Parallel Programming with OpenMP

Alejandro Duran

Barcelona Supercomputing Center

Agenda

-	Thursday
10:00 - 11:15	OpenMP Basics
11:00 - 11:30	Break
11:30 - 13:00	Hands-on (I)
13:00 - 14:30	Lunch
14:30 - 15:15	Task parallelism in OpenMP
15:15 - 17:00	Hands-on (II)
-	Friday
10:00 - 11:00	Data parallelism in OpenMP
11:00 - 11:30	Break
11:30 - 13:00	Hands-on (III)
13:00 - 14:30	Lunch
14:30 - 15:00	Other OpenMP topics
15:00 - 16:00	Hands-on (IV)
16:00 - 16:30	OpenMP in the future

Part I

OpenMP Basics

Outline

- OpenMP Overview
- The OpenMP model
- Writing OpenMP programs
- Creating Threads
- Data-sharing attributes
- Synchronization

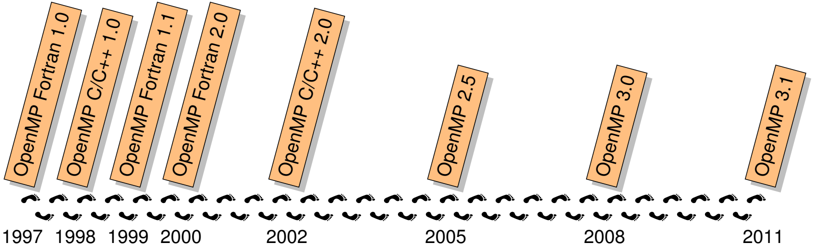
Outline

What is OpenMP?

- It's an API extension to the C, C++ and Fortran languages to write parallel programs for shared memory machines
  - Current version is 3.0 (May 2008)
  - Supported by most compiler vendors
    - Intel,IBM,PGI,Sun,Cray,Fujitsu,HP,GCC,...
- Maintained by the Architecture Review Board (ARB), a consortium of industry and academia

http://www.openmp.org

A bit of history



Advantages of OpenMP

- Mature standard and implementations
  - Standardizes practice of the last 20 years
- Good performance and scalability
- Portable across architectures
- Incremental parallelization
- Maintains sequential version
- (mostly) High level language
  - Some people may say a medium level language :-)
- Supports both task and data parallelism
- Communication is implicit

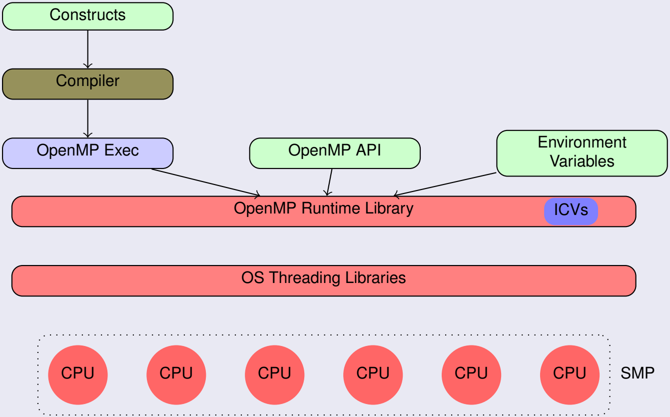
Disadvantages of OpenMP

- Communication is implicit
- Flat memory model
- Incremental parallelization creates false sense of glory/failure
- No support for accelerators
- No error recovery capabilities
- Difficult to compose
- Lacks high-level algorithms and structures
- Does not run on clusters

Outline

OpenMP at a glance

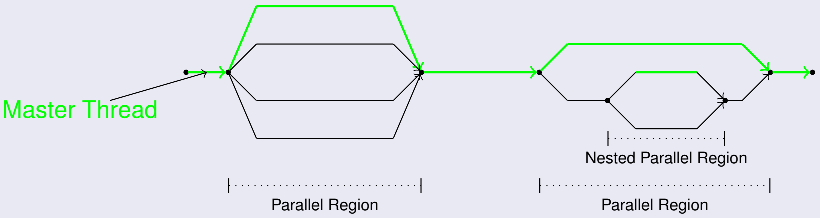
OpenMP components



Execution model

Fork-join model

- OpenMP uses a fork-join model
  - The master thread spawns a team of threads that joins at the end of the parallel region
  - Threads in the same team can collaborate to do work



Memory model

- OpenMP defines a relaxed memory model
  - Threads can see different values for the same variable
  - Memory consistency is only guaranteed at specific points
  - Luckily, the default points are usually enough
- Variables can be shared or private to each thread

Outline

OpenMP directives syntax

In Fortran

Through a specially formatted comment:

```
sentinel construct [clauses]
```

where sentinel is one of:

- `!$OMP` or `C$OMP` or `*$OMP` in fixed format
- `!$OMP` in free format

In C/C++

Through a compiler directive:

```
#pragma omp construct [clauses]
```

- OpenMP syntax is ignored if the compiler does not recognize OpenMP

OpenMP directives syntax

In Fortran

Through a specially formatted comment:

```
sentinel construct [clauses]
```

where sentinel is one of:

- `!$OMP` or `C$OMP` or `*$OMP` in fixed format
- `!$OMP` in free format

In C/C++

Through a compiler directive:

```
#pragma omp construct [clauses]
```

- OpenMP syntax is ignored if the compiler does not recognize

We'll be using C/C++ syntax through this tutorial

Headers/Macros

C/C++ only

- omp.h contains the API prototypes and data types definitions
- The _OPENMP is defined by OpenMP enabled compiler
  - Allows conditional compilation of OpenMP

Fortran only

- The omp_lib module contains the subroutine and function definitions

Structured Block

Definition

Most directives apply to a structured block:

- Block of one or more statements
- One entry point, one exit point
  - No branching in or out allowed
- Terminating the program is allowed

Hello world!

Example

```
int id;
char *message = "Hello␣world!";

#pragma omp parallel private(id)
{
  id = omp_get_thread_num();
  printf("Thread␣%d␣says:␣%s\n", id, message);
}
```

Hello world!

Example

```
int id;
char *message = "Hello world!";

#pragma omp parallel private(id)
{
  id = omp_get_thread_num();
  printf("Thread %d says: %s\n", id, message);
}
```

Directive

Clause

API call

Structured block

Outline

The parallel construct

Directive

```
#pragma omp parallel [clauses]
    structured block
```

where clauses can be:

- **num_threads(*expression*)**
- **if(*expression*)**
- shared(*var-list*) ← Coming shortly!
- private(*var-list*) ← Coming shortly!
- firstprivate(*var-list*) ← Coming shortly!
- default(none|shared| private | firstprivate) ← Coming shortly! (private, firstprivate: Only in Fortran)
- reduction(*var-list*) ← We'll see it later
- copyin(*var-list*) ← Not today

The parallel construct

Specifying the number of threads

- The number of threads is controlled by an internal control variable (**`ICV`**) called **`nthreads-var`**.
- When a parallel construct is found a parallel region with a maximum of **`nthreads-var`** is created
  - Parallel constructs can be nested creating nested parallelism
- The **`nthreads-var`** can be modified through
  - the **`omp_set_num_threads`** API called
  - the **`OMP_NUM_THREADS`** environment variable
- Additionally, the **`num_threads`** clause causes the implementation to ignore the ICV and use the value of the clause for that region.

The parallel construct

Avoiding parallel regions

- Sometimes we only want to run in parallel under certain conditions
  - E.g., enough input data, not running already in parallel, ...
- The **`if`** clause allows to specify an *expression*. When evaluates to false the **`parallel`** construct will only use 1 thread
  - Note that still creates a new team and data environment

Hello world!

Example

```c
int id;
char *message = "Hello world!";

#pragma omp parallel private(id)
{
  id = omp_get_thread_num();
  printf("Thread %d says: %s\n", id, message);
}
```

Hello world!

Example

```
int id;
char *message = "Hello␣world!";

#pragma omp parallel private(id)
{
  id = omp_get_thread_num();
  printf("Thread␣%d␣says:␣%s\n", id, message);
}
```

Creates a parallel region of **OMP_NUM_THREADS**

All threads execute the same code

Hello world!

Example

```
int id;
char *message = "Hello␣world!";

#pragma omp parallel private(id)
{
  id = omp_get_thread_num();
  printf("Thread␣%d␣says:␣%s\n", id, message);
}
```

id is private to each thread

Each thread gets its id in the team

Hello world!

Example

```
int id;
char *message = "Hello␣world!";

#pragma omp parallel private(id)
{
  id = omp_get_thread_num();
  printf("Thread␣%d␣says:␣%s\n", id, message);
}
```

message is shared among all threads

Putting it together

Example

```
void main () {
  #pragma omp parallel
     ...
  omp_set_num_threads(2);
  #pragma omp parallel
     ...
  #pragma omp parallel num_threads(random()%4+1) if(0)
     ...
}
```

Putting it together

Example

```
void main () {
  #pragma omp parallel
    ...
  omp_set_num_threads(2);
  #pragma omp parallel
    ...
  #pragma omp parallel num_threads(random()%4+1) if(0)
    ...
}
```

An unknown number of threads here. Use **OMP_NUM_THREADS**

Putting it together

Example

```
void main () {
  #pragma omp parallel
    ...
  omp_set_num_threads(2);
  #pragma omp parallel
    ...
  #pragma omp parallel num_threads(random()%4+1) if(0)
    ...
}
```

A team of two threads here.

