Introduction to OpenMP Parallel Programming

Jemmy Hu

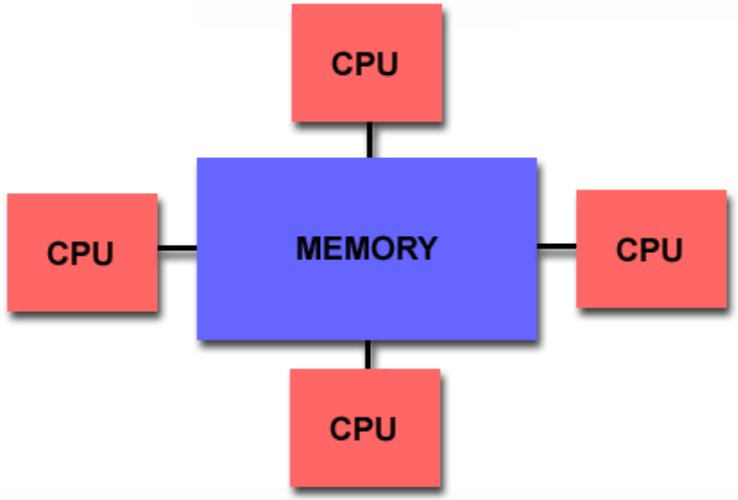
SHARCNET HPC Consultant
University of Waterloo

January 20, 2016

Contents

Parallel Computer Memory Architectures

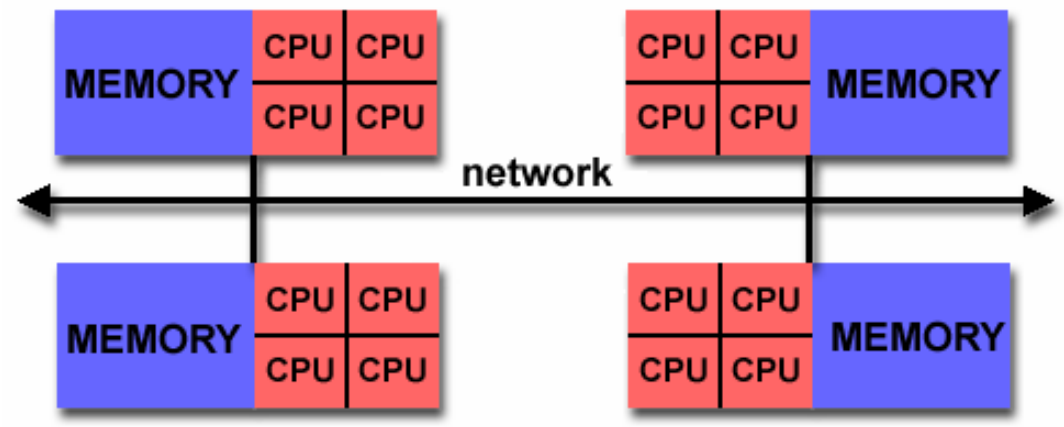
SMP (symmetric multiprocessing), Shared Memory solution



- Shared memory parallel computers vary widely, but generally have in common the ability for all processors to access all memory as global address space.
- Multiple processors can operate independently but share the same memory resources.
- Changes in a memory location effected by one processor are visible to all other processors.
- It could be a single machine (shared memory machine, workstation, desktop), or shared memory node in a cluster.

Parallel Computer Memory Architectures

Hybrid Distributed-Shared Memory (Cluster solution)



- Employ both shared and distributed memory architectures
- The shared memory component is usually a cache coherent SMP machine. Processors on a given SMP can address that machine's memory as global.
- The distributed memory component is the networking of multiple SMPs. SMPs know only about their own memory - not the memory on another SMP. Therefore, network communications are required to move data from one SMP to another.
- Current trends seem to indicate that this type of memory architecture will continue to prevail and increase at the high end of computing for the foreseeable future.

Parallel Computing: What is it?

- **Parallel computing is when a program uses concurrency to either:**

  - decrease the runtime for the solution to a problem.

  - increase the size of the problem that can be solved.

Gives you more performance to throw at your problems.

