An introduction to heterogenous computing

Accelerating numerical codes

Krishna Kumar *[1]

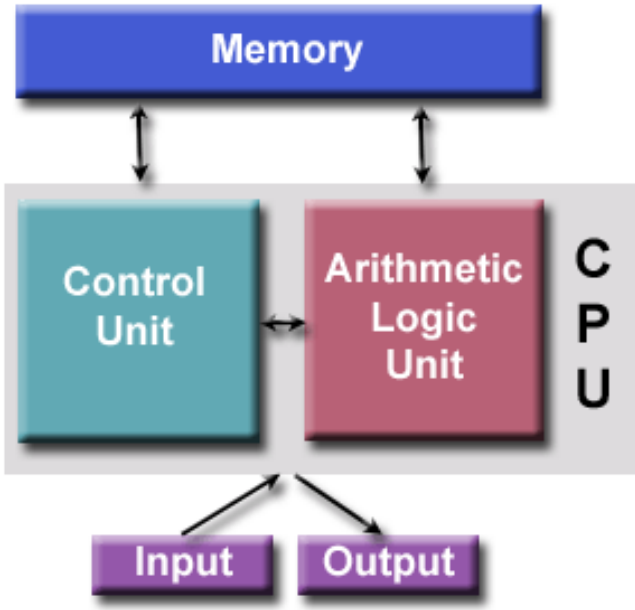
August 11, 2015

[1] github.com/kks32

Outline

von Neumann Architecture

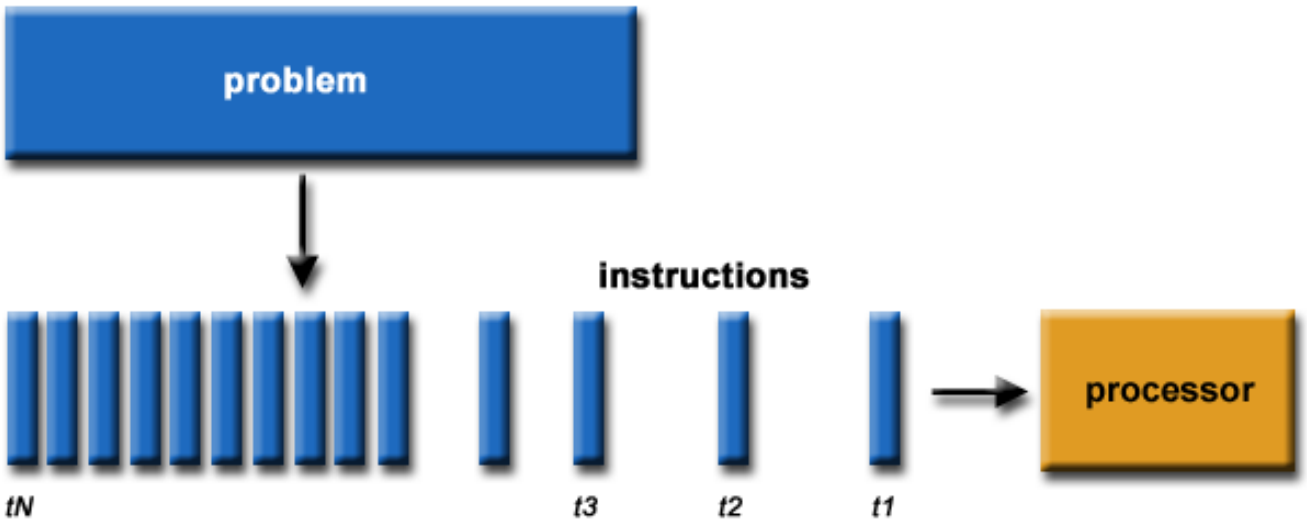
- Hungarian mathematician John von Neumann circa 1940 - the general requirements for an electronic computer.
- "Stored-program computer" - both program instructions and data are kept in electronic memory.
  - *Read/write*, random access memory is used to store both program instructions and data.
  -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