Putting it together

Example

```
void main () {
  #pragma omp parallel
     ...
  omp_set_num_threads(2);
  #pragma omp parallel
     ...
  #pragma omp parallel num_threads(random()%4+1) if(0)
     ...
}
```

A team of 1 thread here.

API calls

Other useful routines

int **omp_get_num_threads**()	Returns the number of threads in the current team
int **omp_get_thread_num**()	Returns the id of the thread in the current team
int **omp_get_num_procs**()	Returns the number of processors in the machine
int **omp_get_max_threads**()	Returns the maximum number of threads that will be used in the next parallel region
double **omp_get_wtime**()	Returns the number of seconds since an arbitrary point in the past

Outline

Data environment

A number of clauses are related to building the data environment that the construct will use when executing.

- **`shared`**
- **`private`**
- **`firstprivate`**
- **`default`**
- **`threadprivate`**
- lastprivate
- reduction
- copyin
- copyprivate

We'll see them later (lastprivate, reduction)

Out of our scope today (copyin, copyprivate)

Data-sharing attributes

Shared

When a variable is marked as **shared**, the variable inside the construct is the same as the one outside the construct.

- In a parallel construct this means all threads see the same variable
  - but not necessarily the same value
- Usually need some kind of synchronization to update them correctly
  - OpenMP has consistency points at synchronizations

Data-sharing attributes

Example

```
int x=1;
#pragma omp parallel shared(x) num_threads(2)
{
   x++;
   printf("%d\n",x);
}
printf("%d\n",x);
```

Data-sharing attributes

Example

```
int x=1;
#pragma omp parallel shared(x) num_threads(2)
{
    x++;
    printf("%d\n",x);
}
printf("%d\n",x);
```

Prints 2 or 3

Data-sharing attributes

Private

When a variable is marked as **`private`**, the variable inside the construct is a new variable of the same type with an undefined value.

- In a parallel construct this means all threads have a different variable
- Can be accessed without any kind of synchronization

Data-sharing attributes

Example

```
int x=1;
#pragma omp parallel private(x) num_threads(2)
{
   x++;
   printf("%d\n",x);
}
printf("%d\n",x);
```

Data-sharing attributes

Example

```
int x=1;
#pragma omp parallel private(x) num_threads(2)
{
   x++;
   printf("%d\n",x);   <-- Can print anything
}
printf("%d\n",x);
```

Data-sharing attributes

Example

```
int x=1;
#pragma omp parallel private(x) num_threads(2)
{
   x++;
   printf("%d\n",x);
}
printf("%d\n",x);
```

Prints 1

Data-sharing attributes

Firstprivate

When a variable is marked as **`firstprivate`**, the variable inside the construct is a new variable of the same type but it is initialized to the original variable value.

- In a parallel construct this means all threads have a different variable with the same initial value
- Can be accessed without any kind of synchronization

Data-sharing attributes

Example

```
int x=1;
#pragma omp parallel firstprivate(x) num_threads(2)
{
   x++;
   printf("%d\n",x);
}
printf("%d\n",x);
```

Data-sharing attributes

Example

```
int x=1;
#pragma omp parallel firstprivate(x) num_threads(2)
{
   x++;
   printf("%d\n",x);   <-- Prints 2 (twice)
}
printf("%d\n",x);
```

Data-sharing attributes

Example

```
int x=1;
#pragma omp parallel firstprivate(x) num_threads(2)
{
   x++;
   printf("%d\n",x);
}
printf("%d\n",x);
```

Prints 1

Data-sharing attributes

What is the default?

- Static/global storage is **`shared`**
- Heap-allocated storage is **`shared`**
- Stack-allocated storage inside the construct is **`private`**
- Others
  - If there is a **`default`** clause, what the clause says
    - **`none`** means that the compiler will issue an error if the attribute is not explicitly set by the programmer
  - Otherwise, depends on the construct
    - For the **`parallel`** region the default is **`shared`**

Data-sharing attributes

Example

```
int x,y;
#pragma omp parallel private(y)
{
    x =
    y =
    #pragma omp parallel private(x)
    {
        x =
        y =
    }
}
```

Data-sharing attributes

Example

```
int x,y;
#pragma omp parallel private(y)
{
    x =
    y =
    #pragma omp                   x)
    {
        x =
        y =
    }
}
```

x is **shared**

y is **private**

Data-sharing attributes

Example

```
int x,y;
#pragma omp parallel private(y)
{
    x =
    y =
    #pragma omp parallel private(x)
    {
        x =     <- x is private
        y =     <- y is shared
    }
}
```

Threadprivate storage

The threadprivate construct

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

- Can be applied to:
  - Global variables
  - Static variables
  - Class-static members
- Allows to create a per-thread copy of “global” variables.
- **threadprivate** storage persist across **parallel** regions if the number of threads is the same

Threadprivate persistence across nested regions is complex

Threaprivate storage

Example

```
char* foo ()
{
  static char buffer[BUF_SIZE];


  ...

  return buffer;
}

void bar ()
{
   #pragma omp parallel
   {
      char * str = foo();
      str[0] = random();
   }
}
```

Threaprivate storage

Example

```
char* foo ()
{
  static char buffer[BUF_SIZE];


  ...

  return buffer;
}

void bar ()
{
   #pragma omp parallel
   {
      char * str = foo();
      str[0] = random();
   }
}
```

Unsafe. All threads access the same *buffer*

Threaprivate storage

Example

```
char* foo ()
{
  static char buffer[BUF_SIZE];
  #pragma omp threadprivate(buffer)

  ...

  return buffer;
}

void bar ()
{
   #pragma omp parallel
   {
      char * str = foo();
      str[0] = random();
   }
}
```

Now foo can be called safely by multiple threads at the same time

Outline

- OpenMP Overview
- The OpenMP model
- Writing OpenMP programs
- Creating Threads
- Data-sharing attributes
- Synchronization

Why synchronization?

Mechanisms

Threads need to synchronize to impose some ordering in the sequence of actions of the threads. OpenMP provides different synchronization mechanisms:

- **`barrier`**
- **`critical`**
- **`atomic`**
- taskwait
- ordered
- locks

We'll see them later

Thread Barrier

The barrier construct

```
#pragma omp barrier
```

- Threads cannot proceed past a barrier point until all threads reach the barrier AND all previously generated work is completed
- Some constructs have an implicit `barrier` at the end
  - E.g., the `parallel` construct

Barrier

Example

```
#pragma omp parallel
{
   foo();
   #pragma omp barrier
   bar();
}
```

Barrier

Example

```
#pragma omp parallel
{
   foo();
   #pragma omp barrier
   bar();
}
```

Forces all *foo* occurrences too happen before all *bar* occurrences

Barrier

Example

```
#pragma omp parallel
{
   foo();
   #pragma omp barrier
   bar();
}
```

Implicit barrier at the end of the **parallel** region

Exclusive access

The critical construct

```
#pragma omp critical [(name)]
    structured block
```

- Provides a region of mutual exclusion where only one thread can be working at any given time.
- By default all critical regions are the same, but you can provide them with names
  - Only those with the same name synchronize

Critical construct

Example

```
int x=1;
#pragma omp parallel num_threads(2)
{
   #pragma omp critical
      x++;
}
printf("%d\n",x);
```

Critical construct

Example

```
int x=1;
#pragma omp parallel num_threads(2)
{
   #pragma omp crit
      x++;
}
printf("%d\n",x);
```

Only one thread at a time here

Critical construct

Example

```
int x=1;
#pragma omp parallel num_threads(2)
{
   #pragma omp critical
      x++;
}
printf("%d\n",x);
```

Only one thread at a time here

Prints 3!

Critical construct

Example

```
int x=1,y=0;
#pragma omp parallel num_threads(4)
{
   #pragma omp critical (x)
      x++;
   #pragma omp critical (y)
      y++;
}
```

Critical construct

Example

```
int x=1,y=0;
#pragma omp parallel num_threads(4)
{
   #pragma omp critical (x)
      x++;
   #pragma omp critical (y)
      y++;
}
```

Different names: One thread can update *x* while another updates *y*

Exclusive access

The atomic construct

```
#pragma omp atomic
    expression
```

- Provides an special mechanism of mutual exclusion to do read & update operations
- Only supports simple read & update expressions
  - E.g., x ++, x -= foo()
- Only protects the read & update part
  - foo() not protected
- Usually much more efficient than a `critical` construct
- Not compatible with `critical`

Atomic construct

Example

```
int x=1;
#pragma omp parallel num_threads(2)
{
   #pragma omp atomic
      x++;
}
printf("%d\n",x);
```

Atomic construct

Example

```
int x=1;
#pragma omp parallel num_threads(2)
{
    #pragma omp atomic
        x++;
}
printf("%d\n",x);
```

Only one thread at a time updates *x* here

Atomic construct

Example

```
int x=1;
#pragma omp parallel num_threads(2)
{
   #pragma omp atomic
      x++;
}
printf("%d\n",x);
```

Prints 3!

