Introduction to OpenMP

Silvio Stanzani , Raphael Cóbe and Jefferson Fialho

UNESP – Center for Scientific Computing

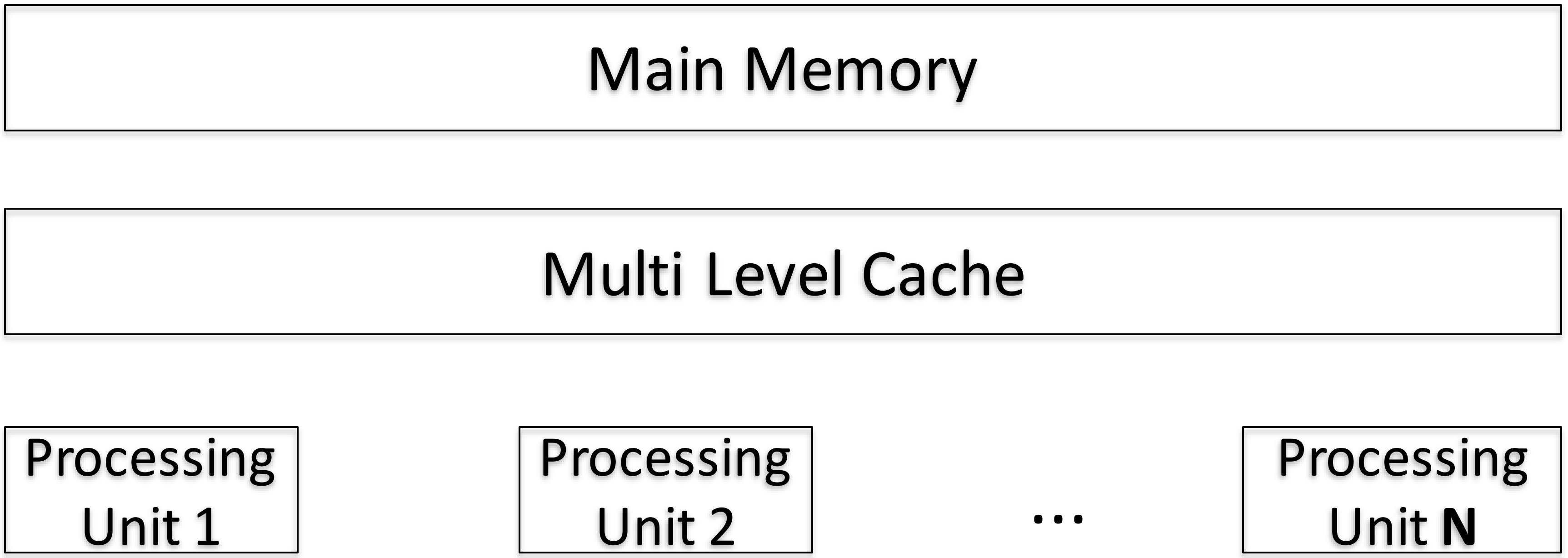
silvio.stanzani@sprace.org.br, raphael.cobe@sprace.org.br, jefferson.fialho@sprace.org.br

Agenda

- Parallel processing
- Concurrency and Synchronization
- Process / Thread
- OpenMP Programming Model
- Race Condition
- Synchronization
- Task Parallelism
- Hands-on

