High-**P**erformance **C**omputing 20**25**

Parallel Computing, Shared/Distributed-Memory Parallelism and **OpenMP**

Aryan Eftekhari | **U**niversità della **S**vizzera **i**taliana (USI) | **I**nstitute of **C**omputing (CI)

Review
SIMD–Single Instruction Multiple Data

An **instruction** is carried out on (multiple) **inputs** resulting in (multiple) **outputs**.

E.g., *Advanced Vector Extension* (AVX512[1]) (*NEON* for ARM[2]).

There are other levels of parallelism, such as instruction-level, thread-level, process-level, etc.

[1] https://www.intel.com/content/dam/develop/external/us/en/documents/31848-compilerautovectorizationguide.pdf
[2] https://developer.arm.com/Architectures/Neon

Review
Bandwidth–A sense of scale

The rate of read/write/transfer.

Cache

L1/L2/L3 cache has a bandwidth in the ~100s of GB/s.

Main Memory

The bandwidth of ram is on the order of ~10 GB/s.

Local Network

High-performance networks in data centers have a bandwidth of ~1GB/s.

Disk

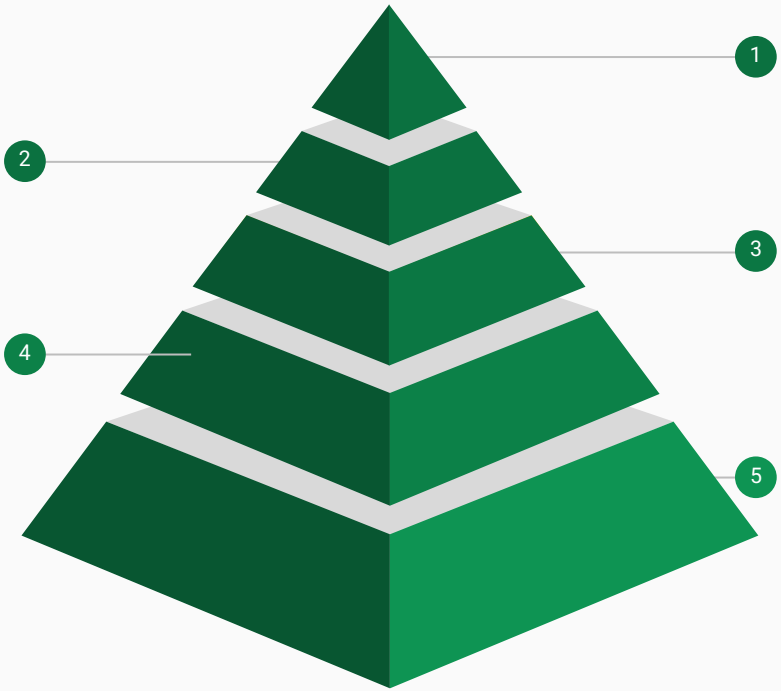
The bandwidth of disks can vary depending on type of SSD ~0.1GB/s.

Internet

4G is about ~0.01 GB/s.

Think about bottlenecks …

Review
Memory Hierarchy



Registers ~ 64–512 bit
Integral to the CPU, lowest latency. Size varies: x86-64 general-purpose registers are 64-bit, Apple's M1 SIMD registers are 128-bit, and some Intel CPUs have 512-bit SIMD registers.

L1 Cache ~16–128 KB
Extremely fast with only few cycles latency.

L2 Cache ~256KB–1MB
Very fast, slower than L1 but much faster than main memory.

L3 Cache ~1–8 MB
Fast, slower than L2 but much faster than main memory. *Shared with all cores.*

Main Memory ~32–128 GB
Compared to the L3 this is slow.

No free lunch…fast memory is expensive and limited in capacity.

The values above are for reference and can vary significantly between products and over time.

