INF560
Algorithmique Parallèle et Distribuée

2021/2022

Patrick CARRIBAULT

CEA, DAM, DIF, F-91297 Arpajon

Lecture Outline - OpenMP

- OpenMP Basics
  - Introduction
  - Parallel region
  - Data flow
  - Worksharing

- Synchronization
  - Introduction
  - Barrier
  - Atomic operation
  - Critical region
  - Exclusive execution
  - Locks

OpenMP Basics

- Introduction

Introduction

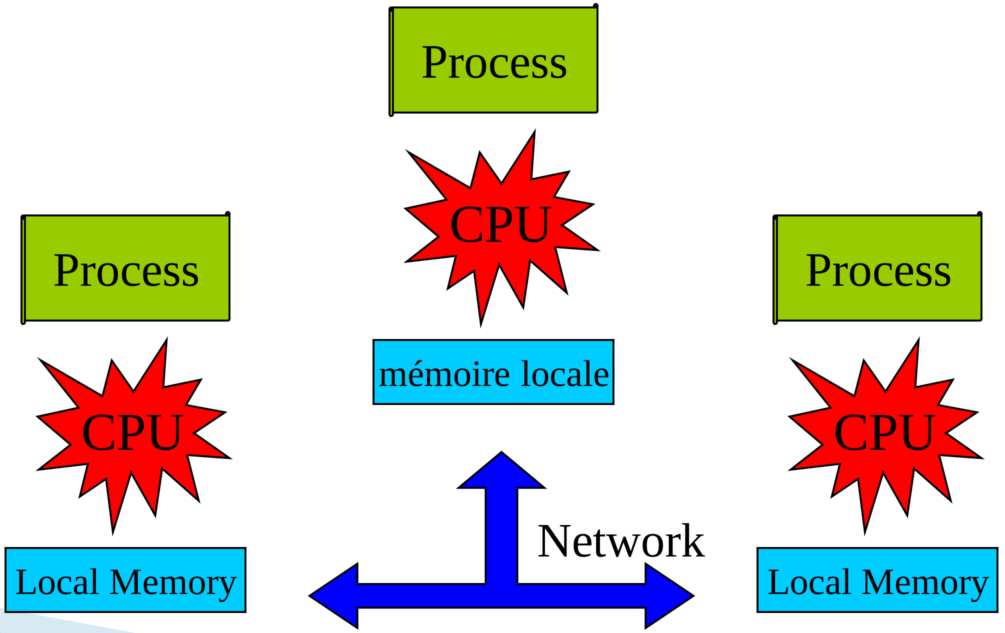
- OpenMP (Open Multi-Processing)
  - Programming model for parallel computing on shared-memory system
  - Supported on numerous platforms
    - Unix, Linux, Windows
  - Enabled on multiple programming languages
    - C/C++ and FORTRAN
  - Set of directives + library + environment variables

- Portable and scalable
  - Fast fine-grain parallelization
    - Support multiple parallelism types
  - Allow scalability (depending on the constructs used in the target application)
  - Main goal: enable parallelism for a whole node!