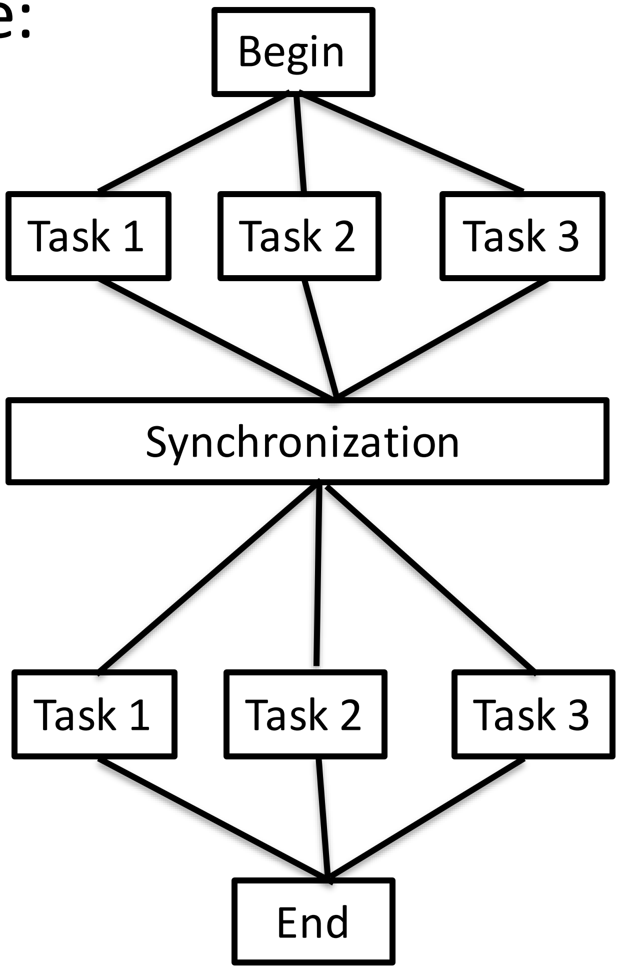
Shared Memory Architecture

- Complex Memory System Architecture
- Transparent to Users



Concurrency and Synchronization

– Example:



- Global Variables
- I/O Operations

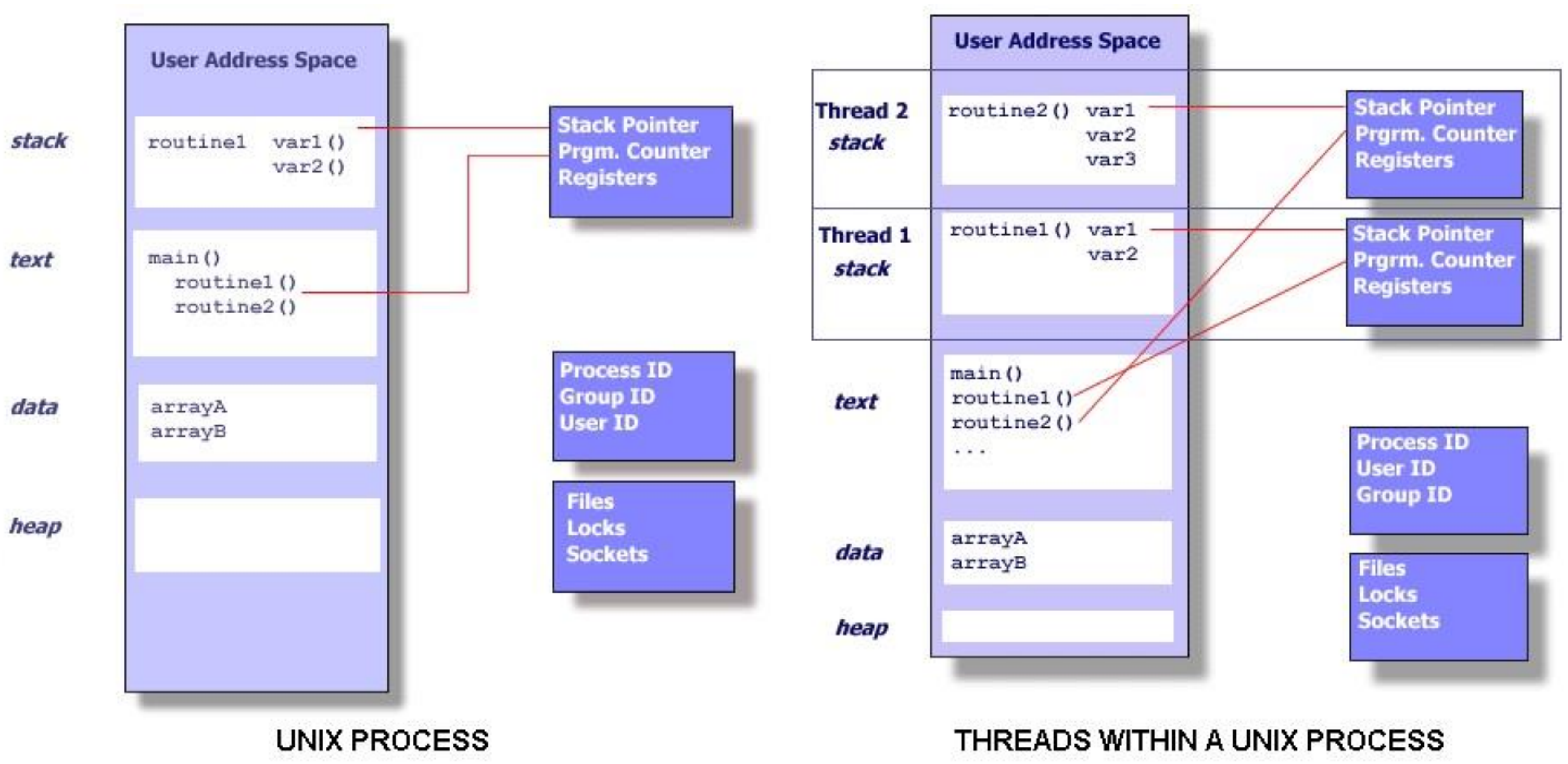
Process and Threads

- Any program running on Operational system is called Process:
  - Is composed of at least of one Thread
  - Can fork several threads which is a copy of itself