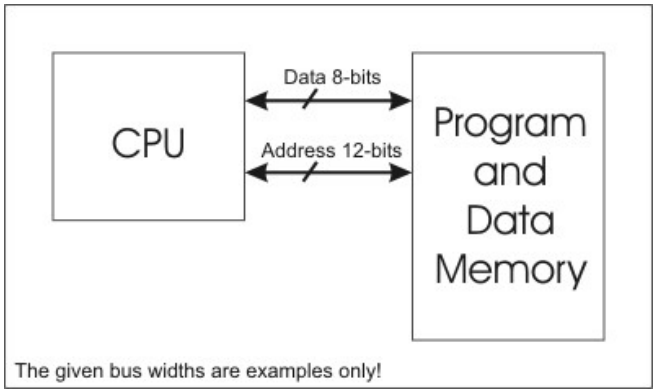
Basic architecture

Serial computing

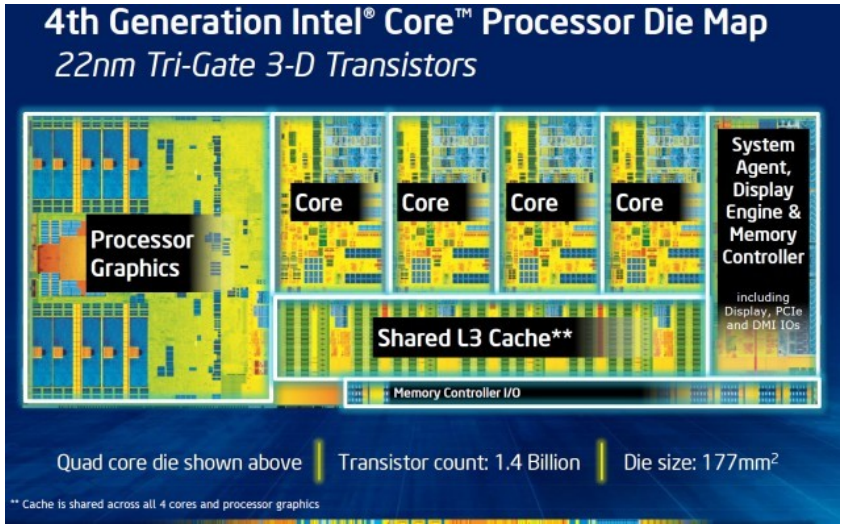
- Traditionally, software has been written for serial computation:
  - A problem is broken into a discrete series of instructions
  - Instructions are executed sequentially one after another
  - Executed on a single processor
  - Only one instruction may execute at any moment in time



Memory model



Ideal memory model:
We write for this architecture



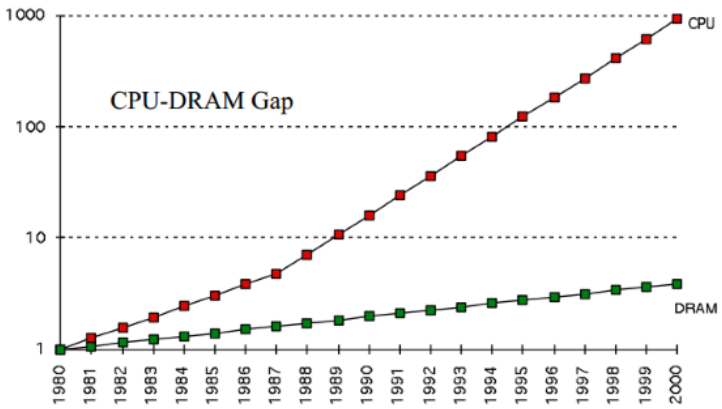
Real memory model: How it actually looks

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.

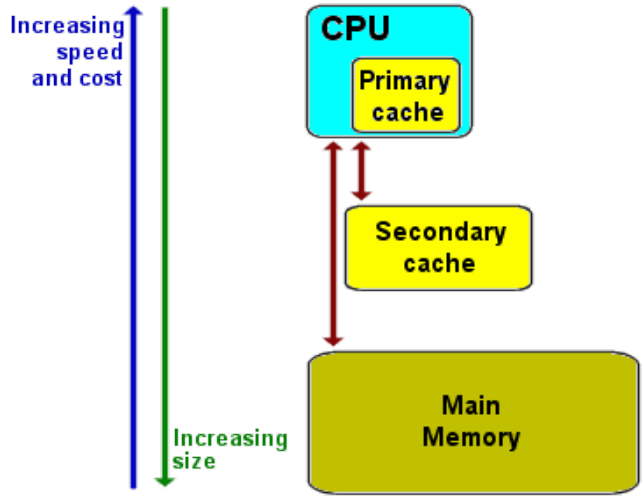
What are caches

- Processor vs Memory Performance



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

CPU vs DRAM



Cache

- 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?

