Parallel programming for large-scale data processing

Hisham Mohamed

Viper group, CVML Laboratory, University of Geneva
December 16, 2012

First biomedical engineering workshop, Cairo, Egypt.
December 16-19, 2012

UNIVERSITÉ DE GENÈVE

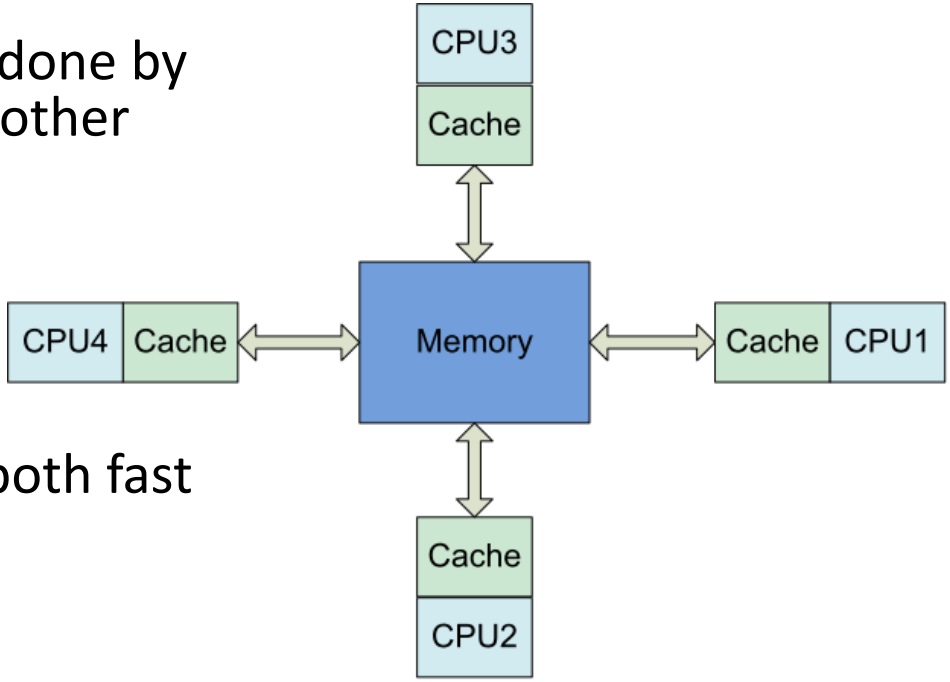
Outline

- Parallel and Distributed computing
  - Architecture
    - Shared
    - Distributed
    - Hybrid
  - Applications
    - Data parallelization
    - Parallel Algorithms
    - Hybrid Applications
- OpenMP
  - Parallel regions
  - Work sharing
  - Data scoping
  - Synchronization
- Message Passing Interface(MPI)
- Conclusion
- References

Architecture…

Architecture: Shared Memory

- Shared memory
  - Processors operate independently but share the same memory.
  - Changes in a memory location done by one processor are visible to all other processors.

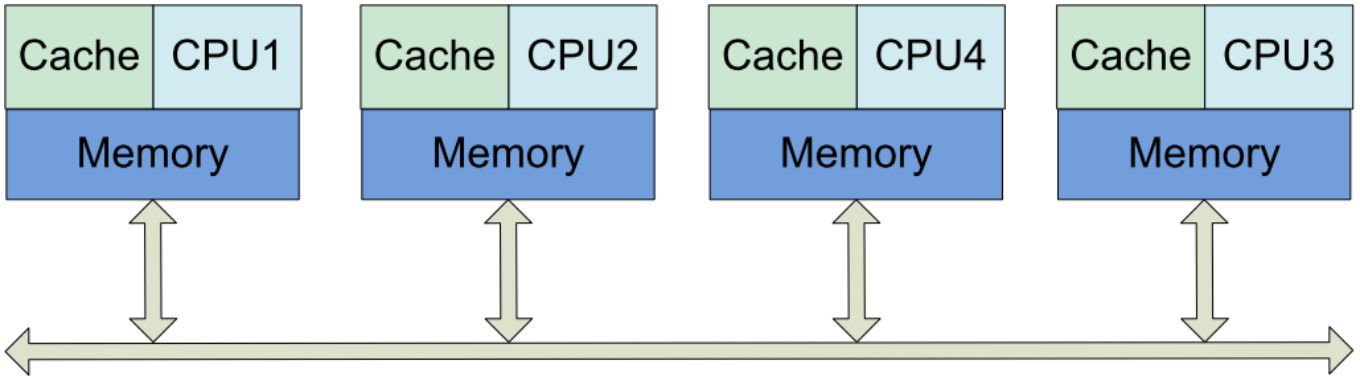


- Advantages
  - Easier in programming.
  - Data sharing between tasks is both fast and uniform.

- Disadvantages
  - Not scalable.
  - Programmer responsibility for synchronization access of global memory.

