Introduction to OpenMP
First Part

Silvio Stanzani , Raphael Cóbe , Rogério Iope, Jefferson Fialho

UNESP – Center for Scientific Computing

silvio@ncc.unesp.br , rmcobe@ncc.unesp.br , rogerio@ncc.unesp.br , jfialho@ncc.unesp.br

UNESP Center for Scientific Computing

- Consolidates scientific computing resources for São Paulo State University (UNESP) researchers
  - It mainly uses Grid computing paradigm

- Main users
  - UNESP researchers, students, and software developers
  - SPRACE (São Paulo Research and Analysis Center) physicists and students
    - Caltech, Fermilab, CERN
    - São Paulo CMS Tier-2 Facility
- P&D Projects
  - Intel and Huawey

Recommendations

- Desired Requirements
  - Basic knowledge about C programming
  - Basic knowledge about how to write algorithms
  - Knowledge about how to run programs in Torque
- Do not let the class continue if you have doubt!
  - Please INTERRUPT if you have doubt (english or spanish or portuguese)

Agenda

- Theory 1:30
  - Parallel processing
  - Process / Thread
  - OpenMP Programming Model
    - ❑ Run-Time Library Routines
    - ❑ Environment Variables
    - ❑ Directives
      - Parallel region
      - Parallel loop
      - Data Environment
- Coffee
- Hands-on 2:00
- Lunch

- Theory 1:30
  - Race conditions
  - Synchronization Constructs
  - Race condition detection
  - Task Parallelism
  - Performance evaluation with Valgrind
- coffee
- Hands-on 2:00
- Overtime

Computer Architecture

- Uniprocessor architecture
  - New processor generations increased performance (Moore's Law)
  - Until processor performance achieved its limit

- Trends to keep increasing performance:
  - More processing units in a single processor
  - More processors in the same machine

- In order to achieve better performance, developers **MUST** explore parallelism.

Parallel Processing

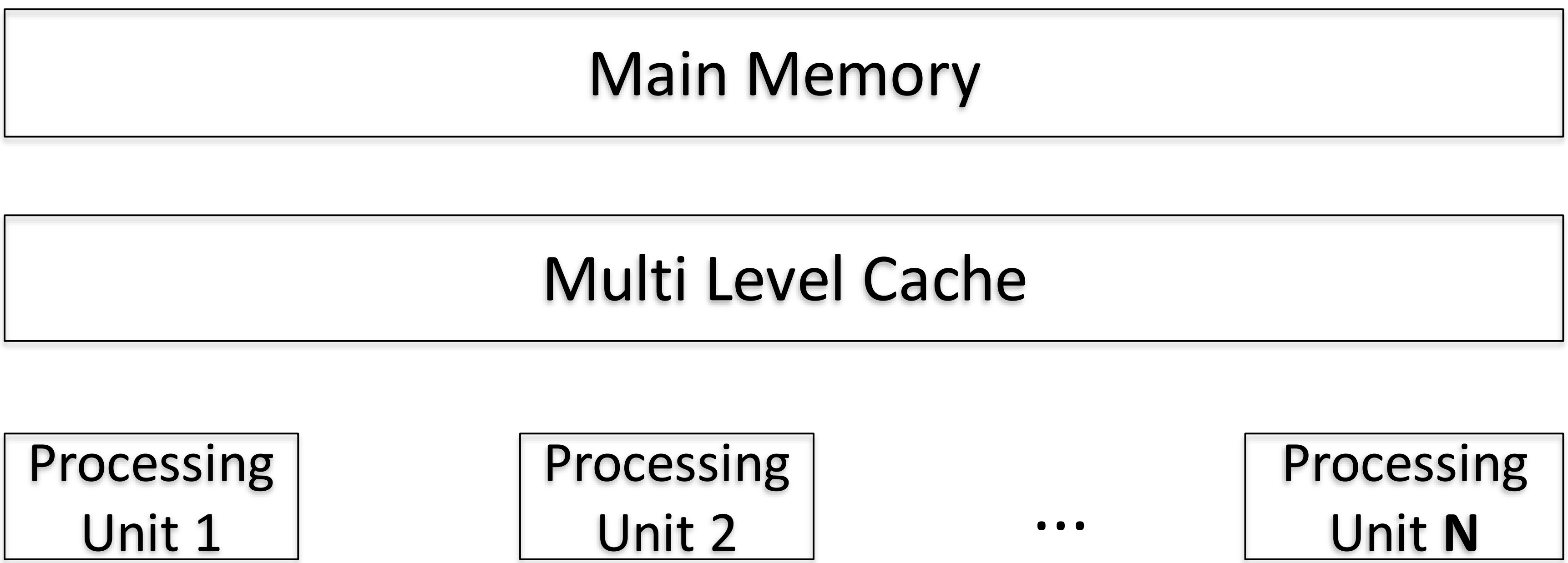
- A parallel computer is a computer system that uses multiple processing elements simultaneously in a cooperative manner to solve a computational problem

