Introduction to Shared Memory Computing using OpenMP

Pascal Paschos Ph.D.
Sr. HPC Specialist,
Research Computing Services

Northwestern
INFORMATION TECHNOLOGY

In this workshop

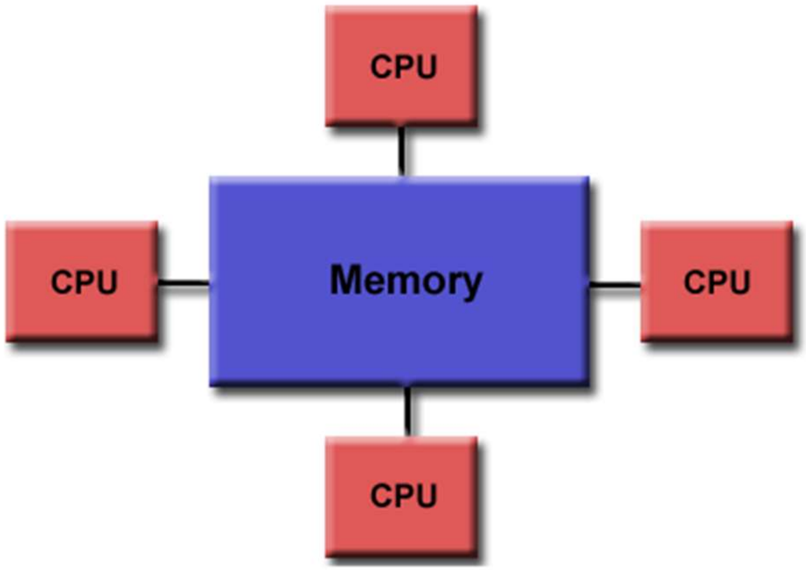
1. We will learn about elements shared memory architecture and programming
2. Introduce the OpenMP API and it’s core components
3. We will use the C-language as the tool to look into code snippets and examples

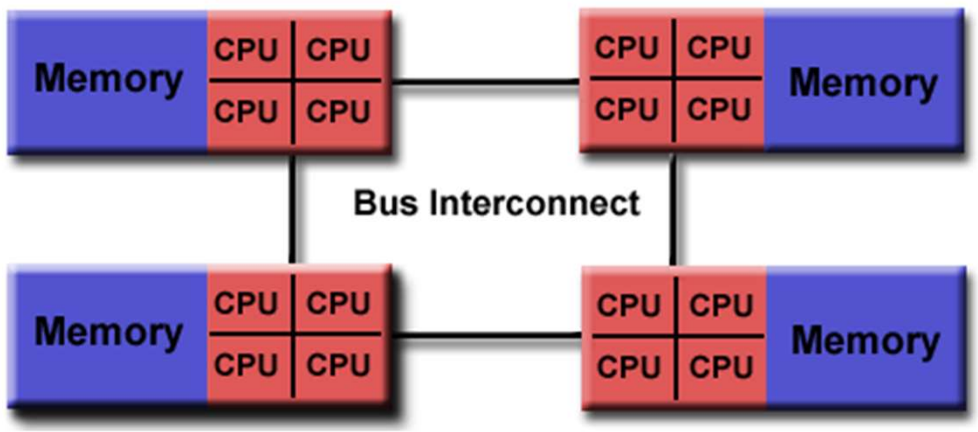
The presentation aims

1. To initiate novice programmers to threaded parallelism
2. To explain the initial means to parallelize serial code
3. Compile, build & run an openmp executable

Shared Memory Architecture

- Symmetric multi-processor (SMP) -uniform memory access
  - Shared memory address space with equal access by all CPUs
- NUMA (non-uniform memory access) multi-processor
  - Shared memory address space but time to access can vary by location -Near & Far memory