Atomic construct

Example

```
int x=1;
#pragma omp parallel num_threads(2)
{
   #pragma omp critical
      x++;
   #pragma omp atomic
      x++;
}
printf("%d\n",x);
```

Atomic construct

Example

```
int x=1;
#pragma omp parallel num_threads(2)
{
   #pragma omp critical
      x++;
   #pragma omp atomic
      x++;
}
printf("%d\n",x);
```

Different threads can update *x* at the same time!

Atomic construct

Example

```
int x=1;
#pragma omp parallel num_threads(2)
{
   #pragma omp critical
      x++;
   #pragma omp atomic
      x++;
}
printf("%d\n",x);
```

Prints 3,4 or 5 :(

Coffee time! :-)

Part II

Hands-on (I)

Outline

- Setup
- Hello world!
- Other

Outline

- Setup
- Hello world!
- Other

Hands-on preparation

Environment

We'll be using ...

- an SGI Altix 4700 System
  - 128 cpus Dual Core Montecito(IA-64). Each one of the 256 cores works at 1,6 GHz, with a 8MB L3 cache and 533 MHz Bus.
    - Unfortunately will be using just 8 of them :-)
  - 2.5 TB RAM.
  - 2 internal SAS disks of 146 GB at 15000 RPMs
  - 12 external SAS disks of 300 GB at 10000 RPMS
- Intel's compiler version 11.0
  - Full support of OpenMP 3.0
  - Other vendors that support 3.0: PGI, IBM, SUN, GCC

Hands-on preparation

Ready...

Copy the exercises from my home:

```
$ cp -a
~aduran/Prace_OpenMP_Handson_1/hello .
```

Hands-on preparation

Ready...

Copy the exercises from my home:

```
$ cp -a
~aduran/Prace_OpenMP_Handson_1/hello .
```

Go!

Now enter the hello directory to start the fun :-)

Outline

- Setup
- Hello world!
- Other

First exercise

Hello world!

Compile

1. Edit the Makefile in the directory and answer the following questions:
   - Which is the compiler name?
   - Which flag does activate OpenMP?
2. Run make and check that it generates a hello program.

First exercise

Hello world!

Run

1. Edit the file hello.c and try to figure out what is going to be the output of the following commands:

```
$ ./hello
```

```
$ OMP_NUM_THREADS=2 ./hello
```

```
$ OMP_NUM_THREADS=4 ./hello
```

2. Now run them. Were you right?

First exercise

Hello world!

Being oneself

Now modify our hello program so that each thread generates a message with its id

Tip: Use `omp_get_thread_num()`

First exercise

Hello world!

Generate extra info

Now modify our hello program so before any thread says hello, it outputs the following information:

1. The number of processors in the system
2. The number of threads that will be available in the parallel region

First exercise

Hello world!

Measuring time

Measure the time that it takes to execute the `parallel` region and output it at the end of the program.

Tip: Use `omp_get_wtime()`

First exercise

One at a time!

Extend the program so that each thread uses C rand to get a random number. Accumulate those numbers in a shared variable and output the result at the end of the program.

- Should the result always be the same given the same seed and number of threads?

Outline

- Setup
- Hello world!
- Other

Second exercise

1. Edit the sync.c file
2. Is correct the access to the variable *x*?
3. Fix it using a critical construct. Compile it:

```
$ make sync
```

4. Run it from 1 to 4 threads and observe how it changes the average time
5. Now change the critical construct with an atomic one.
6. Run it from 1 to 4 threads. How does the averages times compare to the previous ones?

Some more...

One for each thread

1. Compile the tp.c program:

```
$ make tp
```

2. The program is suposed to print three times the tread id
3. Run it with 4 threads. Observe the results
4. Edit tp.c and fix it so it behaves correctly
5. How did you solve the problem for *x*?
6. How did you solve the problem for *y*?
7. If you solved them in the same way, then rethink what you did for *x*

Bon appétit!*

*Disclaimer: actual food may differ from the image! :-)

Part III

Outline

Part IV

The OpenMP Tasking Model

Outline

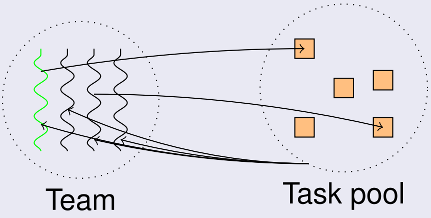
- OpenMP tasks
- Task synchronization
- The single construct
- Task clauses
- Common tasking problems

Outline

- OpenMP tasks
- Task synchronization
- The single construct
- Task clauses
- Common tasking problems

Task parallelism in OpenMP

Task parallelism model



- Parallelism is extracted from “several” pieces of code
- Allows to parallelize very unstructured parallelism
  - Unbounded loops, recursive functions, ...

What is a task in OpenMP ?

- Tasks are work units whose execution may be deferred
  - they can also be executed immediately
- Tasks are composed of:
  - code to execute
  - a data environment
    - Initialized at creation time
  - internal control variables (ICVs)
- Threads of the team cooperate to execute them

Creating tasks

The task construct

```
#pragma omp task [clauses]
    structured block
```

Where clauses can be:

- shared
- private
- firstprivate
    - Values are captured at creation time
- default
- **`if(expression)`**
- **`untied`**

When are task created?

- `Parallel` regions create tasks
  - One implicit task is created and assigned to each thread
    - So all task-concepts have sense inside the parallel region
- Each thread that encounters a `task` construct
  - Packages the code and data
  - Creates a new explicit task

Default task data-sharing attributes

When there are no clauses ...

If no default clause

- Implicit rules apply
  - e.g., global variables are shared
- Otherwise...
  - **`firstprivate`**
  - **`shared`** attribute is lexically inherited

Task default data-sharing attributes

In practice...

Example

```
int a;
void foo() {
  int b,c;
  #pragma omp parallel shared(b)
  #pragma omp parallel private(b)
  {
        int d;
        #pragma omp task
        {
            int e;

            a =
            b =
            c =
            d =
            e =
}}}
```

Task default data-sharing attributes

In practice...

Example

```
int a;
void foo() {
  int b,c;
  #pragma omp parallel shared(b)
  #pragma omp parallel private(b)
  {
        int d;
        #pragma omp task
        {
            int e;

            a = shared
            b =
            c =
            d =
            e =
}}}
```

Task default data-sharing attributes

In practice...

Example

```
int a;
void foo() {
  int b,c;
  #pragma omp parallel shared(b)
  #pragma omp parallel private(b)
  {
        int d;
        #pragma omp task
        {
            int e;

            a = shared
            b = firstprivate
            c =
            d =
            e =
}}}
```

Task default data-sharing attributes

In practice...

Example

```
int a;
void foo() {
  int b,c;
  #pragma omp parallel shared(b)
  #pragma omp parallel private(b)
  {
        int d;
        #pragma omp task
        {
            int e;

            a = shared
            b = firstprivate
            c = shared
            d =
            e =
}}}
```

Task default data-sharing attributes

In practice...

Example

```
int a;
void foo() {
  int b,c;
  #pragma omp parallel shared(b)
  #pragma omp parallel private(b)
  {
        int d;
        #pragma omp task
        {
            int e;

            a = shared
            b = firstprivate
            c = shared
            d = firstprivate
            e =
}}}
```

Task default data-sharing attributes

In practice...

Example

```
int a;
void foo() {
  int b,c;
  #pragma omp parallel shared(b)
  #pragma omp parallel private(b)
  {
        int d;
        #pragma omp task
        {
            int e;

            a = shared
            b = firstprivate
            c = shared
            d = firstprivate
            e = private
}}}
```

Task default data-sharing attributes

In practice...