- **Parallel programming is not generally trivial, 3 aspects:**
  - specifying parallel execution
  - communicating between multiple procs/threads
  - synchronization

  tools for automated parallelism are either highly specialized or absent

- **Many issues need to be considered, many of which don't have an analog in serial computing**
  - data vs. task parallelism
  - problem structure
  - parallel granularity

Distributed vs. Shared memory model

- **Distributed memory systems**
  - For processors to share data, the programmer must explicitly arrange for communication -**"Message Passing"**
  - Message passing libraries:
    - MPI ("Message Passing Interface")
    - PVM ("Parallel Virtual Machine")

- **Shared memory systems**
  - Compiler directives (OpenMP)
  - "Thread" based programming (pthread, …)

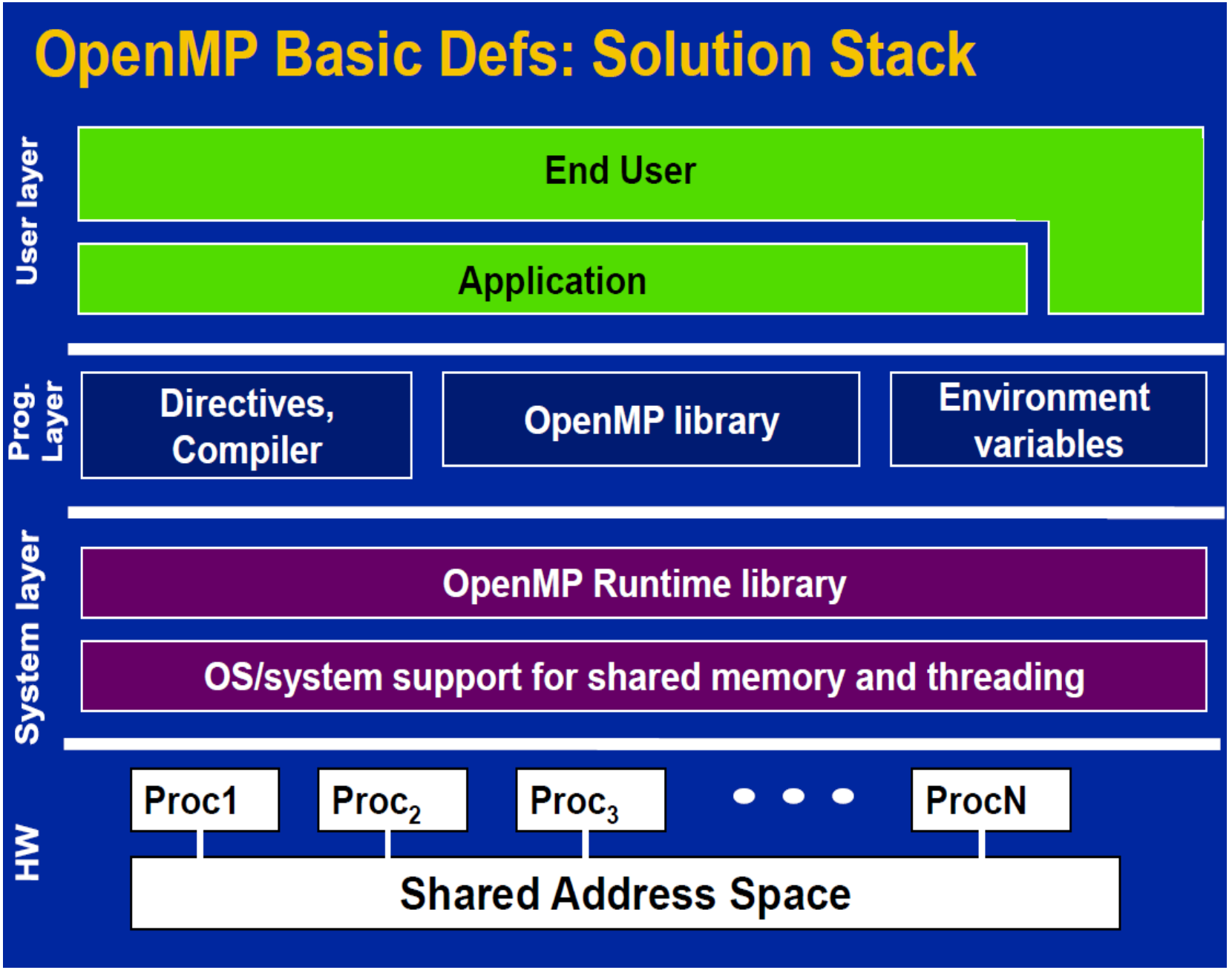
OpenMP Concepts: What is it?

