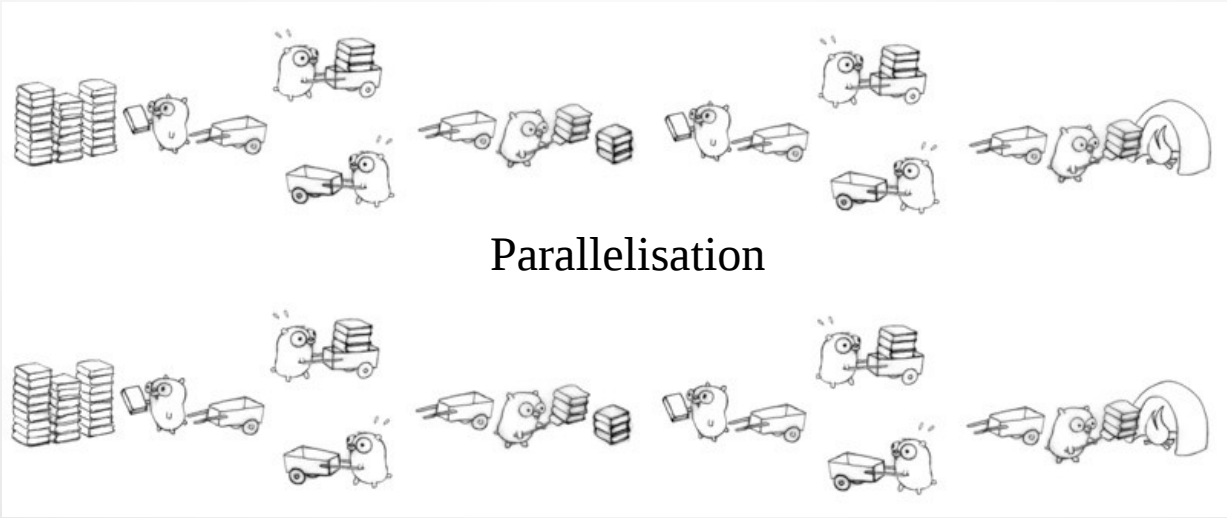
C++ Parallelisation

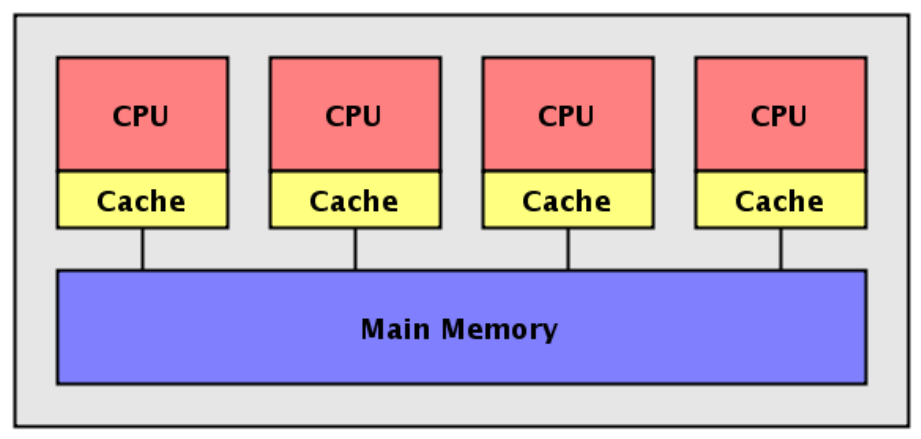
Concurrency

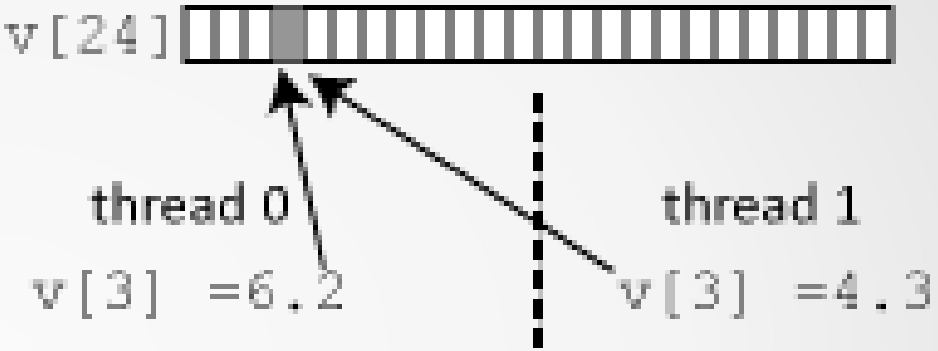


Parallelisation

Krishna Kumar

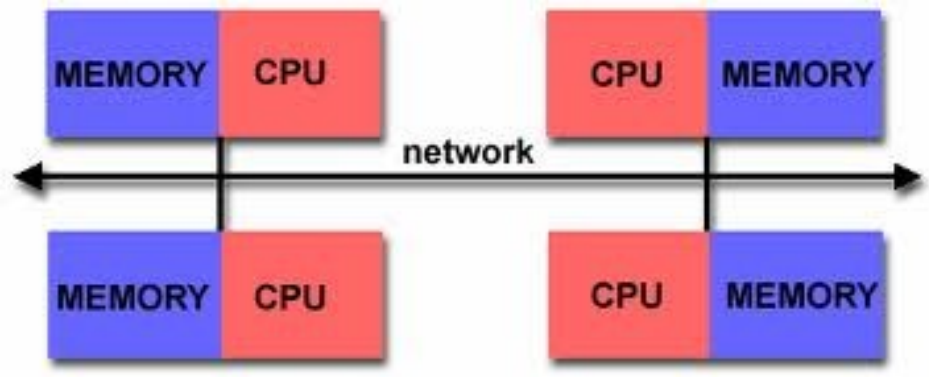
Shared memory

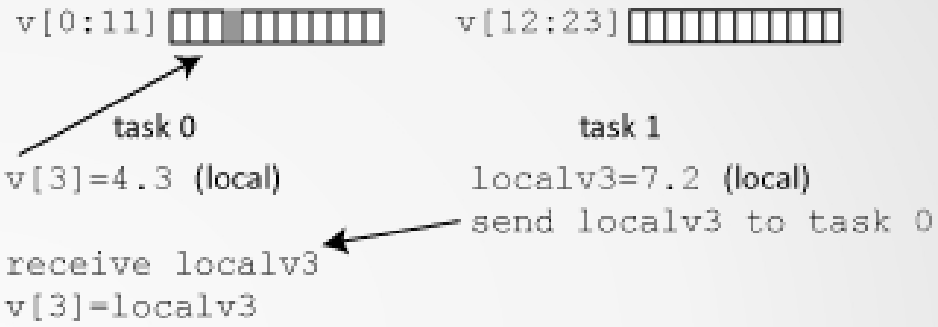




- two threads of execution can both address the same variables in a uniform manner, hereby assigning to an element of a vector whose components are in the virtual memory of the task.
- If the programmer wants thread 0 to use the value placed in the array by thread 1, he needs to use a mechanism which assures him that thread 1 has written the value before thread 0 reads it.

