Introduction to OpenMP Programming

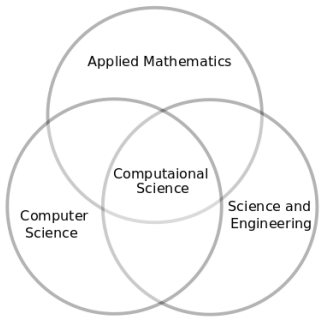
Dr. Ezhilmathi Krishnasamy

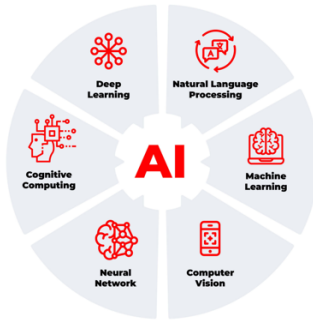
May 31, 2023

Outline

Motivation: why we need parallel programming

- In science & engineering and artificial intelligence, we need to do lots of arithmetic computation with numbers.
- Problems are defined by differential equations, particularly partial differential equations (PDEs).
- In order to solve these PDEs, they will be converted into a system of equations by using numerical methods (e.g., *finite difference, finite element method and finite volume method*).
- Finally, we need to solve the systems of equations using either direct solvers or indirect solvers to find a value for unknown variables.
- Similarly, in Artificial Intelligence (AI), we end up solving matrices and vectors.





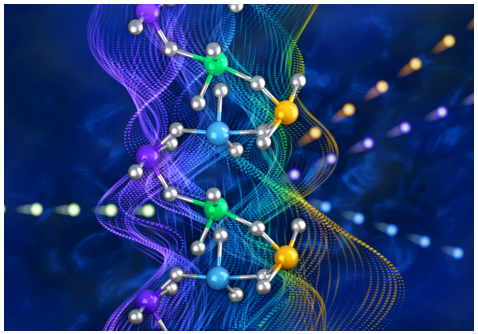
$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \quad \boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \quad \text{and} \quad \boldsymbol{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

Applications and Examples

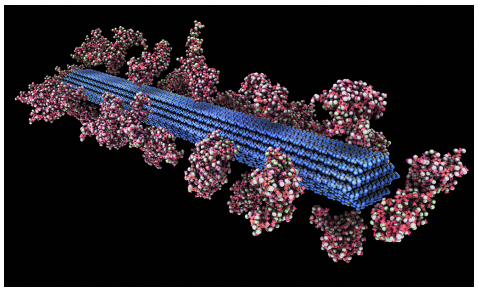
- Materials science: Materials science is an interdisciplinary discipline that integrates chemistry, physics, and engineering to provide a nano-level understanding of current materials and allows for the design of new materials with predetermined properties.

- Proteins are the basic blocks in the human body. The proteins in the cell membrane regulate the flow of material and how they flow into and out of the body. Understanding these further will enable us to find new treatments for diseases and increase human life expenditure.

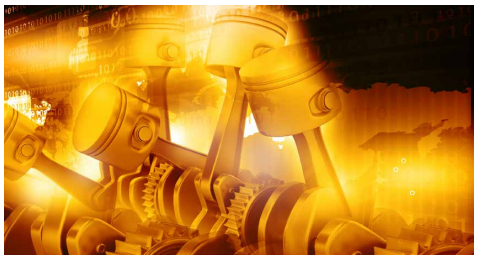
- In engineering, for example, computational fluid dynamics, computational heat transfer and computational solid mechanics need lots of computer power to simulate and design a better and optimised system. Traditionally, these were done using experiments and wind tunnel tests. But these were expensive in terms of cost and human resources.



Materials science: Credit:www.olcf.ornl.gov



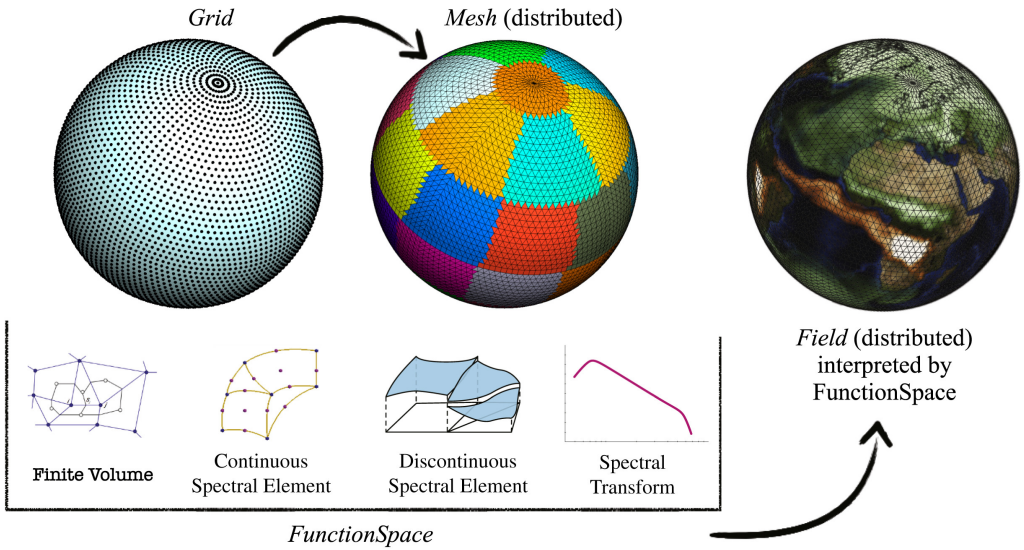
Biology: Credit:www.olcf.ornl.gov



Engineering: Credit:www.olcf.ornl.gov

Applications and Examples

- In weather forecasting, the domain size is discretised by numerical methods. For example, 786 thousand cores from the IBM Blue Gene/Q Mira system at Argonne National Laboratory are needed to simulate a resolution of 3 km (1.8 billion cells) using spectral element methods (one kind of numerical method)[1].



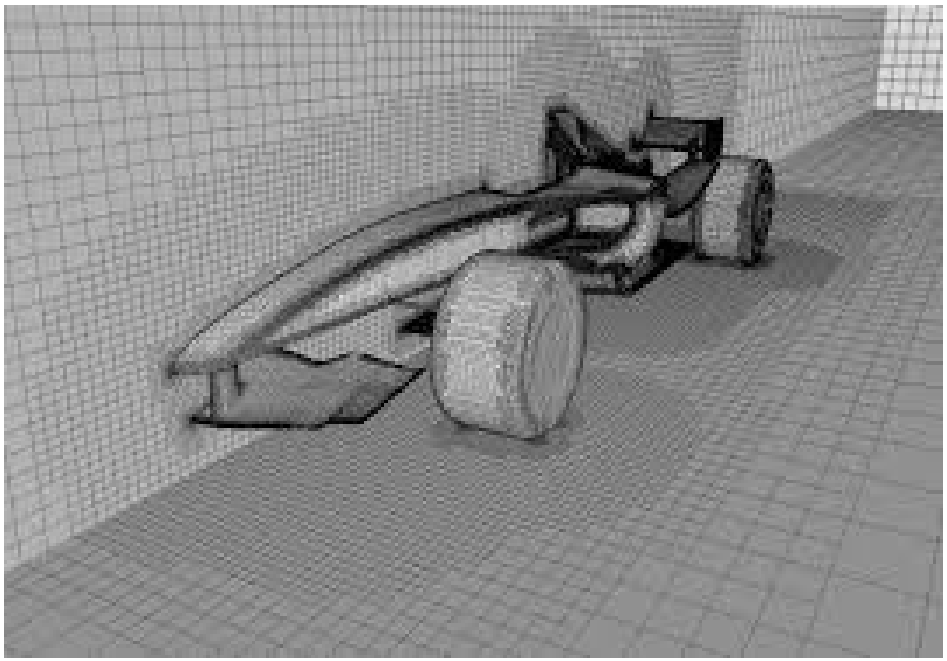
Mesh and simulation settings [2]

