Introduction to OpenMP Programming Model

Dr. Ezhilmathi Krishnasamy

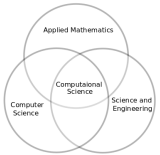
October 29, 2024

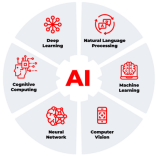
Outline

Outline for section 1

Motivation: why we need parallel programming

1. In the fields of science, engineering, and artificial intelligence, we often engage in extensive arithmetic computations with numbers.
2. Problems are typically defined by differential equations, particularly partial differential equations (PDEs).
3. To solve these PDEs, we convert them into a system of equations using various numerical methods, such as the `finite difference method`, `finite element method`, and `finite volume method`.
4. Ultimately, we must solve these systems of equations using either direct or indirect solvers to determine the values of unknown variables.
5. Similarly, in the realm of artificial intelligence (AI), we frequently work with matrices and vectors.





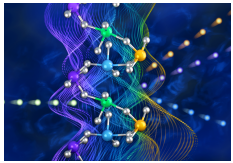
$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \quad \boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \quad \text{and} \quad \boldsymbol{b}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

Applications and examples

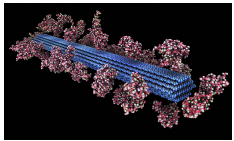
1. Materials Science: Materials science is an interdisciplinary field that combines chemistry, physics, and engineering to provide a nano-level understanding of materials. This knowledge enables the design of new materials with specific, predetermined properties.

2. Proteins are the fundamental building blocks of the human body. Proteins in cell membranes regulate the flow of substances into and out of cells. A deeper understanding of these proteins can lead to new treatments for diseases and potentially extend human lifespan.

3. In engineering, disciplines such as computational fluid dynamics, computational heat transfer, and computational solid mechanics require significant computer power to simulate and design optimized systems. Traditionally, these processes relied on experiments and wind tunnel tests, which could be costly in terms of both finances and human resources.



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

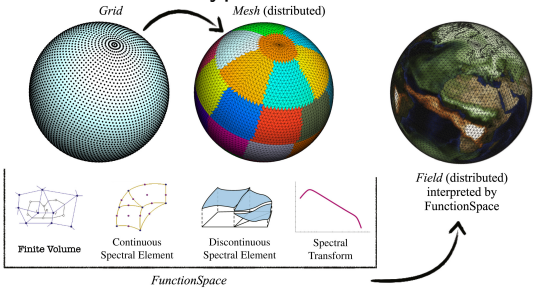


Biology: Credit:www.olcf.ornl.gov

Engineering: Credit:www.olcf.ornl.gov

Applications and examples

In weather forecasting, numerical methods are used to discretize the domain size. For instance, simulating a resolution of 3 kilometers (with 1.8 billion cells) on the IBM Blue Gene/Q Mira system at Argonne National Laboratory requires 786,000 cores. This simulation uses spectral element methods, which are one type of numerical method used in the field[1].



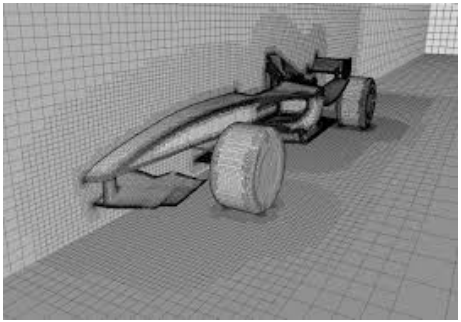
Mesh and simulation settings [2]

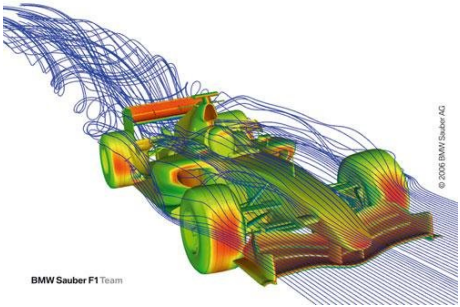
[1] HPC for Weather Forecasting, John Michalakes

