OpenMP threading: parallel regions

Paolo Burgio
paolo.burgio@unimore.it

Outline

- Expressing parallelism
  - Understanding parallel threads
- Memory Data management
  - Data clauses
- Synchronization
  - Barriers, locks, critical sections
- Work partitioning
  - Loops, sections, single work, tasks…
- Execution devices
  - Target

Thread-centric exec. models

- Programs written in C are implicitly sequential
  - One thread traverses all of the instructions
  - Any form of parallelism must be explicitly/manually coded
  - Start sequential..then create a team of threads

- E.g., with Pthreads
  - Expose to the programmer "OS-like" threads
  - Units of scheduling

Underlined: Keywords

- Also OpenMP provides a way to do that
  - OpenMP <= 2.5 implements a <u>thread-centric execution model</u>
  - Specify the so-called <u>parallel regions</u>

pragma omp parallel construct

```
#pragma omp parallel [clause [[,]clause]...] new-line
  structured-block
```

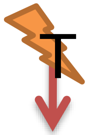
Where clauses can be:

```
if([parallel :] scalar-expression)
num_threads (integer-expression)
default(shared | none)
firstprivate (list)
private (list)
shared (list)
copyin (list)
reduction(reduction-identifier : list)
proc_bind(master | close | spread)
```

Creating a parreg

- Master-slave, fork-join execution model
  - Master thread spawns a team of Slave threads
  - They all perform computation in parallel
  - At the end of the parallel region, implicit barrier

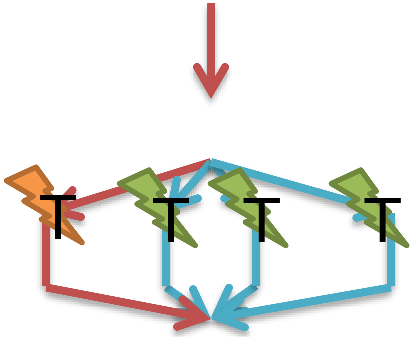
```
int main()
{

  /* Sequential code */

  #pragma omp parallel num_threads(4)
  {


    /* Parallel code */


  } // Parreg end: (implicit) barrier

  /* (More) sequential code */

}
```



Creating a parreg

- Master-slave, fork-join execution model
  - Master thread spawns a team of Slave threads
  - They all perform computation in parallel
  - At the end of the parallel region, implicit barrier

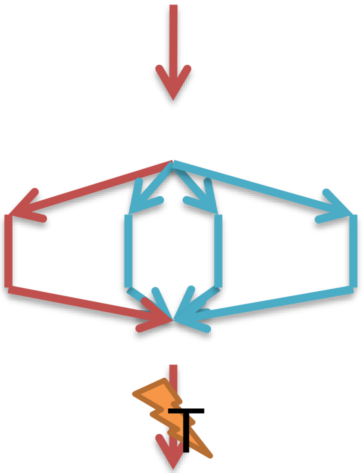
```
int main()
{

  /* Sequential code */

  #pragma omp parallel num_threads(4)
  {


    /* Parallel code */


  } // Parreg end: (implicit) barrier

  /* (More) sequential code */

}
```



Creating a parreg

- Master-slave, fork-join execution model
  - Master thread spawns a team of Slave threads
  - They all perform computation in parallel
  - At the end of the parallel region, implicit barrier

```
int main()
{

  /* Sequential code */

  #pragma omp parallel num_threads(4)
  {


    /* Parallel code */


  } // Parreg end: (implicit) barrier

  /* (More) sequential code */

}
```



Exercise

Let's
code!

- Spawn a team of parallel (OMP)Threads
  - Each printing "Hello Parallel World"
  - No matter how many threads

- Don't forget the `–fopenmp` switch
  - Compiler-dependant!