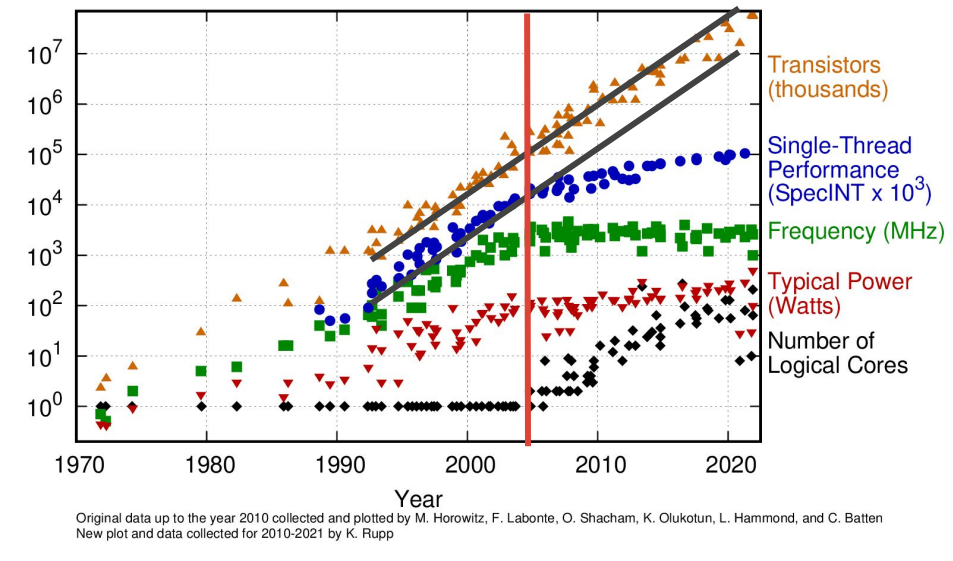
Review
Multi or Manycore Parallelism

Computers have stopped getting “faster” since the mid 2000’s !

Provocative statement to get attention.

1970 to Mind 2000’s - Increasing Clock Speed
No longer an option due to heat and power.

Mind 2000’s to Current - Increasing Cores
More than one core, i.e., multicore processors.



Overview

Parallel computing and **OpenMP**

Parallel Computing
What is parallel computing?

Concurrent Computing
Operations are interleaved/overlapping, instead of sequentially.

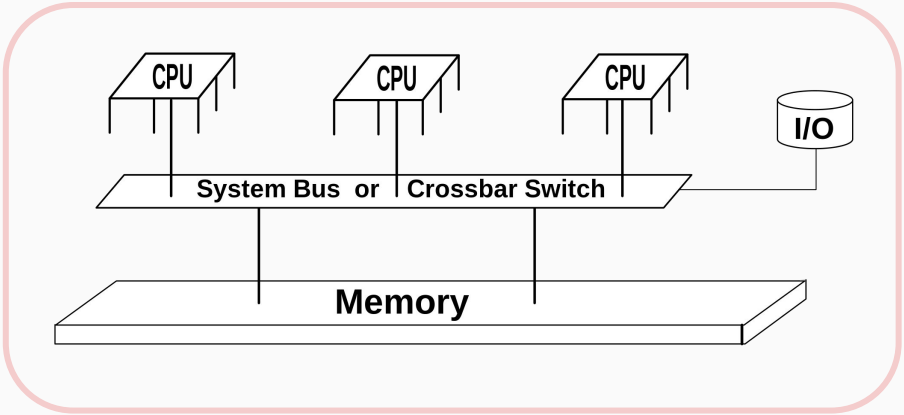
Asynchronous Computing
Operations are executed in near future, i.e., nonblocking.

Parallel Computing
Operations are executed simultaneously.

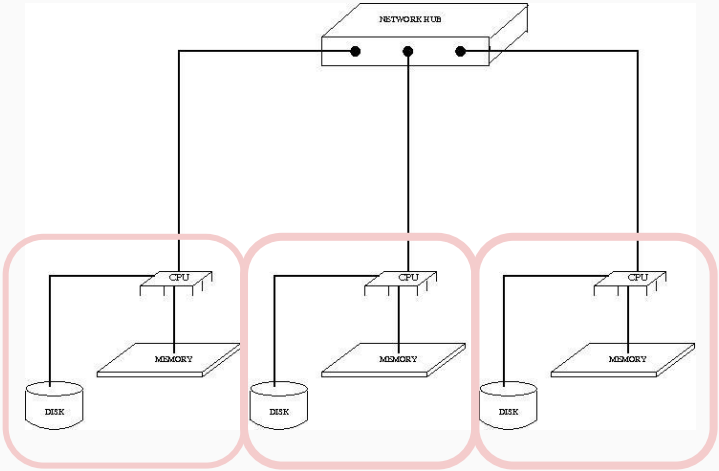
We are concerned with **Parallel Computing**.