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

Applications and examples

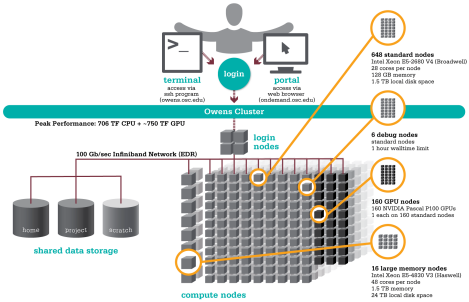
CERFACS in France utilized up to 250,000 cores with grid sizes ranging from 2 to 4 billion cells using Large Eddy Simulation, a numerical method strategy for computational fluid dynamics applications.





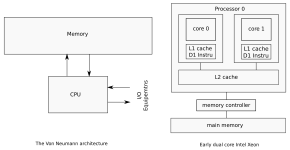
What is supercomputer

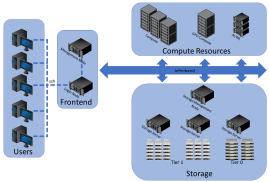
1. A large machine is composed of multiple processors and extensive memory.
2. The processors (CPUs) and their memory are interconnected by network cables.
3. Multicore processors are included within the CPUs.
4. By employing parallel computing techniques, these machines can be utilized more efficiently.



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





Computer speed

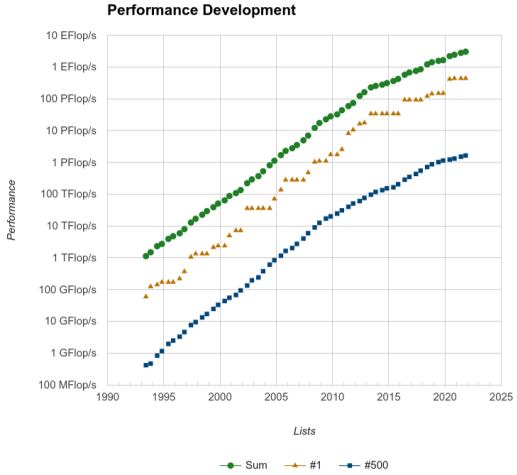
1. FLOPS: Floating Point Operations Per Second - this metric measures the speed of a computer.
2. The speed of a supercomputer is determined by running the LINPACK benchmark applications.
3. Typically, this involves solving N-dimensional linear systems using Gaussian elimination with partial pivoting.
4. The TOP500 regularly lists the fastest supercomputers.
5. As of June 2024, the top 3 supercomputers are:
   - Frontier: HPE Cray EX235a, AMD EPYC 64C at 2GHz, AMD Instinct MI250X, Slingshot-11; DOE/SC/Oak Ridge, USA.
   - Aurora: HPE Cray EX, Intel Xeon CPU Max, Intel Data Center GPU Max, Slingshot-11; Argonne, USA.
   - Eagle: Microsoft NDv5, Intel Xeon Platinum 8480C, NVIDIA H100; Microsoft Azure, USA.

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

Modern supercomputers primarily operate at the levels of petaflops (Pflops) and exaflops (Exaflops).

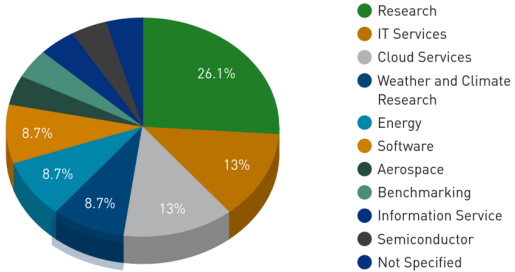
Computing power trend



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

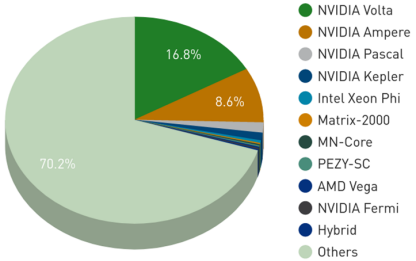
How it is used in different research sectors