- Creating a new thread are much faster than create a new process

- There are libraries that support thread creation such as Pthreads

Process and Threads



Source: https://computing.llnl.gov/tutorials/pthreads/

Multithreaded Programming

- **Multithreading** is the ability of a O.S. to execute one process using several resources simultaneously by the means of threads
- **Multithreaded Programming** is a parallel programming technique that has the objective of prepare your program to be executed as concurrent parts on several threads
- **Pthread** is one library for Multithreaded Programming

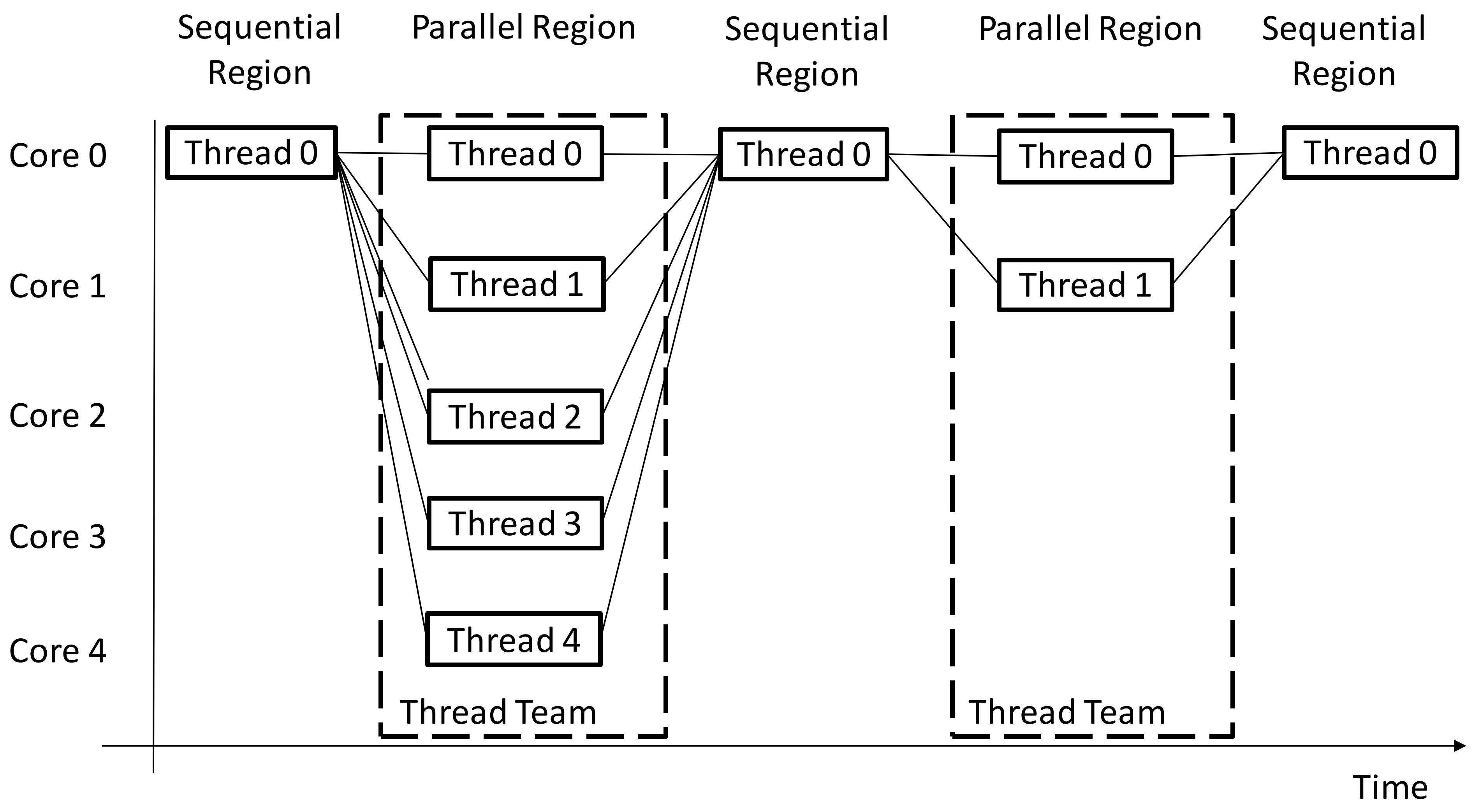
Pthreads Example

```
#include <pthread.h>

void *inc_x(void *x_void_ptr) {
        int *x_ptr = (int *)x_void_ptr;
        while(++(*x_ptr) < 100);
        printf("x increment finished\n");

        return NULL;
}

int main() {

    int x = 0, y = 0;
    pthread_t inc_x_thread;

    printf("x: %d, y: %d\n", x, y);

    pthread_create(&inc_x_thread, NULL, inc_x,
&x)

while(++y < 100);

printf("y increment
finished\n");

pthread_join(inc_x_thread,
NULL)

printf("x: %d, y: %d\n", x, y);

return 0;
}
```