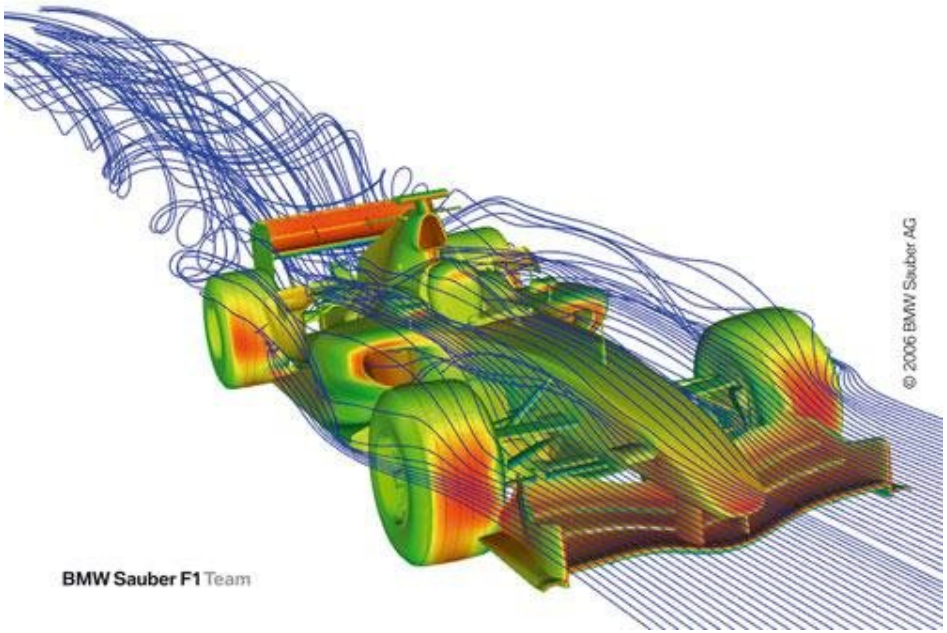
[1] HPC for Weather Forecasting, John Michalakes

[2] A library for numerical weather prediction and climate modelling, Willem Deconinck, et al.

Applications and Examples

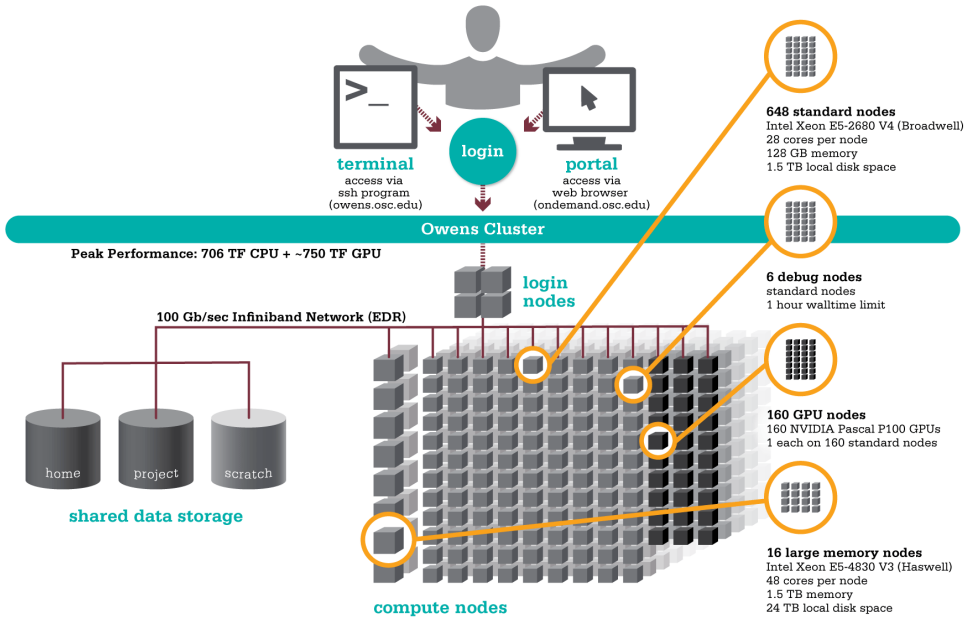
- CERFACS in France used up to 250 000 cores with grids of 2 to 4 billion cells (using Large Eddy Simulation - one kind of numerical methods strategy) for computational fluid dynamics application.





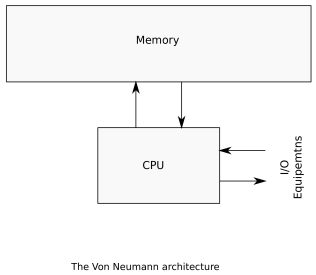
What is supercomputer

- A large machine consists of many processors and large memory.
- Many processors (CPUs) and their memory are connected via network cable.
- Multicore processors are part of the processor (CPU).
- Using parallel computing techniques, one can use these machines efficiently.

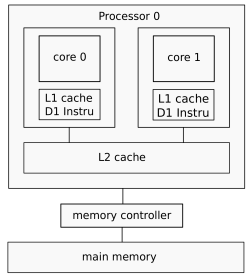


Difference between laptop and supercomputer

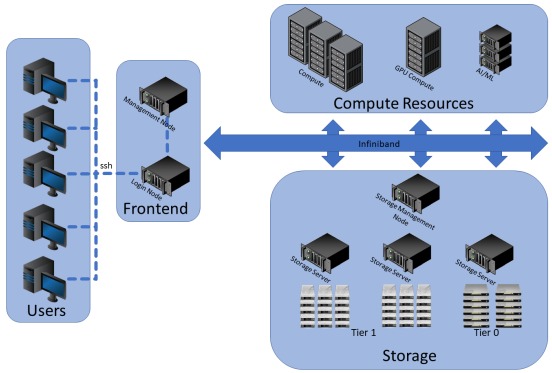
Laptop	Supercomputer
few cores	lots of cores
limited memory	lots of memory
slower than supercomputer	faster than laptop
personalized Operating System (OS)	mostly Linux based
owned by you	owed by public sector and located somewhere
used by owner	many users



The Von Neumann architecture



Early dual core Intel Xeon



Computer speed

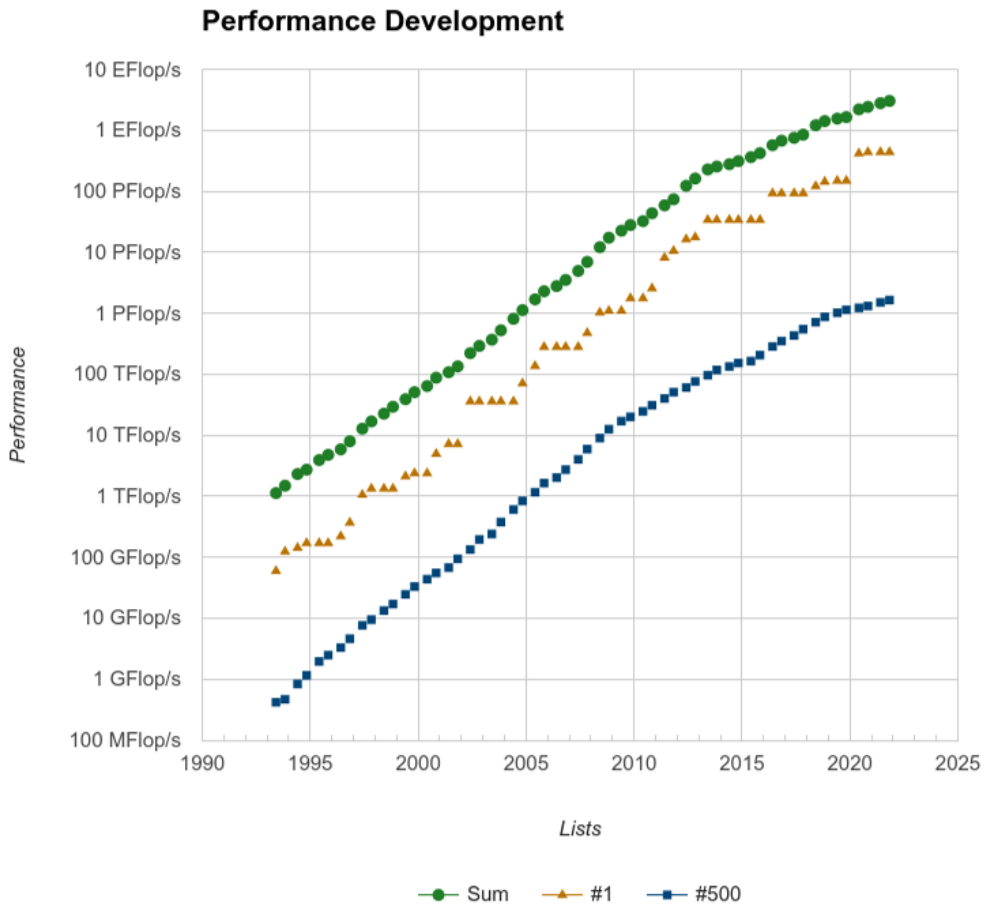
- FLOPS: Floating point Operations Per Second - the speed of the computer is measured by the FLOPS.
- A supercomputer's speed is determined by running LINPACK benchmark applications.
- Typically, solving N-dimensional linear system using Gaussian elimination with partial pivoting
- TOP500 lists the fastest supercomputers on a regular basis.
- Top 3 supercomputers (as of June 2022):
  - Frontier - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States.
  - Supercomputer Fugaku - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu, RIKEN Center for Computational Science, Japan.
  - LUMI - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE, EuroHPC/CSC, Finland.

