Samina Haque

University of Calcutta

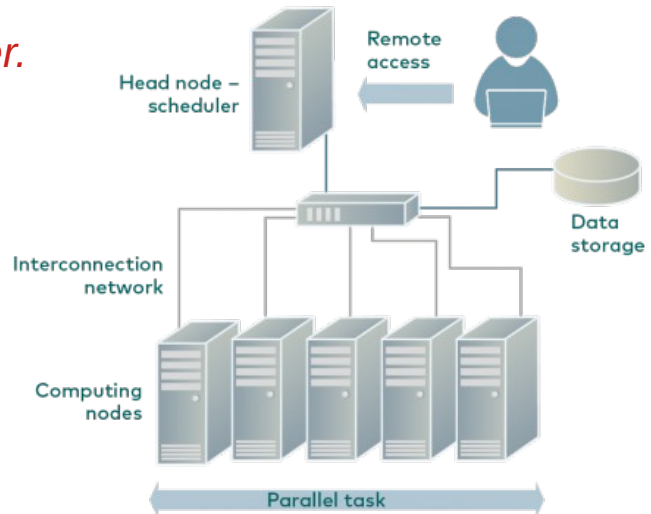
Overview

- What is HPC and why is it important.
- Application and Difference from Desktop Computing
- HPC hardware and software requirments
- Important terminologies
- History
- HPC Layout
- HPC Architecture
- Task Vs Data Parallelism
- Parallel Computing
- OpenMP
- OpenMP 4.0

Introduction to High Performance Computing

High-performance computing (HPC) is the ability to process data and perform complex calculations at high speeds. Basically it is a mechanism that combines a large numbers of processors and makes their combined computing power available to use.

Based fundamentally on parallel computing: using many processors (cores) at the same time to solve a problem.
One of the best-known types of HPC solutions is the *supercomputer.*



Why Is HPC Important?

- HPC is the foundation for scientific, industrial, and societal advancements.
- As technologies like the Internet of Things (IoT), artificial intelligence (AI), and 3-D imaging evolve, the size and amount of data that organizations have to work with is growing exponentially.
- It helps to keep a step ahead of the competition, organizations need lightning-fast, highly reliable IT infrastructure to process, store, and analyze massive amounts of data .
- Scientific simulation and modelling drive the need for greater computing power.
- SingleCore processors can not be made that have enough resource for the simulations needed as the power/heat increases. Higher processors are very expensive.

Applications of HPC

HPC is Driven by demand of computation-intensive applications from various areas

- Biology, neuroscience (e.g. simulation of brains)
- Finance (e.g. modelling the world economy)
- Military and Defence (e.g. modelling explosion of nuclear bombs)
- Engineering (e.g. simulations of a car crash)

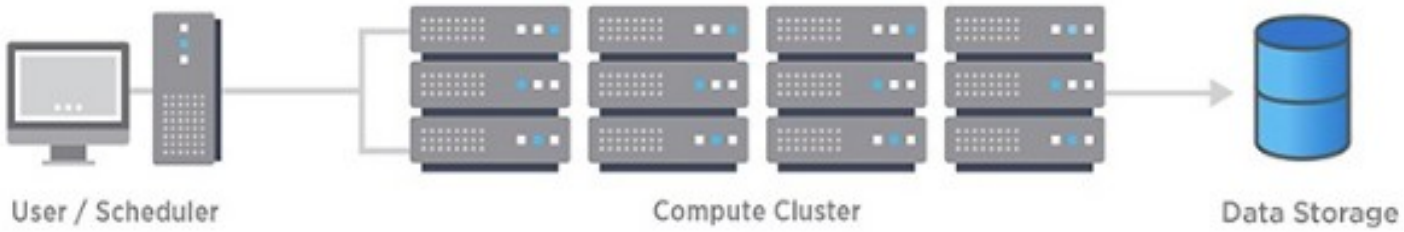
Differences from Desktop Computing

- Do not log on to computernodes directly i.e submit jobs via a batch scheduling system.
- Not a GUI based environment
- Shares the system with many users
- Resources more tightly monitored and controlled.
- Less layers of Software

What Does High Performance Computing Include?

HPC solutions have three main components:

1.Compute
2.Network
3.Storage



Hardware
– Computer Architecture: Vector Computers, MPP, SMP, Distributed Systems, Clusters
– Network Connections: InfiniBand, Ethernet, Proprietary

Software
– Programming models: MPI (Message Passing Interface), SHMEM (Shared Memory), PGAS, etc.
– Applications: Open source, commercia

HPC Terminologies:

Terms	Explanation
Node	A physical computer
Processor	a multi-core processor housing many processing elements
Socket	plug where processor is placed, synonym for the processor
Core	Individual processing element
Task	software process with own data & instructions forking multiple threads
Thread	An instruction stream sharing data with other threads from the task

The Fundamentals:

A multitude of concepts have been developed :