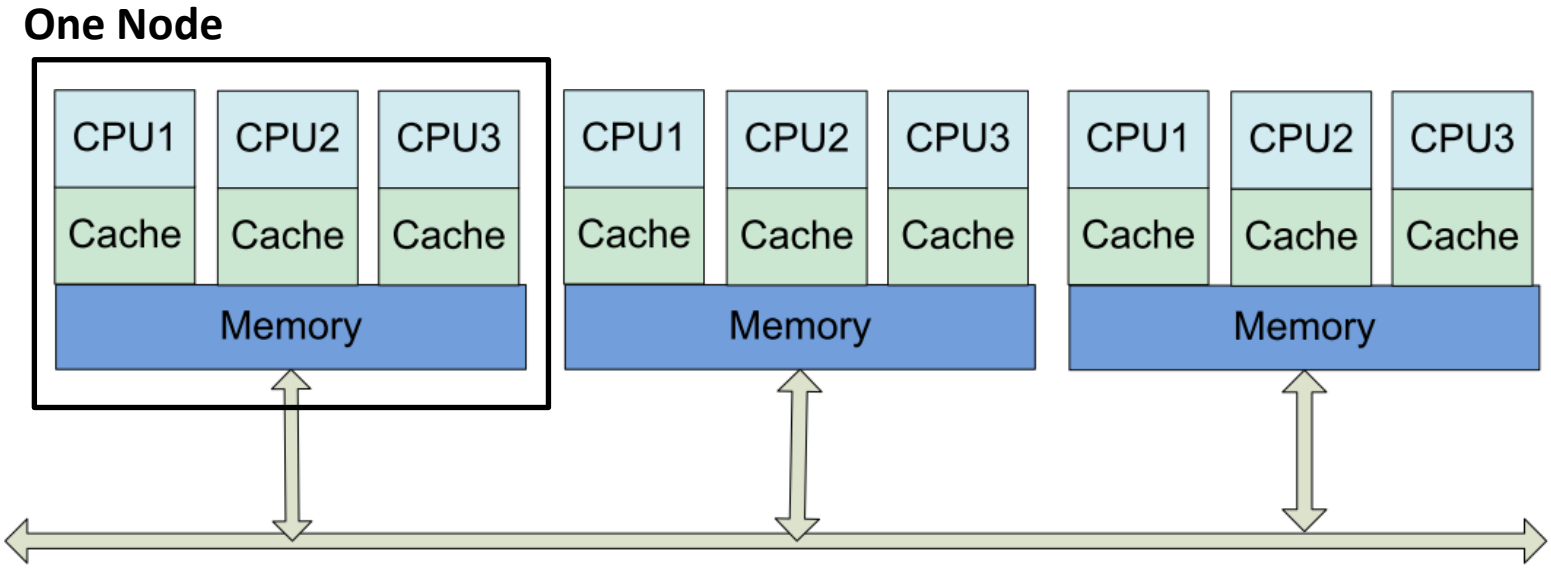
Architecture: Distributed Memory

- Distributed memory architecture
  - Nodes have their own local memory.
  - Network communication is needed to connect inter-processor and transfer data between the nodes, though it can be Ethernet.
- Advantages:
  - Scalable.
  - Each processor can rapidly access its own memory without interference other processors.
- Disadvantages:
  - The programmer is responsible for managing the communication between processors.



Architecture: Distributed-Shared Memory

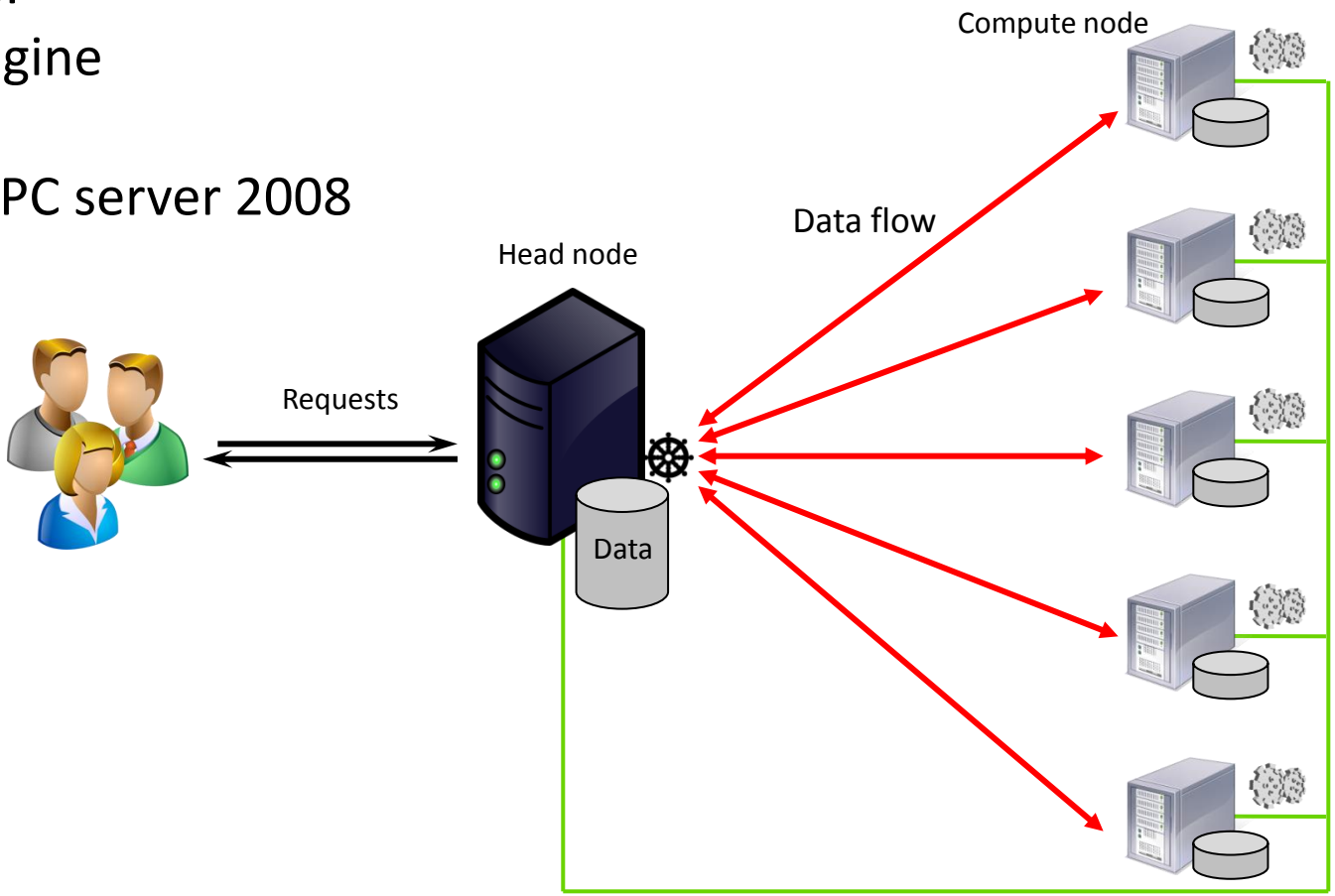
- Cluster Computing:
  - A combination of multiple nodes, which are connected together through network communication.
  - Head Node
  - Compute Node