OpenMP

- OpenMP is an acronym for Open Multi-Processing
- An Application Programming Interface (API) for developing parallel programs in shared memory architectures
  - API based on Pragmas – C code extensions
- Three primary components of the API are:
  - Compiler Directives
  - Runtime Library Routines
  - Environment Variables
- De facto standard - specified for C / C++ and FORTRAN
- http://www.openmp.org/
  - Specification, examples, tutorials and documentation

OpenMP


Sequential Region
Parallel Region
Sequential Region
Parallel Region
Sequential Region
Core 0
Core 1
Core 2
Core 3
Core 4
Thread 0
Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread Team
Thread 0
Thread 0
Thread 1
Thread Team
Thread 0
Time

OpenMP - Core elements

Parallel control structures

Form a team of threads and execute them in parallel

```
omp parallel
```

Work sharing

Distribute work among threads

```
omp [parallel] loop
omp [parallel] sections
omp [parallel] workshare
omp single
```

Synchronization

Coordinates thread execution

```
omp atomic
omp barrier
omp critical
omp flush
omp master
omp ordered
omp taskgroup
omp taskwait
```

Data environment

Control variables scope

`omp threadprivate`
`shared/*private` clauses

OpenMP - Core elements

OpenMP 4.0 - Co-Processors and Accelerators

SIMD vectorization

Offload execution

Thread affinity

Popular model for Intel Co-Processor:

- Xeon Phi
- KNL

Tasking

- Structures for deferring execution
- `omp task`
  `omp taskyield`

Runtime environment

- Runtime functions and environment variables
- `omp_set_num_theads()`, etc.
  `OMP_SCHEDULE`, etc.

Loop

- Serial Application example:

```
Int i=0;
N=25;
for (i=0; i<N; i++)
    a[i] = a[i] + b;
```

- Iterations of a loop represents tasks that can be executed concurrently;

Parallel Region

```
#pragma omp parallel
 {
… //Code that need to be executed concurrently goes here
}
```

- The region enclosed by **pragma omp parallel** will be execute by all threads
- Loop iterations can be divided among threads

OpenMP Sample Program

```
#include <stdio.h>

int main() {
   char hn[600];

   #pragma omp parallel
   {
      gethostname(hn,600);
      printf("hello from hostname %s %d\n",hn);
   }
   return(0);
}
```

Compiling and running an OpenMP application

- Build the application using gcc

gcc <*source-code*> -o <*omp_binary*> -fopenmp

- Build the application using pgi

pgcc <*source-code*> -o <*omp_binary*> -mp

- Launch the application

export OMP_NUM_THREADS=10

./*omp_binary*

OpenMP Functions

- omp_get_max_threads()
  - Amount of processing units (cores)
- omp_get_thread_limit()
  - Amount of threads that O.S. can Manage
- omp_get_thread_num();
  - Get the thread id
- omp_set_num_threads(8);
  - Setup the amount of threads to be used
- Environmental variables:
  - OMP_NUM_THREADS: define the amount of threads to execute a program using OpenMP
    - ❑ Example: export OMP_NUM_THREADS=10

Thread Affinity

- Specify the Process/Cores to map threads
  - **GOMP_CPU_AFFINITY**: specify the cores to execute threads
    - ❑ Uses cores from 0 to 19
      - export GOMP_CPU_AFFINITY=3-15
  - **OMP_PROC_BIND:** specify a pattern to map threads
    - ❑ Keep the threads as close to thread 0 as possible
      - export GOMP_PROC_BIND=close
    - ❑ Spread the threads across processors
      - export GOMP_PROC_BIND=spread

Data Environment

- How threads communicates?
  - Using variables (global and local)

- OpenMP Allows developers to define variables to be private or global among other(Attribute Clauses):
  - shared(list) : global variable across all threads
  - private(list): each thread has its own version, initial value is 0
  - firstprivate(list): each thread has its own version, initial value is the last version before OpenMP region
  - lastprivate(list): each thread has its own version, after the end of OpenMP region the variable receives the value from last thread

Work-Sharing: loop

- 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.
- No implied barrier upon entry to a work sharing construct.
- However, there is an implied barrier at the end of the work sharing construct (unless nowait is used).