Compiler	Compiler Options
GNU (**gcc**, g++, gfortran)	`-fopenmp`
Intel (icc ifort)	`-openmp`
Portland Group (pgcc,pgCC,pgf77,pgf90)	`-mp`

Thread control

- OpenMP provides ways to
  - Retrieve thread ID
  - Retrieve number of threads
  - Set the number of threads
  - Specify threads-to-cores affinity (we won't see this)

Get thread ID

omp.h

```
/*
 * The omp_get_thread_num routine returns
 * the thread number, within the current team,
 * of the calling thread.
 */
int omp_get_thread_num(void);
```

- Function call
  - Returns an integer
  - Can be used everywhere where inside your code
    - Also in sequential parts
- Don't forget to `#include <omp.h>`!!
- Master thread (typically) has ID #0

Exercise

Let's code!

- Spawn a team of parallel (OMP)Threads
  - Each printing "Hello Parallel World. I am thread #<tid>"
  - Also, print "Hello Sequential World. I am thread #<tid>" before and after parreg
  - What do you see?

Get the number of threads

omp.h

```c
/*
 * The omp_get_num_threads routine returns
 * the number of threads in the current team.
 */
int omp_get_num_threads(void);
```

- Function call
  - Returns an integer
  - Can be used everywhere where inside your code
    - Also in sequential parts
  - Don't forget to `#include <omp.h>`!!
- BTW
  - ...thread ID from `omp_get_thread_num` is always < this value..

Exercise

Let's
code!

- Spawn a team of parallel (OMP)Threads
  - Each printing "Hello Parallel World. I am thread #<tid> out of <num>"
  - Also, print "Hello Sequential World. I am thread #<tid> out of <num>" before and after parreg
  - What do you see?

Set the number of threads

- "This, we already saw ☺"
  - NO(t completely)!

- In OpenMP, several ways to do this
  - Implementation-specific default

- In order of priority..
  1. OpenMP `num_threads` clause
  2. Function APIs (explicit function call)
  3. Environmental vars (at the OS level)

Set the number of threads (3)

```
# The OMP_NUM_THREADS environment variable sets
# the number of threads to use for parallel regions

export OMP_NUM_THREADS=4
```

Set the number of threads (2)

omp.h

```c
/*
 * The omp_set_num_threads routine affects the number of threads
 * to be used for subsequent parallel regions that do not specify
 * a num_threads clause, by setting the value of the first
 * element of the nthreads-var ICV of the current task.
 */
void omp_set_num_threads(int num_threads);
```

- Function call
  - Accepts an integer
  - Can be used everywhere where inside your code
    - Also in sequential parts
- Don't forget to `#include <omp.h>`!!
- Overrides value from `OMP_NUM_THREADS`
  - Affects all of the subsequent parallel regions

Set the number of threads (1)

```
#pragma omp parallel [clause [[,]clause]...] new-line
  structured-block
```

Where clauses can be:

```
if([parallel :] scalar-expression)
num_threads (integer-expression)
default(shared | none)
firstprivate (list)
private (list)
shared (list)
copyin (list)
reduction(reduction-identifier : list)
proc_bind(master | close | spread)
```

Exercise

Let's code!

- Spawn a team of parallel (OMP)Threads
  - Each printing "Hello Parallel World. I am thread #<tid> out of <num>"
  - Also, print "Hello Sequential World. I am thread #<tid> out of <num>" before and after parreg
  - Play with
    - `OMP_NUM_THREADS`
    - `omp_set_num_threads`
    - `num_threads`
- Do it at home