- **Pipelined functional units**: By subdividing complex operations (like, e.g., floating point addition and multiplication) into simple components that can be executed using different functional units on the CPU,it is possible to increase instruction throughput, i.e., the number of instructions executed per clock cycle.
- **Superscalar architecture**. Superscalarity provides “direct” instruction-level parallelism by enabling an instruction throughput of more than one per cycle.
- **Data parallelism through SIMD instructions**: SIMD (Single Instruction Multi-ple Data) instructions issue identical operations on a whole array of integer or FP operands, usually in special registers. They improve arithmetic peak per-formance without the requirement for increased superscalarity.
- **Out-of-order execution:** This improves instruction throughput and makes it easier for compilers to arrange machine code for optimal performance.
- **Larger caches:** Small, fast, on-chip memories serve as temporary data storage for holding copies of data that is to be used again “soon,” or that is close to data that has recently been used. This is essential due to the increasing gap between processor and memory speeds.
- **Simplified instruction set**: In the 1980s, a general move from the CISC to the RISC paradigm took place.With RISC, the clock rate of microprocessors could be increased in a way that would never have been possible with CISC. Additionally, it frees up transistors for other uses.

History of High Performance Computing:

1960s: Scalar processor
- Process one data item at a time

1970s: Vector processor
- Can process an array of data items at one go
- Architecture and Overhead

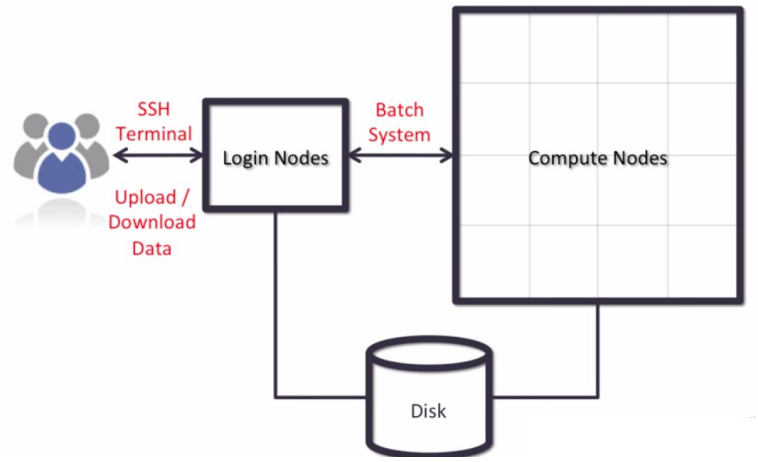
Later 1980s: Massively Parallel Processing (MPP)
- Up to thousands of processors, each with its own memory and OS
- Break down a problem

Later 1990s: Cluster
- Not a new term itself, but renewed interests
- Connecting stand-alone computers with high-speed network

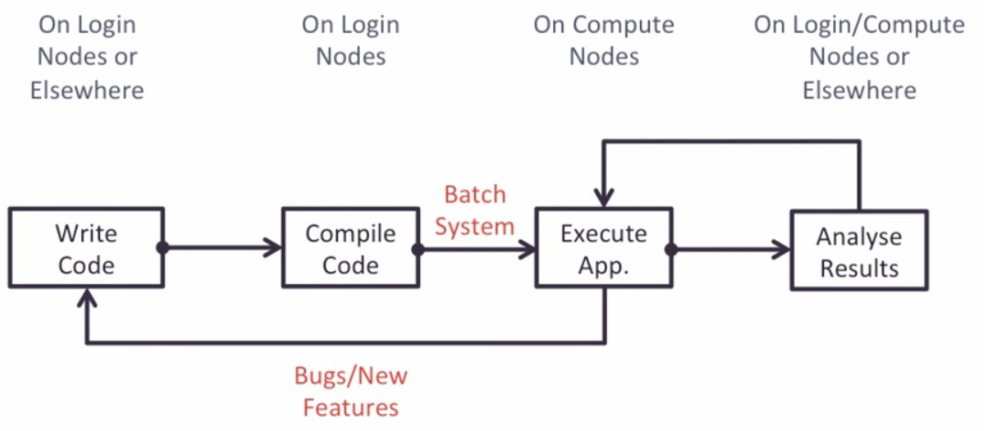
Later 1990s: Grid
- Tackle collaboration among geographically distributed organisations

HPC Layout

HPC System Layout



HPC Software Usage Flow



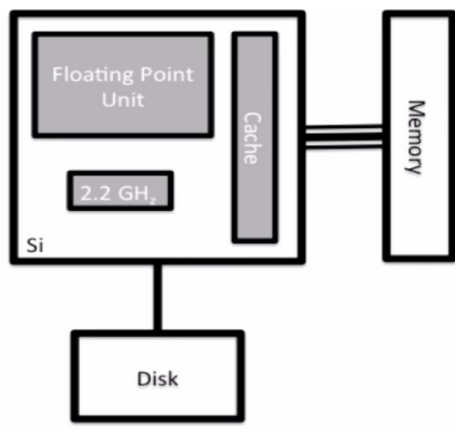
Majority of HPC machines follow this generic conceptual layout for a computer cluster:

- many compute nodes connected together by a network
- each compute node has separate, independent memory

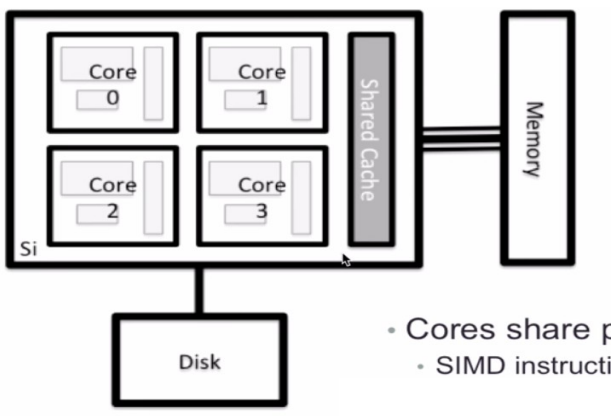
What Type of Hardware HPC Use:

Basically HPC follows the Von Neuman Architecture as the Common computers.