Distributed memory





- Each task owns part of the data, and other tasks must send a message to the owner in order to update that data.
- These may be two tasks on the same computer so that they could just share memory, but the programmer is treating them as though they were not.
- Virtual address space is not shared.

C++ libraries for parallelisation

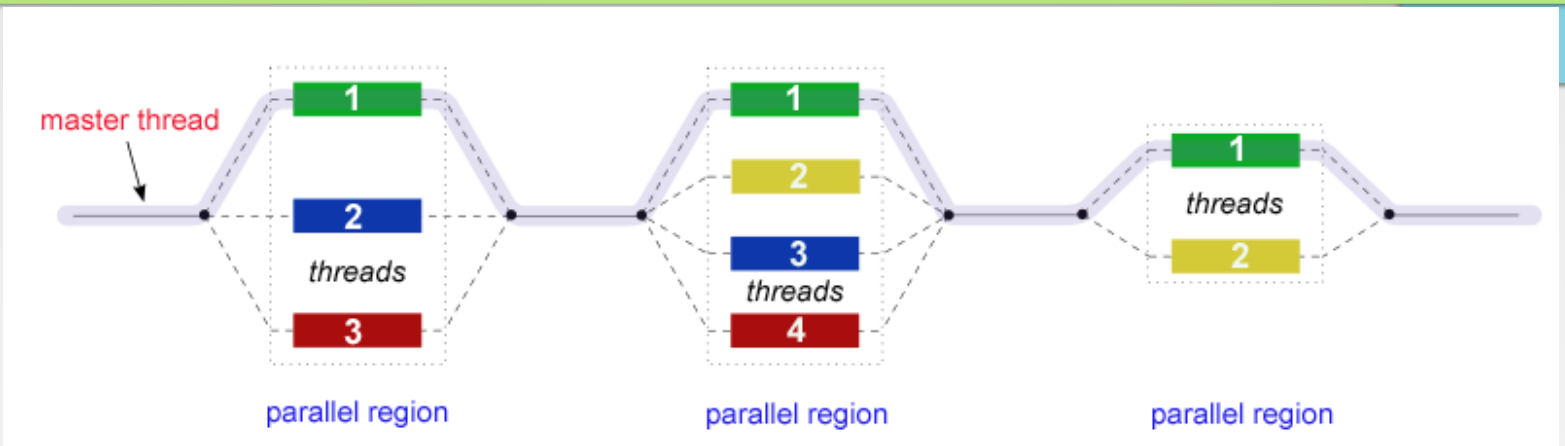
- **Shared memory**
  - OpenMP
  - C++11 Threads
  - Posix Threads
  - Intel TBB
- **Distributed Memory**
  - MPI
- **GPU**
  - CUDA
  - OpenCL
  - OpenACC
  - AMP