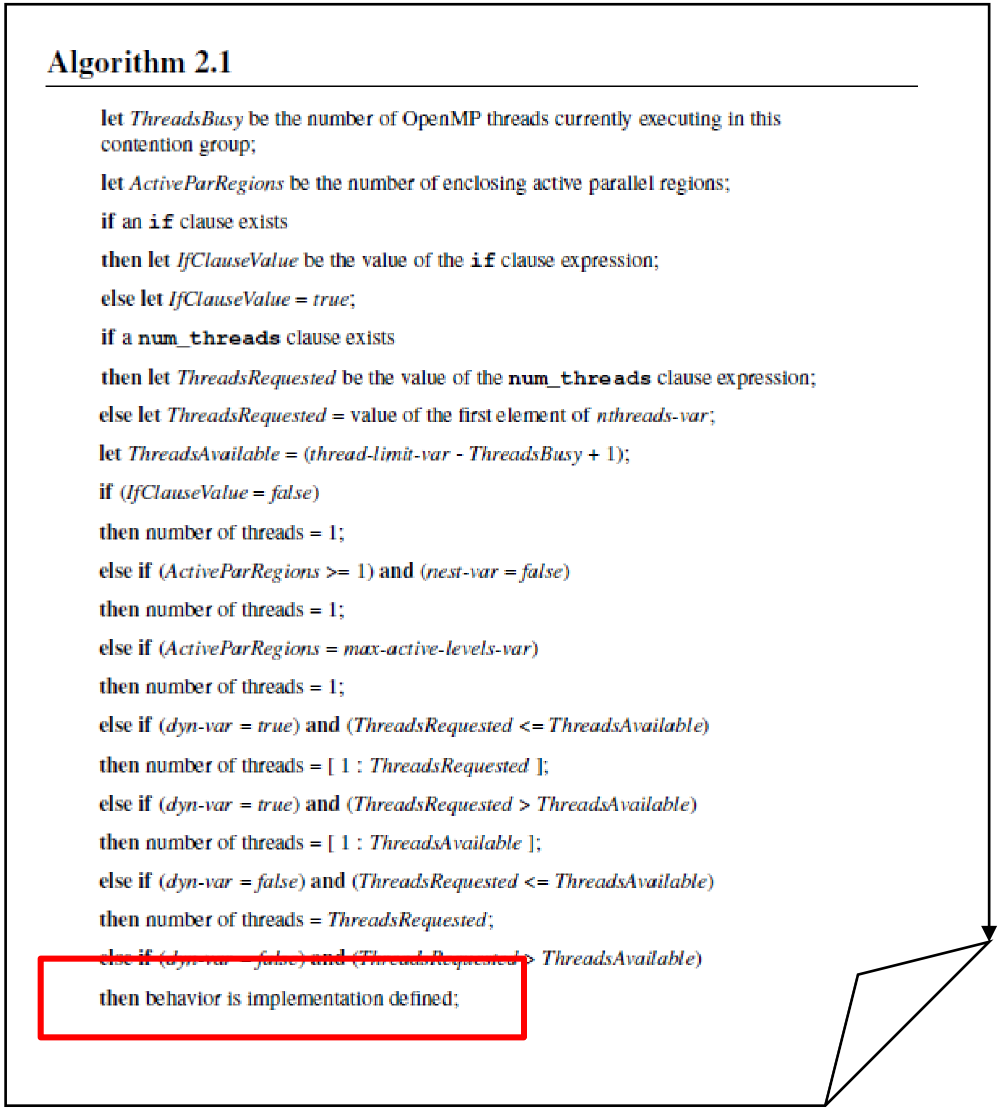
The `if` clause

```
#pragma omp parallel [clause [[,]clause]...] new-line
  structured-block

Where clauses can be:

if([parallel :] scalar-expression)
num_threads (integer-expression)
default(shared | none)
firstprivate (list)
private (list)
shared (list)
copyin (list)
reduction(reduction-identifier : list)
proc_bind(master | close | spread)
```

- If `scalar-expression` is `false`, then spawn a single-thread region
- We will see it also in other constructs…
  - "Can be used in combined constructs, in this case programmer must specify which one it refers to (in this case, with the `parallel` specifier)"