- An Application Program Interface (API) that may be used to explicitly direct **multi-threaded, shared memory parallelism**
- Using **compiler directives**, **library routines** and **environment variables** to automatically generate threaded (or multi-process) code that can run in a concurrent or parallel environment.
- **Portable:**
  - The API is specified for C/C++ and Fortran
  - Multiple platforms have been implemented including most Unix/Linux platforms and Windows NT
- **Standardized:** Jointly defined and endorsed by a group of major computer hardware and software vendors
- **What does OpenMP stand for?**

  Open Specifications for Multi Processing

  http://openmp.org/wp/

OpenMP Basic Defs: Solution Stack



OpenMP: Benefits

- **Standardization:**
  Provide a standard among a variety of shared memory architectures/platforms
- **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.
- **Ease of Use:**
  Provide capability to incrementally parallelize a serial program, unlike message-passing libraries which typically require an all or nothing approach
- **Portability:**
  Supports Fortran (77, 90, and 95), C, and C++
  Public forum for API and membership

OpenMP: hello_omp.c

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

int main (int argc, char *argv[]) {
    int id, nthreads;
    #pragma omp parallel private(id)
    {
            id = omp_get_thread_num();
            printf("Hello World from thread %d\n", id);
            #pragma omp barrier
            if ( id == 0 ) {
                    nthreads = omp_get_num_threads();
                    printf("There are %d threads\n",nthreads);
            }
    }
    return 0;
}
```

Header file

Parallel region directive

Runtime library routines

synchronization

Data types: private vs. shared

OpenMP code structure: C/C++ syntax

```
#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
  . . .
}
```

#pragma omp	directive-name	[clause, ...]	newline
#pragma omp	parallel	private(var1, var2) shared(var3)	

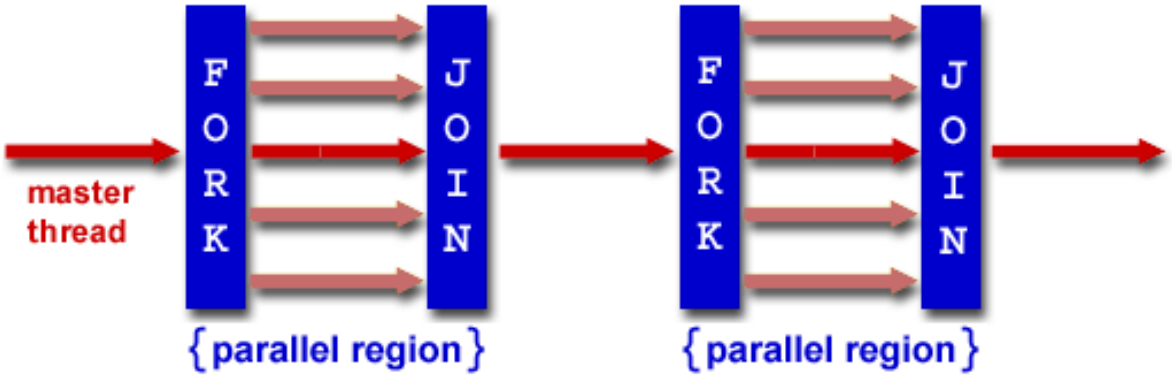
OpenMP code structure: Fortran