The basic models (example)

Shared Memory
CPUs have common memory.



Distributed Memory
CPUs do NOT have common memory.



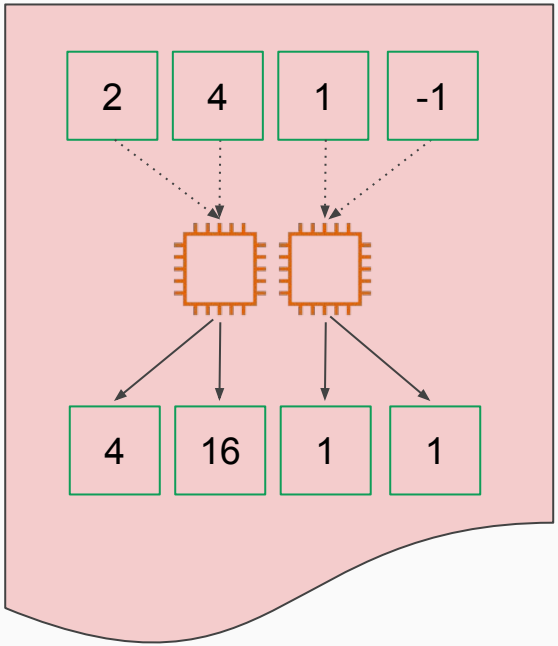
Parallel Computing
Why should I care?

- **Shared-Memory Parallelization**
  - Your computer is multi-core!
  - Efficient utilization of a computer i.e., a node.

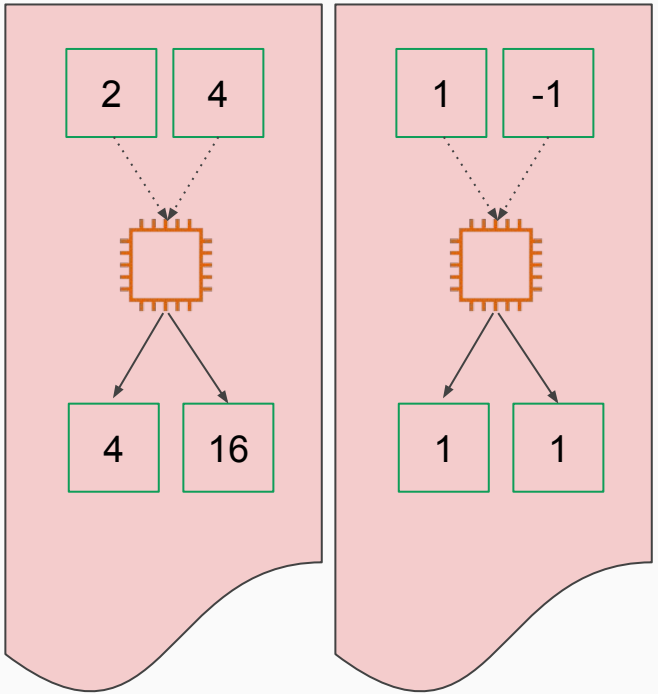
- **Distributed-Memory Parallelization**
  - Fundamental for scientific computing.
  - Base architecture for Supercomputing, i.e., multiple nodes.

Shared/Distributed-Memory Parallelism
Parallelization paradigms

Shared Memory Parallelization
*Computation is distributed along **threads**.*
***Synchronization** between threads.*



Distributed Memory Parallelization
*Computation is distributed along **processes**.*
***Communication** via message passing.*



Shared/Distributed-Memory Parallelism
Programming OpenMP/MPI

Shared-Memory Parallelization

Interface:
OpenMP: an API for C/C++/Fortran

Get it:
It's part of the GNU compiler pkg. (above gcc 4.2)

Use it:
1) Add `#include <omp.h>`
2) ***Add pragma statements to code**
3) Add `-fopenmp` flag
4) Set environment variable `OMP_NUM_THREADS`
5) Execute

**This can be tricky (race conditions).*

Distributed-Memory Parallelization

Interface:
MPI: **M**essage **P**assing **I**nterface a standard designed for message passing parallel computing architectures.