Application Area System Share



credit: *top500*

Accelerator/CP Family System Share



credit: *top500*

Supercomputers in Luxembourg

1. MeluXina: Accelerator Module - BullSequana XH2000, featuring an AMD EPYC 7452 processor with 32 cores at 2.35 GHz, an NVIDIA A100 with 40 GB, and equipped with Mellanox HDR InfiniBand/ParTec ParaStation ClusterSuite, provided by Atos and LuxProvide in Luxembourg. Ranked 36th in 2021 (according to TOP500).
2. Aion is a supercomputer developed by Atos/Bull/AMD, consisting of 318 compute nodes. It totals 40704 compute cores and 81408 GB of RAM, delivering a peak performance of approximately 1.70 PetaFLOP/s.
3. Iris is a Dell/Intel supercomputer comprised of 196 compute nodes, totaling 5824 compute cores and 52224 GB of RAM, with a peak performance of around 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

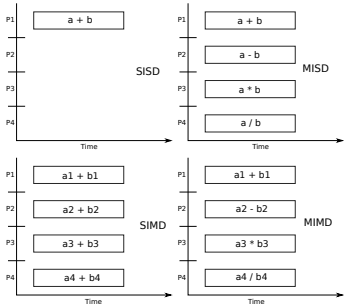
Outline for section 2

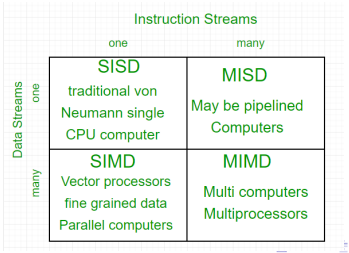
Classification of Parallel Computer Architecture

1. Control Structure: Based on the instruction and data streams.
   - SISD - Single Instruction Stream, Single Data Stream
   - SIMD - Single Instruction Stream, Multiple Data Streams
   - MISD - Multiple Instruction Streams, Single Data Stream
   - MIMD - Multiple Instruction Streams, Multiple Data Streams
2. Memory Organization: Shared Memory and Distributed Memory
3. Network Topology (Connectivity): For example, 3D Grid and Tree

Control Structure - Flynn's Taxonomy

1. SISD - Represents sequential execution, exemplified by conventional single processors in the von Neumann model.
2. SIMD - Refers to array computers and traditional vector processors, where instructions can be executed either in a pipeline or in parallel.
3. MISD - This model is rarely used in practice.
4. MIMD - Operates with both shared and distributed memory models.

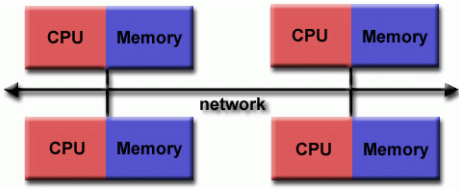




Outline for section 3

Distributed memory architecture

1. A distributed memory architecture is designed using a network connection.
2. There is no global address space; each compute node has its own local memory.
3. There is no cache coherence, meaning that if one processor updates memory, that change is not automatically visible to another processor at the hardware level.
4. It is up to the programmers to determine how and from where data should be accessed or transferred.



Distributed memory architecture

1. Advantages:
   - It is scalable with respect to memory and processors, allowing for large-scale computations.
   - Each processor within a node can quickly access its own memory without the need for global cache coherence.
   - Networking enables access to multiple processors and memory resources.

2. Disadvantages:
   - Accessing local data is faster than accessing data across compute nodes via interconnect networks.
   - It is the programmer's responsibility to manage read and write operations across multiple compute nodes and their remote memory.