```
PROGRAM HELLO
 INTEGER VAR1, VAR2, VAR3
 Serial code . . .
 Beginning of parallel section. Fork a team of threads.
Specify variable scoping
!$OMP PARALLEL PRIVATE(VAR1, VAR2) SHARED(VAR3)
   Parallel section executed by all threads
   . . .
   All threads join master thread and disband
!$OMP END PARALLEL
   Resume serial code
   . . .
   END
```

sentinel	directive-name	[clause ...]
!$OMP	PARALLEL	PRIVATE(VAR1, VAR2) SHARED(VAR3)

OpenMP: Fork-Join Model

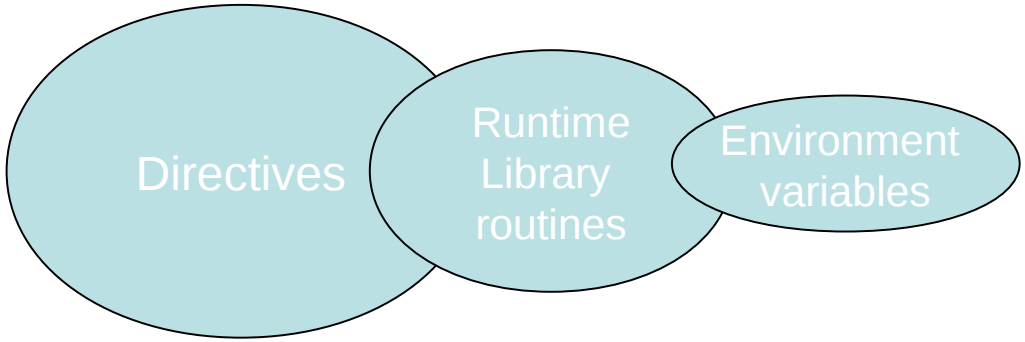
- OpenMP uses the fork-join model of parallel execution:



FORK: the master thread then creates a ***team*** of parallel threads
The statements in the program that are enclosed by the parallel region construct are then executed in parallel among the various team threads

JOIN: When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread

OpenMP Components



Directives

- Parallel regions
- Work sharing
- Synchronization
- Data scope attributes
  - private
  - firstprivate
  - lastprivate
  - shared
  - reduction
- Orphaning

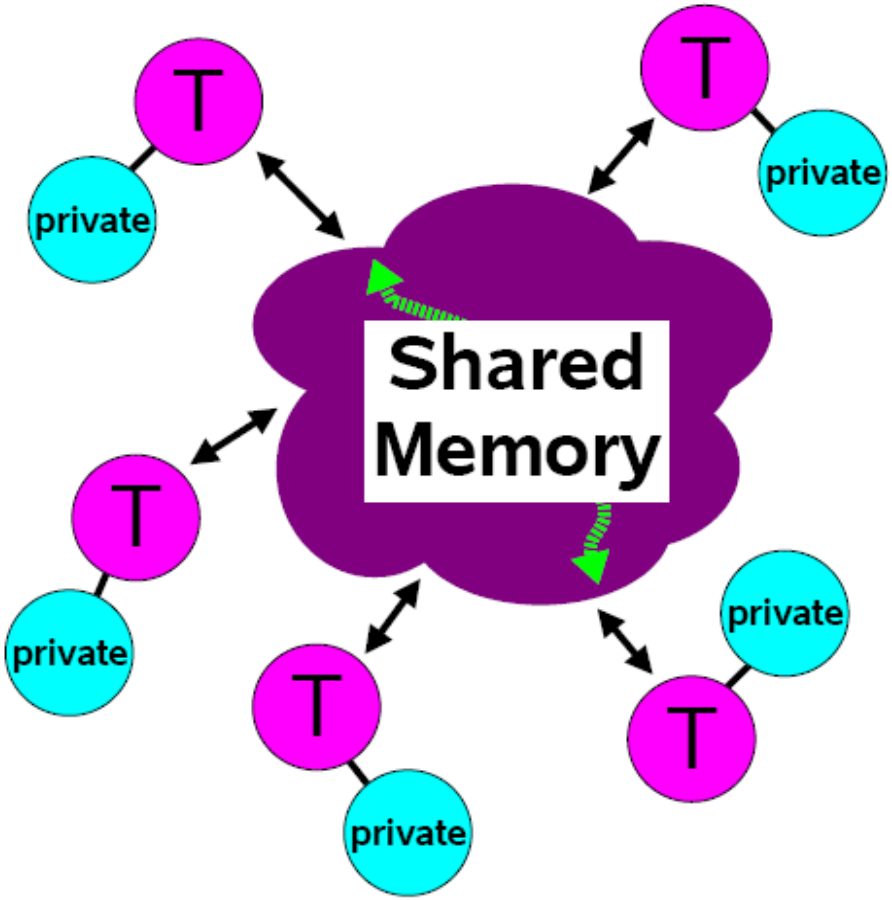
Environment variables

- Number of threads
- Scheduling type
- Dynamic thread adjustment
- Nested parallelism

Runtime environment

- Number of threads
- Thread ID
- Dynamic thread adjustment
- Nested parallelism
- Timers
- API for locking

Shared Memory Model



Programming Model

- ✔ *All threads have access to the same, globally shared, memory*
- ✔ *Data can be shared or private*
- ✔ *Shared data is accessible by all threads*
- ✔ *Private data can be accessed only by the threads that owns it*
- ✔ *Data transfer is transparent to the programmer*
- ✔ *Synchronization takes place, but it is mostly implicit*

OpenMP Directives

OpenMP: Contents

- OpenMP's constructs fall into 5 categories:
  - Parallel Regions
  - Worksharing
  - Data Environment
  - Synchronization
  - Runtime functions/environment variables
- OpenMP is basically the same between Fortran and C/C++

Parallel Region review

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

int main (int argc, char *argv[]) {
    int id, nthreads;
    #pragma omp parallel private(id)
    {
            id = omp_get_thread_num();
            printf("Hello World from thread %d\n", id);
            #pragma omp barrier
            if ( id == 0 ) {
                    nthreads = omp_get_num_threads();
                    printf("There are %d threads\n",nthreads);
            }
    }
    return 0;
}
```