Get it:
Different implementations, OpenMPI is common.

Use it:
1) Add `#include <mpi.h>`
2) ***Add MPI functions to pass messages.**
3) Compile with `mpic++` (not `g++`)
4) Execute `mpirun -np 2` (run with 2 processes)

**This can change your program structure.*

Shared/Distributed-Memory Parallelism
Writing OpenMP vs MPI

```cpp
//OpenMP

int main(){

   int n = 100000000;
   std::vector<double> val(n,0);

   #pragma omp parallel for
   for (int i = 0; i < n; i++){
      val[i] = COSTLY_OPERATION(i);
   }

   return 0;
}
```

```cpp
// MPI

int main(){

   int n = 100000000;
   std::vector<double> val(n,0);

   int size, rank;
   MPI_Init(NULL,NULL);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   int block_size = n/size;
   int start = rank*block_size;
   int end   = (rank+1)*block_size;
   if(rank==size-1){
       end=n;
   }

   for ( int i = start; i < end; i++){
      val[i] = COSTLY_OPERATION(i);
   }

   MPI_Allreduce( MPI_IN_PLACE, &val[0] , n , MPI_DOUBLE,
MPI_SUM,MPI_COMM_WORLD);
   MPI_Finalize();

   return 0;
}
```

OpenMP can be **easy to use** (e.g., `#pragma omp…`),
a lot of the work is done for you…

```
#include <omp.h>
#include <vector>

int main(){
   std::vector<double> val(1e8,0);
   #pragma omp parallel for
   for (int i = 0; i < val.size(); i++)
      val[i] = COSTLY_OPERATION(i);
   return 0;
}
```

Parallel Region

```
// In Terminal/Command line

// Compile via command line (or makefile)
g++ -fopenmp -O3 main.cpp -o main.exe

// Run
export OMP_NUM_THREADS=2; ./main.exe
```

… but debugging can be tricky.

"OpenMP" = ***Directives*** + ***Clauses*** + ***Functions*** + ***Environment Variables***.

Provide control over **parallelization** and **memory**.

```
double res = 0.0;
#pragma omp parallel for default(none) shared(a,b,n) reduction(+:res)
for (int i = 0; i < n; i++) {
  res += a[i] * b[i];
}
```

Parallelization: What do you want to do? (*Parallel for*).

Memory: How will the memory be mapped? (*shared/private*).

- *Private*: The memory is private to the thread (private variables are not initialized).
- *Shared*: The memory is shared among the threads.
- *Reduction*: The variable is "kind of shared and kind of not" ... more on this later.

For details see https://docs.microsoft.com/en-us/cpp/parallel/openmp/reference/openmp-library-reference

OpenMP
Basics–Number of Threads

There are **3** ways

```
// via terminal/command line
export OMP_NUM_THREADS=2; ./main.exe
```

1) **Runtime:** Set the environment variable `OMP_NUM_THREADS`.

```
// global
 omp_set_num_threads(4);
 #pragma omp parallel
 {
   …
 }
```

2) **Compile-Time Globally:** Via function `omp_set_num_threads`.

```
// local
 omp_set_num_threads(4);
 #pragma omp parallel num_threads(3)
 {
   …
 }
```

3) **Compile-Time Locally:** Via directive clause `num_threads`.

***Note**: (2) overrides (1) and (3) overrides (1) and (2). Option (1) is common.*

OpenMP
Directives–Parallel, Sections, For …

```
// all threads enter this parallel region
#pragma omp parallel
{
  std::cout<<omp_get_thread_num()<< "/" << omp_get_num_threads()<<std::endl;
}
```

```
// give each section to any available thread (one thread per section)
#pragma omp parallel sections
{
   #pragma omp section
   {
       std::cout << omp_get_thread_num() << "/" <<omp_get_num_threads()<<std::endl;
   }
   #pragma omp section
   {
    std::cout << << omp_get_thread_num() <<"/"<< omp_get_num_threads()<<std::endl;
   }
}
```

```
// divide work evenly among each thread
#pragma omp parallel for
for (int i = 0; i < a.size(); i++){
  a[i]= my_funct(i);
}
```

There are many more! We will look at other directives in examples `barrier`, `critical`, and `atomic`.