Computer speed

Speed	Flops
1 Flop	$10^0 = 1$
1 Kflops	$10^3 = 1\,Thousand$
1 Mflops	$10^6 = 1$ Million
1 Gflops	$10^9 = 1$ Billion
1 Tflops	$10^{12} = 1$ Trillion
1 **Pflops**	$10^{15} = 1$ Quadrillion
1 **Eflops**	$10^{18} = 1$ Quintillion
1 Zflops	$10^{21} = 1$ Sextillion
1 Yflops	$10^{24} =$ Septillion

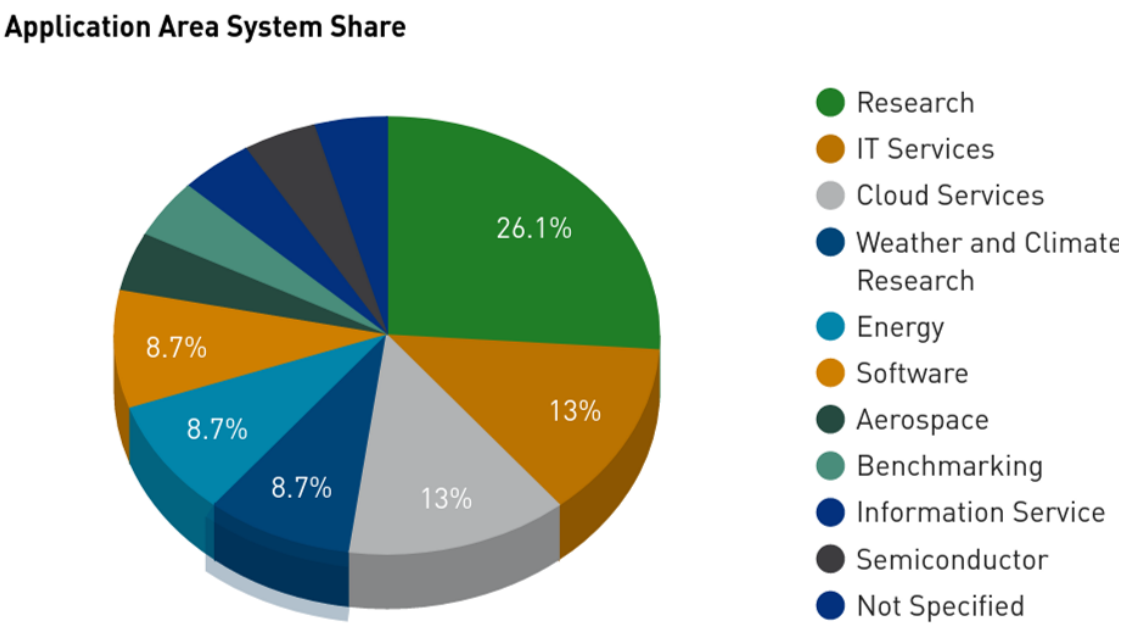
Present-day supercomputers are mainly based on Pflops and Exaflops.

Computing power trend

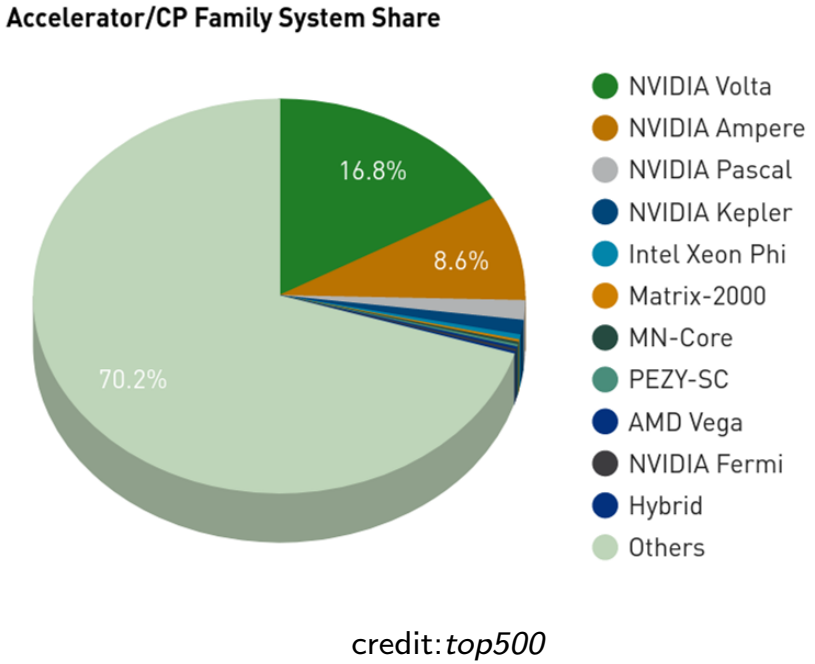


credit:top500;*FLOPS(Floating Point Operations per Second)*

How it is used in different research sectors



credit: *top500*



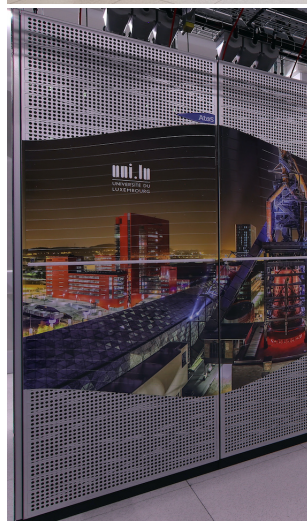
credit: *top500*

Supercomputers in Luxembourg

- MeluXina - Accelerator Module - BullSequana XH2000, AMD EPYC 7452 32C 2.35GHz, NVIDIA A100 40GB, Mellanox HDR InfiniBand/ParTec ParaStation ClusterSuite, Atos, LuxProvide, Luxembourg; Ranked 36 in 2021 (according to TOP500).
- Aion is an Atos/Bull/AMD supercomputer which consists of 318 compute nodes, totalling 40704 compute cores and 81408 GB RAM, with a peak performance of about 1,70 PetaFLOP/s.
- Iris is a Dell/Intel supercomputer which consists of 196 compute nodes, totalling 5824 compute cores and 52224 GB RAM, with a peak performance of about 1,072 PetaFLOP/s.