Every thread executes all code enclosed in the parallel region

OpenMP library routines are used to obtain thread identifiers and total number of threads

OpenMP: Work-sharing constructs:

- A work-sharing construct divides the execution of the enclosed code region among the members of the team that encounter it.
- Work-sharing constructs do not launch new threads
- There is no implied barrier upon entry to a work-sharing construct, however there is an implied barrier at the end of a work sharing construct.

A motivating example

Sequential code

```
for(i=0;I<N;i++)   { a[i] = a[i] + b[i];}
```

OpenMP parallel region

```
#pragma omp parallel
{
        int id, i, Nthrds, istart, iend;
        id = omp_get_thread_num();
        Nthrds = omp_get_num_threads();
        istart = id * N / Nthrds;
        iend = (id+1) * N / Nthrds;
        for(i=istart;I<iend;i++)   { a[i] = a[i] + b[i];}
}
```

OpenMP parallel region and a work-sharing for-construct

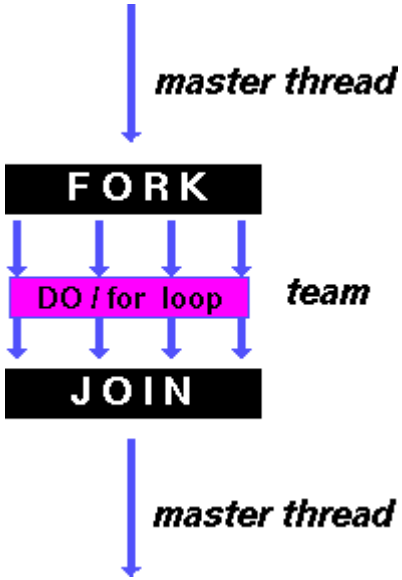
```
#pragma omp parallel
#pragma omp for schedule(static)
        for(i=0;I<N;i++)   { a[i] = a[i] + b[i];}
```

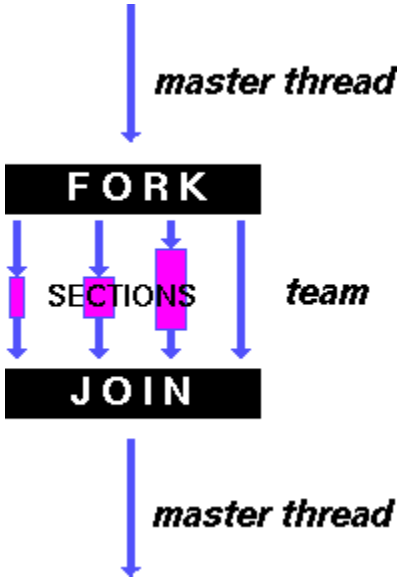
Types of Work-Sharing Constructs:

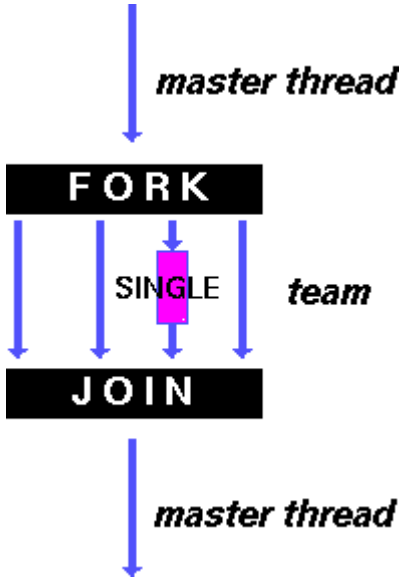
DO / for - shares iterations of a loop across the team. Represents a type of "data parallelism".

