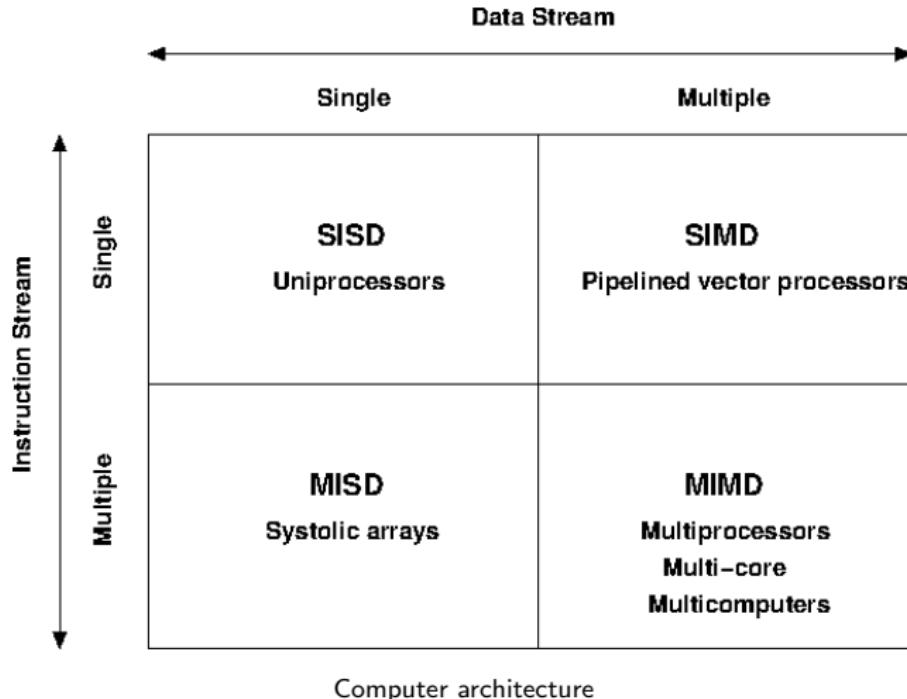


Concurrency



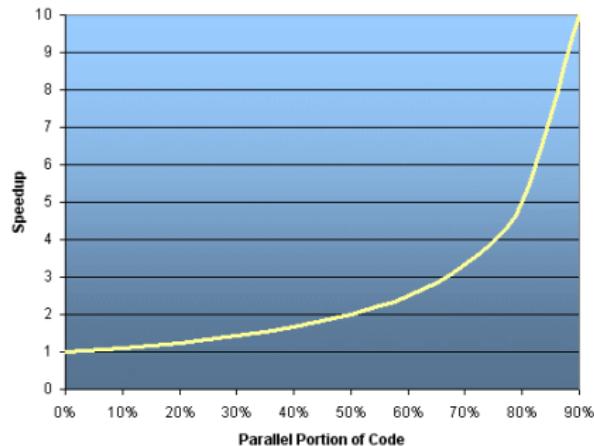
parallelism

Flynn's Classical Taxonomy



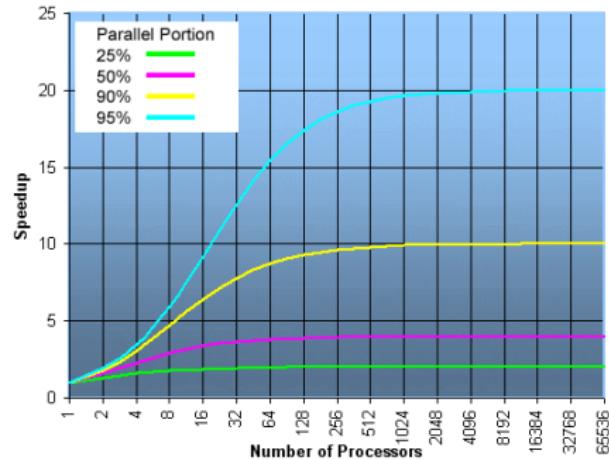
Cost of parallelisation

$$speedup = \frac{1}{1 - \text{parallelpart}}$$



Single processor

$$speedup = \frac{1}{\frac{\text{parallelpart}}{\#\text{processors}} + \text{serialpart}}$$



Multiple processor

Problems that increase the percentage of parallel time with their size are more **scalable** than problems with a fixed percentage of parallel time

Scalability

- *Strong scaling:* The total problem size stays fixed as more processors are added.
- *Weak scaling:* The problem size per processor stays fixed as more processors are added.
- Simply adding more processors is rarely the answer to scalability.
- The algorithm may have inherent limits to scalability. At some point, adding more resources causes performance to decrease.
- Hardware factors play a significant role in scalability. Examples:
 - Memory-cpu bus bandwidth on Symmetric Multi-Processor (SMP) - where multiple processors share a single address space and have equal access to all resources.
 - Communications network bandwidth
 - Amount of memory available on any given machine or set of machines
 - Processor clock speed
- Parallel support libraries and subsystems software can limit scalability independent of your application.