Uniform Memory Access

Non-Uniform Memory Access

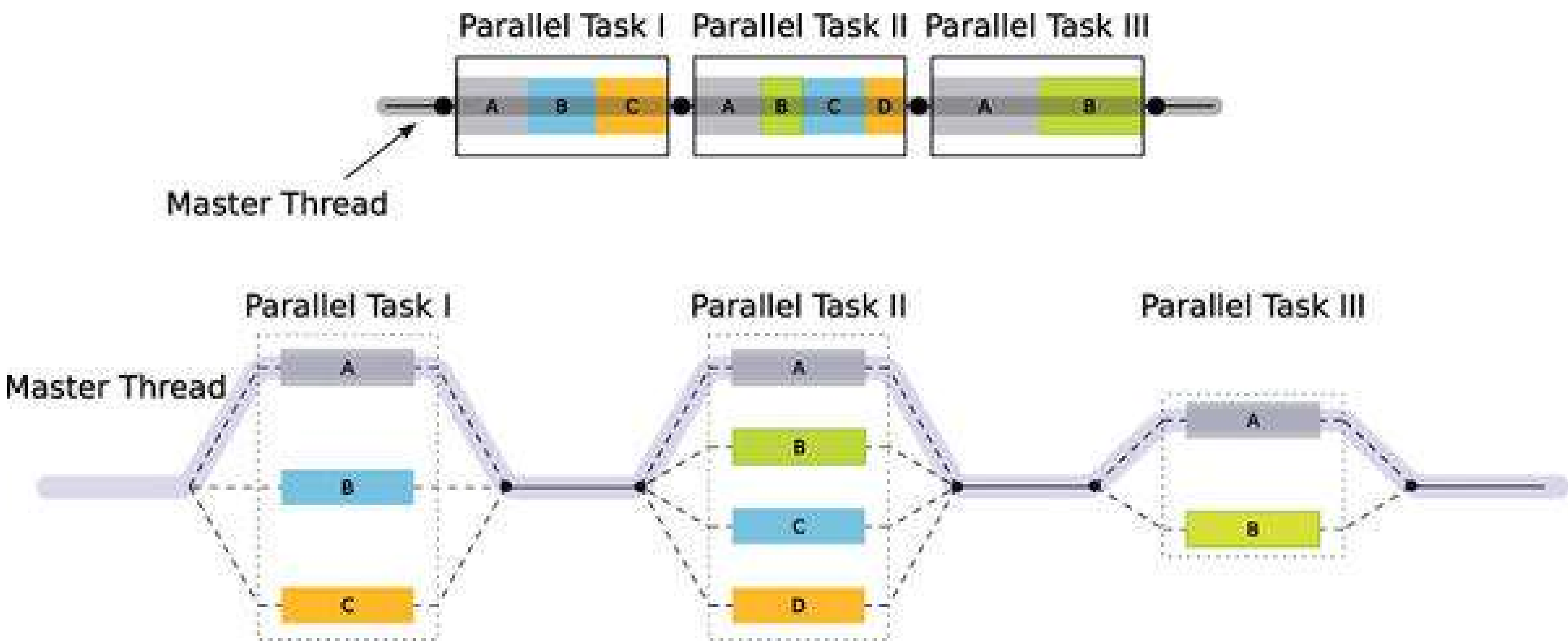
Shared Memory Programming

- An instance of a program execution is a single main (heavy) process
- Threads are lightweight processes that are spawned by the main process
- All threads share the state of the main process
- Threads 'communicate'/interact via reads & writes to the shared memory address space

OpenMP Overview

- OpenMP is an API that supports multithreaded, shared memory parallelization
- Simple to use
- Minimally invasive to parallelize serial program
- Cross platform
- Supports Fortran, C/C++

Fork/join parallelism



Invoking OpenMP extensions at compilation

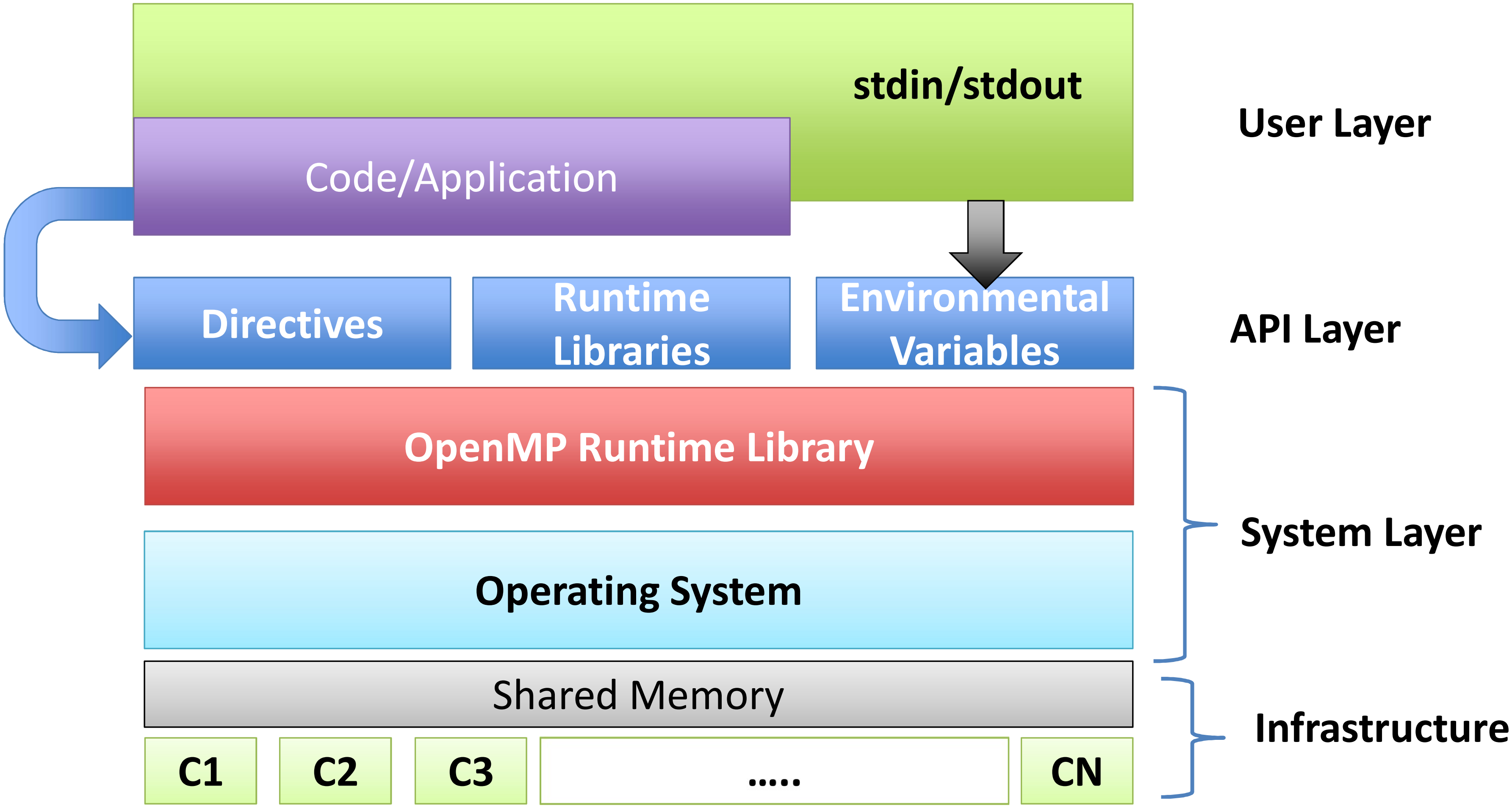
- GNU: gcc -fopenmp <sourcecode>
- Intel: icc -openmp <sourcecode>
- Must #include <omp.h> header file in C/C++ codes