SECTIONS - breaks work into separate, discrete sections. Each section is executed by a thread. Can be used to implement a type of "functional parallelism".

SINGLE - serializes a section of code







Work-sharing constructs: Loop construct

- The DO / for directive specifies that the iterations of the loop immediately following it must be executed in parallel by the team. This assumes a parallel region has already been initiated, otherwise it executes in serial on a single processor.

```
#pragma omp parallel
#pragma omp for
        for (I=0;I<N;I++){
                NEAT_STUFF(I);
        }
```

```
!$omp parallel
!$omp do
                do-loop
!$omp end do
!$omp parallel
```

SECTIONS Directive: Functional/Task parallelism

Purpose:

1. The SECTIONS directive is a non-iterative work-sharing construct. It specifies that the enclosed section(s) of code are to be divided among the threads in the team.

2. Independent SECTION directives are nested within a SECTIONS directive. Each SECTION is executed once by a thread in the team. Different sections may be executed by different threads. It is possible that for a thread to execute more than one section if it is quick enough and the implementation permits such.

Examples: 3-loops

Serial code with three independent tasks, namely, three do loops.
each operating on a different array using different loop counters and temporary scalar variables.

```
program compute
  implicit none
  integer, parameter :: NX = 10000000
  integer, parameter :: NY = 20000000
  integer, parameter :: NZ = 30000000
  real :: x(NX)
  real :: y(NY)
  real :: z(NZ)
  integer :: i, j, k
  real :: ri, rj, rk
  write(*,*) "start"
  do i = 1, NX
      ri = real(i)
     x(i) = atan(ri)/ri
  end do
  do j = 1, NY
      rj = real(j)
      y(j) = cos(rj)/rj
  end do
  do k = 1, NZ
      rk = real(k)
     z(k) = log10(rk)/rk
  end do
  write(*,*) "end"
end program
```

Examples: 3-loops

Instead of hard-coding, we can use OpenMP provides task sharing directives (section) to achieve the same goal.

```
program compute
……

write(*,*) "start"
!$omp parallel
   !$omp sections
      !$omp section
         do i = 1, NX
            ri = real(i)
           x(i) = atan(ri)/ri
         end do
       !$omp section
          do j = 1, NY
             rj = real(j)
            y(j) = cos(rj)/rj
          end do
       !$omp section
           do k = 1, NZ
              rk = real(k)
             z(k) = log10(rk)/rk
           end do
     !$omp end sections
  !$omp end parallel
write(*,*) "end"
end program
```

Data Scope Clauses

- **SHARED (list)**
- **PRIVATE (list)**
- **FIRSTPRIVATE (list)**
- **LASTPRIVATE (list)**
- **DEFAULT (list)**
- **THREADPRIVATE (list)**
- **COPYIN (list)**
- **REDUCTION (operator | intrinsic : list)**

OpenMP: Synchronization

OpenMP has the following constructs to support synchronization:

– atomic
– critical section
– barrier
– flush
– ordered
– single
– master

Example: Calculating π

- **Numerical integration**

$$\int_0^1 \frac{4}{1+x^2}\,dx = \pi$$

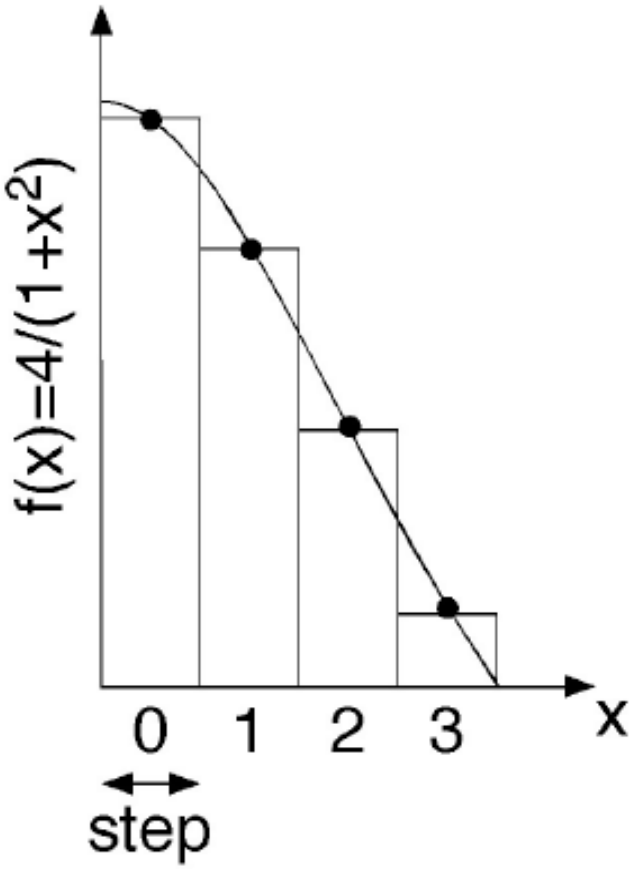
- **Discretization:**

$\Delta = 1/N$: `step = 1/NBIN`

$x_i = (i+0.5)\Delta$ $(i = 0,\ldots,N\text{-}1)$

$$\sum_{i=0}^{N-1} \frac{4}{1+x_i^2}\Delta \cong \pi$$

```
#include <stdio.h>
#define NBIN 100000
void main() {
  int i; double step,x,sum=0.0,pi;
  step = 1.0/NBIN;
  for (i=0; i<NBIN; i++) {
    x = (i+0.5)*step;
    sum += 4.0/(1.0+x*x);}
  pi = sum*step;
  printf("PI = %f\n",pi);
}
```



Parallel Loop, reduction clause

```
#include <stdio.h>
#include <omp.h>
#define NBIN 100000
int main(int argc, char *argv[ ]) {
    int I, nthreads;
    double x, pi;
    double sum = 0.0;
    double step = 1.0/(double) NUM_STEPS;

    /* do computation -- using all available threads */
    #pragma omp parallel
    {
        #pragma omp master
        {
            nthreads = omp_get_num_threads();
        }
        #pragma omp for private(x) reduction(+:sum) schedule(runtime)
        for (i=0; i < NUM_STEPS; ++i) {
            x = (i+0.5)*step;
            sum = sum + 4.0/(1.0+x*x);
        }
        #pragma omp master
        {
            pi = step * sum;
        }
    }
    printf("PI = %f\n",pi);
}
```

OpenMP: compiler

- **Compilers:**
  Intel (icc, ifort): -openmp
  GNU (gcc, g++, gfortran), -fopenmp
  Pathscale (pathcc, pathf90), -openmp
  PGI (pgcc, pgf77, pgf90), -mp

- **compile with –openmp flag with default**

  icc –openmp –o hello_openmp hello_openmp.c
  gcc –fopenmp –o hello_openmp hello_openmp.c

  ifort –openmp –o hello_openmp hello_openmp.f
  gfortran –fopenmp –o hello_openmp hello_openmp.f

OpenMP on SHARCNET

- SHARCNET systems

  https://www.sharcnet.ca/my/systems

  All systems allow for SMP- based parallel programming (i.e., OpenMP) applications,

  but the size (number of threads) per job differ from cluster to cluster (depends on the number of cores per node on the cluster)

Size of OpenMP Jobs on specific system

https://www.sharcnet.ca/my/systems

System	Nodes	CPU/Node	OMP_NUM_THREADS (max, use a full node)
orca	Opteron	16 or 24	16 or 24
saw	Xeon	8	8
redfin	Opteron	24	24
brown	Xeon	8	8