Algorithm that determines #threads

- OpenMP Specifications
  - Section 2.1
  - http://www.openmp.org



Algorithm 2.1

```
let $ThreadsBusy$ be the number of OpenMP threads currently executing in this
contention group;
let $ActiveParRegions$ be the number of enclosing active parallel regions;
if an if clause exists
then let $IfClauseValue$ be the value of the if clause expression;
else let $IfClauseValue = true$;
if a num_threads clause exists
then let $ThreadsRequested$ be the value of the num_threads clause expression;
else let $ThreadsRequested$ = value of the first element of $nthreads\text{-}var$;
let $ThreadsAvailable = (thread\text{-}limit\text{-}var - ThreadsBusy + 1)$;
if $(IfClauseValue = false)$
then number of threads = 1;
else if $(ActiveParRegions >= 1)$ and $(nest\text{-}var = false)$
then number of threads = 1;
else if $(ActiveParRegions = max\text{-}active\text{-}levels\text{-}var)$
then number of threads = 1;
else if $(dyn\text{-}var = true)$ and $(ThreadsRequested <= ThreadsAvailable)$
then number of threads = [ 1 : $ThreadsRequested$ ];
else if $(dyn\text{-}var = true)$ and $(ThreadsRequested > ThreadsAvailable)$
then number of threads = [ 1 : $ThreadsAvailable$ ];
else if $(dyn\text{-}var = false)$ and $(ThreadsRequested <= ThreadsAvailable)$
then number of threads = $ThreadsRequested$;
else if $(dyn\text{-}var = false)$ and $(ThreadsRequested > ThreadsAvailable)$
then behavior is implementation defined;
```

Even more control…

- OpenMP provides fine-grain tuning of all the main "control knobs"
  - Dynamic thread number adjustment
  - Nesting level
  - Threads stack size
  - …

- More and more with every new version of specifications

Nested parallel regions

- One can create a parallel region within a parallel region
  - A new team of thread is created
- Enabled-disabled via environmental var, or library call
- Easy to lose control..
  - Too many threads!
  - Their number explodes
  - Be ready to debug..

Dynamic # threads adjustment

- The OpenMP implementation might decide to dynamically adjust the number of thread within a parreg
  - Aka the team size
  - Under heavy load might be reduced