Applications…

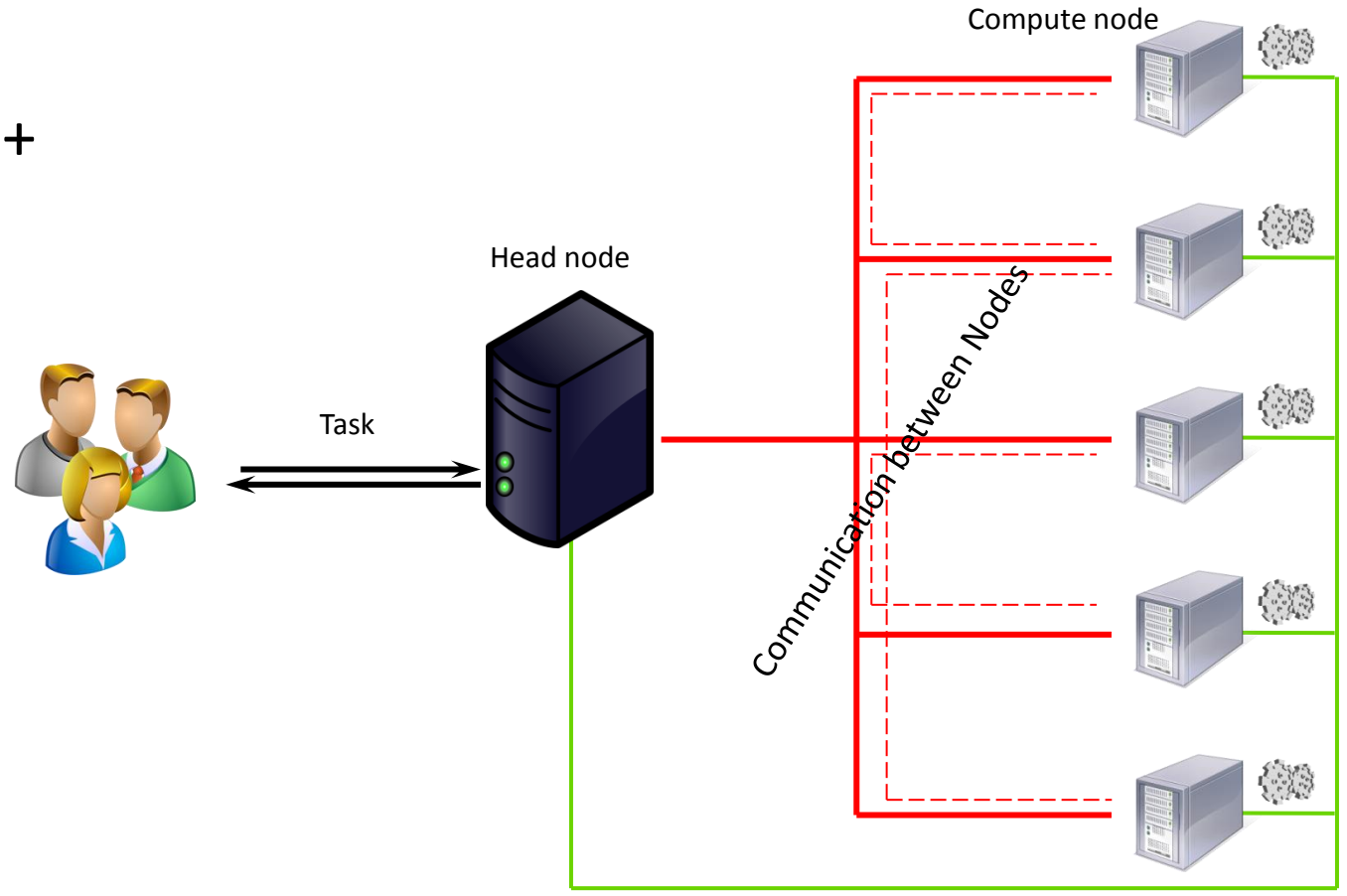
Applications: Data parallelization

- Chunk of data which is divided on multiple nodes and every node processes its part separately.
- Communication between the nodes is not needed.
- Job scheduler
  - Sun Grid Engine
  - PBS-Torque
  - Windows HPC server 2008
- Examples:
  - Google
  - Yahoo!