Supercomputers in Luxembourg

	Meluxina	**Aion**	**Iris**
CPU Architecture	AMD EPYC	AMD EPYC	Intel Xeon
GPU Architecture	Nvidia A100	-	Nvidia V100
Performance *RPEAK*	18 PFLOPS	1,70 PFLOPS	1,072 PFLOPS
Host	EuroHPC & LuxProvide	Luxembourg University	Luxembourg University
Vendor	AMD	AMD	Intel

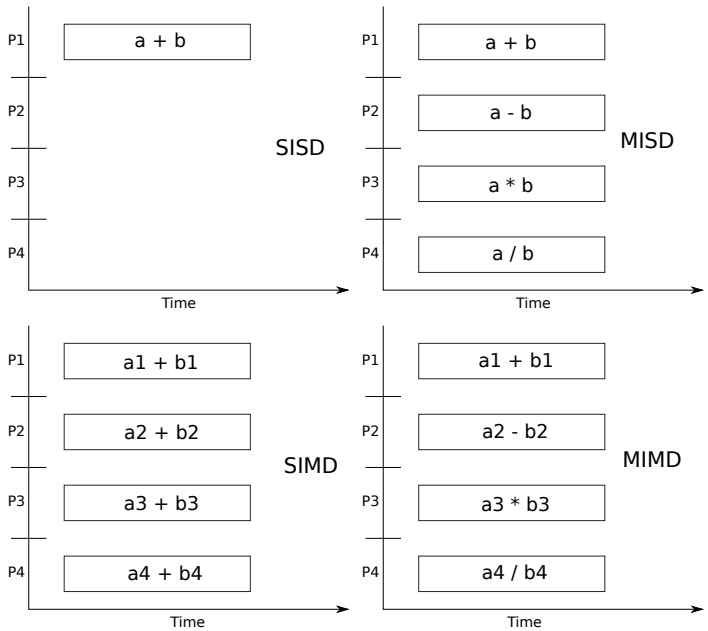


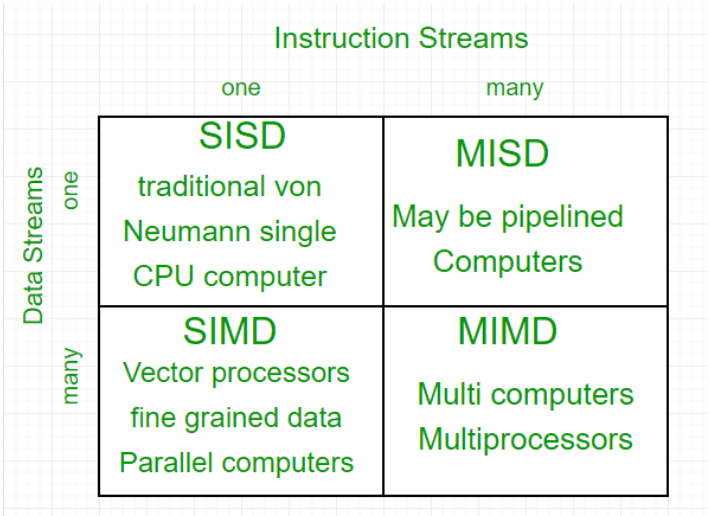
Classification of parallel computer architecture

- Control structure: based on the instructions and data streams.
  - SISD - single instruction stream, single data stream
  - SIMD - single instruction streams, multiple data streams
  - MISD - multiple instruction streams, single data stream
  - MIMD - multiple instruction streams, multiple data streams
- Memory organisation: shared memory and distributed memory
- Network topology (connectivity): for example, 3D grid and tree

Control structure - Flynn's taxonomy

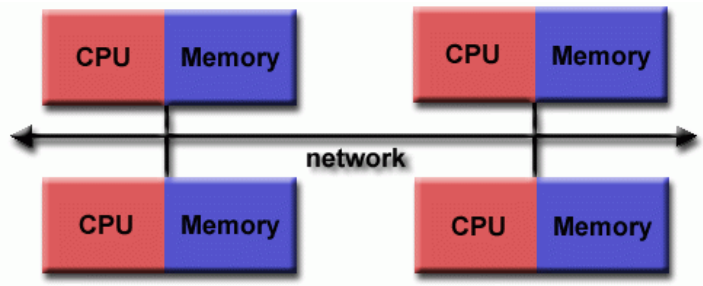
- SISD - an example of sequential execution (conventional single processor - von Neumann model).
- SIMD - an example of array computers and traditional vector processors; instructions can be executed in a pipeline or in parallel.
- MISD - it is not used commonly.
- MIMD - it works with both shared and distributed memory models.





Distributed memory architecture

1. Distributing memory architecture is designed by using a network connection.
2. There is no global address space since each compute node has its own local memory.
3. There is no Cache Coherent (memory update by one processor is visible to another processor - done at the hardware level) existing.
4. Only the programmers decide how and from where the data needs to be accessed/transferred.



Distributed memory architecture

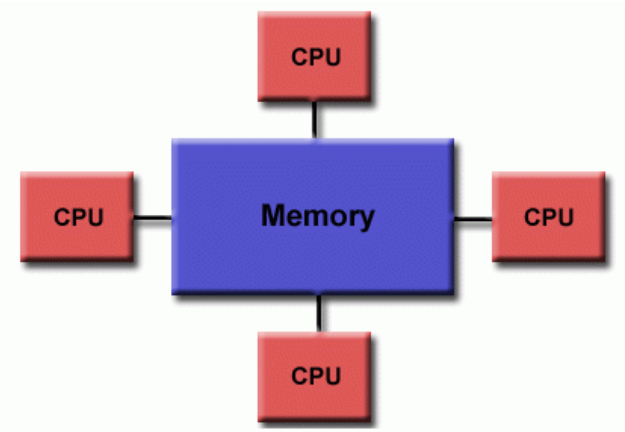
1. Advantages:
   - scalable with memory and processor (large-scale computation is possible)
   - processor within the node can have quick access to its own memory without implementing global cache coherency
   - networking makes it possible to access many processors and memory
2. Disadvantages:
   - accessing the local data is faster than across the compute nodes (via interconnect networks)
   - it will be the programmer's responsibility to access (read and write) across the multiple compute nodes (among the remote memory)

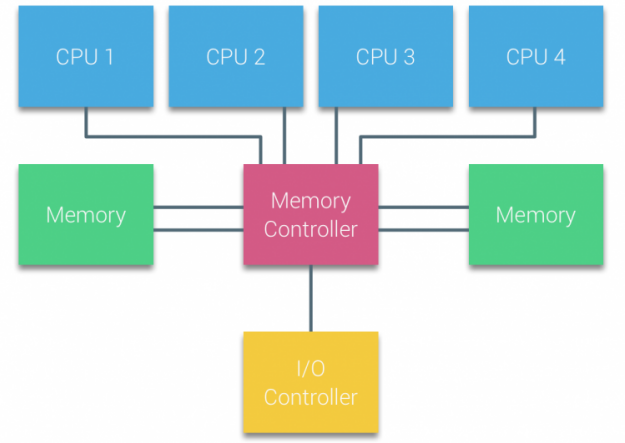
Shared memory architecture

1. Shared memory computer's processors have access to all memory as global address space
2. Multiple processors in the shared memory architecture can act independently but share the same memory resources
3. Changes in a memory location affected by one processor are visible to all other processors
4. Shared memory architecture is categorised into Unified Memory Access (UMA) and Non-Unified Memory Access (NUMA) based on memory access types

Unified Memory Access (UMA)

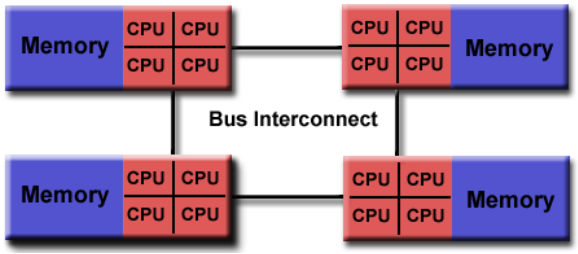
1. All the processors have memory access through an interconnection network (similar to single processor access to its memory).
2. A memory access by all the processors has an equal time (same time) through an interconnection network (e.g. bus).
3. Often, it is represented by Symmetric Multiprocessor (SMP) and has identical processors.
4. It can also be called Cache Coherent UMA (CC-UMA) - where memory update by one processor is visible to another processor, achieved by hardware configuration level.

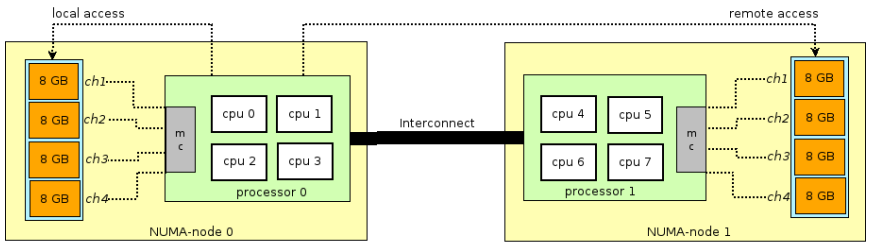




Non-Unified Memory Access (NUMA)

1. Each set of processors (SMP) has its own memory.
2. Processors from one SMP can access another SMP memory.
3. Memory accessing time varies between different processors (across the SMPs), thus slowing memory access across SMPs.
4. NUMA can also have Cache Coherent NUMA (CC-NUMA) with the help of hardware-level configuration.