- In a Makefile typically insert in:
- CFLAGS="$CFLAGS -fopenmp"

Components of OpenMP API

- Compiler Directives
- Runtime Library Routines
- Environmental Variables

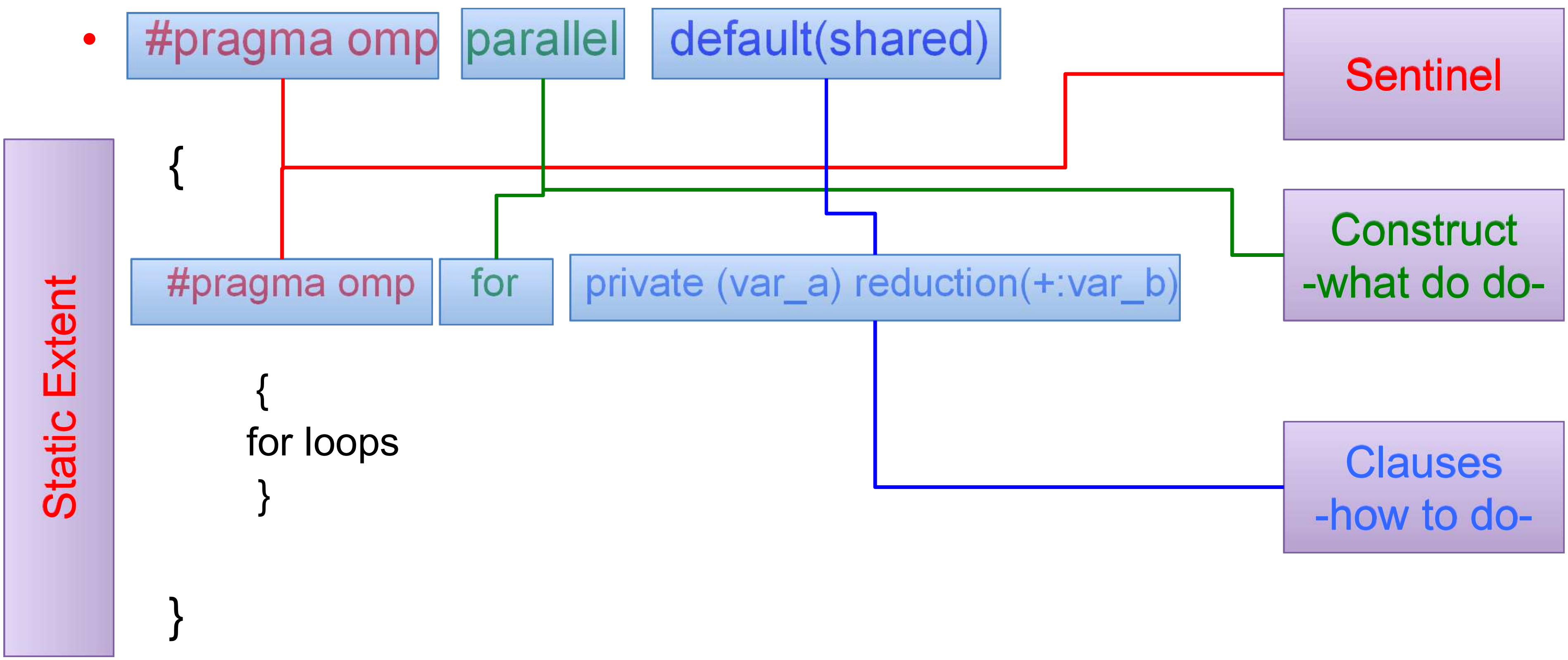
OpenMP computing stack



Compiler Directives

- Define parallel regions
- Use constructs for:
  - Work distribution (loops, blocks of code) to threads
  - Synchronization of work among threads
  - Serialization when imperative
- Use clauses to provide tuning and customization