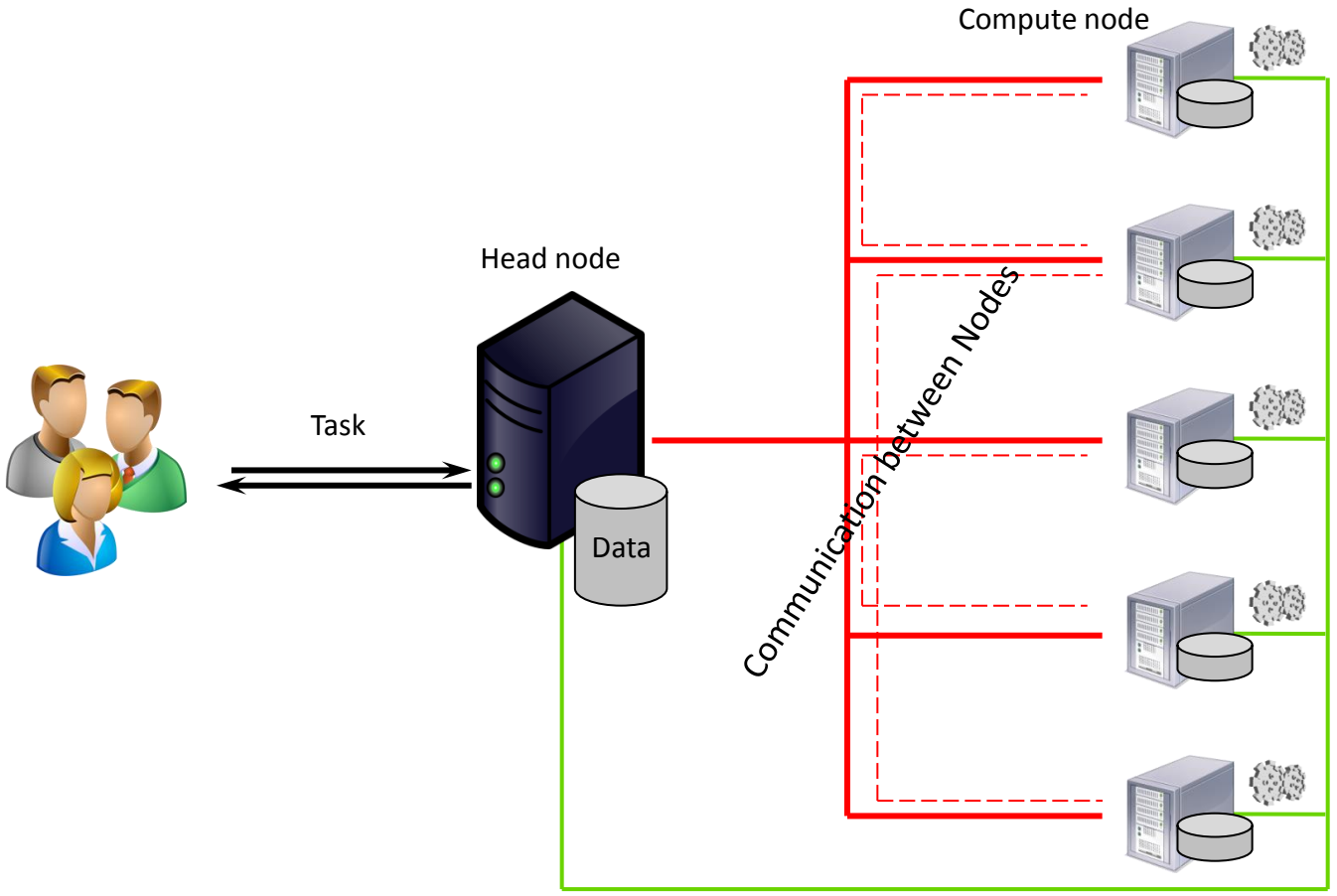


Applications: Parallel Algorithms

- All nodes work together to solve one problem.
- Tools and Libraries:
  - Message passing interface (MPI)
  - X10
  - Charm++



Applications: Data-Algorithms Parallelization



HPC Evolution (1964-2012)