We have some memory which stores dataset,disk which stores instruction ,the processor which manages the data .
It has multicore processing unit each having similar significance and shares a common shared cache.



Von Neuman Architecture



Multicore Architecture

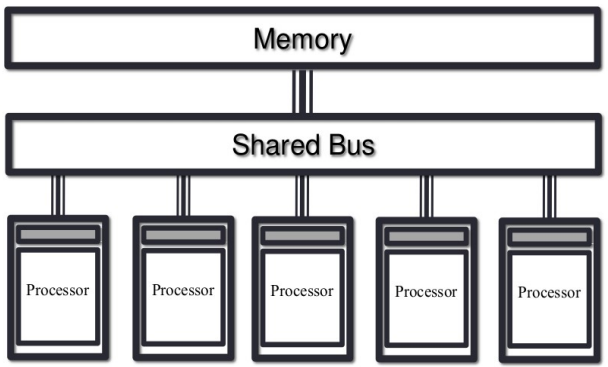
HPC Architecture

Shared memory architectures

Simplest to use, hardest to use.

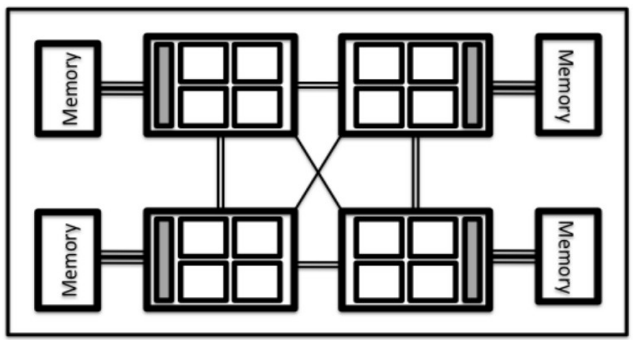
- Multi-processor shared-memory systems have been common since the early 90's
  - originally built from many single-core processors
  - multiple sockets sharing a common memory system
- A single OS controls the entire shared-memory system
- Modern multicore processors are just shared-memory systems on a single chip

Symmetric Multi-Processing Architectures



- All cores have the same access to memory, e.g. a multicore laptop

Non-Uniform Memory Access Architectures



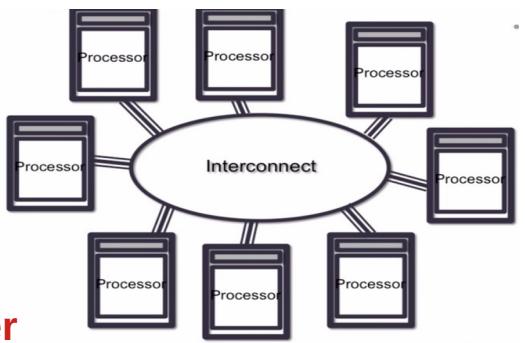
- Cores have faster access to their own local memory

HPC Architecture

Distributed memory architectures

Clusters and interconnects.

- Each selfcontained part is called a node and each node runs its own OS
- Each processors i.e nodes are interconnected.
- The performance of parallel programs often depends on the interconnect performance
  - Although once it is of a certain (high) quality, applications usually reveal themselves to be CPU, memory or IO bound
  - Low quality interconnects do not usually provide the performance required
  - Specialist interconnects are required to produce the largest supercomputers.
  - Infiniband is dominant on smaller systems
- High bandwidth relatively easy to achieve
  - low latency is usually more important single chip



MultiComputer

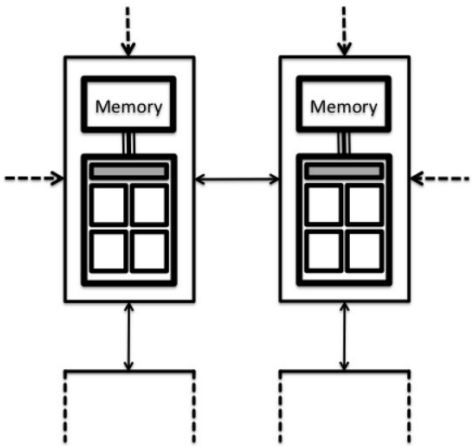
HPC Architeture

Distributed and Shared Hybrid memory architectures

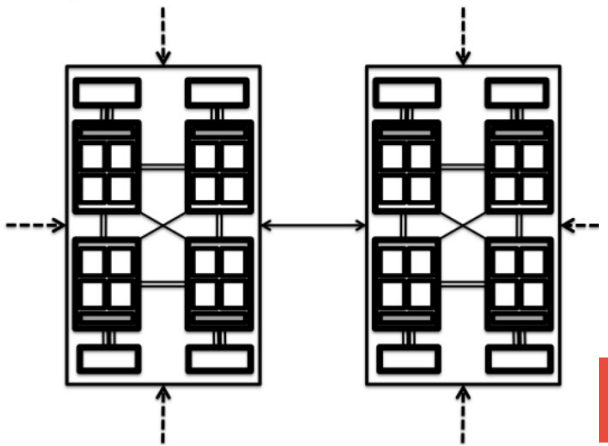
Almost all HPC machines fall in this class