Generic Compiler Directive Structure



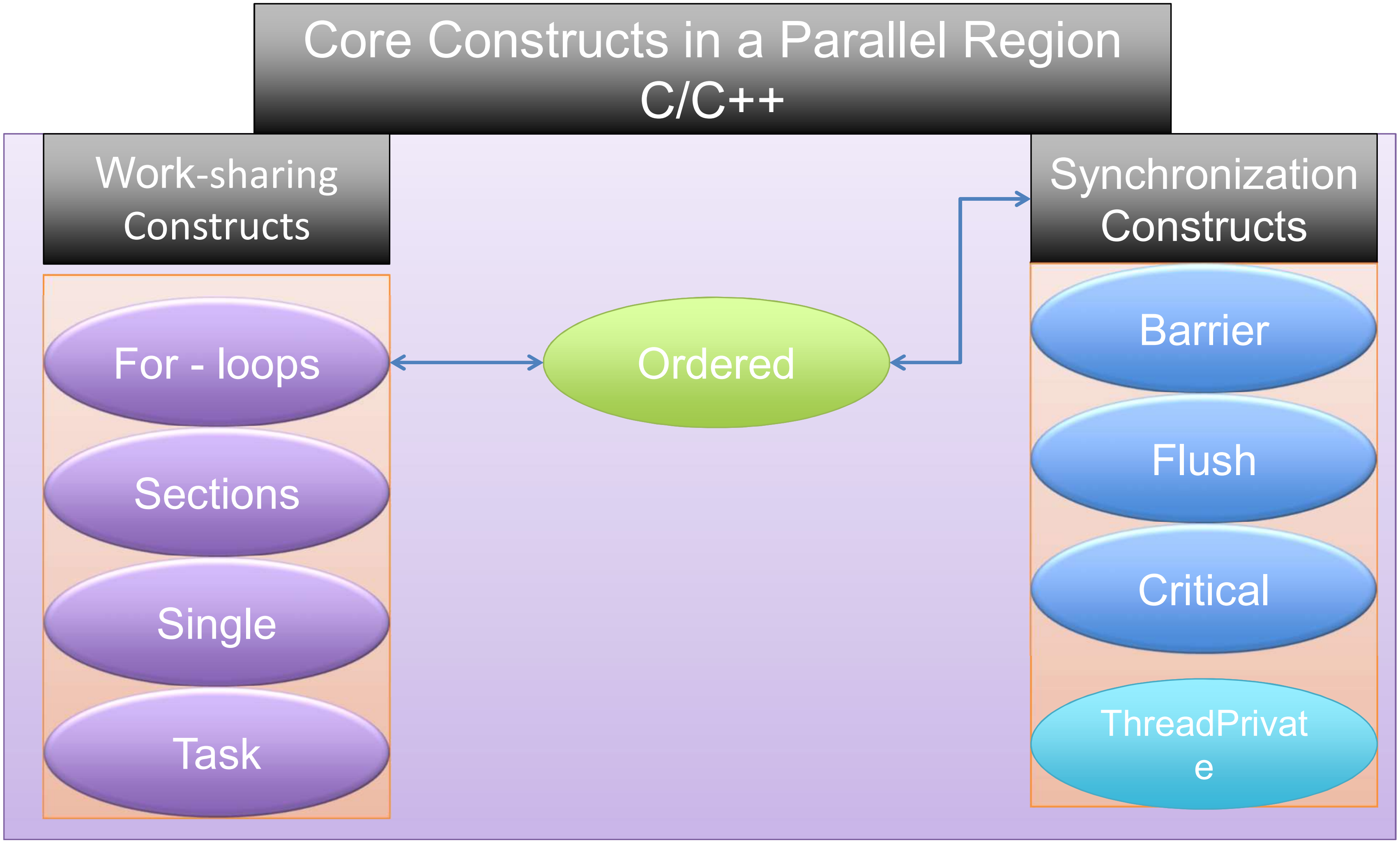
Defining a parallel region

#pragma omp parallel	<Clauses>
	if (scalar_expr)
	private (list)
	shared (list)
	default (shared|none)
	firstprivate (list)
	reduction(operator:list)
	copyin (list)
	num_threads (int)

```
#pragma omp parallel <Clauses>
{
...
...
...
Threaded code here
...
...

}
```

} End of parallel region

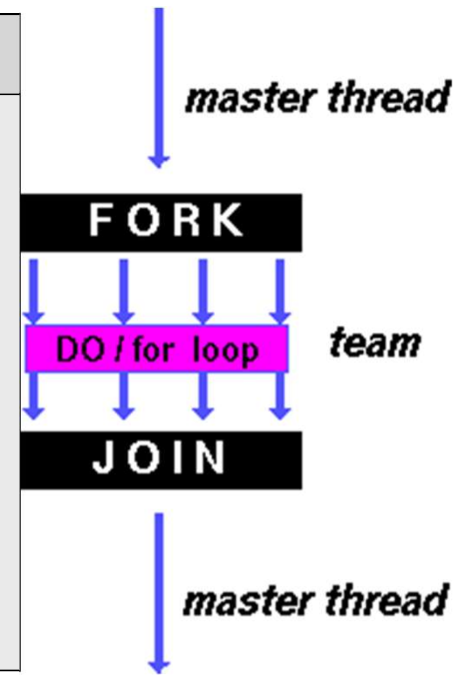
- All variables defined outside parallel regions are shared by default
- private variables in parallel regions
  – Each thread has it’s own copy of the variable and it’s own value. The context is hidden from other threads
- shared variables in parallel regions
  – The values of shared variables are visible and accessible by all threads
  – Programmer is responsible for avoiding *race conditions*


Core Constructs in a Parallel Region
C/C++
Work-sharing Constructs
For - loops
Sections
Single
Task
Ordered
Synchronization Constructs
Barrier
Flush
Critical
ThreadPrivat
e

for worksharing construct

```
#pragma omp for
{
for_loop
...
...
...
...
...

}
```

```
<Clauses>
schedule(type,[chunk])
ordered
private (list)
shared (list)
firstprivate (list)
lastprivate (list)
reduction(operator:list)
collapse (n)
nowait
```

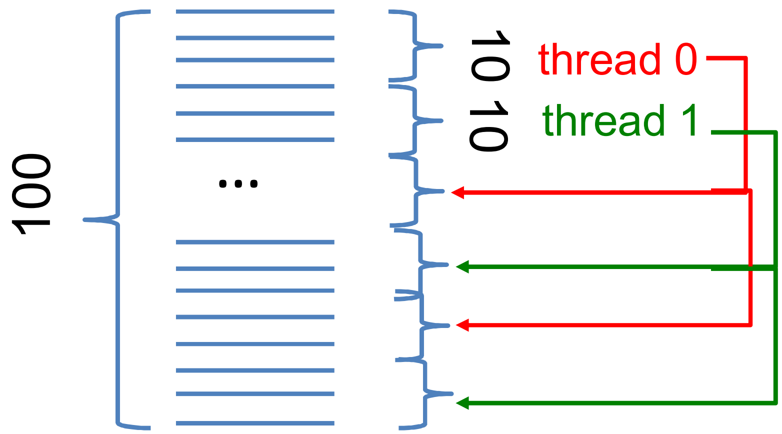


The reduction clause creates private copies of the shared variable list. Private copies are then ‘reduced’, i.e. summed, and the answer is returned back to the global variable