- Hybrid MPI+OpenMP parallelism can be used

Distributed-Memory System

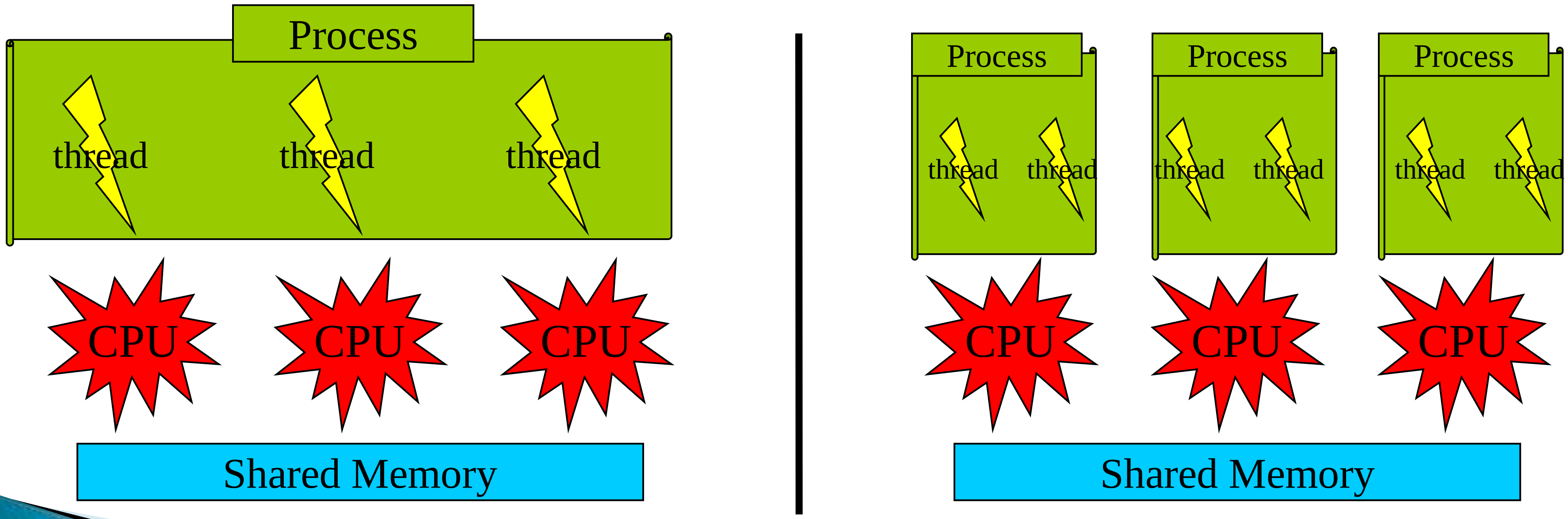
- Distributed-memory system
  - System in which multiple compute units have their own memory space
  - Units cannot directly access other memories
  - Network may represent inter-socket or inter-core or inter-node connections



Shared-Memory System

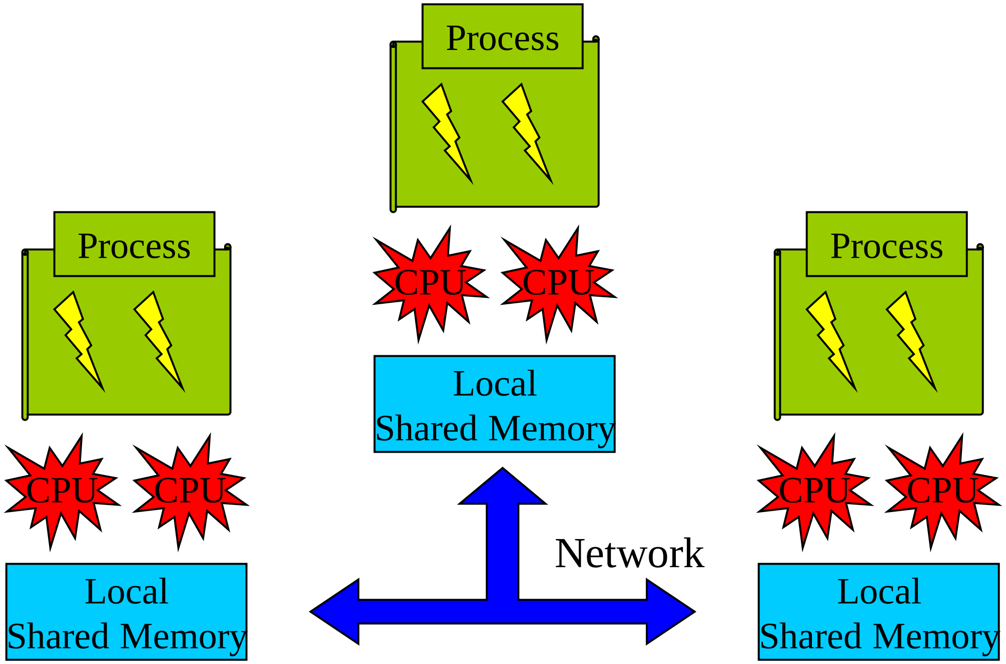
- Shared-memory system
  - System in which multiple compute units share memory
  - Logical or physical