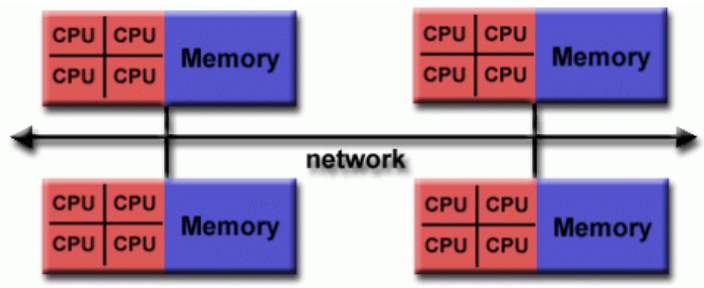


Shared memory architecture

1. Advantages
   - data sharing between processes are both rapid and uniform because of the proximity of memory to CPUs
   - nature of the global address space offers a user-friendly programming perspective to memory access
   - no need to specify the communication of data between processes explicitly
   - easy to learn/program (e.g. OpenMP)
2. Disadvantages
   - scalability is affected by memory and processors
   - programmers are responsible for synchronisation constructs (e.g. locks) for accessing the "correct" data in the global memory

Hybrid distributed-shared memory

1. Present-day supercomputers have shared and distributed memory architecture.
2. Shared memory components can have GPUs and other accelerators that can access shared memory.
3. Distributed memory will connect to shared memory via network connections. So, that memory access from one shared memory node to another will be accessible via interconnect network.
4. Scalability can be increased with increasing compute nodes and memory (advantage).
5. There is programmer's complexity involved - task, data and communication (disadvantage).



What is OpenMP

1. **OpenMP** is an application programming interface (API) to facilitate the parallel programming on shared memory architecture (both UMA and NUMA)
2. It is a directive-based (notation) programming model that is used to parallelize the serial code written in **C/C++** and **Fortran**
3. Its **directive** will instruct the compiler how to execute a code in parallel on the shared memory architecture
4. A single application can have both serial and parallel version
5. Latest **OpenMP API 5.2** specification released on Nov-2021

OpenMP compilers

Vendor/Source	Compiler/Language
AMD	C/C++
ARM	C/C++/Fortran
GNU	GCC - C/C++/Fortran
HPE	CCE - C/C++/Fortran
IBM	XL - C/C++/Fortran
Intel	C/C++/Fortran
NVIDIA HPC compiler	C/C++/Fortran

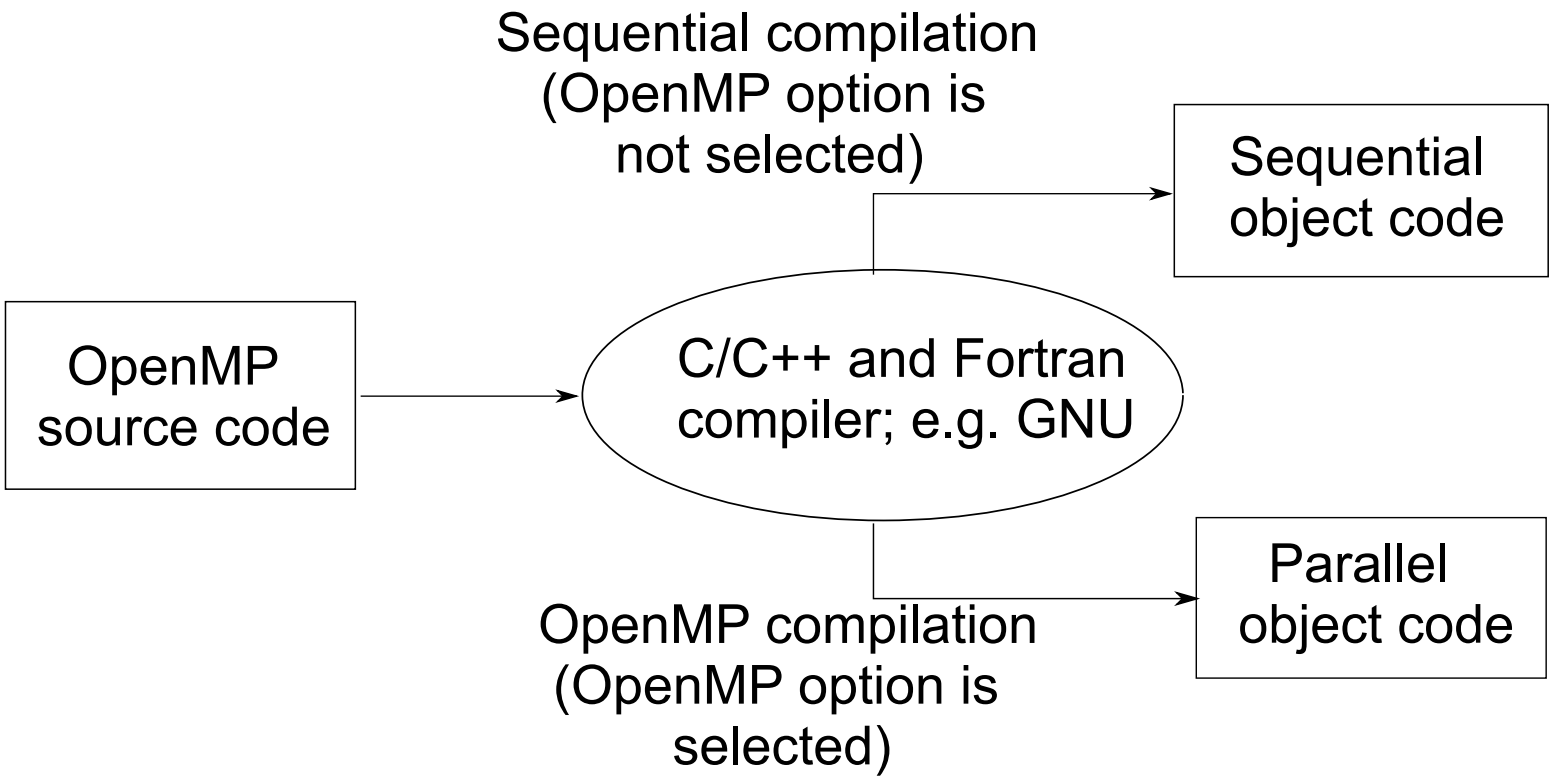
GNU

```
g++ -fopenmp hello-world.cc -o hello-world
gcc -fopenmp hello-world.c -o hello-world
```

Intel

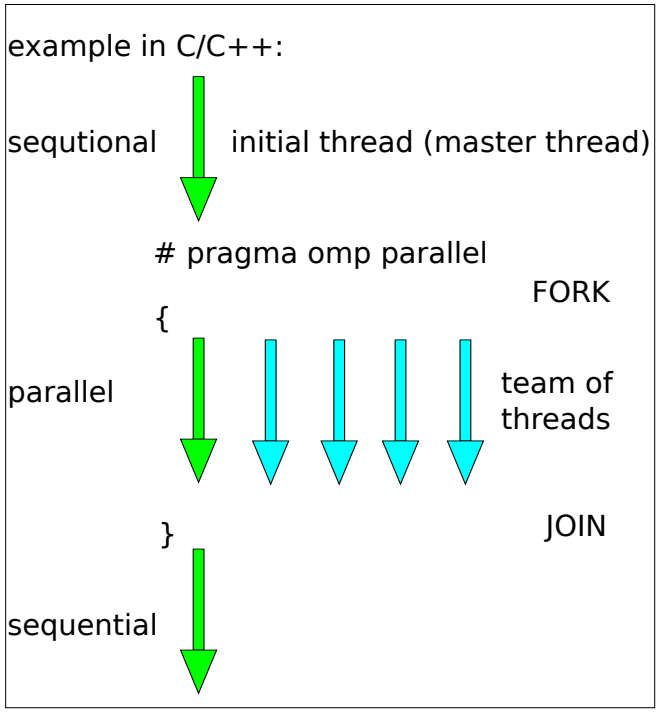
```
icpc -qopenmp hello-world.cc -o hello-world
icc -qopenmp hello-world.c -o hello-world
```

Compilation



Parallel region

1. **Fork-Join:** model of parallel execution.
2. **FORK:** **master** thread creates a **team** of parallel threads (master thread is also part of a team of threads).
   - → in C: **#pragma omp parallel** will create a treads.
   - → by default, thread number is defined by the given architecture.
3. **JOIN:** team threads complete statements in parallel regions.
   - → then they synchronize & terminate, leaving only the master thread to continue.