- Most applications use a message-passing (MPI) model for programming
  - Usually use a single process per core
- Increased use of hybrid message-passing + shared memory (MPI+OpenMP) programming
  - Usually use 1 or more processes per NUMA region and then the appropriate number of shared-memory threads to occupy all the cores
- Placement of processes and threads can become complicated on these machines

Multicore nodes



Hybrid architectures



Task VS Data Parallelism

Task parallel (maps to high-level MIMD machine model)

- Task differentiation
- Communication via shared address space or message passing
- Synchronization is explicit (via locks and barriers)
- Underscores operations on private data, explicit constructs for communication of shared data and synchronization
- Amount of parallelization is proportional to the input size.

Data parallel (maps to high-level SIMD machine model)

- Global actions on data by tasks that execute the same code
- Communication via shared memory or logically shared address space with underlying message passing
- Synchronization is implicit (lock-step execution)
- Underscores operations on shared data, private data must be defined explicitly or is simply mapped onto shared data space
- Amount of parallelization is proportional to the number of independent tasks is performed.

Parallel Computing

Parallel computing refers to the process of breaking down larger problems into smaller, independent, often similar parts that can be executed simultaneously by multiple processors communicating via shared memory, the results of which are combined upon completion as part of an overall algorithm

Parallel Programming Model

Programming model is a conceptualization of the machine that a programmer uses for developing applications

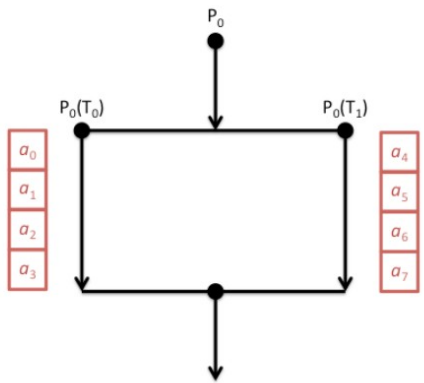
- Shared memory programming
  - Tasks operate and communicate via shared data.
- Message passing programming
  - Explicit point-to-point communication, like phone calls (connection oriented) or email (connectionless, mailbox posts)

OpenMP

Open Multi Processing

OpenMP

- OpenMP is an Application Program Interface (API), jointly defined by a group of major computer hardware and software vendors. OpenMP provides a portable, scalable model for developers of shared memory parallel applications.
- OpenMP is a set of extensions to Fortran, C and C++:
    - Compiler directives
    - Runtime library routines
    - Environment variables
- A directive is a special line of source code with meaning only to certain compilers thanks to keywords (sentinels)
    - Directives are ignored if code is compiled as regular sequential Fortran/C/C++
- Threads “communicate” by having access to the same memory space
    - Any thread can alter any bit of data
    - No explicit communications between the parallel tasks



Goals of OpenMP:

Standardization:

- Provide a standard among a variety of shared memory architectures/platforms
- Jointly defined and endorsed by a group of major computer hardware and software vendors

Lean and Mean:

- Establish a simple and limited set of directives for programming shared memory machines.
- Significant parallelism can be implemented by using just 3 or 4 directives.
- This goal is becoming less meaningful with each new release, apparently.

Ease of Use:

- Provide capability to incrementally parallelize a serial program, unlike message-passing libraries which typically require an all or nothing approach
- Provide the capability to implement both coarse-grain and fine-grain parallelism

Portability:

- The API is specified for C/C++ and Fortra
- Most major platforms have been implemented including Unix/Linux platforms and Windows

Features of OpenMP on Parallelism

- **Thread Based Parallelism:**
  OpenMP programs accomplish parallelism exclusively through the use of threads.

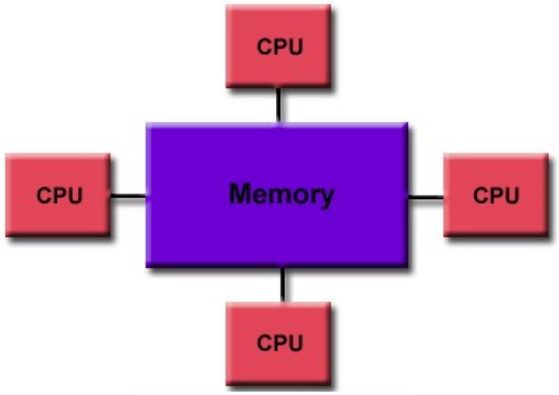
- **Explicit Parallelism:**
  OpenMP is an explicit (not automatic) programming model, offering the programmer full control over parallelization.