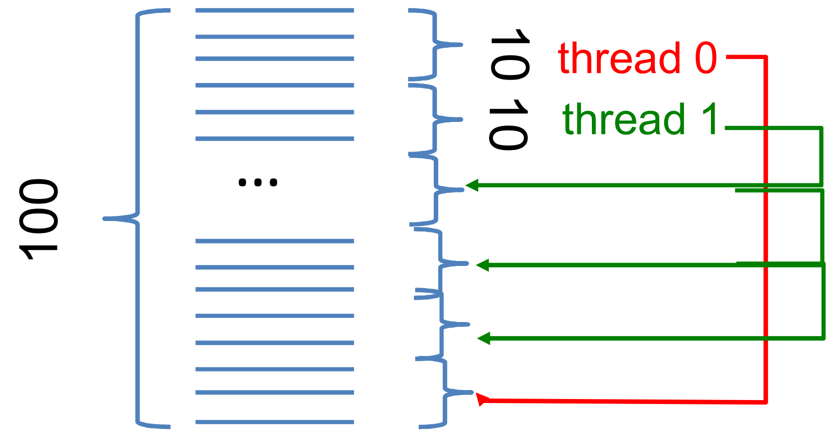
```
#pragma omp parallel for private(i)
reduction(+:result)
{
  for (i=0; i < n; i++)
    result = result + i*i;
}
```

The schedule(type,chunk) clause instructs on how to distribute the loop iterations among the threads.

static: fixed assignment of loop segments (of size chunk) to threads



dynamic: variable assignment of loop segments to threads



The *ordered clause* must be followed by the *ordered construct*. Loop operations are executed sequentially.

```
#pragma omp for ordered
schedule(dynamic)
 for(int n=0; n<100; ++n)
 {
   files[n].compress();

   #pragma omp ordered
   send(files[n]);
 }
```

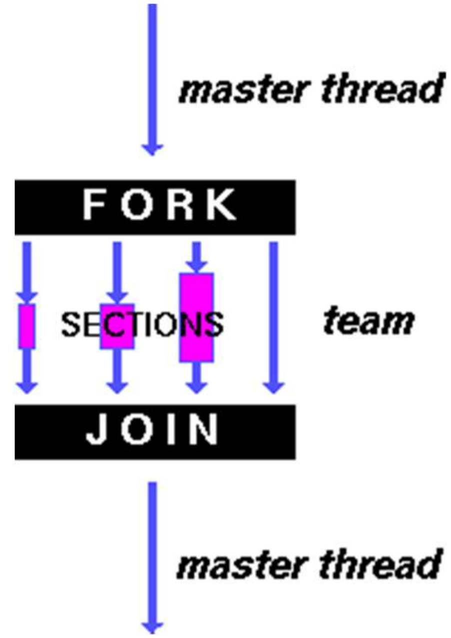
Done in parallel

Done sequentially

sections worksharing construct

#pragma omp sections	<Clauses>
	private (list) shared (list) firstprivate (list) lastprivate (list) reduction(operator:list) nowait

```
{
#pragma omp section
{
...
}
#pragma omp section
{
...
}
}
```



- At the end of ‘for’, ‘sections’ & ‘single’ constructs, threads **<u>by default</u> are** synchronized - they can not proceed unless **all** threads report at the same point in the instruction set.

- The nowait *clause* allows for asynchronous threads.

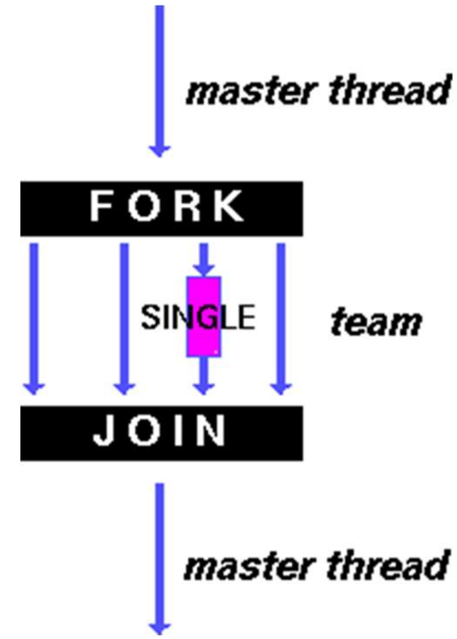
- Bypassing default synchronization points can lead to incorrect updating of shared(global) variables

```
#pragma omp parallel shared(a,b,c,d)
private(i)
 {
 #pragma omp sections nowait
   {
   #pragma omp section
   for (i=0; i < N; i++)
     c[i] = a[i] + b[i];
   #pragma omp section
   for (i=0; i < N; i++)
     d[i] = a[i] * b[i];
   }  /* end of sections */

…. more code …..

 }  /* end of parallel section */
```

single "worksharing" construct

#pragma omp single	<Clauses>
	private (list) firstprivate (list) nowait

```
{
…
block_of_code
…
}
```