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

```
1 // Compiler optimised version
2 for (j = 0; j < cols; ++j) {
3   for (i = 0; i < rows; ++i) {
4     a[j*rows+i]+=42;
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 Fist 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. $ClassObj * obj = newClassObj\{0\}; ...deleteobj;$ $autoobj = std :: make\_shared < ClassObj > (0);$
- 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.

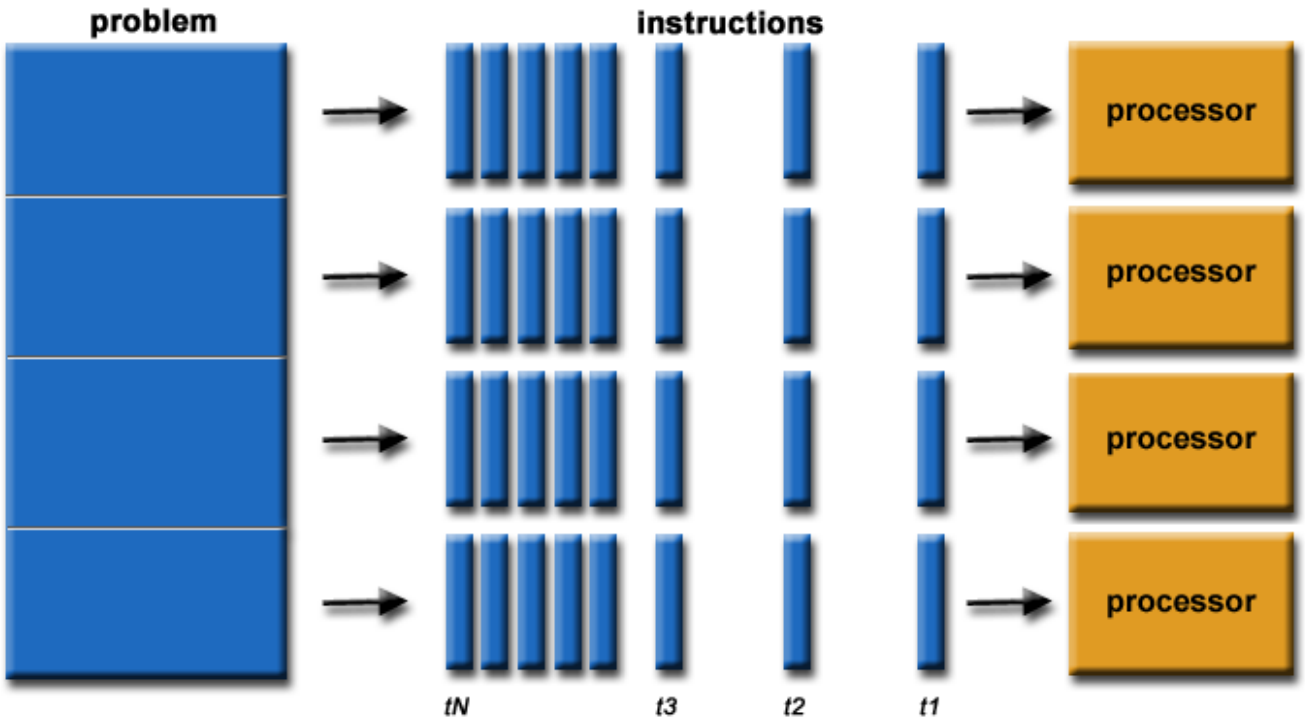
Stack v Heap

Outline

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

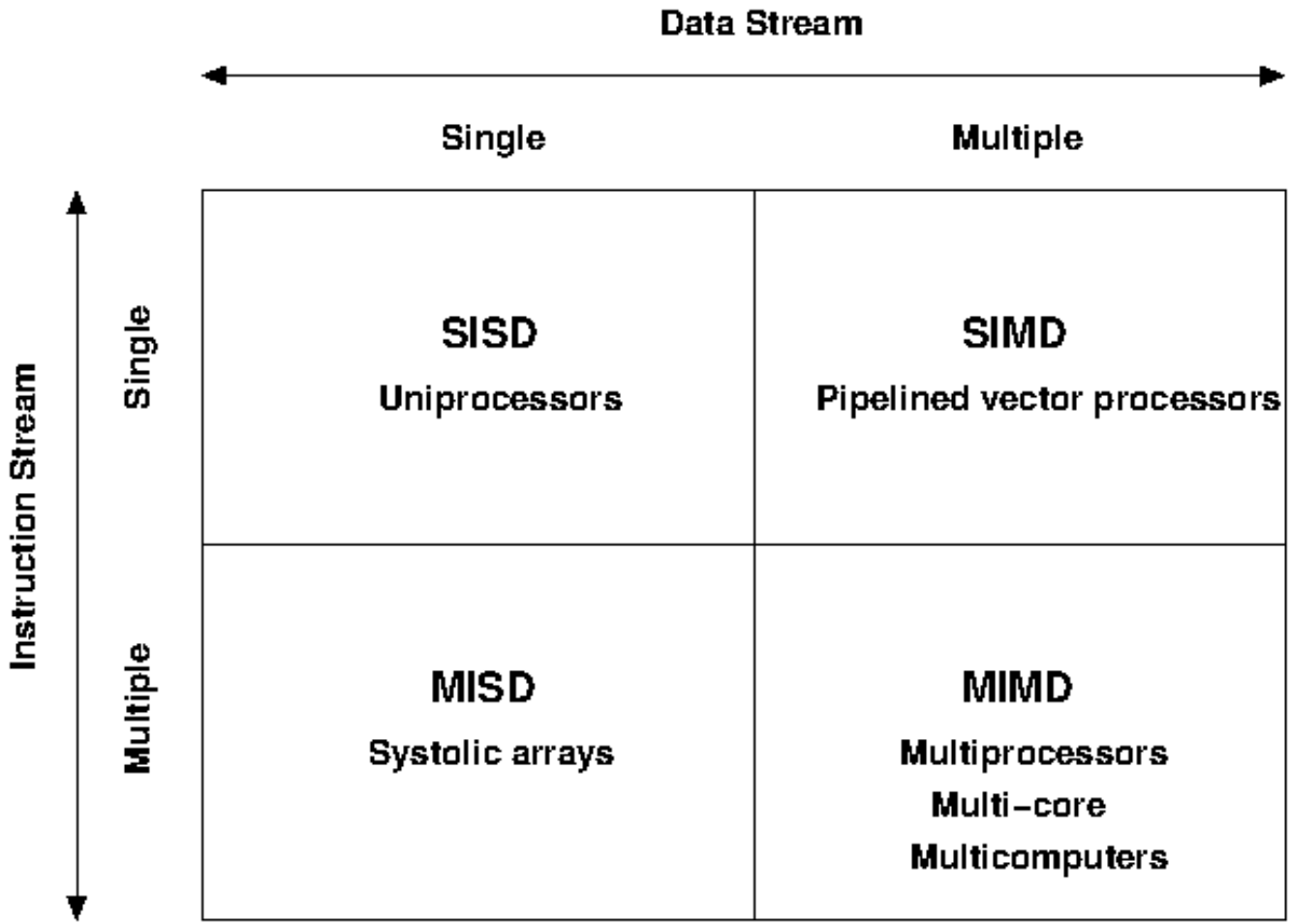


Concurrency v Parallelism

Concurrency

parallelism

Flynn's Classical Taxonomy

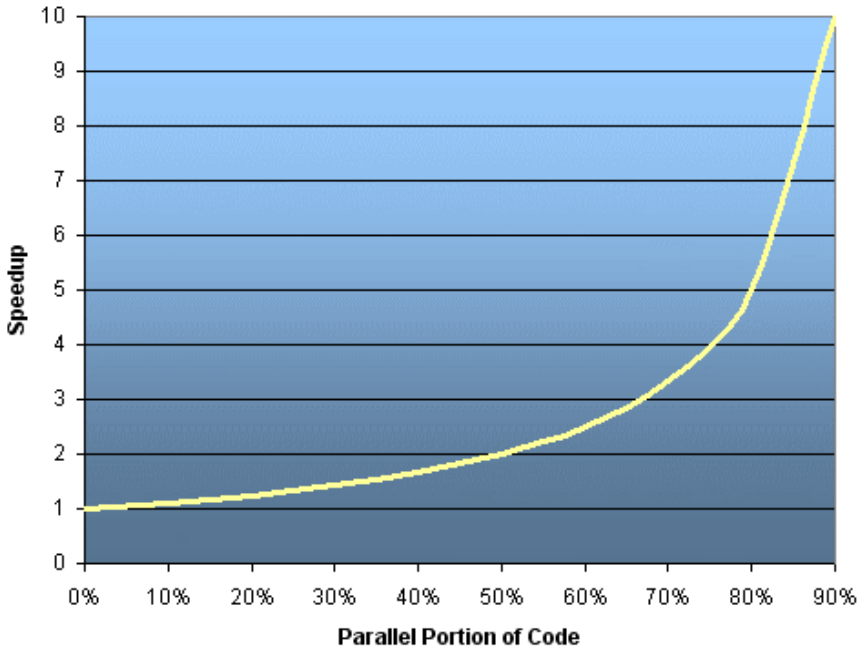


Computer architecture

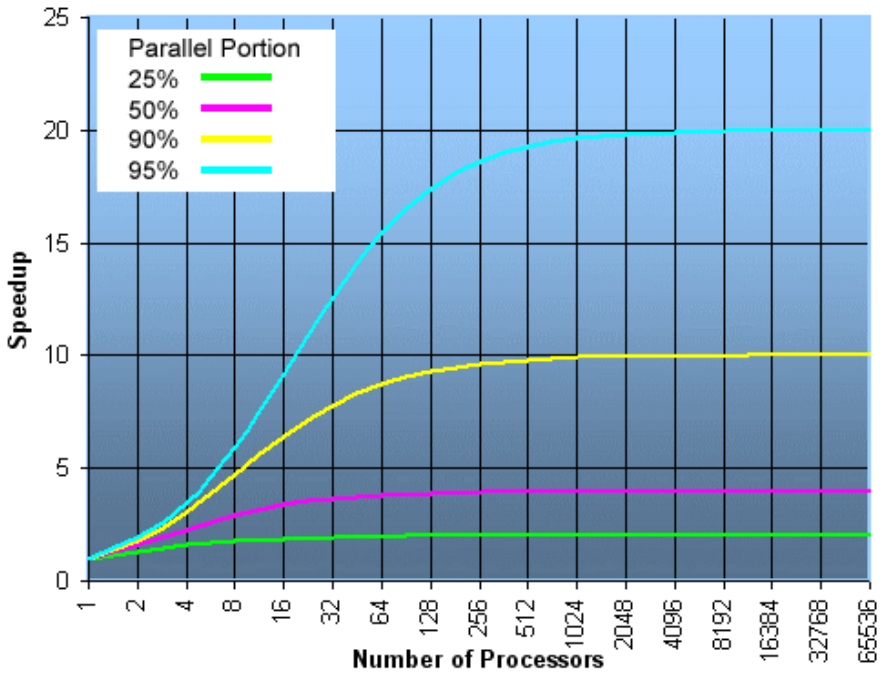
Cost of parallelisation

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

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



Single processor



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