- Parallel processing includes techniques and technologies that make it possible to compute in parallel
  - Hardware, networks, operating systems, parallel libraries, languages, compilers, algorithms, tools, …

- Parallel computing is an evolution of serial computing
  - Parallelism is natural
  - Computing problems differ in level / type of parallelism

Shared Memory Architecture

- Complex Memory System Architecture
- Transparent to Users
- Influences performance!

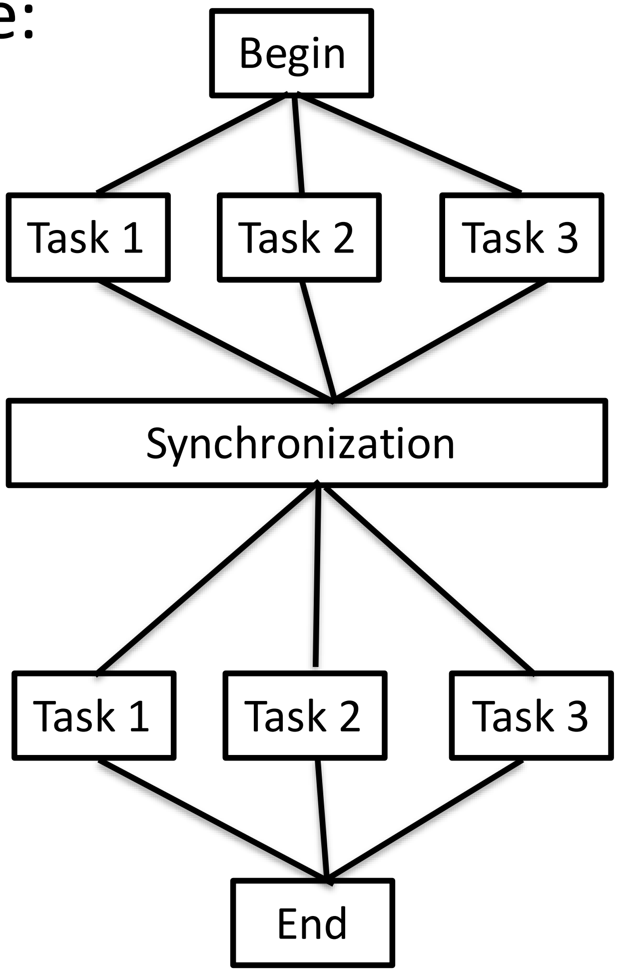


Concurrency and Synchronization

- Consider a program composed of multiple tasks to be executed in a computer
  - Tasks are **concurrent** with they can execute at the same time (concurrent execution)
- If a task requires results produced by other tasks in order to execute correctly, the task's execution is dependent
  - Some form of **synchronization** must be used to enforce (satisfy) dependencies