- **Data Scoping:**
  - Because OpenMP is a shared memory programming model, most data within a parallel region is shared by default.
  - All threads in a parallel region can access this shared data simultaneously.
  - OpenMP provides a way for the programmer to explicitly specify how data is "scoped" if the default shared scoping is not desired.

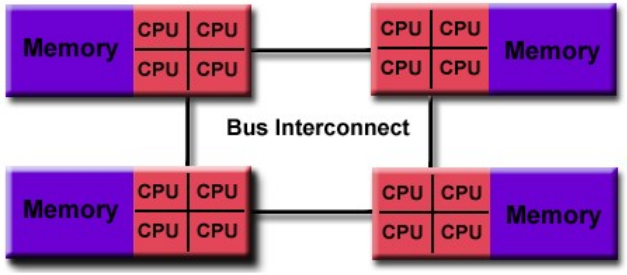
- **Nested Parallelism:**
  The API provides for the placement of parallel regions inside other parallel regions.

OpenMP memory model

- OpenMP supports a shared memory model.
- OpenMP is designed for multi-processor/core, shared memory machines. The underlying architecture can be shared memory UMA or NUMA.



Uniform Memory Access



Non-Uniform Memory Access

- Because OpenMP is designed for shared memory parallel programming, it largely limited to single node parallelism. Typically, the number of processing elements (cores) on a node determine how much parallelism can be implemented.

Execution Model

Fork-join model

- Program starts with a single (master) thread
- Multiple threads are forked by the master thread at a parallel construct

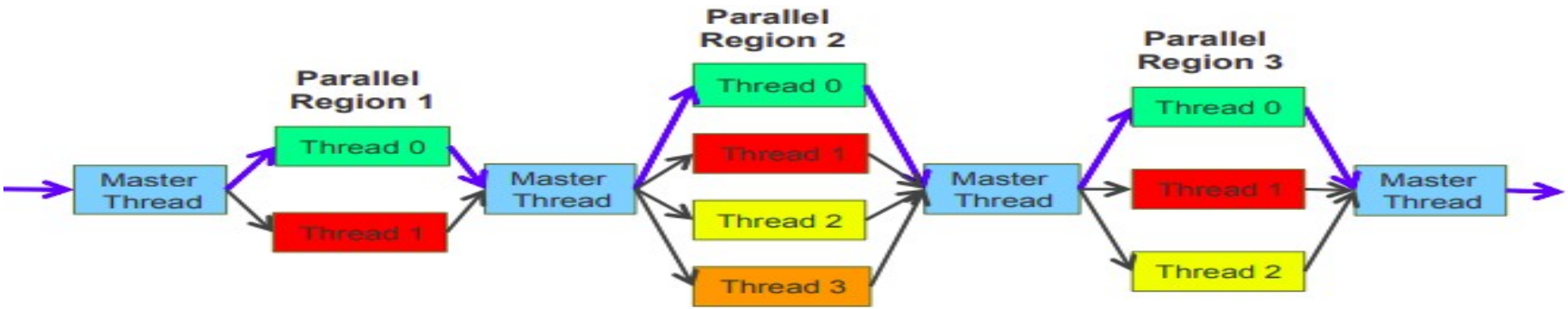
• The master thread is part of the new team of threads

- Threads perform work in the parallel region

• Worksharing constructs distribute work among threads

• Threads may be synchronized with synchronization constructs

- Threads join at the end of the parallel region and the master thread continues



OpenMP Syntax:

- Some functions and types
  #include <omp.h>
- Most apply to a block of code
  Specifically, a “structured block”
  Enter at top, exit at bottom only*
- exit(), abort() permit

- **ParallelRegion:**

```
#pragma omp parallel
        {
        ... code executed by each thread
        }
```

- Effectively a single thread runs before:
  - “fork” at the beginning
  - “join” at the end
- Single thread runs after

Parallel Construct

The fundamental construct to start parallel execution

- Invocation of a team of threads
- Code executed redundantly by every thread until a worksharing construct is encountered
- Number of threads controlled via

• The OMP_NUM_THREADS environment variable
• A call to omp_set_num_threads(), or
• The num_threads clause

```
omp_set_num_threads(4);
 #pragma omp parallel private(myid)
 {      myid = omp_get_thread_num();
        printf("myid is %d\n", myid);
 }
```

Worksharing Construct

• The construct to distribute work among threads

- for (or do): used to split up loop iterations among the threads

```
#pragma omp for for (i=0; i<n; i++)
 a[i] = b[i] + c[i];
```

- sections: assigning consecutive but independent code blocks to different threads (can be used to specify task parallelism)

  • Each code block is indicated by the section directive

```
#pragma omp sections {
        #pragma omp section
                work1();
        #pragma omp section
                work2();
}
```

Worksharing Construct (cont.)