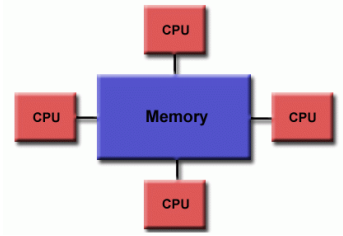
Outline for section 4

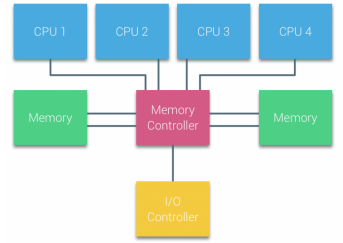
Shared memory architecture

1. In shared memory computers, all processors have access to a global address space of memory.
2. Multiple processors within a shared memory architecture can operate independently while sharing the same memory resources.
3. Changes made to a memory location by one processor are visible to all other processors.
4. Shared memory architecture is categorized into two types: Unified Memory Access (UMA) and Non-Unified Memory Access (NUMA), based on how memory is accessed.

Unified Memory Access (UMA)

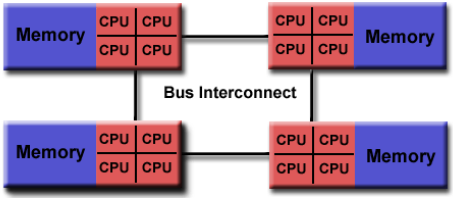
1. All processors access memory through an interconnection network, similar to how a single processor accesses its own memory.
2. Memory access times for all processors are equal, meaning they take the same amount of time when accessing memory through the interconnection network (e.g., bus).
3. This architecture is often represented by Symmetric Multiprocessor (SMP) systems, which feature identical processors.
4. It is also referred to as Cache Coherent UMA (CC-UMA), where memory updates made by one processor are visible to other processors, achieved through hardware-level configurations.

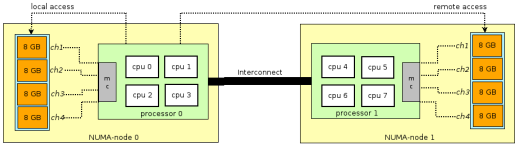




Non-Unified Memory Access (NUMA)

1. Each set of processors (Symmetric Multiprocessing, or SMP) has its own dedicated memory.
2. Processors from one SMP can access the memory of another SMP.
3. The time taken to access memory varies among different processors across the SMPs, leading to slower memory access when communicating between SMPs.
4. NUMA can also include Cache Coherent NUMA (CC-NUMA) through specific hardware-level configurations.





Shared memory architecture

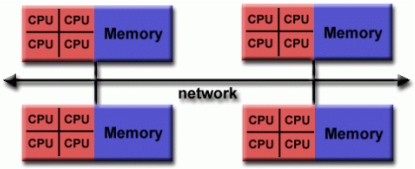
1. Advantages
   - Rapid and uniform data sharing between processes, due to the close proximity of memory to CPUs.
   - The global address space provides a user-friendly perspective for memory access.
   - No need for explicit specification of data communication between processes.
   - Easy to learn and program (e.g., OpenMP).

2. Disadvantages
   - Scalability is limited by memory and processor capabilities.
   - Programmers must implement synchronization constructs (e.g., locks) to ensure access to the correct data in global memory.

Outline for section 5

Hybrid distributed-shared memory

1. Modern supercomputers utilize both shared and distributed memory architectures.
2. Shared memory components may include GPUs and other accelerators that can access this shared memory.
3. Distributed memory connects to shared memory through network connections, allowing memory access from one shared memory node to another via an interconnect network.
4. Scalability can be enhanced by increasing the number of compute nodes and memory—this is an advantage.
5. However, there is increased programmer complexity involving tasks, data management, and communication—this is a disadvantage.



Outline for section 6

What is OpenMP

1. **OpenMP** is an application programming interface (API) designed to facilitate parallel programming on shared memory architectures, including both UMA and NUMA.
2. It uses a directive-based programming model to parallelize serial code written in **C/C++** and **Fortran**.
3. The **directives** instruct the compiler on how to execute code in parallel within the shared memory environment.
4. A single application can contain both serial and parallel versions of its code.
5. The latest specification of the **OpenMP API 5.2** was released in November 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