- Node
  - Largest set of units sharing memory



Mixed Systems

- Mix of shared and distributed memory systems
- Cluster
  - Set of nodes linked with network



Shared-Memory Model

- Requirement
  - **Parallel tasks should have the same view of memory**
- Consequence
  - Concurrent accesses to memory should be handled
- Simple approach but may lead to performance decrease
  - Critical sections/parts are sequential (by definition)
  - Data locality may not be optimal

Shared-Memory Model

- On distributed-memory system
  - Difficult
  - How to share the memory view?
    - DSM (Distributed Shared Memory)
    - May generate a large overhead
      - Depend on the number of remote accesses
- On shared-memory system
  - Easy because of shared memory
  - Inside multithreaded process
    - Every thread have access to the same memory zone
    - Usually, whole node memory

Shared-Memory Model

- API POSIX pthread
  - Standard thread management inside process
  - Suitable for MPMD approach
  - Mainly C/C++

Lecture Focus

- OpenMP
  - Compiler directives
  - Hide some management complexity (thread creation, synchronization…)
  - C/C++/FORTRAN

- TBB, Cilk+…
  - Library-based approach
  - Well integrated to C++ (template, objects…)

History

- Multitask parallelization proposed by various vendors (e.g., CRAY, NEC, IBM...)
  - Everyone provided its own set of directives

- *Standard* definition
  - Motivated by multiprocessor machine

- Tentative w/ PCF
  - Parallel Computing Forum
  - Never adopted

- Industrial & vendor consortium → OpenMP
  - October 28, 1997
  - *De facto* standard
  - Said to be industrial standard

History – Part 1

- Managed by OpenMP ARB *(Architecture Review Board)*
  - http://www.openmp.org
  - http://www.compunity.org

- OpenMP 1.0 for FORTRAN → October 1997
- OpenMP 1.0 for C/C++ → October 1998

- OpenMP 2.0 pour Fortran → 2000
- OpenMP 2.0 pour C/C++ → 2002
- OpenMP 2.5 → May 2005
  - Unified standard for FORTRAN & C/C++

- OpenMP 3.0 → May 2008
  - Task support
- OpenMP 3.1 → July 2011
  - Taskyield construct
  - Extension of atomic operations

History – Part 2

- OpenMP 4.0 → July 2013
  - SIMD constructs
  - PROC_BIND and places
  - Device constructs
  - Task dependencies

- OpenMP 4.5 → November 2015
  - Taskloop constructs
  - Task priority

- OpenMP 5 → November 2018
  - Released SC'18 conference after multiple drafts
  - Support of OMPT (tool interface)

- OpenMP 5.1 → November 2020
  - Extend language support (C11, C++20, Fortran 2008) and include C++ attribute
  - Enhancement of environment and feedback (omp_display_env function + directive error)
  - Loop transformation construct

- OpenMP 6 → November 2023 (tentative)
  - More interaction w/ C++ standard
  - State-less threads (towards workers)

OpenMP Overview

- Programming model
  - Based on directives and API
  - Existing codes can be *augmented* with OpenMP
- Execution model
  - Execution inside one process
  - This process creates and activates threads
  - Based on the fork/join model
- Each thread
  - Executes an implicit task made of instructions
  - Has a specific rank
- Communication mode
  - Based on thread capability