Example

```
int a;
void foo() {
  int b,c;
  #pragma omp parallel shared(b)
  #pragma omp parallel private(b)
  {
        int d;
        #pragma omp task
        {
            int e;

            a = shared
            b = firstprivate
            c = shared
            d = firstprivate
            e = private
}}}
```

Tip: `default(none)` is your friend if you do not see it clearly

List traversal

Example

```
void traverse_list ( List l )
{
  Element e;
  for ( e = l->first; e ; e = e->next )
    #pragma omp task
      process(e);
}
```

e is **firstprivate**

Outline

Task synchronization

There are two main constructs to synchronize tasks:

- **`barrier`**
  - Remember: all previous work (including tasks) must be completed
- **`taskwait`**

Waiting for children

The taskwait construct

```
#pragma omp taskwait
```

Suspends the current task until all children tasks are completed

- Just direct children, not descendants

Taskwait

Example

```
void traverse_list ( List l )
{
  Element e;
  for ( e = l->first; e ; e = e->next )
     #pragma omp task
       process(e);

  #pragma omp taskwait

}
```

Taskwait

Example

```
void traverse_list ( List l )
{
  Element e;
  for ( e = l->first; e ; e = e->next )
     #pragma omp task
       process(e);

  #pragma
}
```

All tasks guaranteed to be completed here

Taskwait

Example

```
void traverse_list ( List l )
{
  Element e;
  for ( e = l->first; e ; e = e->next )
     #pragma omp task
       process(e);

  #pragma omp taskwait

}
```

Now we need some threads to execute the tasks

List traversal

Completing the picture

Example

```
List l

#pragma omp parallel
   traverse_list(l);
```

List traversal

Completing the picture

Example

```
List l

#pragma omp parallel
   traverse_list(l);
```

This will generate multiple traversals

List traversal

Completing the picture

Example

```
List l

#pragma omp parallel
   traverse_list(l);
```

We need a way to have a single thread execute *traverse_list*

Outline

- OpenMP tasks
- Task synchronization
- The single construct
- Task clauses
- Common tasking problems

Giving work to just one thread

The single construct

```
#pragma omp single [clauses]
    structured block
```

- where clauses can be:
  - private
  - firstprivate
  - nowait ← We'll see it later
  - copyprivate ← Not today
- Only one thread of the team executes the structured block
- There is an implicit **barrier** at the end

The single construct

Example

```
int main (int argc, char **argv )
{
    #pragma omp parallel
    {
       #pragma omp single
       {
          printf("Hello world!\n");
       }
    }
}
```

The single construct

Example

```
int main (int argc, char **argv )
{
    #pragma omp parallel
    {
       #pragma omp single
       {
          printf("Hello␣world!\n");
       }
    }
}
```

This program outputs just one "Hello world"

List traversal

Completing the picture

Example

```
List l

#pragma omp parallel
#pragma single
   traverse_list(l);
```

List traversal

Completing the picture

Example

```
List l

#pragma omp parallel
#pragma single
   traverse_list(l);
```

One thread creates the tasks of the traversal

List traversal

Completing the picture

Example

```
List l

#pragma omp parallel
#pragma single
   traverse_list(l);
```

All threads cooperate to execute them

Outline

- OpenMP tasks
- Task synchronization
- The single construct
- **Task clauses**
- Common tasking problems

Task scheduling

How it works?

Tasks are tied by default

- Tied tasks are executed always by the same thread
  - Not necessarily the creator
- Tied tasks have scheduling restrictions
  - Deterministic scheduling points (creation, synchronization, ...)
    - Tasks can be suspended/resumed at these points
  - Another constraint to avoid deadlock problems
- Tied tasks may run into performance problems

The untied clause

A task that has been marked as **`untied`** has none of the previous scheduling restrictions:

- Can *potentially* switch to any thread
- Can *potentially* switch at any moment
- Bad mix with thread based features
  - thread-id, critical regions, threadprivate
- Gives the runtime more flexibility to schedule tasks

The if clause

- If the the expression of an **`if`** clause evaluates to false
  - The encountering task is suspended
  - The new task is executed immediately
    - with its own data environment
    - different task with respect to synchronization
  - The parent task resumes when the task finishes
  - Allows implementations to optimize task creation
    - For very fine grain task you may need to do your own if

Outline

- OpenMP tasks
- Task synchronization
- The single construct
- Task clauses
- **Common tasking problems**

Search problem

Example

```
void search (int n, int j, bool *state)
{
    int i,res;

    if (n == j) {
      /* good solution, count it */
      solutions++;
      return;
    }

    /* try each possible solution */
    for (i = 0; i < n; i++)

    {
       state[j] = i;
       if (ok(j+1,state)) {
         search(n,j+1,state);
       }
    }
}
```

Search problem

Example

```
void search (int n, int j, bool *state)
{
    int i,res;

    if (n == j) {
      /* good solution, count it */
      solutions++;
      return;
    }

    /* try each possible solution*/
    for (i = 0; i < n; i++)
    #pragma omp task
    {
       state[j] = i;
       if (ok(j+1,state)) {
         search(n,j+1,state);
       }
    }
}
```

Search problem

Example

```
void search (int n, int j, bool *state)
{
    int i,res;

    if (n == j) {
      /* good solution, count it */
      solutions++;
      return;
    }

    /* try each possible solution*/
    for (i = 0; i < n; i++)
    #pragma omp task
    {
       state[j] = i;
       if (ok(j+1,state)) {
         search(n,j+1,state);
       }
    }
}
```

Data scoping

Because it's an orphaned task all variables are firstprivate

Search problem

Example

```
void search (int n, int j, bool *state)
{
    int i,res;

    if (n == j) {
      /* good solution, count it */
      solutions++;
      return;
    }

    /* try each possible solution*/
    for (i = 0; i < n; i++)
    #pragma omp task
    {
        state[j] = i;
        if (ok(j+1,state)) {
          search(n,j+1,state);
        }
    }
}
```

Data scoping

Because it's an orphaned task all variables are firstprivate

State is not captured

Just the pointer is captured not the pointed data

Search problem

Example

```
void search (int n, int j, bool *state)
{
    int i,res;

    if (n == j) {
      /* good solution, count it */
      solutions++;
      return;
    }

    /* try each possible solution*/
    for (i = 0; i < n; i++)
    #pragma omp task
    {
       state[j] = i;
       if (ok(j+1,state)) {
         search(n,j+1,state);
       }
    }
}
```

Problem #1

Incorrectly capturing pointed data

Problem #1

Incorrectly capturing pointed data

Problem

firstprivate does not allow to capture data through pointers

Solutions

1. Capture it manually
2. Copy it to an array and capture the array with firstprivate

Search problem

Example

```
void search (int n, int j, bool *state)
{
    int i,res;

    if (n == j) {
      /* good solution, count it */
      solutions++;
      return;
    }

    /* try each possible solution*/
    for (i = 0; i < n; i++)
    #pragma omp task
    {
       bool *new_state = alloca(sizeof(bool)*n);
       memcpy(new_state,state,sizeof(bool)*n);
       new_state[j] = i;
       if (ok(j+1,new_state)) {
         search(n,j+1,new_state);
       }
    }
}
```

Search problem

Example

```
void search (int n, int j, bool *state)
{
    int i,res;

    if (n == j) {
      /* good solution, count it */
      solutions++;
      return;
    }

    /* try each possible solution */
    for (i = 0; i < n; i++)
    #pragma omp task
    {
       bool *new_state = alloca(sizeof(bool)*n);
       memcpy(new_state,state,sizeof(bool)*n);
       new_state[j] = i;
       if (ok(j+1,new_state)) {
         search(n,j+1,new_state);
       }
    }
}
```

Caution!

Will state still be valid by the time memcpy is executed?

Search problem

Example

```
void search (int n, int j, bool *state)
{
    int i,res;

    if (n == j) {
      /* good solution, count it */
      solutions++;
      return;
    }

    /* try each possible solution */
    for (i = 0; i < n; i++)
    #pragma omp task
    {
       bool *new_state = alloca(sizeof(bool)*n);
       memcpy(new_state,state,sizeof(bool)*n);
       new_state[j] = i;
       if (ok(j+1,new_state)) {
         search(n,j+1,new_state);
       }
    }
}
```

Problem #2

Data can go out of scope!

Problem #2

Out-of-scope data

Problem

Stack-allocated parent data can become invalid before being used by child tasks

- Only if not captured with firstprivate

Solutions

1. Use firstprivate when possible
2. Allocate it in the heap
   - Not always easy (we also need to free it)
3. Put additional synchronizations
   - May reduce the available parallelism

Search problem

Example

```
void search (int n, int j, bool *state)
{
    int i,res;

    if (n == j) {
      /* good solution, count it */
      solutions++;
      return;
    }

    /* try each possible solution */
    for (i = 0; i < n; i++)
    #pragma omp task
    {
       bool *new_state = alloca(sizeof(bool)*n);
       memcpy(new_state,state,sizeof(bool)*n);
       new_state[j] = i;
       if (ok(j+1,new_state)) {
         search(n,j+1,new_state);
       }
    }

    #pragma omp taskwait
}
```

Search problem

Example

```
void search (int n, int j, bool *state)
{
    int i,res;

    if (n == j) {
      /* good solution, count it */
      solutions++;
      return;
    }

    /* try each possible solution */
    for (i = 0; i < n; i++)
    #pragma omp task
    {
       bool *new_state = alloca(sizeof(bool)*n);
       memcpy(new_state, state, sizeof(bool)*n);
       new_state[j] = i;
       if (ok(j+1,new_state)) {
         search(n,j+1,new_state);
       }
    }

    #pragma omp taskwait
}
```

Shared variable needs protected access

Search problem

Example

```
void search (int n, int j, bool *state)
{
    int i,res;

    if (n == j) {
      /* good solution, count it */
      solutions++;
      return;
    }

    /* try each possible solution*/
    for (i = 0; i < n; i++)
    #pragma omp task
    {
       bool *new_state = alloca(sizeof(bool)*n);
       memcpy(new_state, state, sizeof(bool)*n);
       new_state[j] = i;
       if (ok(j+1,new_state)) {
         search(n,j+1,new_state);
       }
    }

    #pragma omp taskwait
}
```