- single: specifying a code block executed by only one thread

```
#pragma omp single
 s = 0;
```

• There is an implicit barrier at the end of a worksharing construct

- But can be suppressed with the “nowait” clause

```
#pragma omp for nowait
for (i=0; i<n; i++)
        a[i] = b[i] + c[i];
#pragma omp for
 for (i=0; i<n; i++)
        d[i] = e[i] + f[i];
```

• Master construct

- code block executed by the master thread only, no barrier wait

Loop Scheduling

-Clause to define how loop iterations are distributed among threads of the team

```
#pragma omp for schedule(static)
for (i=0; i<n; i++)
a[i] = b[i] + c[i];
```

- Loop scheduling kinds
  - static: for balanced workload, lowest overhead
    - Default for most compilers
  - dynamic: for unbalanced loop iterations
  - guided: for special monotonically increasing or decreasing workload
  - auto: compiler determines at runtime

Data Sharing

- Accessibility of variables by threads
  - shared: variable is shared by all threads in a team
  - private: variable is private to each thread
  - By default, variables are shared
    - With some exceptions, such as, loop variable is private

- Specifying data sharing attribute in a parallel region or worksharing construct
  - shared clause: for variables shared by threads
  - private clause: for variables private to each thread
  - reduction clause: combining private copies of a variable to the shared copy by an operator. Reduced final value is only guaranteed at a barrier.

Data Sharing (cont.)

• An example

```
s = 0.0;
#pragma omp parallel for private(i,b) \ shared(a) reduction(+:s)
 for (i = 0; i < n; i++) {
        b = a[i] * a[i];
        s+= b; }
printf("sum = %g\n", s);
```

• Threadprivate directive

- Special storage for global variables, private to each thread
- Specified at the variable declaration, valid throughout the program

Synchronization

- Barrier: wait until all of threads of a team have reached this point before continuing
  - Barrier construct specifies an explicit barrier
  - A worksharing construct has an implicit barrier at the end
- Critical construct
  - Code block executed by only one thread at a time, e.g., allows multiplethreads to update shared data

```
#pragma omp critical
{
        s = s + s_local;
}
#pragma omp barrier
        printf("sum = %g\n", s);
```

Synchronization (cont.)

- Atomic construct
  - Update a shared variable atomically, can be more efficient than the critical construct if there is hardware support
  - Only valid for scalar variable and a limited set of operations (+,*,-,…)

```
#pragma omp atomic
      s = s + s_local;
#pragma omp barrier
      printf("sum = %g\n", s);
```

  - Other forms are also available:
    - "atomic read", "atomic write", "atomic capture"
- Locks
  - Similarly to critical but provided by the library routines and more flexible

Data Sharing: tasks (OpenMP 3.0)

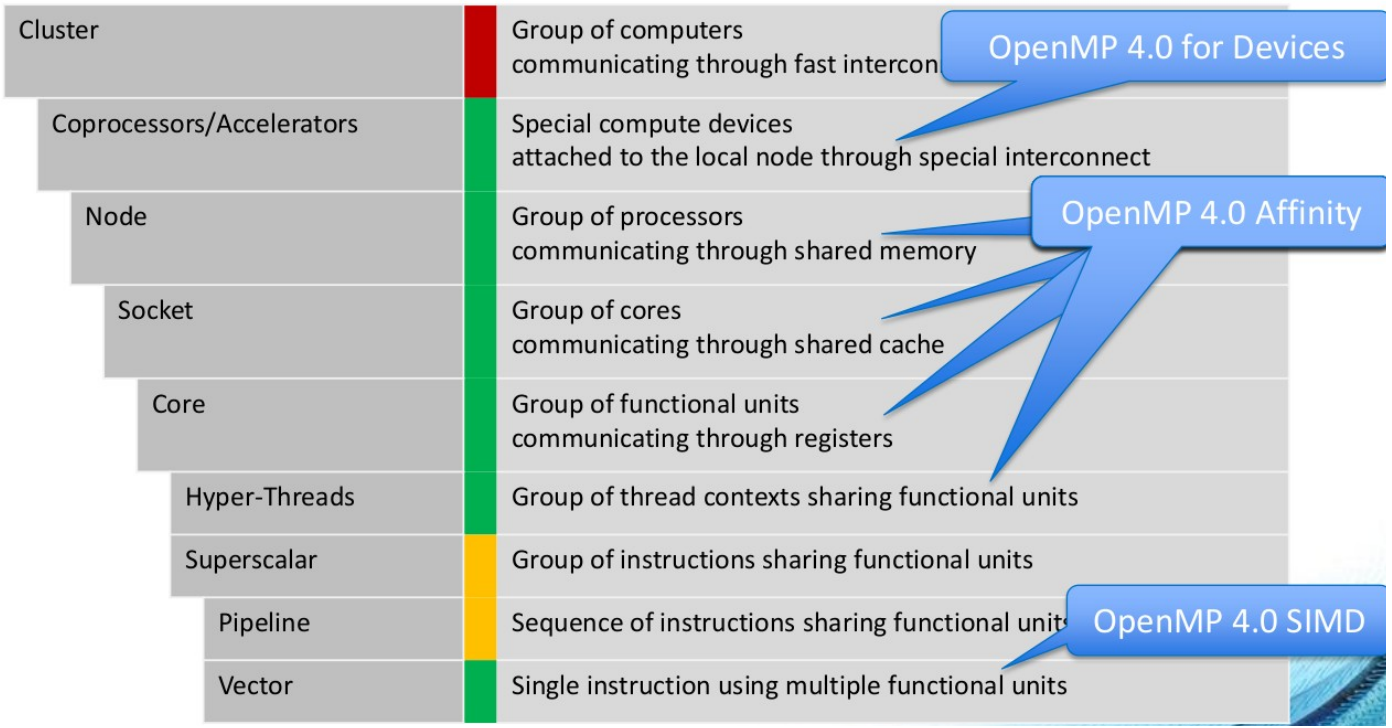
- The default for tasks is usually firstprivate, because the task may not be executed until later (and variables may have gone out of scope).
- Variables that are shared in all constructs starting from the innermost enclosing parallel construct are shared, because the barrier guarantees task completion.

```
#pragma omp parallel shared(A)
private(B)
{
...
        #pragma omp task
        {
                int C;
                compute(A, B, C);
        }
}
```