Only a single thread executes code

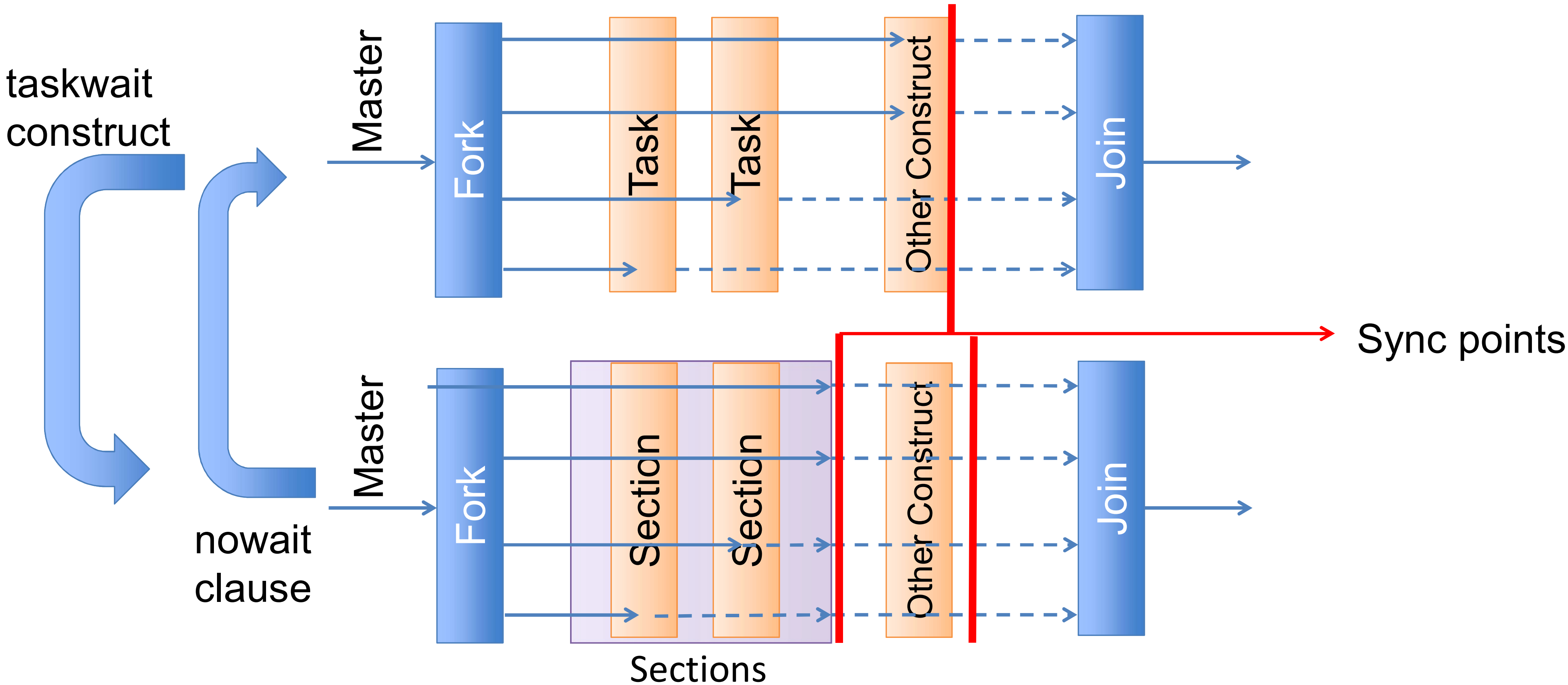
Task worksharing construct -OpenMP 3.0

```
#pragma omp task
{
block_of_code
}
```

<Clauses>
private (list)
shared (list)
firstprivate (list)
if (scalar_expr)
untied
Default(shared|none)
final (scalar_expr)
mergeable

Synchronization can be forced with `#pragma omp taskwait`

Task vs. Sections Construct



Core synchronization directives

```
#pragma omp barrier
```

Barrier synchronizes all threads at that point

```
#pragma omp flush (list)
```

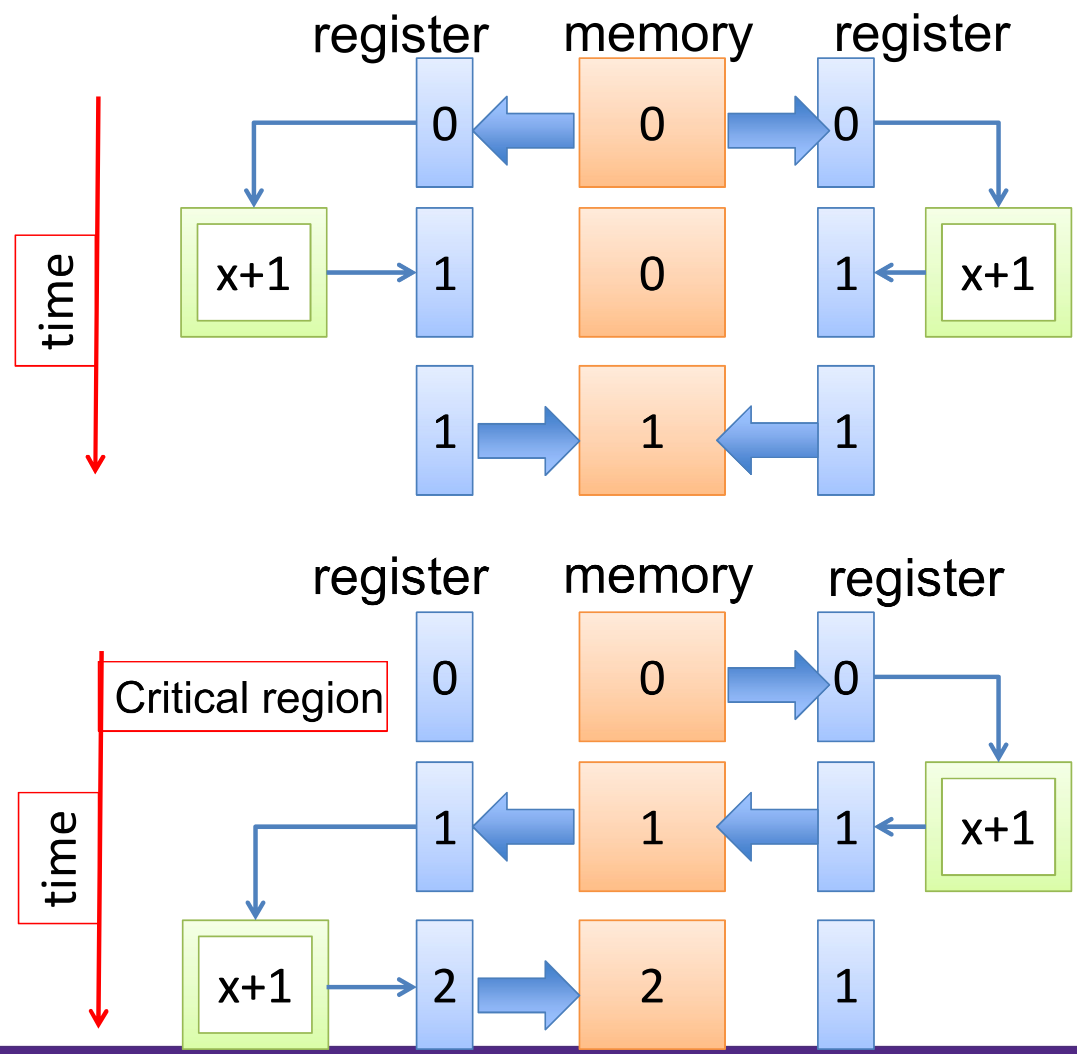
Flush synchronizes all thread-visible (shared) values from local CPU registers to memory. It re-establishes a consistent view of memory across all threads. It is implied upon exiting in:

barrier for
parallel sections
critical single
ordered

```
#pragma omp critical [name]
structured_block
```

The *critical* directive/construct allows access to a code segment one thread at a time. It aims in protecting simultaneous updates to memory locations of shared variables by threads – before threads are made aware of other updates, i.e. race conditions

```
#include <omp.h>
main()
{

int x;
x = 0;

#pragma omp parallel shared(x)
  {
  #pragma omp critical
  x = x + 1;
  }  /* end of parallel section */

}
```



```
#pragma omp threadprivate (list)
```

- The *threadprivate* directive/construct allows private
- variables to retain their thread specific values across
- parallel regions
- Does not apply in dynamic schedules
- Number of threads between parallel regions must be the same
- Defined outside the extend of the parallel regions

Run-time Library Routines

- Explicit management of threads
  - Setting & polling threads
  - Querying thread IDs
  - Querying levels of parallelism (nested threads)
  - Manage thread locking
  - Querying wall clock

Runtime routines

- Can be executed outside parallel regions
- Routines are hardcoded
- Most frequent examples:
- tid = omp_get_thread_num()
  - Query the numeric id number of a thread
- nthreads = omp_get_num_threads()
  - Query the number of available threads

Advantages

Using runtime routines to capture the thread ID allows for:

- Detailed management of thread work and thread synchronization
- Assists in code debugging
- Can emulate practices from distributed computing, i.e. MPI and SPMD (single program multiple data)

Disadvantages

- Inserts non-pragma source code in the form of runtime routines that prevents code from reverting to pure serial
- Use of SPMD practices requires to significantly alter source code
- Some runtime routines, i.e. omp_set_num_threads(), remove code flexibility at execution

Explicit vs. Implicit Casting to threads

```
int a[nelem],b[nelem];
#pragma omp parallel
{
int id, i, nthreads, i_begin, i_end;
id = omp_get_thread_num();
nthreads = omp_get_num_threads();
i_begin = id * nelem / nthreads;
i_end = (id+1) * nelem / nthreads;
if (id == nthreads-1) i_end = nelems;
for(i=i_begin;i<i_end;i++) { a[i] = a[i] + b[i];}
}
```

```
int a[nelem],b[nelem];
int i;
#pragma omp parallel for private(i)
 for( i=1;i<n_elems;i++)
{a[i] = a[i] + b[i];)
```

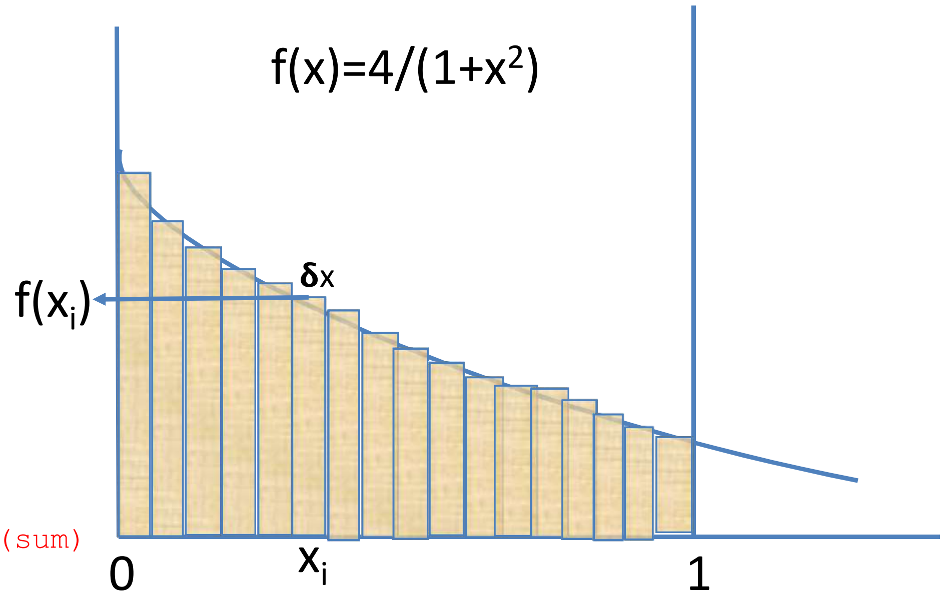
Environment Variables

- Pass information to the executable at runtime, i.e. number of threads or the method of partitioning loop iterations
- Provide an alternative to many runtime routines
- Allows for flexibility at execution time