Performance

20 PF.

10 PF.

7.2 TF. – 2.5 PF.

143 GF. – 2.3 TF.

250 MF.

250 MF.

136 MF.

10 MF.

3 KF.

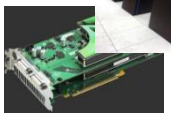
BlueGene Q

1,572,864 cores

K Computer

548,352 cores

BlueGene L/P



GPUS

Beowulf cluster

Goodyear MPP

Goodyear MPP(1983): 128x128 grid one-bit processor

ILLIAC IV

ILLIAC IV: world's fastest computer from 1975-1981, 64 processors, first SIMD

Cray-1

CDC7600

CDC7600: world's fastest computer from 1970-1974

CDC6600

CDC 6600: world's fastest computer from 1964–69

1964-1974 - 1975-1993 – 1994-1999 – 2000-2010 — 2011 — 2012 →

Outline

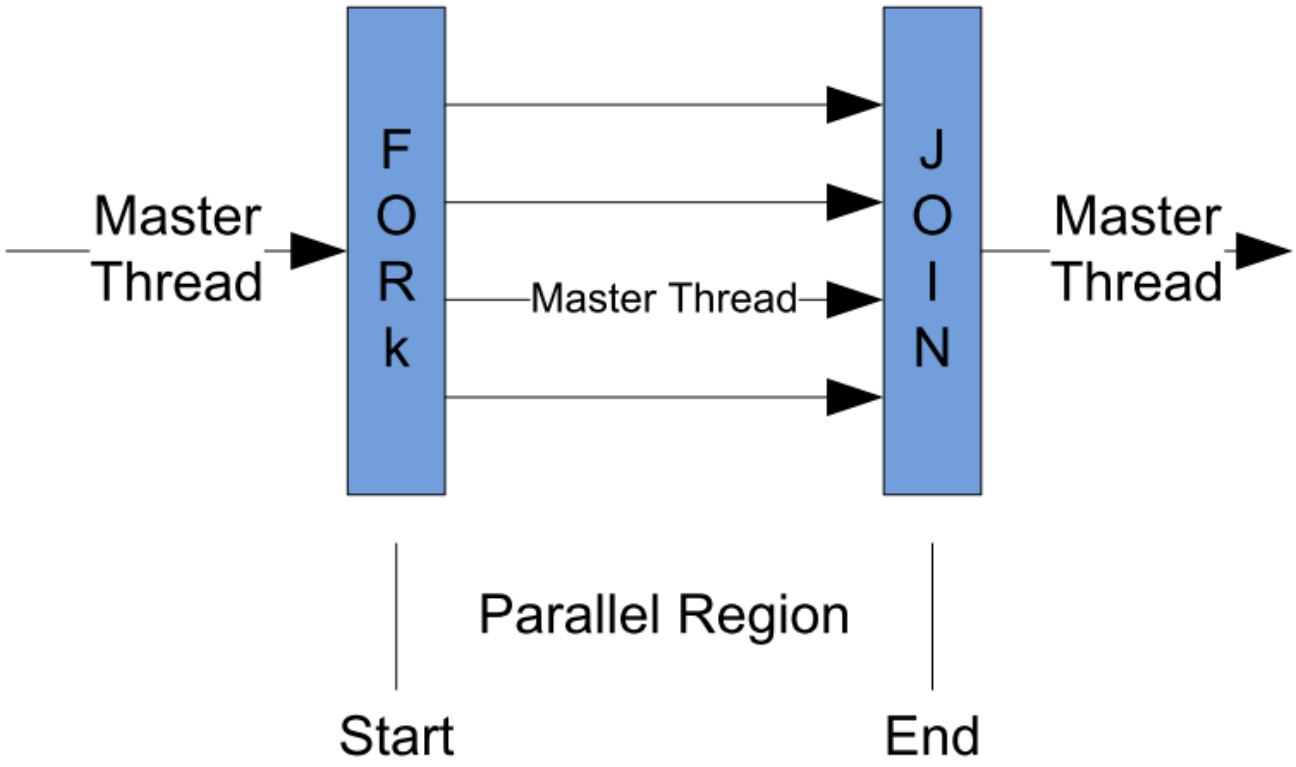
- Parallel and Distributed computing
  - Architecture
    - Shared
    - Distributed
    - Hybrid
  - Applications
    - Data parallelization
    - Parallel Algorithms
    - Hybrid Applications
- OpenMP
  - Parallel regions
  - Work sharing
  - Data scoping
  - Synchronization
- Message Passing Interface(MPI)
- Conclusion
- References

OpenMP

- Stands for : Open Multi-Processing
- API:
  - Allows shared memory parallelism.
  - Specified for C/C++ and Fortran
  - Not for distributed memory parallel systems
- Released 2005.

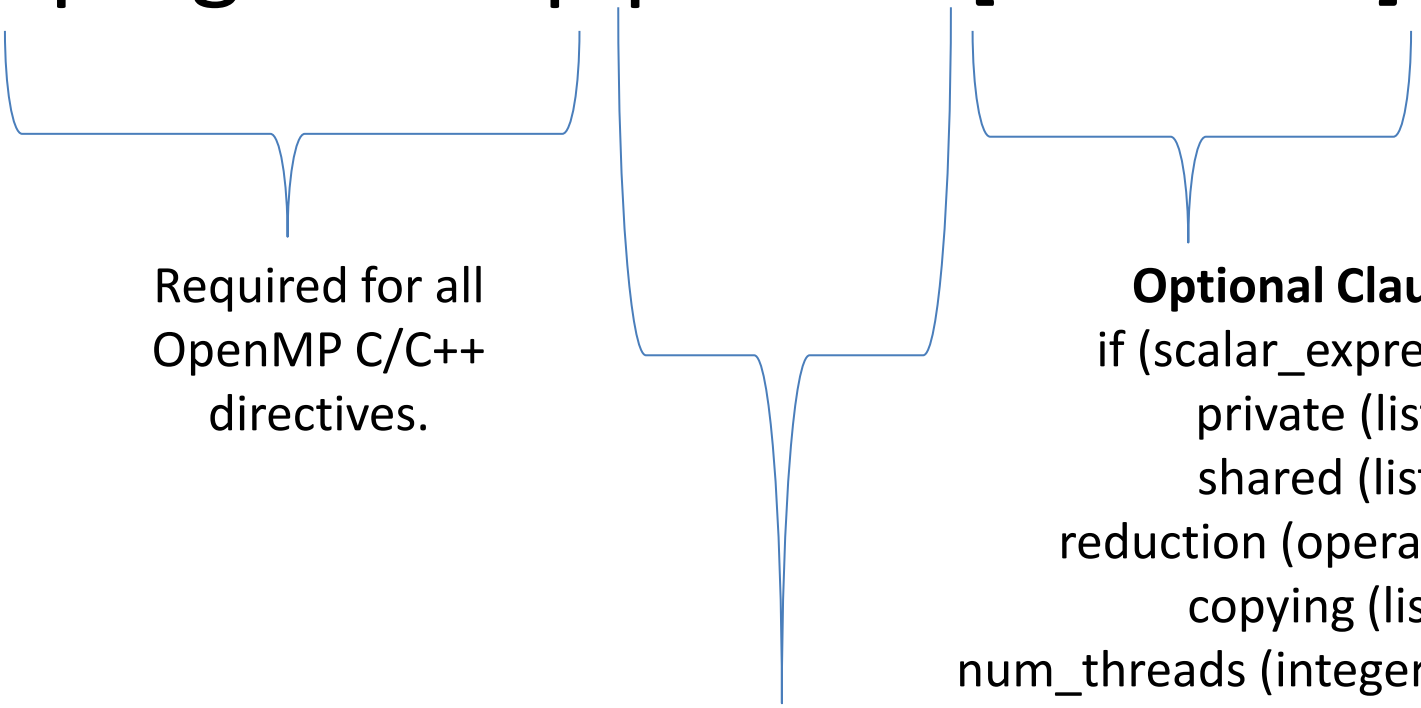
OpenMP: Programming Model

- Fork-Join model



OpenMP Directives

#pragma omp parallel [clause ...] newline



Required for all OpenMP C/C++ directives.