Behavior is dependent on the memory access sequence.

```
//This program is not deterministic
int sum=0;
#pragma omp parallel for default(none) shared(sum)
for (int i = 0; i < 100; i++){
        sum+= 1;
}
```

What is `sum` **before** you update it?

Use reduction clause → **`reduction(+:sum)`**

Challenges of shared-memory …

```
// This code has undefined behavior
   sum=0.0;
   b=5.0;
   #pragma omp parallel for default(none) reduction(+:sum) private(b)
   for (int i = 0; i < 100; i++){
       sum+= 5.0/b;
   }
```

What is `b`, *before*, *in* and *after* the for-loop ?

- **shared/private** – All threads have a shared/private copy of the variable.
- **default** – Behavior of unscoped variables in a parallel region.
- **firstprivate** – Private variable initialized with the value of the variable outside `#pragma`.
- **lastprivate** – local copy of variable set with the value of the last iteration (or section) of parallel loop.

Common synchronization mechanism

```
// This program is deterministic
   sum=0.0;
   b=5.0;
   #pragma omp parallel for default(none) reduction(+:sum) firstprivate(b)
   for (int i = 0; i < 100; i++){
       sum+= 1.0/b;
   }
```

- **atomic** – No simultaneous reading and writing threads.
- **critical** – Execution restricted to a single thread at a time (more general than atomic).
- **barrier** – Wait for all threads in parallel region to reach the same point.

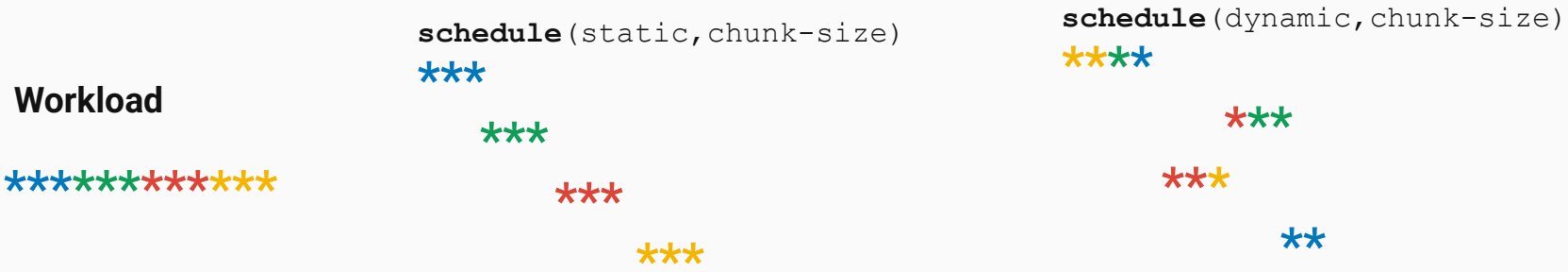
Consider these operations:

```
#pragma omp atomic
sum+= 5.0/b;
```

```
#pragma omp critical
sum+= 5.0/b;
```

```
//Not allowed (will not compile)…
even if it compiled it won't work!
     #pragma omp barrier
     sum+= 5.0/b;
```

Which thread does what ?



Static: Allocated workload at “compile time” (default chunk-size=work_load/num_threads)
Divide the iterations by chunk-size and distributes chunks to threads in a circular order (round-robin).

- *Thread is fixed to specific iteration.*
- *No performance overhead.*

Dynamic: Allocated workload at “run time” (default chunk-size=1)
Divide the iterations by chunk-size and request chunks until there are no more (first come, first serve).

- *Thread is NOT fixed to specific iteration.*
- *Performance overhead.*

There are other scheduling methods, see e.g, OpenMP 5.0 Technical Report 6

OpenMP **simplifies** shared-memory parallelism by abstracting the low-level details.