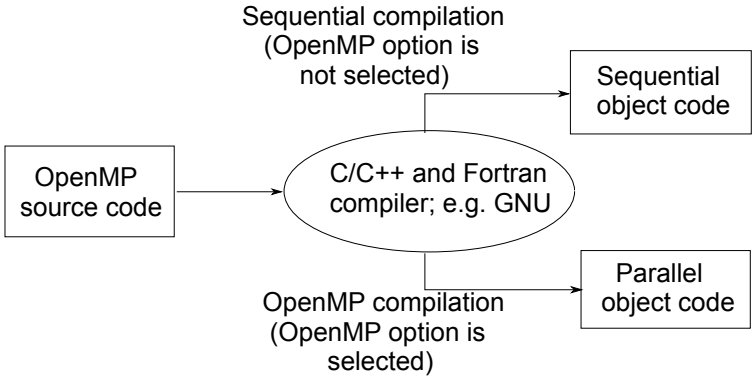
GNU

```
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
```

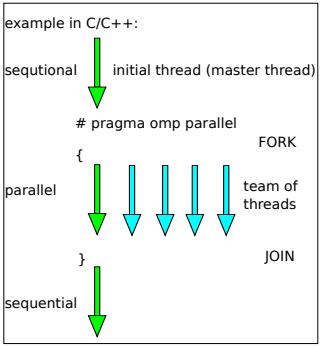
Compilation



Outline for section 7

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
```

Outline for section 8

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
```

Outline for section 9

Shared variable

1. All threads can access the shared variable.
2. By default, in a parallel region, all variables are treated as shared.
3. Shared variables must be managed with care to avoid race conditions in the program.
4. In this context, all threads within the parallel region can access the elements of 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 has its own copy of the private variable.
2. The private variable is accessible only within the parallel region, not outside of it.
3. By default, loop iteration counters are considered private.
4. Changes made by one thread are 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 similar to a private clause.
2. Each thread will have an uninitialized copy of the variables specified as lastprivate.
3. At the end of the parallel loop or sections, the value of the lastprivate variable will be the final value accessed by the last thread that modified it within the section or 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:` behaves like a private clause.
2. Each thread receives an initialized copy of the variables passed as firstprivate.
3. It is 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;
```

Outline for section 10

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:** The 'for' directive executes a loop in parallel.
2. This directive should be placed within the parallel region using `#pragma omp parallel`.
3. By default, the loop counter index is private.
4. `Loop Scheduling:` This feature includes the following clauses:
   - → static
   - → dynamic
   - → guided
   - → auto
   - → runtime

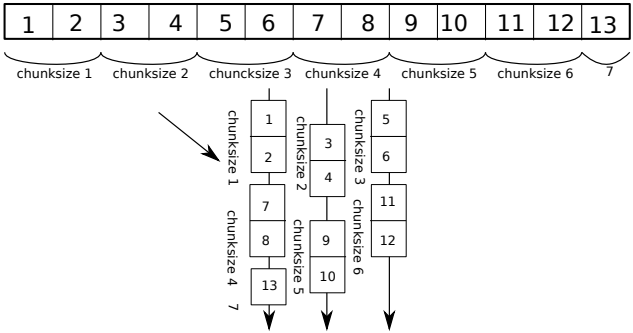
Functionality	Syntax in C/C++	Syntax in Fortran
Distribute iterations over the threads	`#pragma omp for`	`!$omp do`

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;
}
```

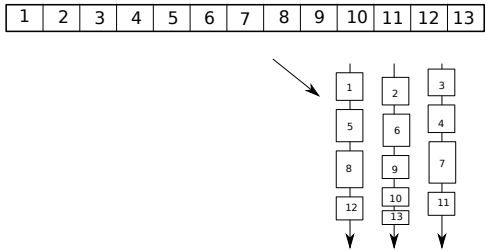
Static clause

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 approach is beneficial when the computational cost is similar for each iteration.



Dynamic clause

1. The number of iterations is divided by the chunk size.
2. If the chunk size is not specified, the default value of 1 will be used.
3. The system will request a chunk of data until no more chunks are available.
4. There is no specific pattern for how different threads access the various chunks of data.
5. This approach is beneficial when the computational cost varies across iterations.
6. This method efficiently places chunks of data into the queue.