Optional Clauses:
if (scalar_expression)
private (list)
shared (list)
reduction (operator: list)
copying (list)
num_threads (integer-expression)

OpenMP directive. After the "**pragma**" and before any clauses.

OpenMP: Code Example

```
#include <omp.h>

main () {

        int var1, var2, var3;

        Serial code

        Beginning of parallel section. Fork a team of threads.
        Specify variable scoping

        #pragma omp parallel private(var1, var2) shared(var3)
        {
                Parallel section executed by all threads
                .
                .
                .
            All threads join master thread and disband
        }
        Resume serial code
}
```

OpenMP-Parallel regions

C / C++ - Parallel Region Example

```
#include <omp.h>

main ()  {

int nthreads, tid;

#pragma omp parallel private(tid)
    {
            tid = omp_get_thread_num();
            printf("Hello World from thread = %d\n", tid);
            if (tid == 0)
             {
                        nthreads = omp_get_num_threads();
                        printf("Number of threads = %d\n", nthreads);
             }
     }

}
```

OpenMP-Parallel regions

- Structured block: a block with one point of entry at the top and one point of exit at the bottom.
- OpenMP directives apply to structured blocks.

```
#pragma omp parallel
{
  int id = omp_get_thread_num();

more: res[id] = do_big_job (id);

  if (conv (res[id]) do_anotherjob;
}
printf ("All done\n");
```

A structured block

```
if (go_now()) goto more;
#pragma omp parallel
{
  int id = omp_get_thread_num();
more:  res[id] = do_big_job(id);
  if (conv (res[id]) goto done;
  goto more;
}
done: if (!really_done()) goto more;
```

Unstructured block

OpenMP: Work-Sharing

- Distribution of the work across number of threads.
- Divides the code into regions.
- Enclosed dynamically within a parallel region to execute in parallel.
- Work-sharing constructs must be encountered by all the threads or none of them.
- Three examples of work-sharing construct in OpenMP are:
  - ***#pragma omp for***
  - ***#pragma omp sections***
  - ***#pragma omp task***

Work-Sharing: “for” loops

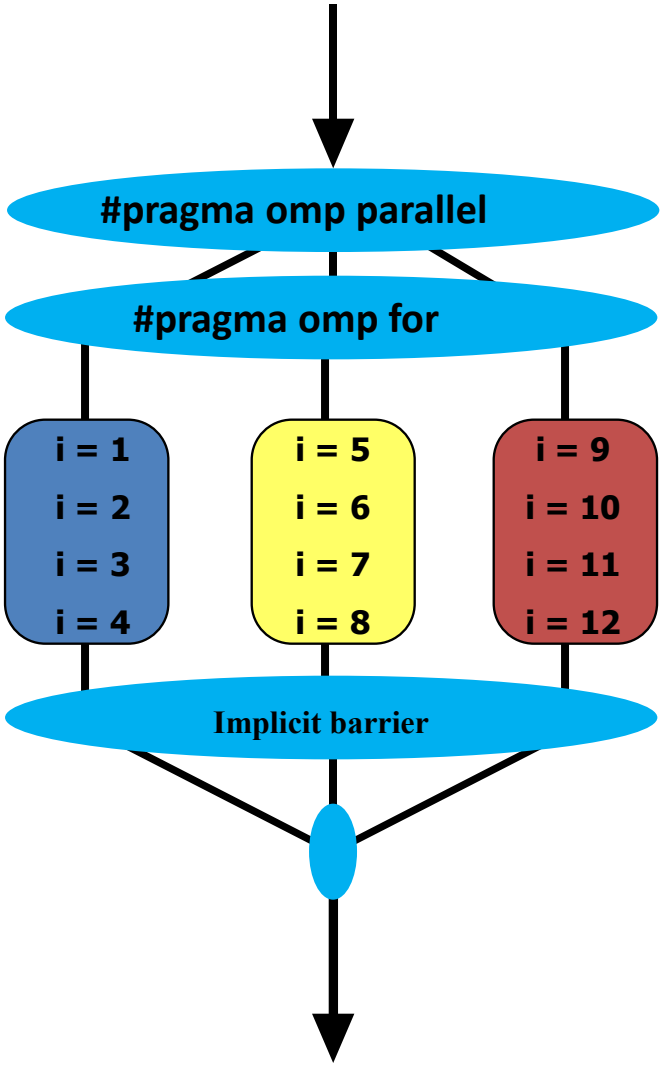
#pragma omp for *[clause ...] newline*

for_loop

Work-Sharing: "for" loops

```
// assume N=12 with 3 threads
#pragma omp parallel
#pragma omp for
  for(i = 1, i < N+1, i++)
    c[i] = a[i] + b[i];
```

- Threads are assigned to an independent set of iterations.
- Threads must wait at the end of work-sharing construct.



Work-Sharing: “for” loops

```
#pragma omp parallel
#pragma omp for
   for(i = 1, i < N+1, i++)
      c[i] = a[i] + b[i];
```

```
#pragma omp parallel for
   for(i = 1, i < N+1, i++)
      c[i] = a[i] + b[i];
```

The same!