OpenMP 4.0

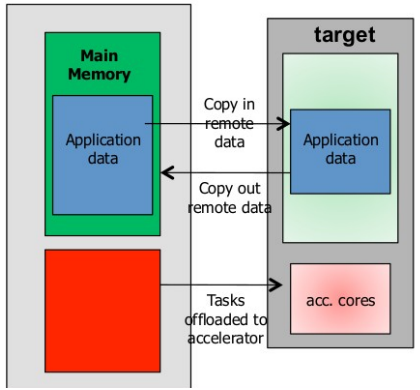
Open Multi Processing

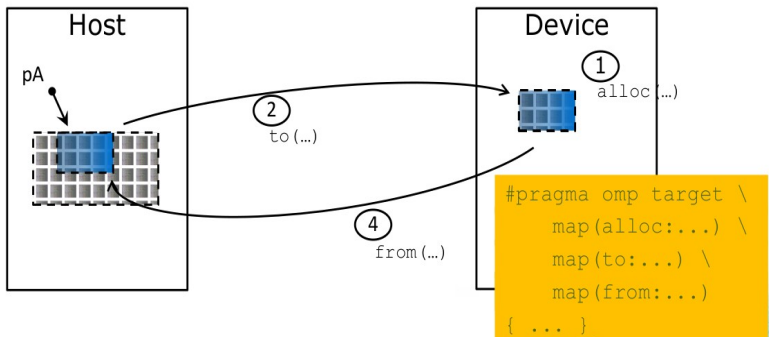
Levels of Parallelism in OpenMP 4.0



Execution Model:

- The target construct transfers the control flow to the target device
  - Transfer of control is sequential and synchronous
  - The transfer clauses control direction of data flow
  - Array notation is used to describe array length
- The target data construct creates a scoped device data environment
  - Does not include a transfer of control
  - The transfer clauses control direction of data flow
  - The device data environment is valid through the lifetime of the target data region
- Use target update to request data transfers from within a target data region





Accelerator: explicit data mapping

- Relatively small number of truly shared memory accelerators so far
- Require the user to explicitly map data to and from the device memory
- Use array region

```
#pragma omp target data map(to:a) \\
  map(tofrom:b,anArray[0:64])
{
   /* a, b and anArray are mapped
     * to the device */

  /* work on the device */
#pragma omp target ...
  {
        ...
   }|
}
/* b and anArray are mapped
  * back to the host */
```

target date example

```
void vec_mult(float *p, float *v1, float *v2, int N)
{
   int i;
   init(v1, v2, N);
   #pragma omp target data map(from: p[0:N])
   {
      #pragma omp target map(to: v1[:N], v2[:N])
      #pragma omp parallel for
      for (i=0; i<N; i++)
        p[i] = v1[i] * v2[i];
      init_again(v1, v2, N);
      #pragma omp target map(to: v1[:N], v2[:N])
      #pragma omp parallel for
      for (i=0; i<N; i++)
        p[i] = p[i] + (v1[i] * v2[i]);
   }
   output(p, N);
}
```

Note mapping inheritance

Accelerator: hierarchical parallelism

- Organize massive number of threads
  – teams of threads,e.g. map to CUDA grid/block

- Distribute loops over teams

```
#pragma omp target

#pragma omp teams num_teams(2)
      num_threads(8)
{
   //-- creates a "league" of teams
   //-- only local barriers permitted
#pragma omp distribute
for (int i=0; i<N; i++) {

}
```

teams and distribute loop example

```
float dotprod_teams(float B[], float C[], int N, int num_blocks,
  int block_threads)
{
    float sum = 0;
    int i, i0;
    #pragma omp target map(to: B[0:N], C[0:N])
    #pragma omp teams num_teams(num_blocks) thread_limit(block_threads)
      reduction(+:sum)
    #pragma omp distribute
    for (i0=0; i0<N; i0 += num_blocks)
       #pragma omp parallel for reduction(+:sum)
       for (i=i0; i< min(i0+num_blocks,N); i++)
           sum += B[i] * C[i];
    return sum;
}
```

Double-nested loops are mapped to the two levels of thread hierarchy (league and team)