Concurrency and Synchronization

– Example:



- Global Variables
- I/O Operations

Parallelism

- Parallelism = concurrency + “parallel” hardware
  - Find concurrent execution opportunities
  - Develop application to execute in parallel
  - Run application on parallel hardware

- Motivations for parallelism
  - Faster time to solution (response time)
  - Solve bigger computing problems (in same time)
  - Effective use of machine resources

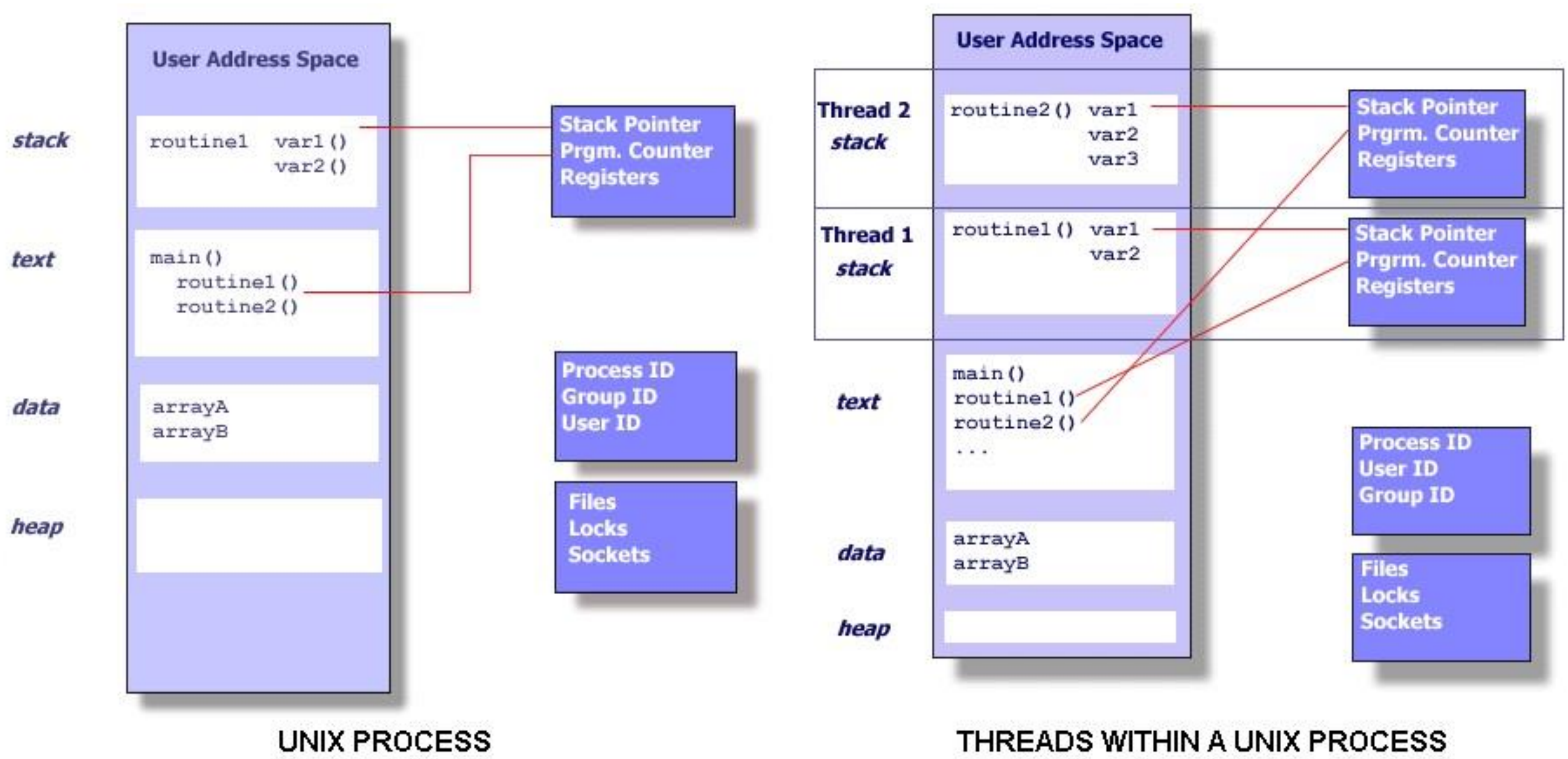
Process and Threads

- 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

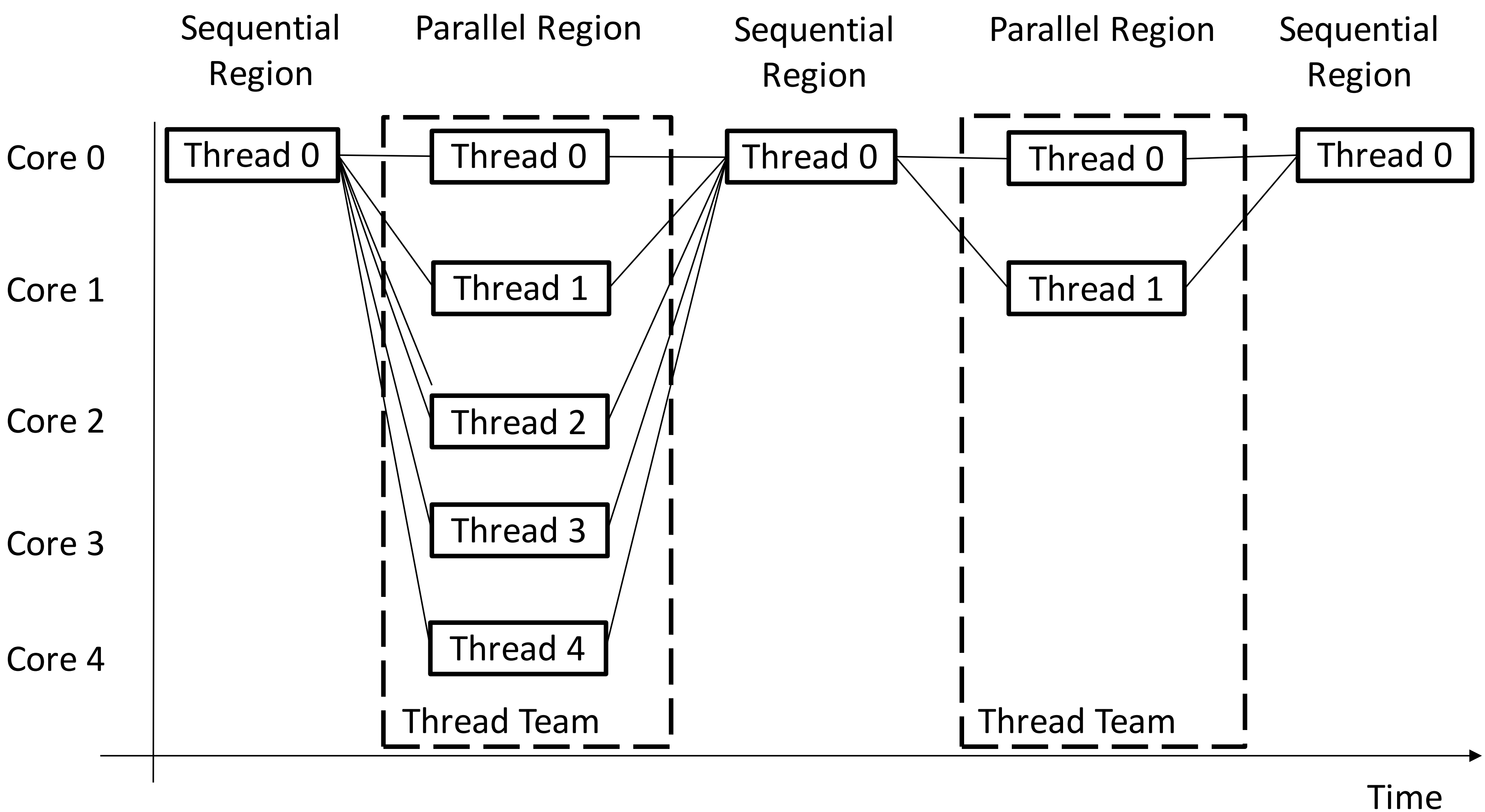
Pthreads Example

```
#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**

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*

Executing on Los Andes Cluster

- Create job file:
  - Ex: job1