Solutions

- Use **critical**
- Use **atomic**
- Use **threadprivate**

Reductions for tasks

Example

```
int solutions=0;
int mysolutions=0;
#pragma omp threadprivate(mysolutions)

void start_search ()
{
  #pragma omp parallel
  {
     #pragma omp single
     {
         bool initial_state[n];
         search(n,0,initial_state);
     }
     #pragma omp atomic
         solutions += mysolutions;
  }

}
```

Use a separate counter for each thread

Accumulate them at the end

Search problem

Example

```
void search (int n, int j, bool *state)
{
    int i,res;

    if (n == j) {
      /* good solution, count it */
      mysolutions++;
      return;
    }

    /* try each possible solution */
    for (i = 0; i < n; i++)
    #pragma omp task
    {
       bool *new_state = alloca(sizeof(bool)*n);
       memcpy(new_state,state,sizeof(bool)*n);
       new_state[j] = i;
       if (ok(j+1,new_state)){
         search(n,j+1,new_state);
       }
    }

    #pragma omp taskwait
}
```

Search problem

Example

```
void search (int n, int j, bool *state)
{
    int i,res;

    if (n == j) {
      /* good solution, count it */
      mysolutions++;
      return;
    }

    /* try each possible solution */
    for (i = 0; i < n; i++)
    #pragma omp task
    {
       bool *new_state = alloca(sizeof(bool)*n);
       memcpy(new_state,state,sizeof(bool)*n);
       new_state[j] = i;
       if (ok(j+1,new_state))
         search(n,j+1,new_state);
       }
    }

    #pragma omp taskwait
}
```

Pruning mechanism potentially introduces imbalance in the tree

Search problem

Example

```
void search (int n, int j, bool *state)
{
    int i,res;

    if (n == j) {
      /* good solution, count it */
      mysolutions++;
      return;
    }

    /* try each possible solution*/
    for (i = 0; i < n; i++)
    #pragma omp task untied
    {
       bool *new_state = alloca(sizeof(bool)*n);
       memcpy(new_state, state, sizeof(bool)*n);
       new_state[j] = i;
       if (ok(j+1,new_state)){
         search(n,j+1,new_state);
       }
    }

    #pragma omp taskwait
}
```

Untied clause

- Allows the implementation to easier load balance

Search problem

Example

```
void search (int n, int j, bool *state)
{
    int i,res;

    if (n == j) {
      /* good solution, count it */
      mysolutions++;
      return;
    }

    /* try each possible solution */
    for (i = 0; i < n; i++)
    #pragma omp task untied
    {
       bool *new_state = alloca(sizeof(bool)*n);
       memcpy(new_state,state,sizeof(bool)*n);
       new_state[j] = i;
       if (ok(j+1,new_state)) {
         search(n,j+1,new_state);
       }
    }

    #pragma omp taskwait
}
```

Because of untied this is not safe!

Pitfall #3

Unsafe use of untied tasks

Problem

Because tasks can migrate between threads at any point thread-centric constructs can yield unexpected results

Remember

When using untied tasks avoid:

- Threadprivate variables
- Any thread-id uses

And be very careful with:

- Critical regions (and locks)

Simple solution

Create a task tied region with `#pragma omp task if(0)`

Search problem

Example

```
void search (int n, int j, bool *state)
{
    int i,res;

    if (n == j) {
      /* good solution, count it */
      #pragma omp task if(0)
      mysolutions++;
      return;
    }

    /* try each possible solution */
    for (i = 0; i < n; i++)
    #pragma omp task untied
    {
       bool *new_state = alloca(sizeof(bool)*n);
       memcpy(new_state,state,sizeof(bool)*n);
       new_state[j] = i;
       if (ok(j+1,new_state)) {
         search(n,j+1,new_state);
       }
    }

    #pragma omp taskwait
}
```

Now this statement is tied and safe

Task granularity

Granularity is a key performance factor

- Tasks tend to be fine-grained
- Try to “group“ tasks together
  - Use if clause or manual transformations

Using the if clause

Example

```
void search (int n, int j, bool *state, int depth)
{
    int i,res;

    if (n == j) {
      /* good solution, count it */
      #pragma omp task if(0)
      mysolutions++;
      return;
    }

    /* try each possible solution */
    for (i = 0; i < n; i++)
    #pragma omp task untied if(depth < MAX_DEPTH)
    {
       bool *new_state = alloca(sizeof(bool)*n);
       memcpy(new_state, state, sizeof(bool)*n);
       new_state[j] = i;
       if (ok(j+1,new_state)) {
         search(n,j+1,new_state,depth+1);
       }
    }
    #pragma omp taskwait
}
```

Using an if statement

Example

```
void search (int n, int j, bool *state, int depth)
{
    int i,res;

    if (n == j) {
      /* good solution, count it */
      #pragma omp task if(0)
      mysolutions++;
      return;
    }

    /* try each possible solution */
    for (i = 0; i < n; i++)
    #pragma omp task untied
    {
       bool *new_state = alloca(sizeof(bool)*n);
       memcpy(new_state, state, sizeof(bool)*n);
       new_state[j] = i;
       if (ok(j+1,new_state)) {
         if ( depth < MAX_DEPTH )
            search(n,j+1,new_state,depth+1);
         else
            search_serial(n,j+1,new_state);
       }
    }
    #pragma omp taskwait
}
```

Part V

Hands-on (II)

Outline

- List traversal
- Computing Pi
- Finding Fibonacci

Before you start

Copy the exercises to your directory:

```
$ cp -a
~aduran/Prace_OpenMP_Handson_1/tasking .
```

Enter the tasking directory to do the following exercises.

Outline

- List traversal
- Computing Pi
- Finding Fibonacci

List traversal

Examine the code

Take a look at the list.cc file which implements a parallel list traversal with OpenMP.

1. What should be the output of executing this program?
2. Run it with one thread:

```
$ ./list
```

3. Do you get the expected result?
4. Run it with two threads:

```
$ OMP_NUM_THREADS=2 ./list
```

5. Does it work?

List traversal

Fix it

Fix the list traversal so it gets the correct result with two threads (or more). Use the following questions as a guide to help you:

1. How many tasks are being generated?
2. Which is the data scoping in each construct?
3. Are memory accesses properly synchronized?

Outline

- List traversal
- Computing Pi
- Finding Fibonacci

Computing Pi

Our algorithm

We will use an algorithm that computes the pi number through numerical integration.

- Take a look at the pi.c file
- Because iterations are independent we will create one **`task`** per iteration

When you run make it will generate two programs: pi.serial and pi.omp. We will use the serial version to evaluate our parallel version.

Computing Pi

Measuring time

- To get reliable execution times will use the Altix batch system. Use the following command to launch your executions:

```
$ make run-$program-$threads
```

- It sets up **OMP_NUM_THREADS** for you
- It will generate an output file in your directory when it finishes.
- You can check your status with mnq
- Run both versions with one thread

```
$ make run-pi.ser-1
```

```
$ make run-pi.omp-1
```

- When they finish compare the results. Now run it with 2 threads.
  - What do you observe? How is this possible?

Computing Pi

Problems

Our version of pi has two main problems:

- Tasks are too fine grain. The overheads associated with creating a task cannot be overcome.
- There is too much synchronization. Hidden synchronization and communications are a common source of performance problems.

Computing Pi

Increase the granularity

1. Modify the pi program so that each **task** executes a chunk of N iterations,
2. Experiment with different numbers of N and see how the execution time changes
   - Which would be the optimal number for N?

Computing Pi

Reduce the number of synchronizations

1. Modify the pi program so that instead of using `critical` uses an `atomic` construct
   - Does the execution time improve?
2. We can improve it further by reducing the number of `atomic` accesses
   - Use a `private` variable and only do one `atomic` update at the end of the task

Computing Pi

Final numbers

1. Run our improved version up to 8 threads.
   - Does it scale?
   - How does it compare to the serial version?
2. Now increase the total number of iterations by 10 and run it again.
   - How it behaves now?

Computing Pi

Some conclusions

- It's difficult to go further than this with `tasks`
  - Task parallelism is very flexible but we need to overcome the overheads
- Beware hidden communication and synchronizations
- OpenMP parallelization is an incremental process
  - As every other paradigm, sometimes we need effort to obtain optimal performance
- We'll see later how to improve further our pi program

Outline

- List traversal
- Computing Pi
- Finding Fibonacci

Fibonacci

The algorithm

We used a recursive implementation to find the Fibonacci number in the fib.c file.

- It's very inefficient
- But useful for educational purposes :-)

To compile it use:

```
$ make fib
```

To submit jobs use:

```
$ make run-fib-threads
```

Fibonacci

First

- Complete the code so all the branches are computed in parallel
  - Use the serial version to check you have the correct result
- Add code to measure the time it takes to compute the number
  - To be more precise put the code inside the single region

Fibonacci

Evaluate

1. Run the code from 1 to 8 threads.
2. Compare it to the time of the serial version
3. What do you observe?

Fibonacci

Incresing granularity

As in the pi program, Fibonacci because it recursive nature ends generating to fine grain tasks.