OpenMP SIMD Loop Construct

- Vectorize a loop nest
- Cut loop into chunks that fit a SIMD vector register
- No parallelization of the loop body
- Syntax (C/C++)

  #pragma omp simd [clause[[,] clause],...]

for-loops

- Syntax (Fortran)

  !$omp simd [clause[[,] clause],...]

  do-loops

```
void sprod(float *a, float *b, int n)
{

      float sum = 0.0f;
      #pragma omp simd reduction(+:sum)
      for (int k=0; k<n; k++)
      sum += a[k] * b[k];
      return sum;
}
```

Data Sharing Clauses

• private(var-list):
Uninitialized vectors for variables in var-list

• firstprivate(var-list):
Initialized vectors for variables in var-list

• reduction(op:var-list):
Create private variables for var-list and apply reduction operator op at the end of the construct

SIMD Loop Clauses

- safelen (length)
  - Maximum number of iterations that can run concurrently without breaking a dependence
  - in practice, maximum vector length
- linear (list[:linear-step])
  - The variable's value is in relationship with the iteration number

  x i = x orig + i * linear-step
- aligned (list[:alignment])
  - Specifies that the list items have a given alignment
  - Default is alignment for the architecture
- collapse (n)

SIMD Worksharing Construct

- Parallelize and vectorize a loop nest
  - Distribute a loop's iteration space across a thread team
  - Subdivide loop chunks to fit a SIMD vector register
- Syntax (C/C++)

```
#pragma omp for simd [clause[[,] clause],...]
for-loops
```

- Syntax (Fortran)

```
!$omp do simd [clause[[,] clause],...]
do-loops
```

SIMD Function Vectorization

- Declare one or more functions to be compiled for calls from a SIMD-parallel loop
- Syntax (C/C++):

```
#pragma omp declare simd [clause[[,] clause],...]
[#pragma omp declare simd [clause[[,] clause],...]]
```

SIMD Function Vectorization

- simdlen (length)
  - generate function to support a given vector length
- uniform (argument-list)
  - argument has a constant value between the iterations of a given loop
- inbranch
  - function always called from inside an if statement
- notinbranch
  - function never called from inside an if statement
- linear (argument-list[:linear-step])
- aligned (argument-list[:alignment])
- reduction (operator:list)

Thread Affinity in OpenMP 4.0

- OpenMP 4.0 introduces the concept of places...
  - set of threads running on one or more processors
  - can be defined by the user
  - pre-defined places available:
    - threads: one place per hyper-thread
    - cores: one place exists per physical core
    - sockets: one place per processor package

and affinity policies...

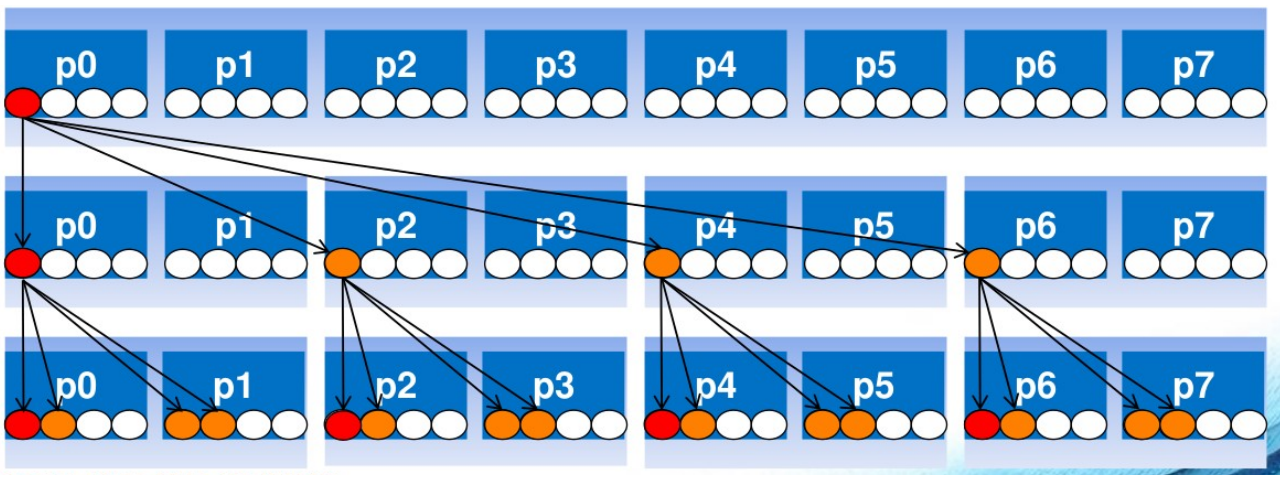
  - spread: spread OpenMP threads evenly among the places
  - close: pack OpenMP threads near master thread
  - master: collocate OpenMP thread with master thread

and means to control these settings

  - Environment variables OMP_PLACES and OMP_PROC_BIND
  - clause proc_bind for parallel regions

Thread Affinity Example

- Example
  Distribute outer region, keep inner regions close
  OMP_PLACES=cores(8); OMP_NUM_THREADS=4,4
  #pragma omp parallel proc_bind(spread)
  #pragma omp parallel proc_bind(close)



Thank You