Parallel construct

Example of parallel region

```
#include<iostream>
#include<omp.h>
using namespace std;
int main()
{
#pragma omp parallel //*-- create a parallel region "Fork" --*//
  {
    cout << " Hello world from thread id "
         << omp_get_thread_num()
         << " from the team size of "
         << omp_get_num_threads()
         << endl;
  } //*-- parallel region is closed "Join" --*//
  return 0;
  cout << "end of the program from the master thread" << endl;
}
```

g++ -fopenmp hello-world.cc -o hello-world

OpenMP - environment routines

1. `void omp_set_num_threads(int num_threads);`
   → number of threads to be used in the parallel region

2. `int omp_get_num_threads(void);`
   → gets the number of threads in the current parallel region

3. `int omp_get_max_threads(void);`
   → get available maximum threads (system default)

OpenMP - environment routines

1. `int omp_get_thread_num(void);`
   → gets the thread numbers (e.g., 1, 4, etc.)

2. `int omp_get_num_procs(void);`
   → gets the number of the available processor in the system

3. `int omp_in_parallel(void);`
   → boolean variable to check if the parallel region is active (returns 1) or not (returns 0)

OpenMP - environment routines

OMP_NUM_THREADS

```
$ setenv OMP_NUM_THREADS 4
```

```
$ export OMP_NUM_THREADS=4
```

```
$ OMP_NUM_THREADS=4 ./omp_code.exe
```

Shared variable

1. All the threads have access to the shared variable.
2. By default, in the parallel region, all the variable is considered as shared variable.
3. Shared variables should be handled carefully; otherwise, it causes race conditions in the program.
4. Here, all the threads within the parallel region have access to elements in vector a.

```
#pragma omp parallel for shared(a)
  for (int i = 0; i < N; i++)
    a[i] += i;
/*-- End of parallel region --*/
```

Private variable

1. Each thread will have its own copy of the private variable.
2. The private variable is only accessible within the parallel region, not outside of the parallel region.
3. By default, the loop iteration counters are considered private.
4. A change made by one thread is not visible to other threads.

```
#pragma omp parallel for private(a)
  for (int i = 0; i < N; i++)
    a = 1 + i;
/*-- End of parallel region --*/
```

```
default(shared | none)
#pragma omp parallel default(shared) private(var_list)
```

Lastprivate

1. `lastprivate:` is also similar to a private clause
2. But each thread will have an uninitialized copy of the variables passed as lastprivate
3. At the end of the parallel loop or sections, the final variable value will be the last thread accessed value in the section or in a parallel loop.

```
int var = 5;
#pragma omp parallel for lastprivate(var)
   for(int i = 0; i < n; i++)
     {
       var += omp_get_thread_num();
       cout << " lastprivate in parallel region " << var << endl;
     } /*-- End of parallel region --*/
 cout << "lastprivate after parallel region " << var <<endl;
```

Firstprivate

1. `firstprivate:` is similar to a private clause
2. But each thread will have an initialized copy of the variables passed as firstprivate
3. Available for `parallel` constructs, `loop`, `sections` and `single` constructs

```
int var = 5;
#pragma omp parallel firstprivate(var)
  {
    var += omp_get_thread_num();
    cout << " firstprivate variable within the parallel region " << var << endl;
  } /*-- End of parallel region --*/
  cout << "firstprivate varibale after the parallel region " << var <<endl;
```

Work-sharing constructs

1. Loop construct
2. Sections construct
3. Single construct
4. Master construct
5. Workshare construct (only available in Fortran)

Loop construct

1. **For:** for directive execute for loop as a parallel loop
2. It should be placed within the parallel region #pragma omp parallel
3. By default loop counter index becomes private
4. `Loop-scheduling:` has the following clauses:
   - → static
   - → dynamic
   - → guided
   - → auto
   - → runtime

Functionality	Syntax in C/C++	Syntax in Fortran
Distribute iterations over the threads	`#pragma omp for`	`!$omp do`

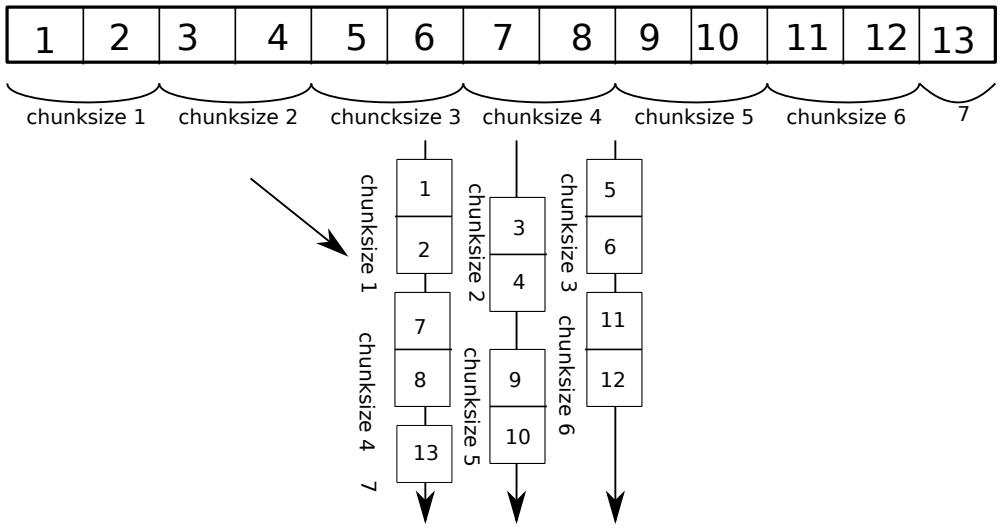
Loop construct

Example of work-sharing loop

```
#include<iostream>
#include<omp.h>
using namespace std;
int main()
{
  int N = 10;
#pragma omp parallel num_threads(2)
  {
#pragma omp for
    for (int i = 0; i < N; i++)
      {
        cout << " Thread " << omp_get_thread_num()
             << " executes loop iteration " << i << endl;
      }
  } /*-- End of parallel region --*/
  return 0;
}
```

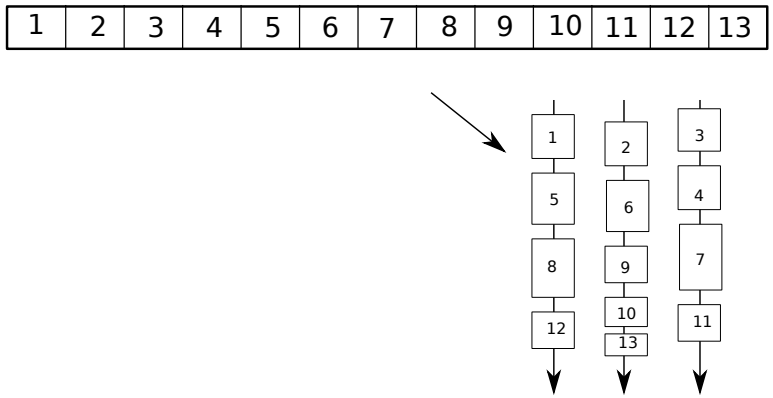
Static clause

1. Number of iterations is divided by chunksize (granularity of the workload)
2. If the chunksize is not provided, a number of iterations will be divided by the size of the team of threads
   - e.g., n=64, num_threads=4; each thread will execute the 16 iterations in parallel
3. This is useful when the computational cost is similar to each iteration



Dynamic clause

1. Number of iterations is divided by chunksize
2. If the chunksize is not provided, it will consider the default value as 1
3. It will request a chunk of data until there are no more chunks of data available
4. There is no pattern for how different threads access the different chunks of data
5. This is useful when the computational cost is different in the iteration
6. This will quickly place the chunk of data in the queue



Guided, Auto and Runtime schedule clause

1. **Guided:**
   - Similar to dynamic scheduling that is the number of iterations is divided into chunksize
   - But the chunk of the data size is decreasing, which is proportional to the number of unsigned iterations divided by the number of threads
   - If the chunksize is not provided, it will consider the default value as 1
   - This is useful when there is poor load balancing at the end of the iteration
2. **Auto:**
   - Compiler chooses the optimized chunksize for a number of iterations.
3. **Runtime:**
   - Environmental variable `OMP_SCHEDULE` is used to define parallel for loop scheduling.