Work-Sharing: loop

Sequential code

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

OpenMP // 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 for schedule(static) private(i)

for(i = 0;i < N; i++ ) {

      a[ i ] = a[ i ] + b[ i ];

}
```

Work-Sharing: loop

Sequential code

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

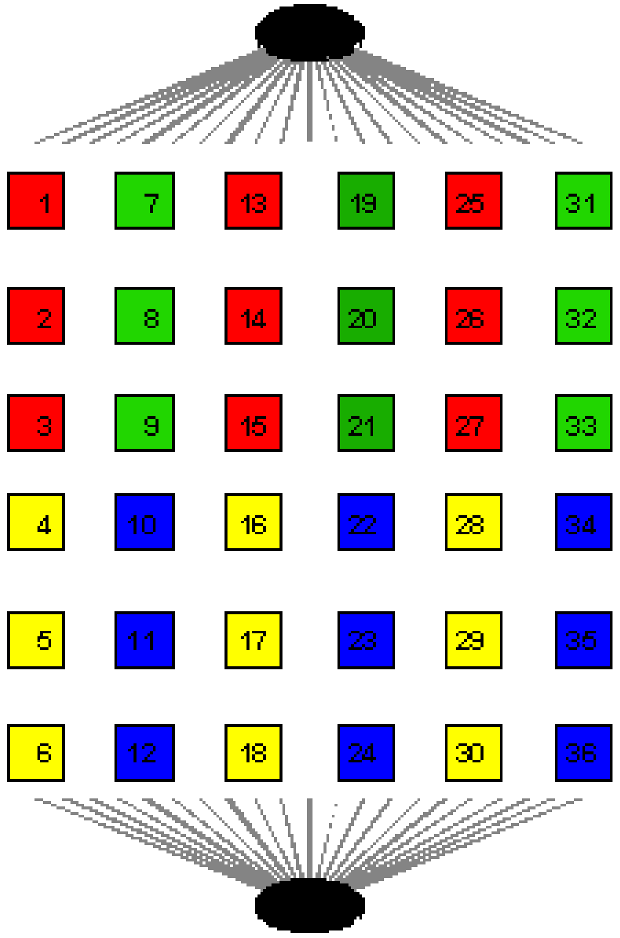
OpenMP // Region

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

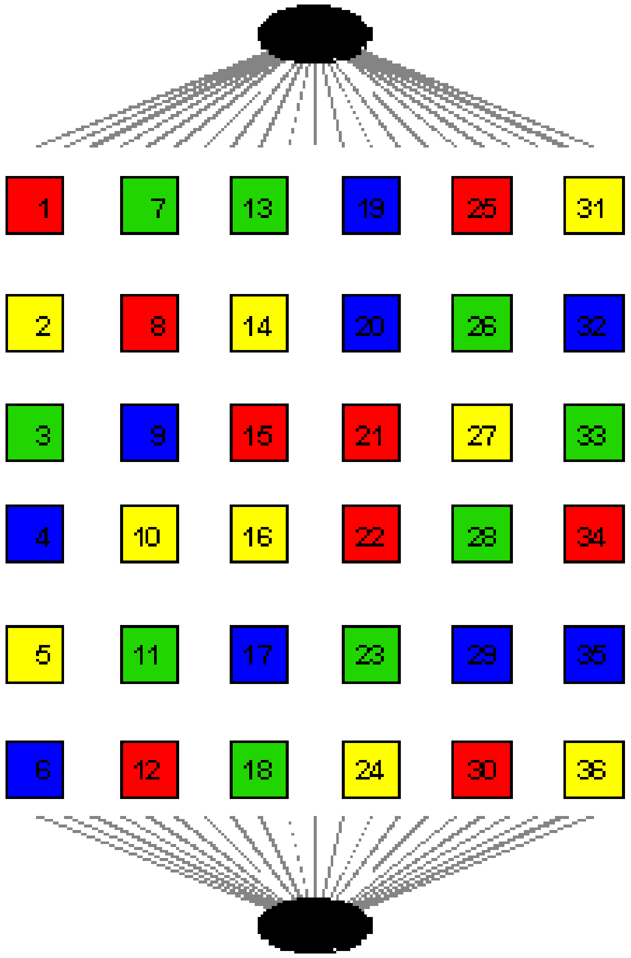
schedule(static [,chunk])



- Deal-out blocks of iterations of size "chunk" to each thread
- Iterations are divided evenly among threads
- If chunk is specified, divides the work into chunk sized parcels
- If there are N threads, each thread does every Nth chunk of work.