- 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
- OpenMP is not:
  - For distributed memory parallel systems (by itself)
  - Guaranteed to make the most efficient use of shared memory
- **The programmer is responsible for synchronizing input and output.**

Thread based parallelism

- OpenMP programs accomplish parallelism exclusively through the use of 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: 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: General code structure

```
#include <omp.h>
int main ()  {
      int var1, var2, var3;
      // Serial code . . .
      // Beginning of parallel section. Fork a team of threads.
      //Specify variable scoping
#pragma omp parallel private(var1, var2) shared(var3) {
      //  Parallel section executed by all threads
      // Run-time Library calls
      // All threads join master thread and disband
  }
// Resume serial code ..
}
```

Hello World Example

“g++ -fopenmp hello.cc -o hello”

PARALLEL region construct

#pragma omp parallel [clause ...] newline

private (list)

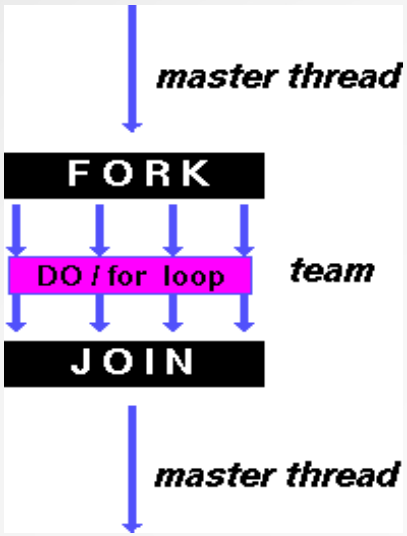
shared (list)

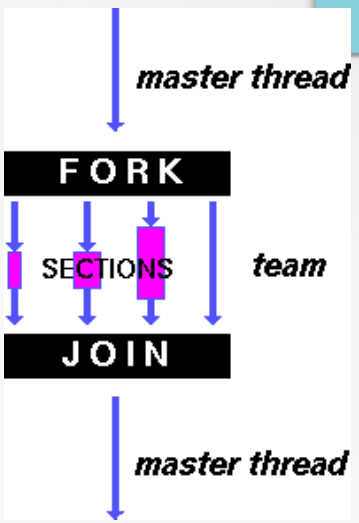
- default (private|shared)

- **Private –** Variable is private to each thread
- **Shared –** variable is shared between threads
- **omp_set_num_threads(4);**

- A parallel region is a block of code that will be executed by multiple threads
- When a thread reaches a PARALLEL directive, it creates a team of threads and becomes the master of the team. The master is a member of that team and has thread number 0 within that team.

Work sharing construct





- A work-sharing construct divides the execution of the enclosed code region among the members of the team that encounter it.
- Work-sharing constructs do not launch new threads
- There is no implied barrier upon entry to a work-sharing construct, however there is an implied barrier at the end of a work sharing construct.

#pragma omp for [clause ...]
schedule (type [,chunk])

- SCHEDULE: Describes how iterations of the loop are divided among the threads in the team.
- STATIC: Loop iterations are divided into pieces of size chunk and then statically assigned to threads.
- DYNAMIC: Loop iterations are divided into pieces of size chunk, and dynamically scheduled among the threads; when a thread finishes one chunk, it is dynamically assigned another.
- GUIDED: Iterations are dynamically assigned to threads in blocks as threads request them until no blocks remain to be assigned. The size of the initial block is proportional to:
  - number_of_iterations / number_of_threads
- Subsequent blocks are proportional to
  - number_of_iterations_remaining / number_of_threads

Adding two arrays of c[10] = a[10]+b[10]