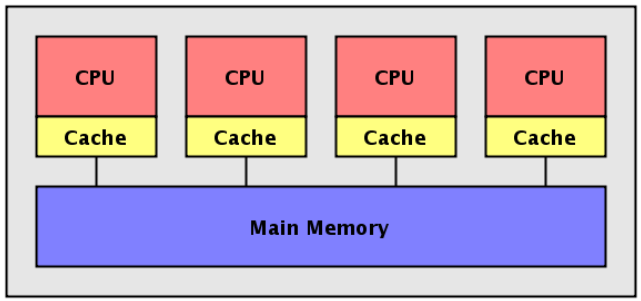
Scalability

- *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.

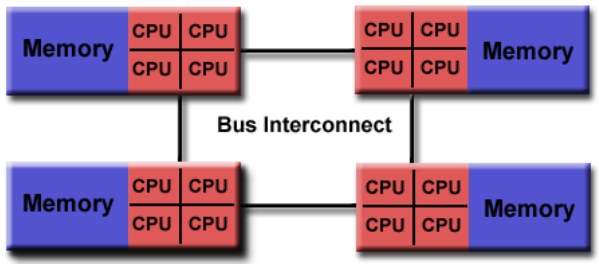
Outline

Shared memory

- Multiple processors can operate independently but share the same memory resources.
- Changes in a memory location effected by one processor are visible to all other processors.

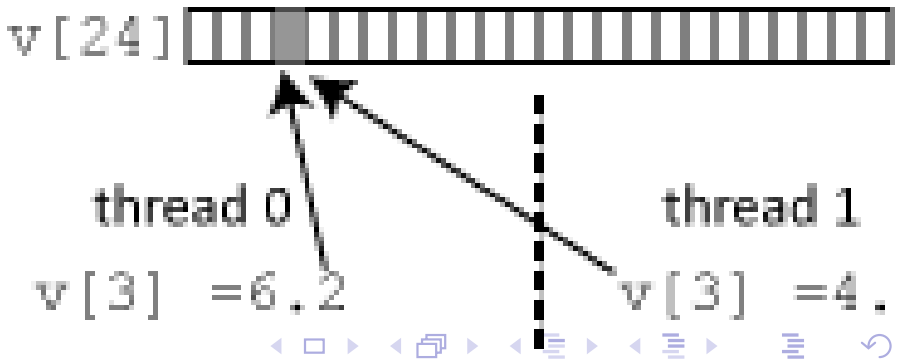


Shared memory



Non Uniform Memory access

If we want thread 0 to use the value placed in the array by thread 1, we needs to use a mechanism which assures that thread 1 has written the value before thread 0 reads it.



Shared memory frameworks

- **OpenMP**
  - API that supports multi-platform shared memory multiprocessing programming
  - Model: master thread forks
  - Simple to use
- **P-threads**
  - A set of C programming language types, functions and constants.
  - A thread can be created with much less operating system overhead.
  - No advanced threading algorithm. Use C++11 with same set of features.
- **C++11 threads**