Schedule clause

Example of schedule clause

```
#include<iostream>
#include<omp.h>
using namespace std;
int main()
{
  int i, N = 1000;
  int *A, *B, *C;
  A = new int[N]; B = new int[N]; C = new int[N];
#pragma omp parallel for schedule(runtime)
    for (i= 0; i < N; i++)
      {
        C[i] = A[i] + B[i];
      }
/*-- End of parallel region --*/
  return 0;
}
```

```
$ export OMP_SCHEDULE="DYNAMIC"
$ setenv OMP_SCHEDULE "GUIDED,4"
```

Collapse clause

1. Useful for nested loops
2. Total number of iterations will be partitioned into an available number of threads
3. Example: if the outer loop index is equal to the threads, then innermost loop (size - more than a number of threads work will be divided by a number of threads)
4. Only one collapse clause is allowed within the nested loop

```
  int i, N = 1000;
  int *A, *B, *C;
  A = new int[N]; B = new int[N]; C = new int[N];
#pragma omp parallel for schedule(runtime) collapse(2)
  for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
      {
        C[i]=A[i]*B[j];
      }
  /*-- End of parallel and collapse region --*/
```

Section(s) construct

1. Different tasks can be executed in parallel
2. Sections can be used when more tasks (functions or subroutines) need to be computed in parallel
3. Each thread will execute one task (only once)
4. Each task should be independent of the other within the sections

```
#pragma omp parallel sections num_threads(9)
  {
#pragma omp section
      cout <<"section-1 from thread id " << omp_get_thread_num () <<endl;
#pragma omp section
      cout <<"section-2 from thread id " << omp_get_thread_num () <<endl;
#pragma omp section
      cout <<"section-3 from thread id " << omp_get_thread_num () << endl;
#pragma omp section
      cout <<"section-4 from thread id " << omp_get_thread_num () <<endl;
  } /*-- End of parallel and sections region --*/
```

Single construct

1. Order of the threads accessing the single construct is random
2. Only one thread executing the single construct only once
3. Single construct waits for all the single constructs to finish their job

---

```
#pragma omp parallel num_threads(6)
{
#pragma omp single
  {
  cout <<"Thread id in single " << omp_get_thread_num()
       << " from team of threads "<< omp_get_num_threads() <<endl;
  }
  cout <<"Parallel region thread id - "
       << omp_get_thread_num() << " from team of threads "
       << omp_get_num_threads() <<endl;
  } /*-- End of parallel and single region --*/
```

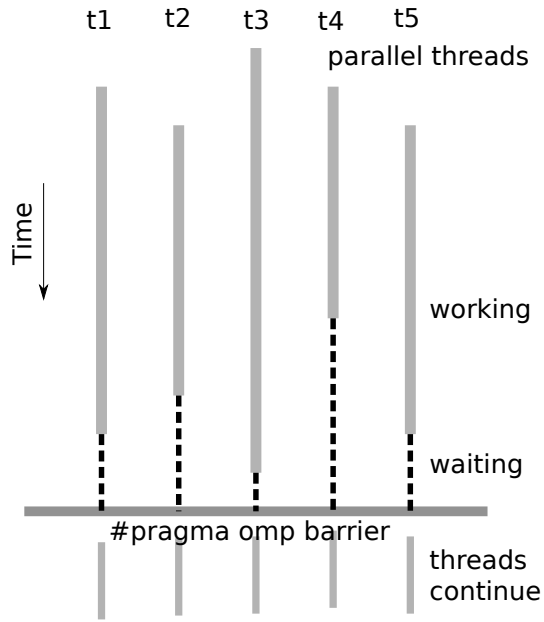
Master construct

1. Only the master thread executed the master construct
2. It is similar to a single construct; however, it does not have an implied barrier
3. It has to be used along with `barrier` when there is a data dependency

```
#pragma omp parallel shared(a,b)
  {
    #pragma omp master
    {
      a = 1;
      cout << "master thread " << omp_get_thread_num() << endl;
    }
#pragma omp barrier
#pragma omp for
    for (int i = 0; i < n; i++)
      b[i] = a;
  } /*-- End of the parallel region --*/
```

Synchronization constructs

1. #pragma omp barrier will synchronize all the threads (waiting for other threads to finish their work) before it continues.
2. Synchronization is useful to avoid data race in the application




```
int *x = new int[10];
#pragma omp parallel num_threads(10)
  {
    int tid = omp_get_thread_num();
    x[tid] = my_task(tid);
    cout <<" before barrier" << x[tid] << endl;
#pragma omp barrier
    cout <<" after barrier" << x[tid] << endl;
  } /*-- End of the parallel region --*/
```

Critical construct

1. It will allow only one thread at once (one-by-one) within the critical section
2. A multiple threads can not access the critical section in parallel
3. This will avoid the race condition

```
int local_sum, sum = 0;
#pragma omp parallel num_threads(5) private(local_sum)
  {
    for (int i = 0; i < n; i++)
      local_sum += i;
#pragma omp critical
    {
      sum += local_sum;
    }
  } /*-- End of the parallel region --*/
  cout << "value of local sum is " << sum << endl;
```

Atomic constructs

1. It is similar to critical but has a performance advantage
2. Just a single statement followed by the atomic declaration
3. Safely updates the shared numeric variable and supports the memory location update

```
int total = 0;
#pragma omp parallel num_threads(5)
  {
    for(int i = 0; i < 10; i++)
      {
#pragma omp atomic
        total++;
      }
  } /*-- End of the parallel region --*/
  cout <<"Total = " << total <<endl;
```

Reduction clause

1. Useful for incrementing or summation of an array into a shared numerical variable.
2. `reduction(operators: variable)`
   - arithmetic reductions: `'+,*,-,max,min'`
   - logical operator reductions : `'& && | ∧ ||'`
3. Shall be used instead of critical (only one thread allows to update the variable)

```
omp_set_num_threads(5);
int sum = 0;
#pragma omp parallel for schedule(runtime) reduction(+:sum)
  for (int i = 0; i < N; i++)
    sum += a[i];
/*-- End of parallel region --*/
```