Environmental Variable Examples

- Setting the number of threads
  - export OMP_NUM_THREADS=4
- Setting the thread stack size
  - export OMP_STACKSIZE="1G"
- Binding threads to processors (thread affinity)
  - export OMP_PROC_BIND=TRUE
- Specifying how loop operations are partitioned across threads
  - export OMP_SCHEDULE="static"

```c
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>
#define NSTEPS  500000000
long i,num_steps=NSTEPS;
double x,step,sum,pi;

int main(int argc, char **argv)
{
x=0;
sum = 0.0;
step = 1.0/(double) num_steps;
#pragma omp parallel private(i,x) shared(sum)
{
#pragma omp for schedule(static) reduction(+:sum)
for (i=0; i < num_steps; ++i){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
}
}
pi = step * sum;
printf("Computed PI %.24f\n", pi);
return 0;
}
```



$$\pi = \int_0^1 f(x)dx \cong \sum_i f(x_i)\boldsymbol{\delta}x$$

```
[ppk652@quser12 openmp_source]$ gcc -o pi_red.exe -fopenmp pi_red.c


[ppk652@quser12 openmp_source]$ export OMP_NUM_THREADS=4

[ppk652@quser12 openmp_source]$ time ./pi_red.exe

Computed PI 3.141592653589826422688702
real    0m3.794s
user    0m14.599s
sys     0m0.006s

[ppk652@quser12 openmp_source]$ export OMP_NUM_THREADS=1

[ppk652@quser12 openmp_source]$ time ./pi_red.exe
Computed PI 3.141592653590016936959728
real    0m13.833s
user    0m13.830s
sys     0m0.003s
```

```
top - 12:21:06 up 42 days, 23:36, 86 users,  load average: 0.84, 0.34, 0.28Tasks: 1128
total,   2 running, 1099 sleeping,  27 stopped,   0 zombie%Cpu(s): 20.2 us,  0.2 sy,  0.0 ni,
79.6 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 stKiB Mem : 13180445+total, 33825440 free,
96267984 used,  1711036 buff/cacheKiB Swap: 10737459+total, 28133764 free,
79240824 used. 33926724 avail Mem   PID USER      PR  NI    VIRT    RES    SHR S
%CPU %MEM     TIME+ COMMAND
19861 ppk652    20   0   16796    416    320 R 398.7  0.0   0:14.02 pi_red.exe
```

```
top - 12:22:06 up 42 days, 23:37, 86 users,  load average: 0.81, 0.39, 0.30Threads: 3539
total,   5 running, 3259 sleeping, 275 stopped,   0 zombie%Cpu(s): 15.5 us,  0.5 sy,  0.0
ni, 83.9 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 stKiB Mem : 13180445+total, 33760304 free,
96328160 used,  1715988 buff/cacheKiB Swap: 10737459+total, 28133764 free,
79240824 used. 33864044 avail Mem   PID USER      PR  NI    VIRT    RES    SHR S
%CPU %MEM     TIME+ COMMAND
20060 ppk652    20   0   16796    412    320 R 73.1  0.0   0:02.25 pi_red.exe
20061 ppk652    20   0   16796    412    320 R 73.1  0.0   0:02.25 pi_red.exe
20063 ppk652    20   0   16796    412    320 R 73.1  0.0   0:02.25 pi_red.exe
20064 ppk652    20   0   16796    412    320 R 73.1  0.0   0:02.25 pi_red.exe
```

Example of building and running an OpenMP C program with threadprivate directive

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

int  a, b, i, tid;

#pragma omp threadprivate(a)

main ()  {

/* Explicitly turn off dynamic threads */
  omp_set_dynamic(0);

  printf("1st Parallel Region:\n");
#pragma omp parallel private(b,tid)
  {
  tid = omp_get_thread_num();
  a = tid;
  b = tid;
  printf("Thread %d:   a,b= %d %d \n",tid,a,b);
  }  /* end of parallel section */

  printf("************************************\n");
  printf("Master thread doing serial work here\n");
  printf("************************************\n");

  printf("2nd Parallel Region:\n");
#pragma omp parallel private(tid)
  {
  tid = omp_get_thread_num();
  printf("Thread %d:   a,b= %d %d \n",tid,a,b);
  }  /* end of parallel section */

}
```

```
[ppk652@quser12 openmp_source]$ gcc -fopenmp -o thrpv.exe thrpv.c
[ppk652@quser12 openmp_source]$ export OMP_NUM_THREADS=4
[ppk652@quser12 openmp_source]$ ./thrpv.exe
1st Parallel Region:
Thread 2:   a,b= 2 2
Thread 3:   a,b= 3 3
Thread 0:   a,b= 0 0
Thread 1:   a,b= 1 1
************************************
Master thread doing serial work here
************************************
2nd Parallel Region:
Thread 2:   a,b= 2 0
Thread 3:   a,b= 3 0
Thread 1:   a,b= 1 0
Thread 0:   a,b= 0 0
```

Instructions for running OpenMP codes on a Mac

From a terminal type: ($ is the prompt)

```
$/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"

$brew install gcc --without-multilib
```

Then you can compile from a terminal as: gcc-7 -fopenmp etc

Conclusions

- The OpenMP API enables thread-based parallelism on shared memory machines using low-level programming languages like C
- Any change in the code requires recompiling
- Parallelism is limited by available memory and number of processors (cores)
- Thread based parallelism is implemented in all programming languages in some form, e.g. Python, R, java

Threading example in Python

```
import threading

def worker():
      print("Hello world")
      return

threads = []
for i in range(5):
      t = threading.Thread(target=worker)
      threads.append(t)
      t.start()
```

```
[ppk652@quser12 ~]$ python pythread.py
Hello world
Hello world
Hello world
Hello world
```

Questions about OpenMP?
Email us at:
quest-help@northwestern.edu
Check us out at:
www.it.northwestern.edu/research