1. Modify the program so it does not generate tasks at all when *n* is too small (e.g. 20)
2. Run again this improved version up to 8 threads
3. How does it compare with respect to the serial version?
4. Try changing the cut-off value from 20 and how affects performance

Part VI

Data Parallelism in OpenMP

Outline

- The worksharing concept
- Loop worksharing

Outline

- The worksharing concept
- Loop worksharing

Worksharings

Worksharing constructs divide the execution of a code region among the threads of a team

- Threads cooperate to do some work
- Better way to split work than using thread-ids
- Lower overhead than using **`tasks`**
  - But, less flexible

In OpenMP, there are four worksharing constructs:

- single
- loop worksharing
- section
- workshare

We'll see them later

Restriction: worksharings cannot be nested

Outline

- The worksharing concept
- Loop worksharing

Loop parallelism

The for construct

```
#pragma omp for [clauses]
    for( init-expr ; test-expr ; inc-expr )
```

where clauses can be:

- private
- firstprivate
- **lastprivate(*variable-list*)**
- **reduction(*operator:variable-list*)**
- **schedule(*schedule-kind*)**
- **nowait**
- **collapse(*n*)**
- ordered ← We'll see it later

The for construct

How it works?

The iterations of the loop(s) associated to the construct are divided among the threads of the team.

- Loop iterations must be independent
- Loops must follow a form that allows to compute the number of iterations
- Valid data types for inductions variables are: integer types, pointers and random access iterators (in C++)
  - The induction variable(s) are automatically privatized
- The default data-sharing attribute is **shared**

It can be merged with the **parallel** construct:

```
#pragma omp parallel for
```

The for construct

Example

```
void foo (int *m, int N, int M)
{
  int i;
  #pragma omp parallel for private(j)
  for ( i = 0; i < N; i++ )
     for ( j = 0; j < M; j++ )
        m[i][j] = 0;
}
```

The for construct

Example

```
void foo (int *m, int N, int M)
{
  int i;
  #pragma omp parallel for private
  for ( i = 0; i < N; i++ )
     for ( j = 0; j < M; j++ )
        m[i][j] = 0;
}
```

New created threads cooperate to execute all the iterations of the loop

The for construct

Example

```
void foo (int *m, int N, int M)
{
   int i;
   #pragma omp par
   for ( i = 0; i
      for ( j = 0; j < M; j++ )
         m[i][j] = 0;
}
```

The *i* variable is automatically privatized

The for construct

Example

```
void foo (int *m, int N, int M)
{
   int i;
   #pragma omp parallel for private(j)
   for ( i = 0; i < N
      for ( j = 0; j
         m[i][j] = 0;
}
```

Must be explicitly privatized

The for construct

Example

```
void foo ( std::vector<int> &v )
{
  #pragma omp parallel for
  for ( std::vector<int>::iterator it = v.begin() ;
        it < v.end() ;
        it ++ )
    *it = 0;
}
```

The for construct

Example

```
void foo ( std::vector<int> &v )
{
  #pragma omp parallel for
  for ( std::vector<int>::iterator it = v.
        it < v.end() ;
        it ++ )
    *it = 0;
}
```

random access iterators (and pointers) are valid types

The for construct

Example

```
void foo ( std::vector<int> &v )
{
  #pragma omp parallel for
  for ( std::vector<int>::ite
        it < v.end() ;
        it ++ )
     *it = 0;
}
```

!= cannot be used in the test expression

Removing dependences

Example

```
x = 0;
for ( i = 0; i < n; i++ )
{
    v[i] = x;
    x += dx;
}
```

Removing dependences

Example

```
x = 0;
for ( i = 0; i < n; i++ )
{
    v[i] = x;
    x += dx;
}
```

Each iteration *x* depends on the previous one. Can't be parallelized

Removing dependences

Example

```
x = 0;
for ( i = 0; i < n; i++ )
{
    x = i * dx;
    v[i] = x;
}
```

But *x* can be rewritten in terms of *i*.
Now it can be parallelized

Removing dependences

Example

```
x = 0;
#pragma omp parallel for private(x)
for ( i = 0; i < n; i++ )
{
    x = i * dx;
    v[i] = x;
}
```

The lastprivate clause

When a variable is declared **`lastprivate`**, a private copy is generated for each thread. Then the value of the variable in the last iteration of the loop is copied back to the original variable.

- A variable can be both **`firstprivate`** and **`lastprivate`**

The lastprivate clause

Example

```
int i
#pragma omp for lastprivate(i)
for ( i = 0; i < 100; i++ )
   v[i] = 0;

printf("i=%d\n",i);
```

The lastprivate clause

Example

```
int i
#pragma omp for lastprivate(i)
for ( i = 0; i < 100; i++ )
   v[i] = 0;

printf("i=%d\n",i);
```

prints 100

The reduction clause

A very common pattern is where all threads accumulate some values into a shared variable

- E.g., n += v[i], our pi program, ...
- Using `critical` or `atomic` is not good enough
  - Besides being error prone and cumbersome

Instead we can use the `reduction` clause for basic types.

- Valid operators for C/C++: +,-,*,|,||,&,&&,^
- Valid operators for Fortran: +,-,*,.and.,.or.,.eqv.,.neqv.,max,min
  - also supports reductions of arrays
- The compiler creates a `private` copy that is properly initialized
- At the end of the region, the compiler ensures that the `shared` variable is properly (and safely) updated.

We can also specify `reduction` variables in the `parallel` construct.

The reduction clause

Example

```
int vector_sum (int n, int v[n])
{
   int i, sum = 0;
   #pragma omp parallel for reduction(+:sum)

      for ( i = 0; i < n; i++ )
         sum += v[i];

   return sum;
}
```

The reduction clause

Example

```
int vector_sum (int n, int v[n])
{
   int i, sum = 0;
   #pragma
      for (
         sum
   return
}
```

Private copy initialized here to the identity value

Shared variable updated here with the partial values of each thread

Also in parallel

Example

```
int nt = 0;

#pragma omp parallel reduction(+:nt)
   nt++;

printf("%d\n",nt);
```

Also in parallel

Example

```
int nt = 0;

#pragma omp parallel reduction(+:nt)
   nt++;

printf("%d\n",nt);
```

reduction available in parallel as well

Also in parallel

Example

```
int nt = 0;

#pragma omp parallel reduction(+:nt)
   nt++;

printf("%d\n",nt);
```

Prints the number of threads

The schedule clause

The **schedule** clause determines which iterations are executed by each thread.

- If no **schedule** clause is present then is implementation defined

There are several possible options as schedule:

- **STATIC**
- **STATIC,chunk**
- **DYNAMIC[,chunk]**
- **GUIDED[,chunk]**
- **AUTO**
- **RUNTIME**

The schedule clause

Static schedule

The iteration space is broken in chunks of approximately size $N/num-threads$. Then these chunks are assigned to the threads in a Round-Robin fashion.

Static,N schedule (Interleaved)

The iteration space is broken in chunks of size N. Then these chunks are assigned to the threads in a Round-Robin fashion.

Characteristics of static schedules

- Low overhead
- Good locality (usually)
- Can have load imbalance problems

The schedule clause

Dynamic,N schedule

Threads dynamically grab chunks of N iterations until all iterations have been executed. If no chunk is specified, $N = 1$.

Guided,N schedule

Variant of `dynamic`. The size of the chunks deceases as the threads grab iterations, but it is at least of size N. If no chunk is specified, $N = 1$.

Characteristics of dynamic schedules

- Higher overhead
- Not very good locality (usually)
- Can solve imbalance problems

The schedule clause

Auto schedule

In this case, the implementation is allowed to do whatever it wishes.

- Do not expect much of it as of now

Runtime schedule

The decision is delayed until the program is run through the **sched-nvar** ICV. It can be set with:

- The **OMP_SCHEDULE** environment variable
- The **omp_set_schedule**() API call

False sharing

- When a thread writes to a cache location, and another thread reads the same location the coherence protocol will copy the data from one cache to the other. This is called true sharing
- But it can happen that this communication happens even if two threads are not working on the same memory address. This is false sharing

Scheduling

Example

```
int v[N];

#pragma omp for
for ( int i = 0; i < N; i++ )
   for ( int j = 0; j < i ; j++ )
      v[i] += j;
```

Scheduling

Example

```
int v[N];

#pragma omp for
for ( int i = 0; i < N; i++ )
   for ( int j = 0; j < i ; j++ )
      v[i] += j;
```

i loop quite unbalaced

Scheduling

Example

```
int v[N];

#pragma omp for
for ( int i = 0; i < N; i++ )
   for ( int j = 0; j < i ; j++ )
      v[i] += j;
```

dynamic schedule?

Scheduling

Example

```
int v[N];

#pragma omp for
for ( int i = 0; i < N; i++ )
   for ( int j = 0; j < i
      v[i] += j;
```

lots of false sharing!

The nowait clause

When a worksharing has a **`nowait`** clause then the implicit **`barrier`** at the end of the loop is removed.