Job uses less than the max threads available on a cluster will be preferable

OpenMP: compile and run at Sharcnet

- **Compiler flags:**

Intel (icc, ifort) -openmp

GNU(gcc gfortran, >4.1) -fopenmp

SHARCNET compile: cc, CC, f77, f90

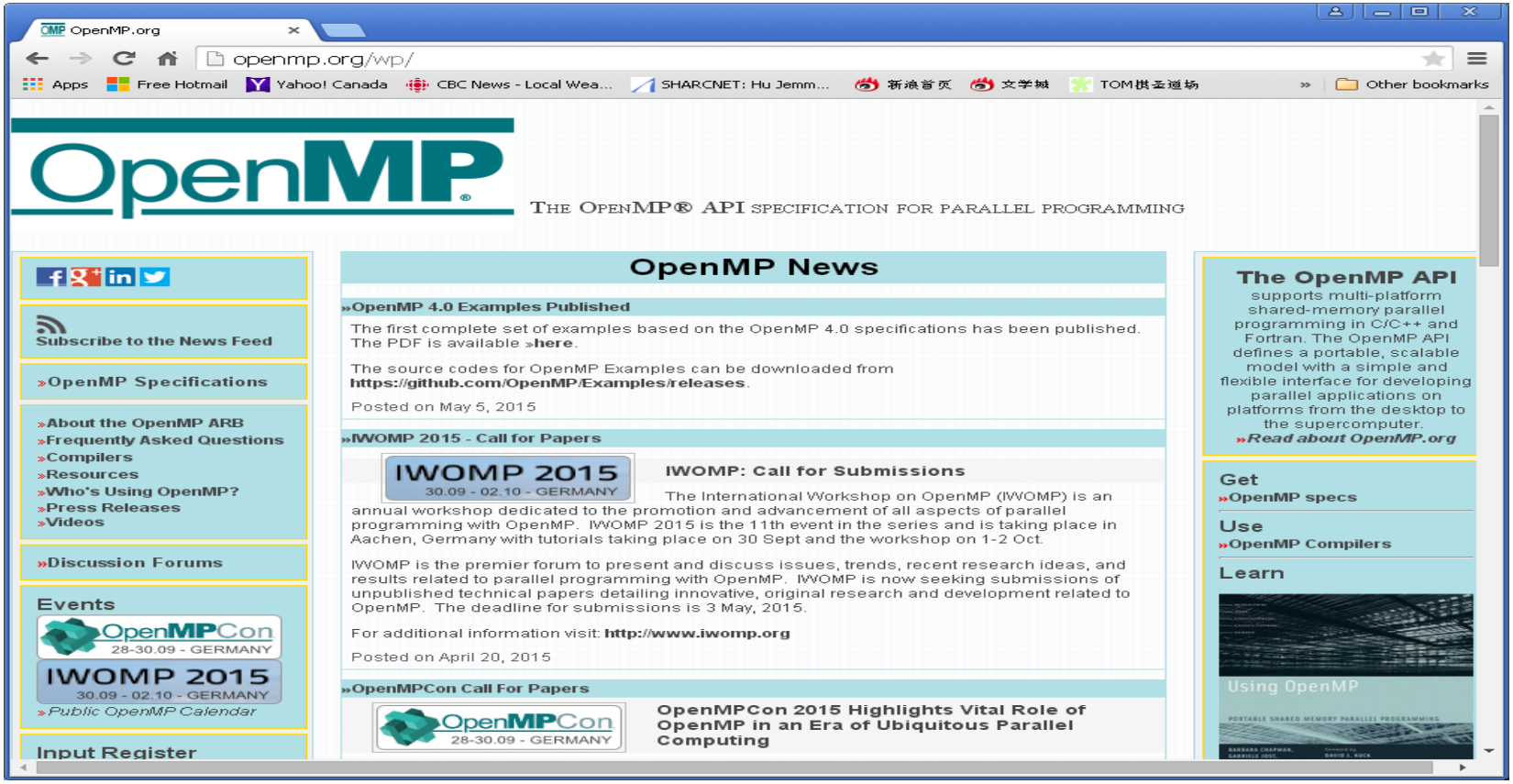
e.g., f90 –openmp –o hello_openmp hello_openmp.f

- **Run OpenMP jobs in the threaded queue**

Submit OpenMP job on a cluster with 4-cpu nodes
(The size of threaded jobs varies on different systems as discussed in previous page)

sqsub –q threaded –n 4 –r 1.0h –o hello_openmp.log ./hello_openmp

OpenMP 4.0



https://terboven.files.wordpress.com/2012/09/parallel2013-openmp-4-0.pdf

OpenMP 4 – What's New?
https://support.scinet.utoronto.ca/wiki/images/9/9b/Ds-openmp.pdf

References

1. OpenMP specifications for C/C++ and Fortran, http://www.openmp.org/

2. OpenMP tutorial:
http://openmp.org/mp-documents/Intro_To_OpenMP_Mattson.pdf

3. Parallel Programming in OpenMP by Rohit Chandra, Morgan Kaufman Publishers, ISBN 1-55860-671-8