```
#!/bin/bash
#PBS -k o
#PBS -l nodes=1:ppn=1,walltime=30:00
#PBS -M jthutt@tatooine.net
#PBS -m abe
#PBS -N FirstOMPjob
#PBS -j oe
/hpcfs/home/sshpc/jobs/execjob.sh
```

- Submit job qsub
  - Qsub job1

- Verfify job status qstat
  - qstat

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

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 accross 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

Loop sharing

- Create a parallel region and distribute the loop interations among the threads
  - Complete version:
    - Pragma omp parallel
      - Pragma omp for
  - short version:
    - Pragma omp parallel for

- lastprivate(list): each thread has its own version, after the end of openmp region the variable receives the value from last thread

Hands-on

- option-price
- Matrix Transposition
- IronBar (ERAD-SP 2018 warmup)

Introduction to OpenMP
Second Part

Silvio Stanzani , Raphael Cóbe , Rogério Iope, Jefferson Fialho

UNESP – Center for Scientific Computing

silvio@ncc.unesp.br , rmcobe@ncc.unesp.br , rogerio@ncc.unesp.br , jfialho@ncc.unesp.br

Agenda

- Theory 1:30
- Parallel processing
- Process / Thread
- OpenMP Programming Model
  - Run-Time Library Routines
  - Environment Variables
  - Directives
    - ❑ Parallel region
    - ❑ Parallel loop
    - ❑ Data Environment
- Coffee
- Hands-on 2:00
- lunch

- Theory 1:30
  - Race condition
  - Synchronization Constructs
  - Performance evaluation with Valgrind
  - Race condition detection
  - Task parallelism
- coffee
- Hands-on 2:00
- Overtime

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 using a different algorithm
  - Enforce synchronization: the execution is performed sequentially by all threads
- OpenMP provides several options for synchronization
- Synchronization enforce performance penalties!

Breaking Dependencies

- Example...

- Algorithm from first challenge;

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

```
#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

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

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

- Barrier, master and single example

Detect Race Condition

- Detect race condition can be difficult!

- How to detect race condition?
  - Manual Debugging
  - profile tools: ex: valgrind