- This allows to overlap the execution of non-dependent loops/tasks/worksharings

The nowait clause

Example

```
#pragma omp for nowait
for ( i = 0; i < n ; i++ )
   v[i] = 0;
#pragma omp for
for ( i = 0; i < n ; i++ )
   a[i] = 0;
```

First and second loop are independent so we can overlap them

The nowait clause

Example

```
#pragma omp for nowait
for ( i = 0; i < n ; i++ )
   v[i] = 0;
#pragma omp for
for ( i = 0; i < n ; i++ )
   a[i] = 0;
```

On a side note, you would be better by fusing the loops in this case

The nowait clause

Example

```
#pragma omp for nowait
for ( i = 0; i < n ; i++ )
   v[i] = 0;
#pragma omp for
for ( i = 0; i < n ; i++ )
   a[i] = v[i]*v[i];
```

First and second loop are dependent!. No guarantees that the previous iteration is finished

The nowait clause

Exception: static schedules

If the two (or more) loops have the same **static** schedule and all have the same number of iterations.

Example

```
#pragma omp for schedule(static,2) nowait
for ( i = 0; i < n ; i++ )
   v[i] = 0;
#pragma omp for schedule(static,2)
for ( i = 0; i < n ; i++ )
   a[i] = v[i]*v[i];
```

The collapse clause

Allows to distribute work from a set of *n* nested loops.

- Loops must be perfectly nested
- The nest must traverse a rectangular iteration space

The collapse clause

Allows to distribute work from a set of *n* nested loops.

- Loops must be perfectly nested
- The nest must traverse a rectangular iteration space

Example

```
#pragma omp for collapse(2)
for ( i = 0; i < N; i++ )
   for ( j = 0; j < M; j++ )
      foo (i,j);
```

i and *j* loops are folded and iterations distributed among all threads. Both *i* and *j* are privatized

Coffee time! :-)

Part VII

Hands-on (III)

Outline

- Matrix Multiply
- Computing Pi (revisited)
- Mandelbrot

Before you start

Copy the exercises to your directory:

```
$ cp -a
~aduran/Prace_OpenMP_Handson_2/worksharing
.
```

Enter the worksharing directory to do the following exercises.

Outline

- Matrix Multiply
- Computing Pi (revisited)
- Mandelbrot

Matrix Multiply

Parallel loops

The file matmul implements a sequential matrix multiply.

1. Use OpenMP worksharings to parallelize the application.
   - check the init_mat and matmul functions
2. Run it up to 8 threads to check the scalability

Remember: To submit it use make run-matmul.omp-$threads

Matrix Multiply

Memory matters!

To optimize accesses to the cache in these kind of algorithms, it is a common practice to "logically" split the matrix in blocks of size *BxB*, and do computation block-a-block instead of going through all the matrix at once.

1. Implement such a blocking scheme for our matrix multiply
2. Experiment with different sizes of *B*
3. Run it up to 8 threads and compare the results with the previous version

Tip: You need three additional inner loops

Outline

- Matrix Multiply
- Computing Pi (revisited)
- Mandelbrot

Computing Pi

Using data parallelism

1. Complete the implementation of our pi algorithm using data parallelism
2. Execute with 1 and 2 threads.
   - Does it scale?
   - How does it compare to our previous implementation with tasks?
   - What is the problem?

Computing Pi

Problem

The number of synchronizations is still very high for this program to scale.

Using `reduction`

1. Change the program to make use of the `reduction` clause
2. Run it up to 8 threads
3. How it compares to the previous version?

Outline

- Matrix Multiply
- Computing Pi (revisited)
- Mandelbrot

Mandelbrot

More data parallelism

We will now parallelize an algorithm that generates sections of the Mandelbrot function.

1. Edit file mandel.c and complete the parallelization in function mandel
   - Note that there is a dependence on the variable x

Mandelbrot

Uncover load imbalance

We can see that each point in the final output is computed through the mandel_point function. If we check the code of that function we can see that the number of iterations it takes will be different from one point to another.

We want to know how many iterations (this also happens to be the result of mandel_point) each thread does.

1. Add a private counter to each thread
2. Add to this counter the result of each mandel_point call by that thread
3. Output the count for each thread at the end of the parallel region
4. What do you observe?

Mandelbrot

Playing with schedules

To overcome the observed load imbalance we can use a different loop schedule.

- Use the clause `schedule(runtime)` so the schedule is not fixed at compile time
- Now run different experiments with different schedules and number of threads
  - Try at least `static`, `dynamic` and `guided`
- Which one obtains the best result?

Tip: Change `OMP_SCHEDULE` before doing make run-...

Part VIII

Other OpenMP Topics

Outline

- The master construct
- Other synchronization mechanisms
- Nested parallelism
- Other worksharings
- Other environment variables and API calls

Outline

- The master construct
- Other synchronization mechanisms
- Nested parallelism
- Other worksharings
- Other environment variables and API calls

Only the master thread

The master construct

```
#pragma omp master
    structured block
```

- The structured block is only executed by the master thread
    - Useful when we want always the same thread to execute something
- No implicit barrier at the end

Master construct

Example

```
void foo ()
{
   #pragma omp parallel
   {
      #pragma omp single
         printf("I␣am␣%d\n", omp_get_thread_num());

      #pragma omp master
         printf("I␣am␣%d\n", omp_get_thread_num());
   }
}
```

Master construct

Example

```
void foo ()
{
   #pragma omp parallel
   {
      #pragma omp single
         printf("I␣am␣%d\n", omp_get_thread_num());   <- Can be any thread

      #pragma omp master
         printf("I␣am␣%d\n", omp_get_thread_num());   <- It's always thread 0
   }
}
```

Outline

- The master construct
- Other synchronization mechanisms
- Nested parallelism
- Other worksharings
- Other environment variables and API calls

Ordering

The ordered construct

```
#pragma omp ordered
    structured block
```

- Must appear in the dynamic extend of a loop worksharing
  - The worksharing must also have the `ordered` clause
- The structured block is executed in the iteration's sequential order

Locks

OpenMP provides lock primitives for low-level synchronization

`omp_init_lock`	Initialize the lock
`omp_set_lock`	Acquires the lock
`omp_unset_lock`	Releases the lock
`omp_test_lock`	Tries to acquire the lock (won't block)
`omp_destroy_lock`	Frees lock resources

Locks

OpenMP provides lock primitives for low-level synchronization

omp_init_lock	Initialize the lock
omp_set_lock	Acquires the lock
omp_unset_lock	Releases the lock
omp_test_lock	Tries to acquire the lock (won't block)
omp_destroy_lock	Frees lock resources

OpenMP also provides nested locks where the thread owning the lock can reacquire the lock without blocking.

Locks

Example

```
#include <omp.h>
void foo ()
{
   omp_lock_t lock;

   omp_init_lock(&lock);
   #pragma omp parallel
   {
       omp_set_lock(&lock);
       // mutual exclusion region
       omp_unset_lock(&lock);
   }
   omp_destroy_lock(&lock);
}
```

Locks

Example

```
#include <omp.h>
void foo ()
{
   omp_lock_t lock;

   omp_init_lock(&lock);
   #pragma omp parallel
   {
      omp_set_lock(&lock);
      // mutual exclusion region
      omp_unset_lock(&lock);
   }
   omp_destroy_lock(&lock);
}
```

Lock must be initialized before being used

Locks

Example

```
#include <omp.h>
void foo ()
{
   omp_lock_t lock;

   omp_init_lock(&lock);
   #pragma omp parallel
   {
       omp_set_lock(&lock);
       // mutual exclusion region
       omp_unset_lock(&lock);
   }
   omp_destroy_lock(&lock);
}
```

Only one thread at a time here

Locks

Example

```
#include <omp.h>

omp_lock_t lock;

void foo ()
{
   omp_set_lock(&lock);
}

void bar ()
{
   omp_unset_lock(&lock);
}
```

Locks

Example

```
#include <omp.h>

omp_lock_t lock;

void foo ()
{
    omp_set_lock(&lock);
}

void bar ()
{
    omp_unset_lock(&lock);
}
```

Locks are unstructured

Outline

- The master construct
- Other synchronization mechanisms
- Nested parallelism
- Other worksharings
- Other environment variables and API calls

Nested parallelism

- OpenMP `parallel` constructs can dynamically be nested. This creates a hierarchy of teams that is called nested parallelism.
- Useful when not enough parallelism is available with a single level of parallelism
  - More difficult to understand and manage
  - Implementations are not required to support it

Controlling nested parallelism

Related Internal Control Variables

- The ICV **`nest-var`** controls whether nested parallelism is enabled or not.
  - Set with the **`OMP_NESTED`** environment variable
  - Set with the **`omp_set_nested`** API call
  - The current value can be retrieved with **`omp_get_nested`**.
- The ICV **`max-active-levels-var`** controls the maximum number of nested regions
  - Set with the **`OMP_MAX_ACTIVE_LEVELS`** environment variable
  - Set with the **`omp_set_max_active_levels`** API call
  - The current value can be retrieved with **`omp_get_max_active_levels`**.