```
!$OMP PARALLEL DO &
!$OMP SCHEDULE(STATIC,3)

DO J = 1, 36
Work (j)
END DO

!$OMP END DO
```

schedule(dynamic [,chunk])



- Each thread grabs "chunk" iterations off a queue until all iterations have been handled
- Divides the workload into chunk sized parcels.
- As a thread finishes one chunk, it grabs the next available chunk.
- Default value for chunk is one.
- More overhead, but potentially better load balancing.

```
!$OMP PARALLEL DO &
!$OMPSCHEDULE(DYNAMIC,1)

DO J = 1, 36
Work (j)
END DO

!$OMP END DO
```

Optimization Example

- Performance comparison using command "time"
  - **Time** return the amount of time spent by your application

- Serial version:
  - gcc OMP-matrix-sum.c -o OMP-matrix-sum
  - time ./OMP-matrix-sum

- Parallel version:
  - gcc OMP-matrix-sum.c -o OMP-matrix-sum -fopenmp
  - time ./OMP-matrix-sum

Race Condition

- When two or more threads perform operations on shared data, it is impossible to know the order in which this operations will be performed;
- This is a condition in which one or more threads are "racing" to perform the same operation
- The program will not end with a bug, but in some cases will return with incorrect results

Race Condition

- Example of a race condition:

```
#pragma omp parallel for
for(i=0 ; i<size_of_input_array; i++)
{
    Int *tmpsum = input+i;
    sum += *tmpsum;
}
```

- Every execution return different results!

Race Condition

- Solution to solve race condition problems:
  - Break the dependency changing the algorithm
  - Enforce synchronization: the execution is performed sequentially by all threads
- OpenMP provides several options for synchronization
- Synchronization enforce performance penalties!

Synchronization

- Synchronization directives:
  - omp atomic:
    - ❑ Ensures that a specific memory location is updated atomically, which prevents the possibility of multiple, simultaneous reading and writing of threads.
  - omp critical
    - ❑ Specifies a code block that is restricted to access by only one thread at a time.
  - omp ordered
    - ❑ Specifies a code block in a worksharing loop that will be run in the *order* of the loop iterations

Synchronization: `atomic` directive

- atomic enables mutual exclusion for some simple operations
- these are converted into special hardware instructions if supported
- however, it only protects the read/update of the target location

```
#pragma omp parallel
{
    // compute my_result

    #pragma omp atomic
    x += my_result;
}
```

acceptable operations

- `x++`
- `x--`
- `++x`
- `--x`
- `x binop= expr`
- `x = x binop expr`
- `x = expr binop x`

where `binop` is one of

`+ * - / & ^ | << >>`

```
#pragma omp parallel
{
    #pragma omp atomic
    x += func(); // warning func() is not atomic!
}
```

Synchronization

```
#pragma omp parallel for
for(i=0 ; i<size_of_input_array; i++) {
    Int *tmpsum = input+i;
    #pragma omp critical
    {
        sum += *tmpsum;
    }
}
```

You can enclose a **code region** inside critical clause

Synchronization

```
#pragma omp parallel for ordered
for(i=0 ; i<size_of_input_array; i++) {
    Int *tmpsum = input+i;
    #pragma omp atomic
    sum += *tmpsum;
}
```

Atomic can embrace a **single line only**

- **Atomic** presents better performance then **critical**
- if synchronization is unavoidable use **atomic** instead of **critical** when possible

Synchronization

- Synchronization directives:
  - omp barrier
    - ❑ Specifies a point in the code where each thread must wait until all threads in the team arrive.
  - omp master
    - ❑ Specifies the beginning of a code block that must be executed only once by the master thread of the team.
  - omp single
    - ❑ Only one thread execute the code block

Synchronization

```
#pragma omp master
{
        thid=    omp_get_thread_num();
        printf("master thread only: thread %d \n", thid);
}

thid=    omp_get_thread_num();

printf("ALL threads: BE CAREFULL! thread  %d \n",
thid);
```

Synchronization

```
#pragma omp single
{
        thid=    omp_get_thread_num();
        printf("some thread execute this part (only one): thread  %d \n", thid);
}

#pragma omp barrier

thid=    omp_get_thread_num();
printf("after omp barrier! thread %d \n", thid);
```

All Threads wait in this barrier

Synchronization: `reduction` clause

- creates a private variable for each thread
- each thread works on private copy
- finally all thread results are accumulated using operator
- allowed operators: `+, -, *, &, |, ^, &&, ||, min, max`
- each operator has a default initialization value (e.g. 0 for addition, 1 for multiplication)