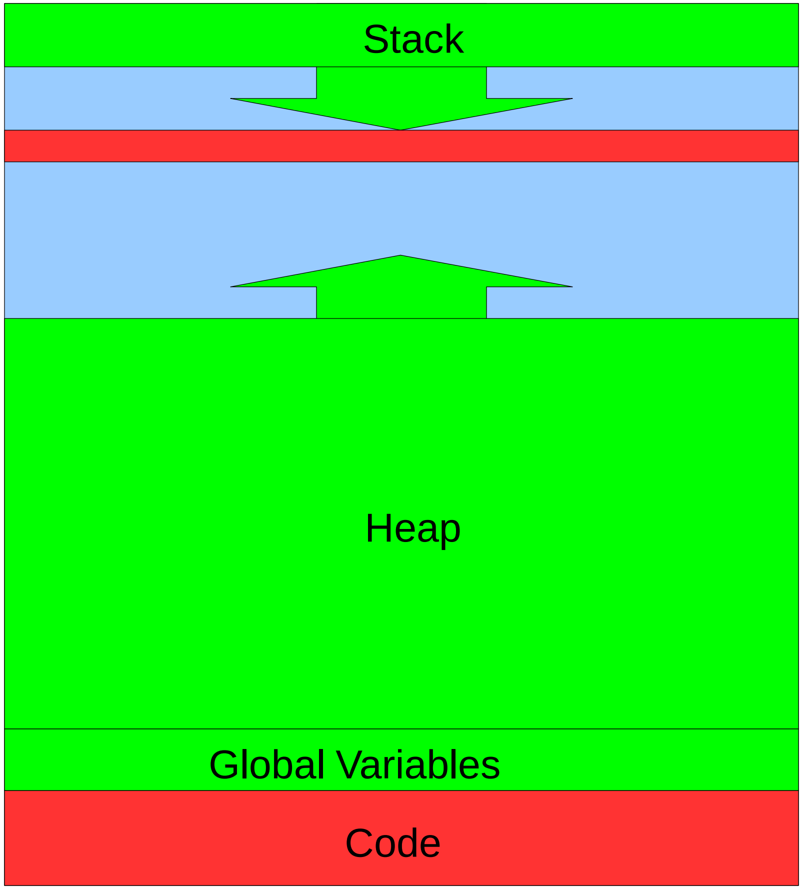
Thread vs. Process

- Thread = lightweight process

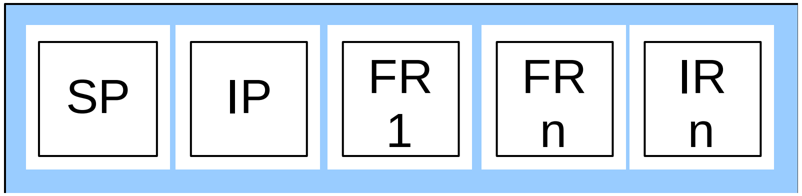
- Parts of a thread
  - Stack
  - Context
    - Include set of registers