- Also this can be disabled

Threads stack size

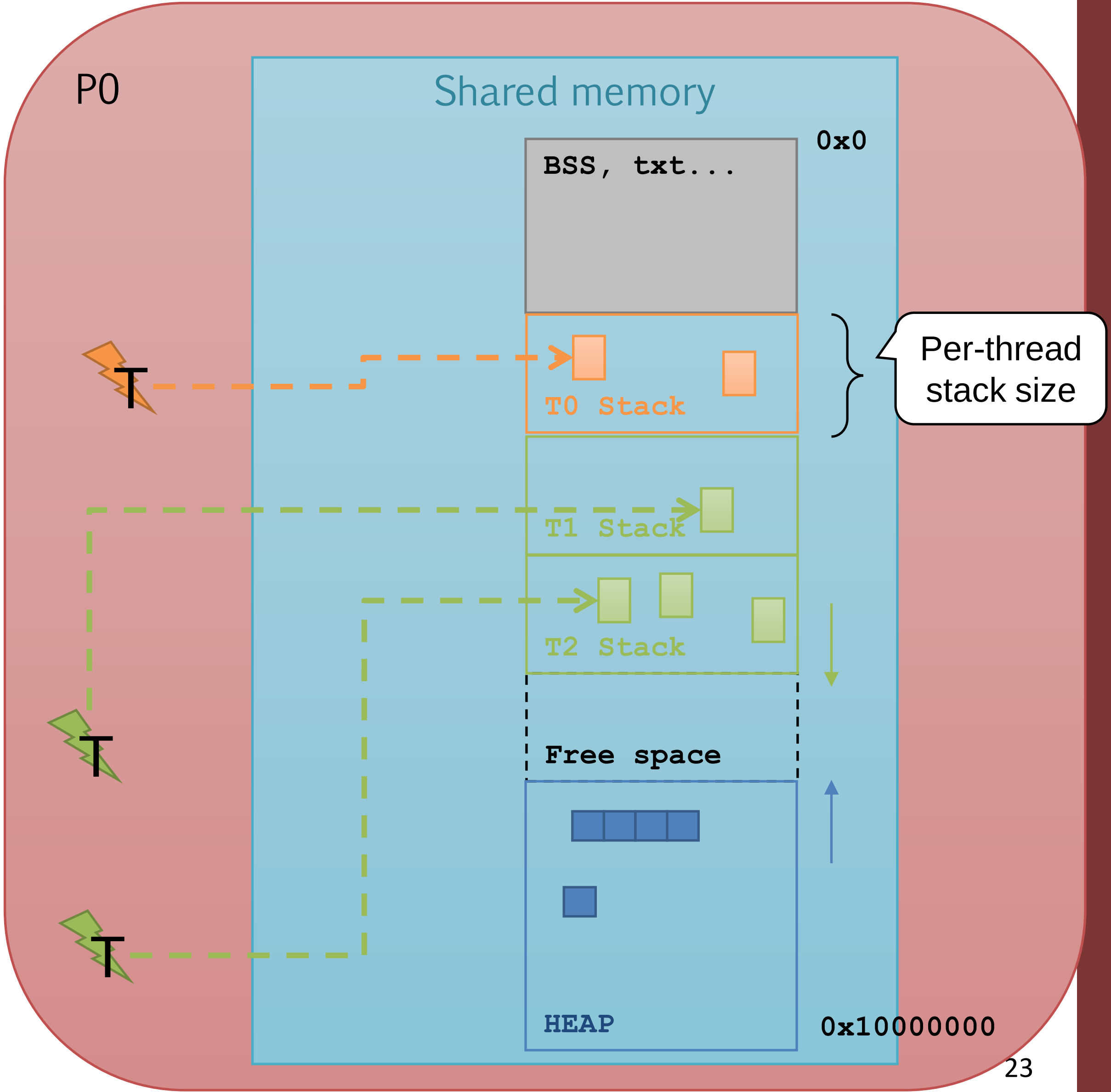
- Can specify low-level details such as the stack size
  - Why only via environmental var?

Bash Shell

```
# The OMP_STACKSIZE environment variable controls the size of the stack
# for threads created by the OpenMP implementation,
# by setting the value of the stacksize-var ICV.
# The environment variable does not control the size of the stack
# for an initial thread.
# The value of this environment variable takes the form:
#      size | sizeB | sizeK | sizeM | sizeG

setenv OMP_STACKSIZE 2000500B
setenv OMP_STACKSIZE "3000 k "
setenv OMP_STACKSIZE 10M
setenv OMP_STACKSIZE " 10 M "
setenv OMP_STACKSIZE "20 m "
setenv OMP_STACKSIZE " 1G"
setenv OMP_STACKSIZE 20000
```

Process (shared) memory space

- Per-thread stack
  - Still, accessible
  - `auto` vars
  - Stack overflow!!
- Common heap
  - `malloc/new`
- BSS, text
  - …