  - C++11 language provides a memory model that supports threading.
  - This library provides everything from thread management to mutexes and synchronization.
- **Intel Cilk**
  - Language extension.
  - Simple to parallelise.
  - No advanced algorithm support.
- **Intel Threaded Building Blocks**
  - C++ template library for parallelisation.
  - Support for parallel algorithms, data structures and is scalable.

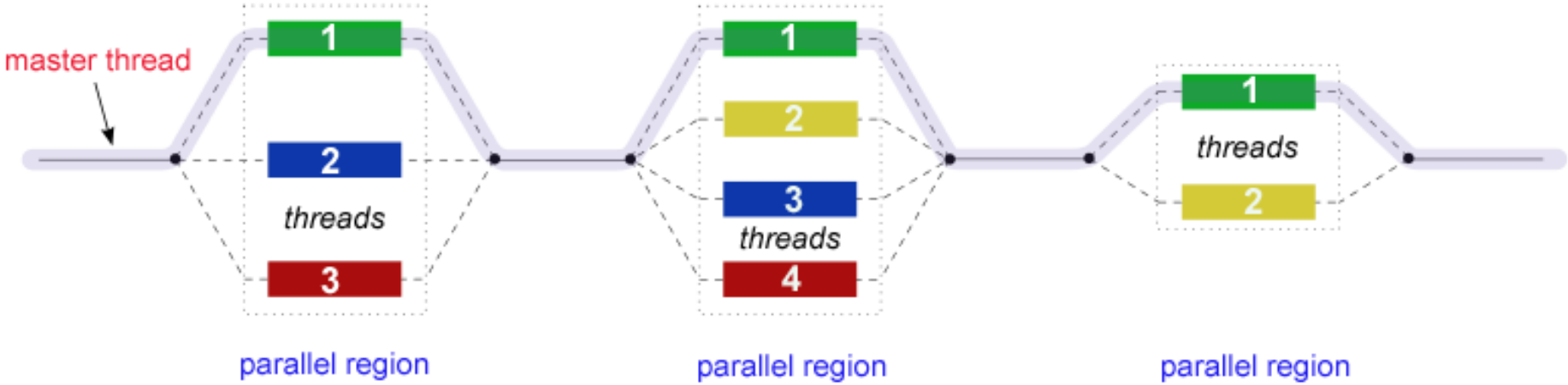
Shared memory: Choosing the right framework

OpenMP	Intel Cilk Plus	Intel TBB	C++11 Threads	PThreads
+ Options	+ Performance	+ Possibilities	+ Flexibility	+ Flexibility
+ Portable	+ Scaling	+ Management	+ Type-Safety	+ Low-Level
+ Languages	+ Variables	+ Maintenance	+ Possibilities	+ Compatibility
- Performance	- Fortran	- Language	- Fortran	- Efforts
- Memory	- Control	- OOP	- Compiler	- Type-Safety
- Unreliable	- Possibilities	- Control	- Scaling	- Management

OpenMP

- Open Multi-Processing is an API to explicitly direct multi- threaded, shared memory parallelism
- Comprised of three primary API components:
  - Compiler Directives Pragmas (pre-processor macros)
  - Runtime Library Routines
  - Environment Variables
- The programmer is responsible for synchronizing input and output.
- OpenMP uses threads. A thread of execution is the smallest unit of processing that can be scheduled by an operating system.
- Threads exist within the resources of a single process. Without the process, they cease to exist.
- Typically, the number of threads match the number of machine processors/cores. However, the actual use of threads is up to the application.

Fork Join Model



Fork - Join model

- **FORK**: the master thread then creates a team of parallel threads. The statements in the program that are enclosed by the parallel region construct are then executed in parallel among the various team threads.
- **JOIN**: When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread.
- The number of parallel regions and the threads that comprise them are arbitrary.

OpenMP: Code structure