- Parts of multi-threaded process
  - Page table
  - Set of threads

Process

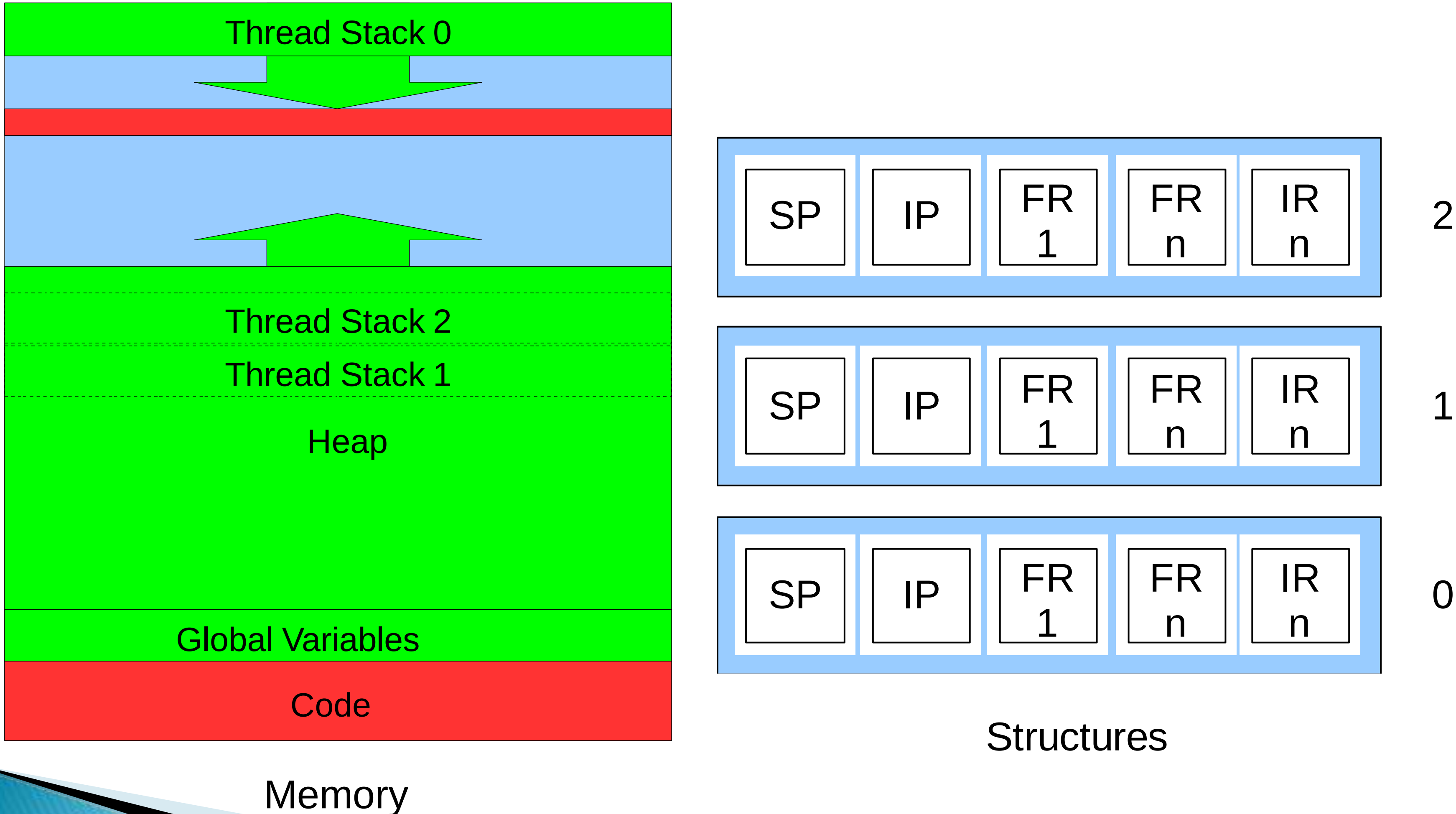


Memory



Structures

Multi-Threaded Process

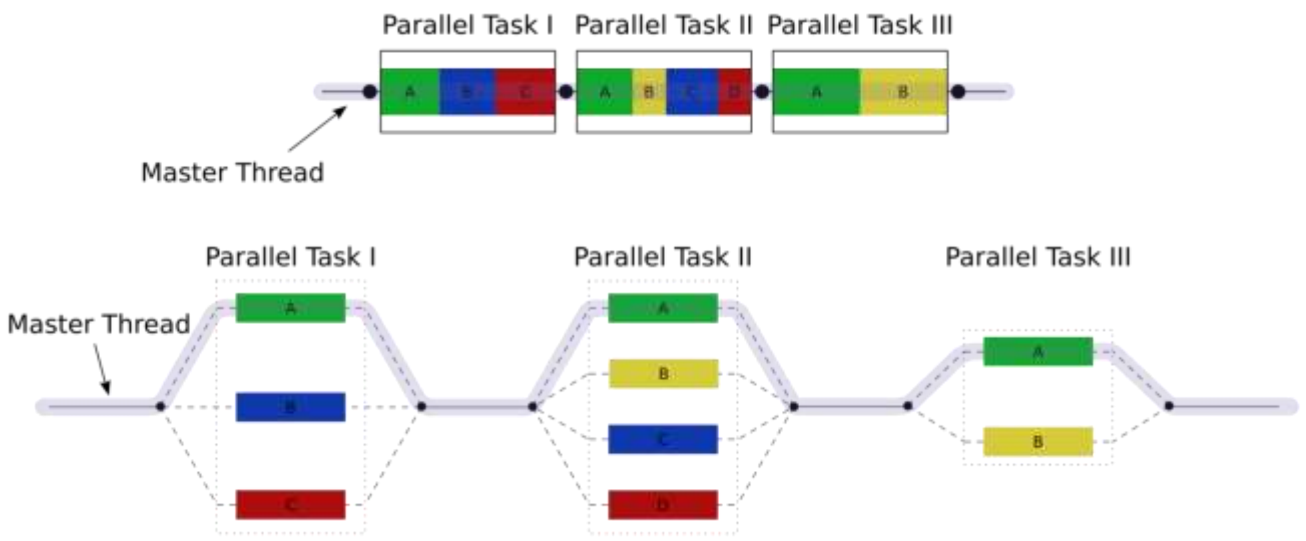


OpenMP Basics

- Parallel Region

OpenMP Execution Model

- Fork/join model
  - Program starts w/ serial part (by master thread)
  - Entering a parallel region → fork
  - Exiting a parallel region → join (barrier)
- Expressed through directives & function calls



General Syntax

- Main directive syntax for C/C++
  - `#pragma omp directive-name [clause[ [,] clause] ... ] new-line`
- Main directive syntax for FORTRAN
  - `sentinel directive-name [clause[ [,] clause]...]`

  - Sentinel for fixed source form: **`!$omp | c$omp | *$omp`**
  - Sentinel for free source form: **`!$omp`**
- Impact of directives
  - Processed by compiler if OpenMP mode is on
  - Ignored by the compiler if OpenMP mode if off
- Library
  - Some functions are available
  - C/C++ header file: **`omp.h`**
  - FORTRAN module: **`OMP_LIB`**

Hello World!

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


void main() {
  #pragma omp parallel
   {
    printf( “Hello from thread %d\n”,
      omp_get_thread_num() ) ;
   }
}
```

Header

Directive

Parallel Region

OpenMP functions

```
int omp_get_thread_num() ;
int omp_get_num_threads() ;
```

Hello World!

Execution flow

```
void main() {

#pragma omp parallel
   {

      printf( "Hello
         from thread %d\n",
         omp_get_thread_num()
         ) ;

   }

}
```

Sequential

Parallel

Sequential

Compilation

- Underlying compiler should be aware of OpenMP
- Directives are processed by the compiler
  - Need a flag to activate OpenMP support
  - Example: `-fopenmp` for GNU compiler
  - Classical bug: OpenMP compiler flag missing
- Compiler adds correct flags to link to OpenMP library
  - Link to function API
  - Link to internal functions (calls generated by compiler during directive lowering)
- More details of compilation process for OpenMP in next lecture!

Hello World!

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


void main() {
  #pragma omp parallel
  {
    printf( "Hello from thread %d\n",
      omp_get_thread_num() ) ;
  }
}
```

```
$ gcc –o test –fopenmp test.c

$
```

Execution

- Code execution
  - Regular execution
  - As any multithreaded program!
- Local execution
  - Just run the executable
- Remote/Cluster execution
  - Use SLURM with one process (on one node)

Hello World!