```
double global_result = 0.0;

#pragma omp parallel reduction(+:global_result)
{
    double h = (b - a) / n;
    int tid = omp_get_thread_num();
    int nthreads = omp_get_num_threads();
    int local_n = n / nthreads;
    double local_a = a + tid * local_n * h;
    double local_b = local_a + local_n * h;

    global_result += local_sum(local_a, local_b, local_n, h);
}
```

Task Parallelism

- Tasks are independent units of work
  - code to execute
  - Input/output data
- Threads are assigned to perform the work of each task.
- Tasks can be defined as a relation of dependency

Synchronization: `nowait` clause

Work sharing constructs have a implict barrier at their end. With nowait you can allow them to continue after they finish their part.

```
#pragma omp parallel
{
    #pragma omp for
    for(...) {
    }
    // implicit barrier

    #pragma omp for
    for(...) {
    }
}
```

```
#pragma omp parallel
{
    #pragma omp for nowait
    for(...) {
    }
    // threads can continue

    #pragma omp for
    for(...) {
    }
}
```

Synchronization: `master` clause

```
#pragma omp parallel
{
    #pragma omp master
    {
        // only master thread should execute this
        // useful for I/O or initialization
        // there is NO implicit barrier!
    }

    // add explicit barrier if needed
    #pragma omp barrier
    ...
}
```

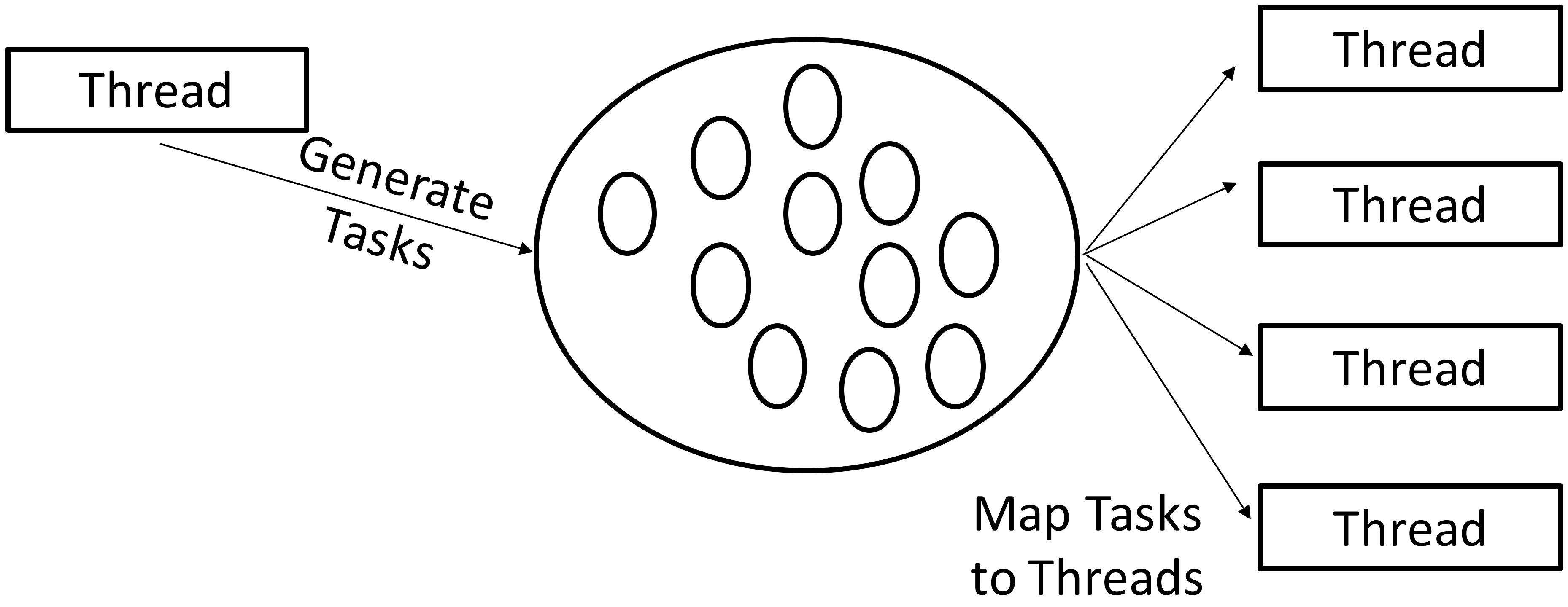
Synchronization: `single` clause