Guided, Auto, and Runtime schedule clause

1. **Guided:**
   - Similar to dynamic scheduling, the number of iterations is divided into a "chunksize."
   - However, the size of each chunk decreases proportionally to the number of unsigned iterations divided by the number of threads.
   - If a "chunksize" is not specified, the system will use a default value of 1.
   - This approach is useful when there is poor load balancing towards the end of the iterations.
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 managing nested loops.
2. The total number of iterations is divided among the available threads.
3. For example, if the outer loop index equals the number of threads, the innermost loop (with a size larger than the number of threads) will have its work distributed among those threads.
4. Only one collapse clause is permitted 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 are useful when multiple tasks (functions or subroutines) need to be processed in parallel.
3. Each thread will execute one task only once.
4. Tasks within the sections should be independent of each other.

```
#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. The order in which the threads access the single construct is random.
2. Only one thread will execute the single construct, and only once.
3. The single construct waits for all other constructs to finish their tasks.

```
#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

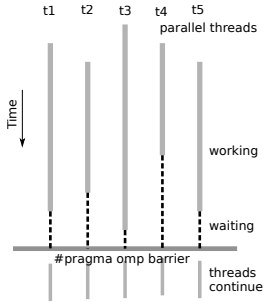
1. Only the master thread executes the master construct.
2. It functions similarly to a single construct; however, it does not include an implied barrier.
3. It must be used in conjunction with a `barrier` when there is a data dependency.

```
#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 --*/
```

Outline for section 11

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 allows only one thread at a time within the critical section.
2. Multiple threads cannot access the critical section in parallel.
3. This prevents race conditions.

```
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. This method is similar to critical sections but offers a performance advantage.
2. It consists of a single statement followed by the atomic declaration.
3. This method safely updates a shared numeric variable and supports memory location updates.

```
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 summing the values of an array into a shared numerical variable.
2. The syntax for reduction is: `reduction(operators: variable)`
   - Supported arithmetic reductions include: `'+, *, -, max, min'`
   - Supported logical operator reductions include: `', |, ∧, ||'`
3. This should be used instead of critical sections (which allow only one thread to update the variable at a time).

```
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

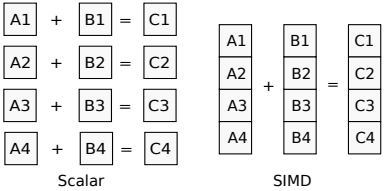
1. The `nowait` clause disables the implicit barrier in worksharing constructs.
2. It allows for fine-tuning the program's performance.
3. However, it will not affect the `#pragma omp barrier` directive.

```
#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 --*/
```

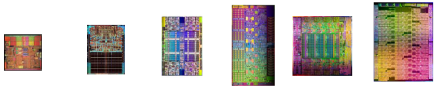
Outline for section 12

Single Instruction Multiple Data

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

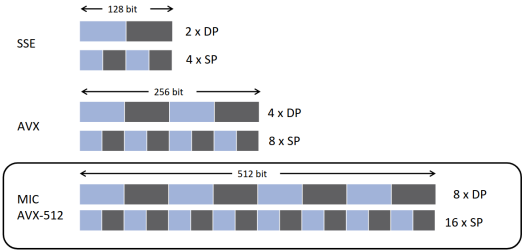


Intel's SIMD width



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 assists with debugging, profiling, and analysis.
   - → It supports multiple programming models, including MPI, UPC, CUDA, and OpenMP, across various architectures and compilers.
   - → Both DDT and MAP offer a graphical user interface (GUI).
2. Intel
   - → VTune: Identifies time-consuming sections of code, cache misses, and latency issues.
   - → Advisor: A collection of tools for gathering metrics and traces, which can be used for further code optimization. It can help analyze and explore ideas for efficient vectorization.
   - → Intel Inspector: Detects and identifies memory issues, deadlocks, and data races, such as memory access problems and leaks.
   - → ASP: Used to monitor CPU utilization, memory access efficiency, and vectorization.
3. AMD uProf employs a statistical sampling-based approach to collect profiling data, helping to identify performance bottlenecks in applications.

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