Work-Sharing: “omp for schedule”

#pragma omp for *[clause ...] newline*

schedule *(type [,chunk]),* Ordered, private *(list),* firstprivate *(list),* lastprivate *(list),* shared *(list),* reduction *(operator: list),* collapse *(n)* ,nowait

Work-Sharing: “omp for schedule”

- The schedule clause affects how loop iterations are mapped onto threads:
  - schedule(static [,chunk])
    - Blocks of iterations of size “chunk” to threads.
    - Round robin distribution.
    - Low overhead, may cause load imbalance.
  - schedule(dynamic[,chunk])
    - Threads grab “chunk” iterations .
    - When done with iterations, thread requests next set.
    - Higher threading overhead, can reduce load imbalance.
  - schedule(guided[,chunk])
    - Dynamic schedule starting with large block .
    - Like dynamic, but chunk size is exponentially reduced.

Work-Sharing: “omp for schedule”

```
#pragma omp parallel for schedule (static, 8)
   for( int i = 0; i <= 16; i++)
   {
            c[i] = a[i] + b[i];
   }
```

Iterations are divided into chunks of 8 using 2 threads:

- Thread 0, then first chunk is **i**={0,1,2,3,4,5,6,7}
- Thread 1, then second chunk is **i**= {8,9,10,11,12,13,14,15}

```
#pragma omp parallel for schedule (dynamic, 4)
   for( int i = 0; i <= 16; i++)
   {
            c[i] = a[i] + b[i];
   }
```

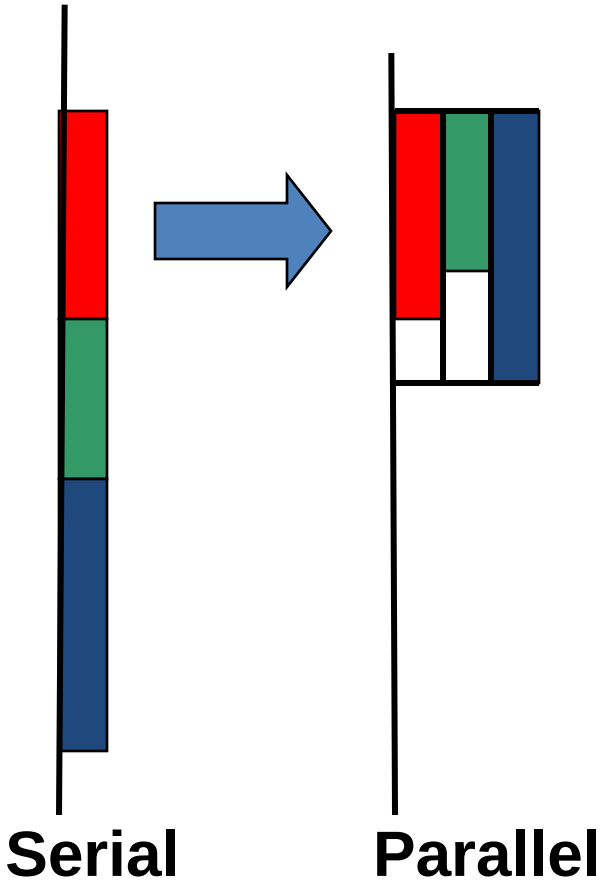
Iterations are divided into chunks of 4 using 2 threads:

- Thread 0, first chunk is **i**={0,1,2,3, 8,9,10,11,12,13,14,15}
- Thread 1, second chunk is **i**= {4,5,6,7}

Work-Sharing: Sections

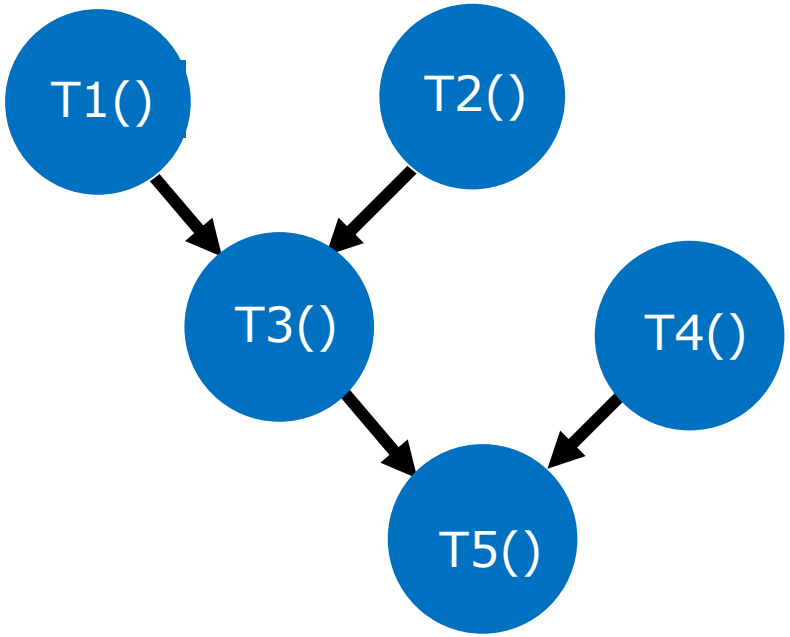
Work-Sharing: Sections

- Independent sections of code which can be executed concurrently to reduce execution time



Work-Sharing: Sections

```
a = T1();
b = T2();
s = T3(a, b);
c = T4();
printf ("%6.2f\n",T5 (s,c));
```