Per-thread stack size

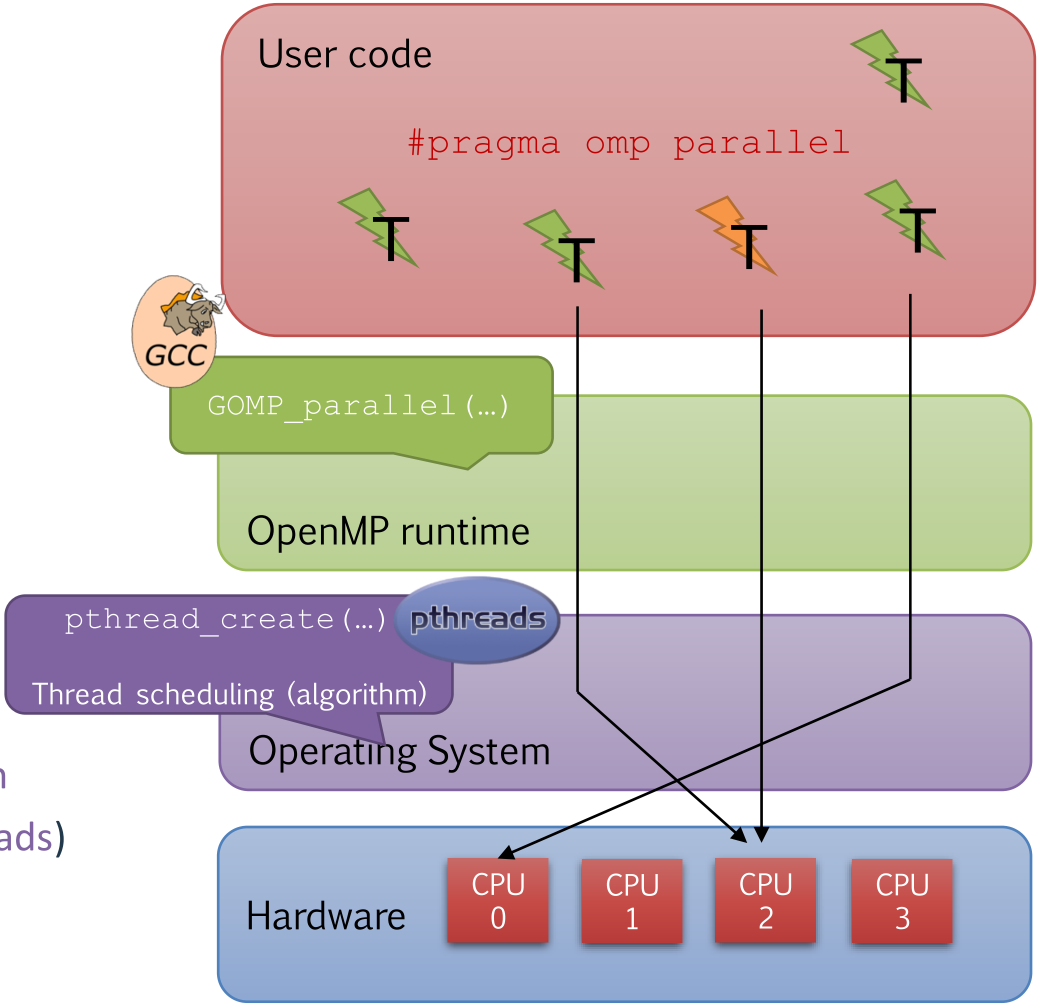
Under the hood

- You have control on # threads
  - Partly
- You have parial control on where the threads are scheduled
  - Affinity
- You have no control on the actual scheduling!
  - Demanded to OS + runtime
- …"OS and runtime"?

OpenMP software stack

Multi-layer stack, engineered for portability

- Application code
  - Compliant to OMP standard
- Runtime (e.g., GCC-OpenMP)
  - Provides services for parallelism
  - Compiler replaces pragma with runtime-specific function calls
- OS (e.g., Linux)
  - Provides basic services
  - Threading, memory mgmt, synch
  - Can be standardized (e.g., PThreads)



How to run the examples

Let's code!

- Download the `Code/` folder from the course website

- Compile

```
$ gcc -fopenmp code.c -o code
```

- Run (Unix/Linux)

```
$ ./code
```

- Run (Win/Cygwin)

```
$ ./code.exe
```

References

- "Calcolo parallelo" website
  - http://hipert.unimore.it/people/paolob/pub/PhD/index.html
- My contacts
  - paolo.burgio@unimore.it
  - http://hipert.mat.unimore.it/people/paolob/
- Useful links
  - http://www.google.com
  - http://www.openmp.org
  - https://gcc.gnu.org/
- A "small blog"
  - http://www.google.com