```
#pragma omp parallel
{
    #pragma omp single
    {
       // only one thread will execute this block
       // all others wait until it completes
       // implicit barrier!
    }
}
```

```
#pragma omp parallel
{
    #pragma omp single nowait
    {
       // only one thread will execute this block
       // others will go right past it
    }
}
```

Task Parallelism

- Task Parallelism model of OpenMP.



Task Parallelism

- tasks must be created inside of a parallel region:
  - **#pragma omp task**

```
#pragma omp parallel
{
   #pragma omp single
   {
      #pragma omp task
      printf("hello world\n");

      #pragma omp task
      printf("hello again!\n");
   }}
```

Task Parallelism

- Fibonacci Sequence:
  - A sequence of number in which every number after the first two is the sum of the two preceding ones
- F(n) = F(n-1) + f(n-2);
- F(1)=1 and F(2)=1
- Example F(10): 1 1 2 3 5 8 13 21 34 55

Task Parallelism

- Fibonacci serial version:

```
fibs[0]=1;
fibs[1]=1;
sum=2;
for (i = 2; i < N; i++) {
  fibs[i] = fibs[i - 1] + fibs[i - 2];
  sum+=fibs[i];
}
```

Task Parallelism

Recursive Version

```
int x,y;

if ( n < 2 ) return n;

x = fib(n-1);

y = fib(n-2);

return x+y;
```

Omp task

```
int x,y;

if ( n < 2 ) return n;
#pragma omp task shared(x)
x = fib(n-1);
#pragma omp task shared(y)
y = fib(n-2);
#pragma omp taskwait
return x+y;
```

Task Parallelism

Recursive

```
int main() {

  for (c = 1; c <= n; c++)
      fib(NN);

}
```

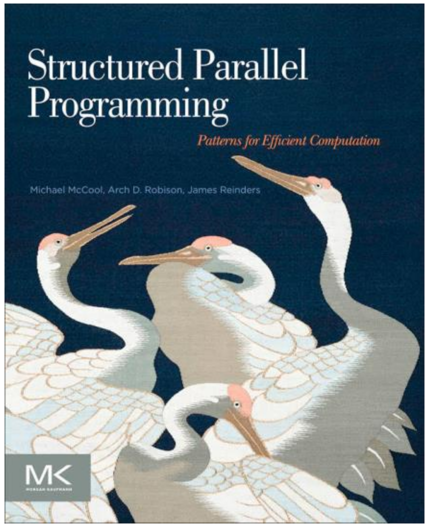
Omp task

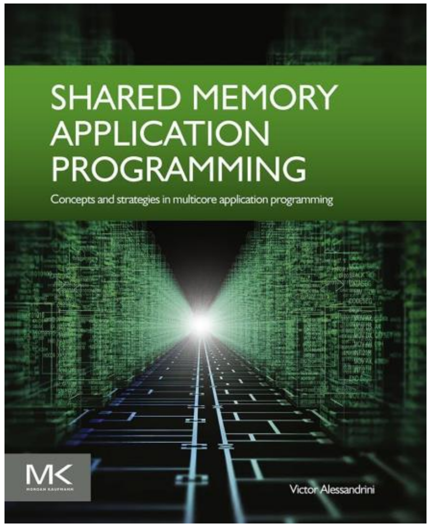
```
int main() {

  #pragma omp parallel
  {
    #pragma omp master
    {
     for (c = 1; c <= n; c++)
         fib(NN);
    }
  }
}
```

Reference

- "Structured parallel programming"
  - *McCool, Michael*

- "Shared memory application programming"
  - *Victor Alessandrini*


Structured Parallel Programming
Patterns for Efficient Computation
Michael McCool, Arch D. Robison, James Reinders
MK


SHARED MEMORY APPLICATION PROGRAMMING
Concepts and strategies in multicore application programming
MK
Victor Alessandrini

Hands-on

- Download source code to your home at cluster:
  - git clone https://github.com/silviostanzani/ICTP-HPC.git

- In the folder: "Introduction-to-OpenMP/hands-on" there are four applications:
  - Transposition
  - ironbar
  - Quicksort
  - Sum
  - Nbody
  - Optionprice

- Use OpenMP to parallelize each code
- Identify data dependencies
- Compare the performance between serial and parallel version
  - Plot the speedup curve (0-8 cores in the same node)
- Compare the execution of parallel version using different thread affinity scenarios

Synchronization

```
#pragma omp parallel for ordered
for(i=0 ; i<size_of_input_array; i++) {
    Int *tmpsum = input+i;
    #pragma omp ordered
    sum += *tmpsum;
}
```

You must put ordered clause in a loop with ordered clause