Nested parallelism info API

To obtain information about nested parallelism

- How many nested parallel regions at this point?
  - `omp_get_level`()
- How many active (with 2 or more threads) regions?
  - `omp_get_active_level`()
- Which thread-id was my ancestor?
  - `omp_get_ancestor_thread_num`(level)
- How many threads there are at a previous region?
  - `omp_get_team_size`(level)

Outline

- The master construct
- Other synchronization mechanisms
- Nested parallelism
- Other worksharings
- Other environment variables and API calls

Static tasks

The sections construct

```
#pragma omp sections [clauses]
#pragma omp section
   structure block
   ...
```

- The different **section** are distributed among the threads
- There is an implicit barrier at the end
- Clauses can be:
  - **private**
  - **lastprivate**
  - **firstprivate**
  - **reduction**
  - **nowait**

Sections

Example

```
#pragma omp parallel sections num_threads(3)
{
#pragma omp section
     read(data);
#pragma omp section
#pragma omp parallel
    work(data);
#pragma omp section
    write(data);
}
```

Sections

Example

```
#pragma omp parallel sections num_thr
{
#pragma omp section
     read(data);
#pragma omp section
#pragma omp parallel
    work(data);
#pragma omp section
    write(data);
}
```

Combined construct

Sections

Example

```
#pragma omp parallel sections num_threads(3)
{
#pragma omp section
     read(data);
#pragma omp section
#pragma omp parallel
    work(data);
#pragma omp section
    write(data);
}
```

Sections distributed among threads

Sections

Example

```
#pragma omp parallel sections num_threads(3)
{
#pragma omp section
     read(data);
#pragma omp section
#pragma omp parallel
    work(data);
#pragma omp section
    write(data);
}
```

Nested parallel region

Supporting array syntax

The workshare construct

```
$!OMP WORKSHARE
    array syntax
!$OMP END WORKSHARE [NOWAIT]
```

- Only for Fortran
- The array operation is distributed among threads

Example

```
$!OMP WORKSHARE
   A(1:M) = A(1:M) * B(1:M)
!$OMP END WORKSHARE NOWAIT
```

Outline

- The master construct
- Other synchronization mechanisms
- Nested parallelism
- Other worksharings
- Other environment variables and API calls

Other Environment variables

OMP_STACKSIZE	Controls the stack size of created threads
OMP_WAIT_POLICY	Controls the behaviour of idle threads
OMP_THREAD_LIMIT	Limit of threads that can be created
OMP_DYNAMIC	Turns on/off thread dynamic adjusting

Other API calls

omp_in_parallel	Returns true if inside a parallel region
omp_get_wtick	Returns the precision of the **wtime** clock
omp_get_thread_limit	Returns the limit of threads
omp_set_dynamic	Returns whether thread dynamic adjusting is on or off
omp_get_dynamic	Returns the current value of dynamic adjusting
omp_get_schedule	Returns the current loop schedule

Part IX

Hands-on (IV)

Outline

Before you start

Copy the exercises to your directory:

```
$ cp -a
~aduran/Prace_OpenMP_Handson_2/other .
```

Enter the other directory to do the following exercises.

Nested parallelism

First take

1. Edit the file nested.c and try to understand what it does
2. Run make
3. Execute the programe nested with differents numbers of threads
   - How many messages are printed? Does it match your expectations?
4. Run the program again the defining the OMP_NESTED variable. E.g.:

```
$ OMP_NUM_THREADS=2 OMP_NESTED=true
./nested
```

5. What is the difference? Why?

Nested parallelism

Shaping the tree

1. Now, change the code so the nested level only creates as many threads as the parent id+1
   - Thread 0 creates a nested parallel region of 1
   - Thread 1 creates a nested parallel region of 2
   - ...

Tip: Use either `omp_set_num_threads` or `num_threads`

Locks

Exclusive access

1. Edit the file lock.c and take a look at the code
2. Parallelize the first two loops of the application
3. Now run it several times with different numbers of threads
4. We see that result differs because of improper synchronization
5. Use critical to fix it
   - What problem do we have?

Locks

Locks to the help

1. Use locks to implement a fine grain locking scheme
2. Assign a lock to each position of the array *a*
3. Then use it to lock only that position in the main loop
   - Does it work better?
4. Now compare it to an implementation using atomic

Part X

OpenMP in the future

Outline

- How OpenMP evolves
- OpenMP 3.1
- OpenMP 4.0
- OpenMP is Open

Outline

- How OpenMP evolves
- OpenMP 3.1
- OpenMP 4.0
- OpenMP is Open

The OpenMP Language Committee

Body that prepares new standard versions for the ARB.

- Composed by representatives of all ARB members
  - Lead by Bronis de Supinski from LLNL
- Integrates the information about the different subcommittees
- Currently working on OpenMP 3.1

The OpenMP Subcommittees

When a topic is deemed important or too complex usually a separate group is formed (with a subset of the same people usually).

Currently, the following subcommittees exist:

1. Error model subcommittee
   - In charge of defining an error model for OpenMP
2. Tasking subcommittee
   - In charge of defining new extensions to the tasking model
3. Affinity subcommittee
   - In charge of breaking the flat memory model
4. Accelerators subcommittee
   - In charge of integrating accelerator computing into OpenMP
5. Interoperability and Composability subcommittee

What can we expect in the future?

Disclaimer

- This are my subjective appreciations.
- All these dates and topics are my guessings.
- They might or might not happen.

Tentative Timeline

November 2010	3.1 Public comment version
May 2011	3.1 Final version
June 2012	4.0 Public comment version
November 2012	4.0 Final version

Outline

- How OpenMP evolves
- OpenMP 3.1
- OpenMP 4.0
- OpenMP is Open

Clarifications

Several clarifications to different parts of the specification

- Nothing exciting but needs to be done

Atomic extensions

Extensions to the **atomic** construct to allow:

- to do atomic writes

```
#pragma omp atomic
    x = value;
```

- to capture the value before/after the atomic update

```
#pragma omp atomic
    v = x, x--;
```

User-defined reductions

Allow the users to extend reductions to cope with non-basic types and non-standard operators.

- In 3.1
  - Including pointer reductions in C
  - Including class members and operators in C++
- In 4.0
  - Array for C
  - Template reductions for C++

User-defined reductions

Example

```
#pragma omp declare reduction(+:std::string:omp_out += omp_in)

void foo ()
{
  std::string s;

  #pragma omp parallel reduction(+:s)
  {
     s += "I'm a thread"
  }

  std::cout << s << std::endl;

}
```

Affinity extensions

New environment variables

- `OMP_PROCBIND`=true, false
  - Portable mechanism to bind threads
- Extend `OMP_NUM_THREADS` to support multiple levels of parallelism
- `OMP_AFFINITY`=scatter,compact
  - Specifies how threads should be distributed in the machine
- `OMP_MEMORY_PLACEMENT`=first_touch|round_robin|random
  - Portable mechanisms to specify memory placement policies

Tasking extensions

New constructs/clause

- the **`taskyield`** construct to allow user-defined scheduling points
- the **`final`** clause to allow the optimization of leaf tasks

Outline

Error model

- Allow the programmer to catch and react to runtime errors
- Integrate C++ exceptions into this model
- Allow the programmer to cancel nicely the parallel computation

It looks like we are leaning towards a model based on callbacks

Error model

Example

```
void error_handler ( omp_err_info_t *info , int *nths )
{
   if ( omp_get_error_type(info) == OMP_ERR_NOT_ENOUGH_THREADS )
      *nths = *nths > 1 ? *nths -1 : 1;
   return OMP_RETRY;
}

nths = 4;
#pragma omp parallel onerror(error_handler,&nths) num_threads(nths)
{
  ....
}
```

Other tasking improvements

- Tasking reductions
  - Add a **`reduction`** clause to the **`task`** construct
- Tasking dependences
  - Allow finer tasking synchronizations by means of expressing data dependences among tasks
- Scheduling hints for the runtime
  - Allow the programmer to express some kind of task priority

Task dependences

Example

```
for ( ; ; ) {
   char *buffer;
   #pragma omp task output(buffer)
   {
       buffer = malloc(...);
       stage1(buffer);
   }
   #pragma omp task inout(buffer)
   {
       stage2(buffer)
   }
   #pragma omp task input(buffer)
   {
       stage3(buffer)
   }
}
```

Accelerators support

- Discussion is in the very early stages.
  - Several proposals on the table
- Cover both data and task parallelism
- Will probably take care of the backend compilation

A glimpse into BSC proposal

Example

```
int main( void ){
for (int i = 0; i < NB; i++)
   for (int j = 0; j < NB; j++)
      for (int k = 0; k < NB; k++)
      #pragma omp target device(smp,cell) \
                  copy_in([BS][BS] A, [BS][BS] B, [BS][BS] C) \
                  copy_out([BS][BS] C)
      #pragma omp task inout([BS][BS] C)
         matmul ( A[i][k], B[k][j], C[i][j] );
}
```

Outline

- How OpenMP evolves
- OpenMP 3.1
- OpenMP 4.0
- OpenMP is Open

OpenMP is Open

Compunity

Compunity represents the OpenMP User's Group.

- It is an special ARB member
  - Representative: Barbara Chapman from Univ of Houston
- Anyone can join and participate
  - and also give feedback

OpenMP Forum

- Forum oversighted by ARB members
  - OpenMP usage forum
  - Spec clarifications forum
- Several 3.1 clarifications have its origin in comments from users

Where to go now?

- http://www.openmp.org
- http://www.compunity.org
- http://nanos.ac.upc.edu