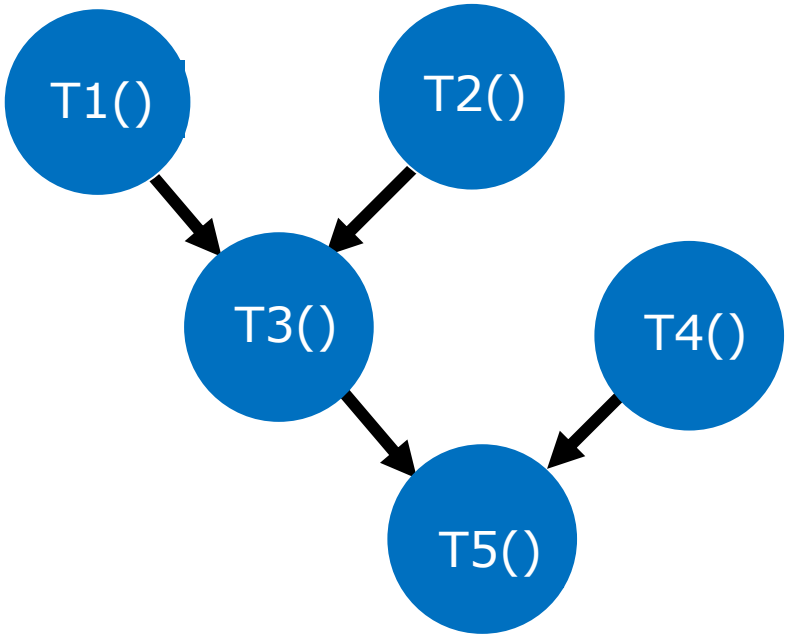
T1(),T2(), and T4() can be computed in parallel

Work-Sharing: Sections

- #pragma omp sections
  - Must be inside a parallel region
  - Precedes a code block containing of N blocks of code that may be executed concurrently by N threads
- #pragma omp section
  - Precedes each block of code within the encompassing block described above
  - Parallel execution among available threads

Work-Sharing: Sections

```
#pragma omp parallel sections
{
   #pragma omp section /* Optional */
    a = T1();
   #pragma omp section
    b = T2();
   #pragma omp section
   c = T4();
}
 s = T3(a, b);
 printf ("%6.2f\n",T5 (s,c));
```



Data scoping

Data scoping

- **Shared variable**
  - **#pragma omp parallel shared**(j,k)

- **Private variable**
  - Variables are initialized with zero.
  - **#pragma omp parallel private**(j,k)

Data scoping

- **First private-** variables are initialized according to the value of their original objects prior to entry into the parallel or work-sharing construct.
  - **#pragma omp parallel firstprivate**(j,k)

```
incr=0;
#pragma omp parallel for firstprivate(incr)
for (I=0;I<=MAX;I++) {
        if ((I%2)==0) incr++;
        A(I)=incr;
}
```

Data scoping

- **LastPrivate-** value copied back into the original variable.
  - **#pragma omp parallel lastprivate**(j,k)

```
void sq2(int n, double *lastterm)
{
  double x; int i;
  #pragma omp parallel for lastprivate(x)
  for (i = 0; i < n; i++){
     x = a[i]*a[i] + b[i]*b[i];
     b[i] = sqrt(x);
  }
  lastterm = x;
}
```

Data scoping

- Threadprivate
  - Specifies that a variable is private to a thread.
  - Legal for name-space-scope and file-scope.
  - Use copying to initialize from master thread

```
struct MyType { ~MyType(); };
MyType threaded_var;
 #pragma omp threadprivate(threaded_var)
int main()
 {
#pragma omp parallel {}
 }
```

Synchronization...

Synchronization

- **Race Condition:**
  - Two or more threads access a shared variable.

```
float dot_prod(float* a, float* b, int N)
{
  float sum = 0.0;
#pragma omp parallel for shared(sum)
   for(int i=0; i<N; i++) {
     sum += a[i] * b[i];
   }
  return sum;
}
```

Synchronization

- **Race Condition:**
  - Critical region.
  - Sometimes it causes overhead.

```
float dot_prod(float* a, float* b, int N)
{
  float sum = 0.0;
#pragma omp parallel for
   for(int i=0; i<N; i++) {
#pragma omp critical
     sum += a[i] * b[i];
   }
  return sum;
}
```

Synchronization

- **Single Construct:**
  - Denotes block of code to be executed by only one thread
    - First thread to arrive is chosen
  - Implicit barrier at end

```
#pragma omp parallel
{
  DoManyThings();
#pragma omp single
  {
    ExchangeBoundaries();
  }  // threads wait here for single
  DoManyMoreThings();
}
```

Synchronization

- **Master Construct:**
  - Denotes block of code to be executed only by the master thread
  - No implicit barrier at end

```
#pragma omp parallel
{
  DoManyThings();
#pragma omp master
  {          // if not master skip to next stmt
    ExchangeBoundaries();
  }
  DoManyMoreThings();
}
```

Synchronization

- **Barrier Construct:**
  - Explicit barrier synchronization.
  - Each thread waits until all threads arrive.

```
#pragma omp parallel shared (A, B, C)
{
        DoSomeWork(A,B);
        printf("Processed A into B\n");
#pragma omp barrier
        DoSomeWork(B,C);
        printf("Processed B into C\n");
}
```

OpenMP Library Routines

- Runtime environment routines:
  - Modify/check the number of threads
    - **`omp_[set|get]_num_threads()`**
    - **`omp_get_thread_num()`**
    - **`omp_get_max_threads()`**
  - Are we in a parallel region?
    - **`omp_in_parallel()`**
  - How many processors in the system?
    - **`omp_get_num_procs()`**

Conclusion

- Main concepts of parallel and distributed computing.
- OpenMP
  - Parallel regions
  - Work Sharing
  - Data scoping

References

- https://computing.llnl.gov/tutorials/parallel_comp/
- http://www.ece.rutgers.edu/~parashar/Classes/06-07/ece451-566/
- Intel-OpenMP course