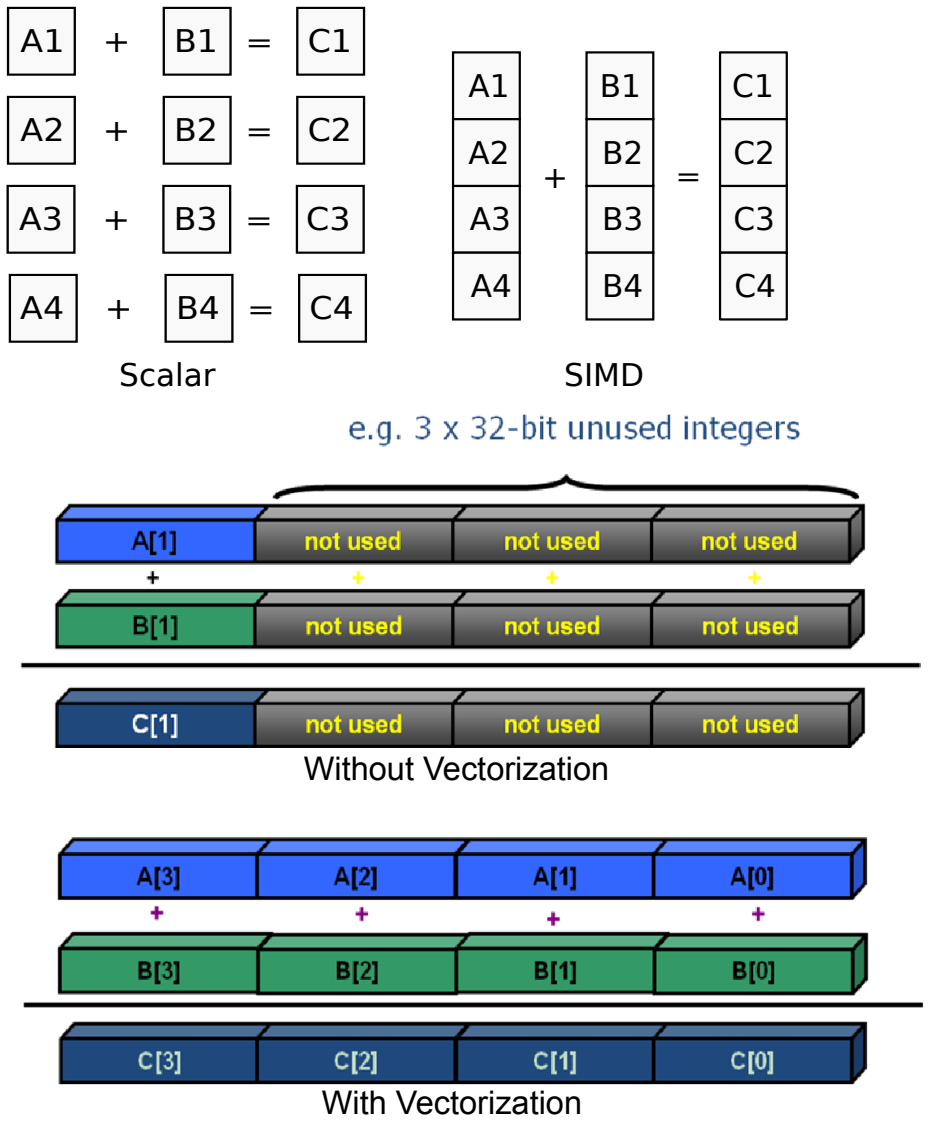
Nowait clause

1. `nowait` will disable the implicit barrier from the worksharing constructs
2. It enables fine-tuning the performance of the program
3. However, it will not suppress the #pragma omp barrier

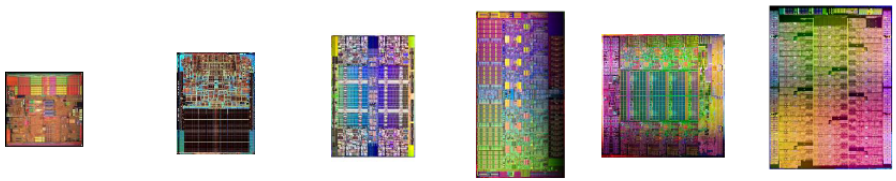
```
#pragma omp parallel num_threads(5)
  {
#pragma omp for schedule(runtime) nowait
    for (int i = 0; i < 10; ++i)
      {
      court << " no wait " << endl;
      }
    cout << " after no wait " << endl;
  } /*-- End of the parallel region --*/
```

Single Instruction Multiple Data

1. According to Flynn's taxonomy, computer architecture can be classified into four
   - one of them is Single Instruction Multiple Data (SIMD)
   - same instruction is executed on the multiple data
2. OpenMP 4.x directives support SIMD code generation

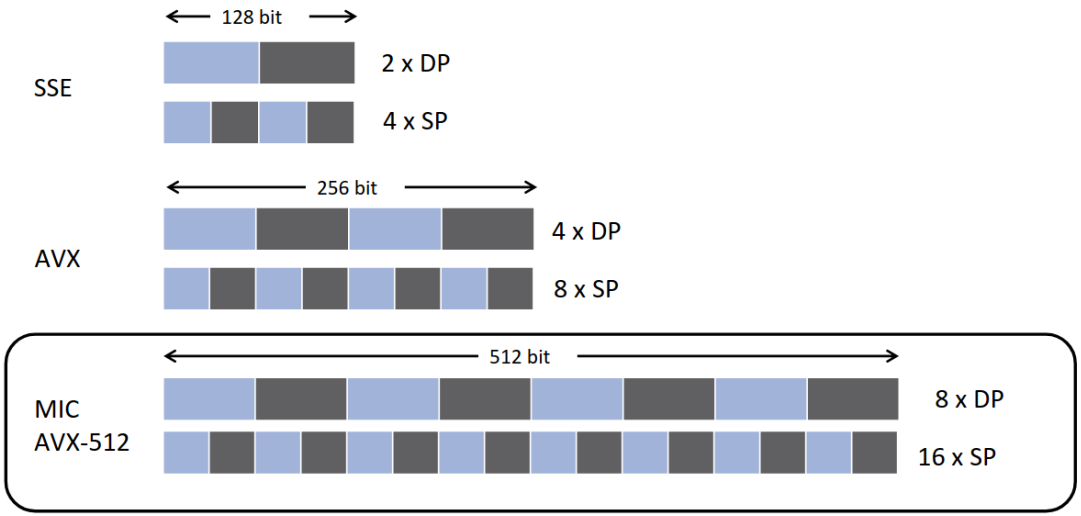


Intel's SIMD width



Images not intended to reflect actual die sizes

	64-bit Intel® Xeon® processor	Intel® Xeon® processor 5100 series	Intel® Xeon® processor 5500 series	Intel® Xeon® processor 5600 series	Intel® Xeon® processor E5-2600v2 series	Intel® Xeon Phi™ Co-processor 7120P
Frequency	3.6 GHz	3.0 GHz	3.2 GHz	3.3 GHz	2.7 GHz	1.238 MHz
Core(s)	1	2	4	6	12	61
Thread(s)	2	2	8	12	24	244
SIMD width	128 (2 clock)	128 (1 clock)	128 (1 clock)	128 (1 clock)	256 (1 clock)	512 (1 clock)



SIMD and Collapse

Example of SIMD and collapse

```
/*-- loop vectorization --*/
#pragma omp simd
  for (int i = 0; i < N; i++) {
    a[i] = a[i] + b[i] * c[i];
    } /*-- End of SIMD operation --*/
```

```
/*-- loop vectorization with scheduling --*/
#pragma omp parallel for simd schedule(static,4)
  for (int i = 0; i < N; i++) {
    a[i] = a[i] + b[i] * c[i];
    } /*-- End of the parallel region --*/
```

```
/*-- SIMD clause with Collapse --*/
#pragma omp simd collapse(2)
for (int i = 0; i < 10; ++i){
    for (int j = 0; j < 10; ++j){
        c[i][j] = a[i][j] / b[i][j];
    }
} /*-- End of the SIMD and collapse region --*/
```

Reduction and Declare

Example of reduction and declare

```
/*-- SIMD with Reduction --*/
int total = 0;
#pragma omp simd reduction(+:total)
   for (int i = 0; i < size; ++i){
           total += a[i] + b[i];}
```

```
/*-- SIMD with Declare --*/
#pragma omp declare simd
double square (double x){
   return x * x;
   }
#pragma omp simd
   for (int i = 0; i < n; ++i){
       c[i] = square(a[i];
   }
```

Utilities

Example for time measurement in OpenMP application

```
double start;
double end;
start = omp_get_wtime();
... work to be timed ...
end = omp_get_wtime();
printf("Work took %f seconds\n", end - start);
```

Profiling and Performance Analysis

1. ARM Forge
   - → Arm forge will help with debugging, profiling, and analyzing.
   - → It also supports the MPI, UPC, CUDA, and OpenMP programming models for a different architecture with different variety of compilers.
   - → Both DDT and MAP are supporting the GUI interface.
2. Intel
   - → VTune: identifying the time-consuming part in the code. Also, the identity of the cache misses and latency.
   - → Advisor: a set of collection tools for the metrics and traces that can be used for further tuning in the code. survey: analyse and explore an idea about where to add efficient vectorisation.
   - → Intel Inspector: detects and locates the code's memory, deadlocks, and data races. For example, memory access and memory leaks can be found.
   - → ASP: can be used to see the CPU utilisation, memory access efficiency and vectorisation.
3. AMD uProf profiler follows a statistical sampling-based approach to collect profile data to identify the performance bottlenecks in the application.

Important - when using OpenMP

1. Create a team of parallel threads
   → determine or have an idea about how many threads needed
2. Properly share the work among the parallel threads
   → choosing appropriate scheduling clause for worksharing
3. Identify the shared variables and private variables
   → to avoid race condition
4. Finally, use synchronization constructs where it is necessary
   → avoid computational error