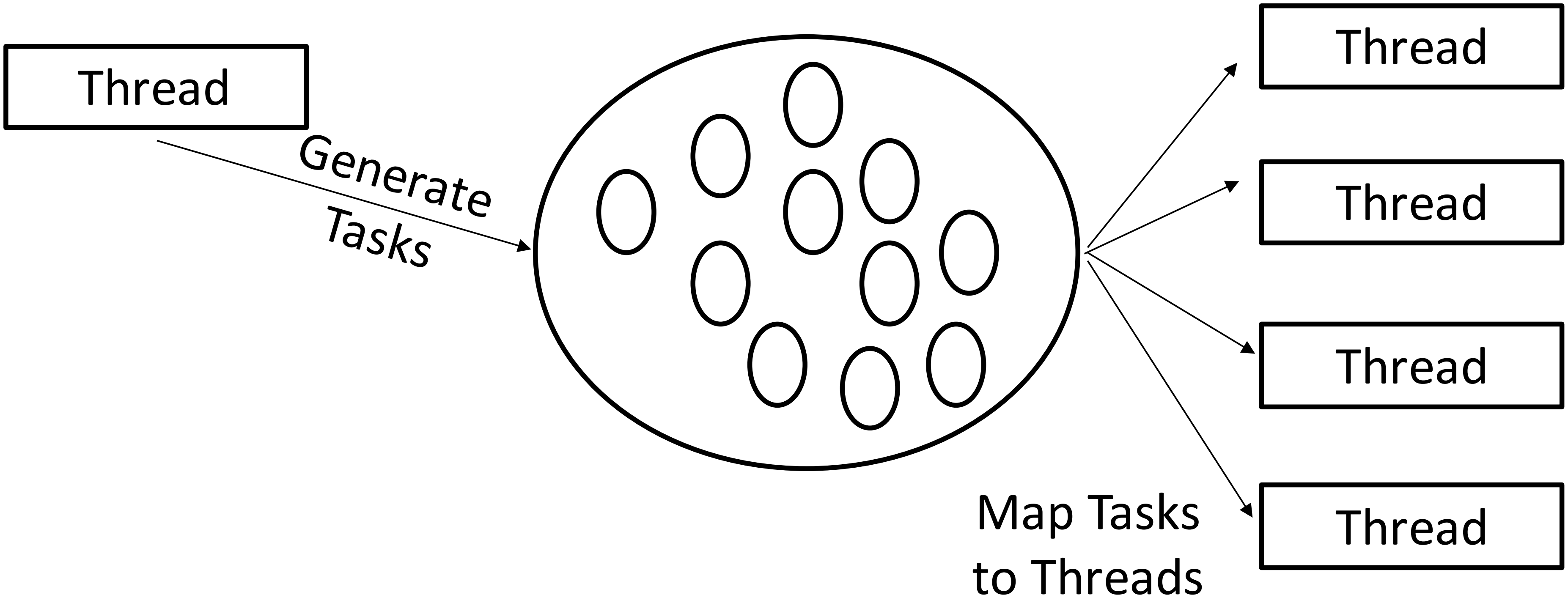
- Valgrind:
  - Compile using flag -g
    - Example: g++ -g -fopenmp sum.cpp -o sum
  - Execute with helgrind
    - valgrind --tool=helgrind ./sum < sum.in &> t

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

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

```
int fib (int n) {
  int x,y;
  if ( n < 2 ) return n;

  x = fib(n-1);

  y = fib(n-2);

  return x+y;
}

int main() {
  int NN = 46;



  fib(NN);

}
```

```
int fib (int n) {
  int x,y;
  if ( n < 2 ) return n;
  #pragma omp task shared(x) if(n>30)
  x = fib(n-1);
  #pragma omp task shared(y) if(n>30)
  y = fib(n-2);
  #pragma omp taskwait
  return x+y;
}

int main() {
  int NN = 460;
  #pragma omp parallel
  {
    #pragma omp master
    fib(NN);
  }
}
```

Hands-on

- Harmonic Progression SUM (WSCAD 2016 warmup)
  - Best place for synchronization
- Apply task parallelism to QuickSort algorithm

Harmonic progression sum

- g++ -O3 -g –fopenmp sum.cpp -o sum

Problem A

Harmonic progression sum

The simplest harmonic progression is

$$^1/_2, ^1/_3, ^1/_4, ^1/_5, \ldots$$

Let $S_n = \sum_{i=1}^{n} (^1/_i)$, compute this sum to arbitrary precision after the decimal point.

Input

The input contains only one test case. The first line contains two values: the first is the number of digits D and the second is the value of N. Consider $(1 \leq D \leq 10^7)$ and $(1 \leq N \leq 10^8)$.

The input must be read from the standard input.

Output

The output contains only one line printing the value of the sum with exact D precision.

The output must be written to the standard output.

Reference

- Structured parallel programming
- Shared memory application programming