```
1  #include <omp.h>
2  int main () {
3    int var1, var2, var3;
4    // Serial code . . .
5    // Beginning of parallel section. Fork a team of threads.
6    //Specify variable scoping
7  #pragma omp parallel for private(var1, var2) shared(var3) {
8      // Parallel section executed by all threads
9      // Run-time Library calls
10     // All threads join master thread and disband
11   }
12   // Resume serial code ..
13 }
```

OpenMP: Dot product

Dot product $sum+ = a[N] * b[N]$

```
// serial code
const int size = 100;
int main() {
  int i;
  float a[size], b[size];
  float sum = 0.0;
  // Some initializations
  for (i = 0; i < size; i++)
    a[i] = b[i] = 1.0;
  // computation loop
  for (i = 0; i < size; i++)
    sum += (a[i] * b[i]);

  printf("Sum = %f\n", sum);
}
```

```
// parallel code
const int size = 100;
int main() {
  int i;  float sum = 0.0;
  float a[size], b[size];

  for (i = 0; i < size; i++)
    a[i] = b[i] = 1.0;

#pragma omp parallel for \
default(shared) private(i)
  for (i = 0; i < size; i++)
    sum += (a[i] * b[i]);

  printf("Sum = %f\n", sum);
}
```

Parallel dot product: What went wrong?

```
1  // wrong code
2  const int size = 100;
3  int main() {
4    int i;  float sum = 0.0;
5    float a[size], b[size];
6    for (i = 0; i < size; i++)
7      a[i] = b[i] = 1.0;
8
9  #pragma omp parallel for \
10 default(shared) private(i)
11   for (i = 0; i < size; i++)
12     sum += (a[i] * b[i]);
13
14   printf("Sum = %f\n", sum);
15 }
```

```
1  // parallel code
2  const int size = 100;
3  int main() {
4    int i;  float sum = 0.0;
5    float a[size], b[size];
6    for (i = 0; i < size; i++)
7      a[i] = b[i] = 1.0;
8
9  #pragma omp parallel for \
10 default(shared) private(i) \
11     reduction(+ : sum)
12   for (i = 0; i < size; i++)
13     sum += (a[i] * b[i]);
14
15   printf("Sum = %f\n", sum);
16 }
```

Synchronisation! Don't forget to synchronise the threads