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


void main() {
  #pragma omp parallel
  {
    printf( "Hello from thread %d\n",
      omp_get_thread_num() ) ;
  }
}
```

```
$ salloc –n 1 ./test
Hello from thread 5
Hello from thread 6
Hello from thread 7
Hello from thread 4
Hello from thread 1
Hello from thread 2
Hello from thread 3
Hello from thread 0
```

Runtime Behavior

- OpenMP runtime can be controlled
  - With directives (inside program)
  - With function calls (inside program)
  - With environment variable (outside program)

- Control the number of threads
  - Environment variable `OMP_NUM_THREADS`
  - Accept an integer for the target number of threads
    - Can provide a list of integers for nested parallelism…
  - Apply to all parallel regions
  - Possibility to specify the target number of threads per parallel region
    - Clause `num_threads(int)`
  - Be careful: number of threads not guaranteed!

Hello World!

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


void main() {
  #pragma omp parallel
  {
    printf( "Hello from thread %d\n",
      omp_get_thread_num() ) ;
  }
}
```

```
$ OMP_NUM_THREADS=4 salloc –n 1 ./test
Hello from thread 1
Hello from thread 2
Hello from thread 3
Hello from thread 0
```

Hello World!

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


void main() {
  #pragma omp parallel num_threads(2)
  {
    printf( "Hello from thread %d\n",
      omp_get_thread_num() ) ;
  }
}
```

```
$ salloc –n 1 ./test
Hello from thread 1
Hello from thread 0

$ OMP_NUM_THREADS=4 salloc –n 1 ./test
Hello from thread 1
Hello from thread 0
```

Parallel Region Restrictions

- Before parallel region
  - Only master thread exists
  - Other threads might be asleep, but they cannot be controlled
  - No restriction on ordering when starting a parallel region
- Inside a parallel region
  - All threads execute the same block of instructions
  - Of course, dynamic control flow can be controlled with thread rank
- Implicit synchronization (barrier) at the end of the parallel region
- Branching inwards or outwards of a parallel region is forbidden
  - Example: no `goto` instruction inside a parallel region branching after the parallel region for error message processing
  - Block of parallel region should be inside the same function

Second OpenMP Program

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

int main() {
  float a;
  int   p;
  a = 91680. ; p = 0;
#pragma omp parallel
  {
#ifdef _OPENMP
    p=omp_in_parallel();
#endif
    printf("a: %f ; p: %d\n",a,p);
  }
  return 0;
}
```

OpenMP macro

Defined if OpenMP is processed (value is supported version in format YYYYMM)

→ 201511 = OpenMP 4.5

(see History section for dates and versions)

OpenMP function

`int omp_in_parallel() ;`

```
$ export OMP_NUM_THREADS=4
$ srun –n 1 ./prog
a: 91680. ; p: 1
a: 91680. ; p: 1
a: 91680. ; p: 1
a: 91680. ; p: 1
```

OpenMP Basics

- Data Flow

Data Flow

- Data accessed inside parallel region must have a deterministic behavior
  - Depends on data scope (including declaration location)
  - Depends on data clauses
- By default ➔ data are shared inside a parallel region
  - One variable declared before the parallel region and used inside will be shared by all threads of the region by default
  - Equivalent to clause shared for these variables
    - Example: `shared(a,b)`
- Default behavior can be changed with `default` clause
- Concurrent accesses to shared data
  - Be careful to concurrent read/write
  - Be careful to compiler code generation → volatility

Private Data

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

int main() {
  float a;
  a = 91000.;
  printf( "Out region: %p\n", &a);
#pragma omp parallel
  {
    printf("In region: %p thread %d\n",
      &a, omp_get_thread_num() );
  }
  return 0;
}
```

```
$ gcc –fopenmp –o test test.c

$ OMP_NUM_THREADS=4
$ salloc –n 1 ./test
Out region: 0xbf8d9a9c
In region: 0xbf8d9a9c thread 1
In region: 0xbf8d9a9c thread 2
In region: 0xbf8d9a9c thread 3
In region: 0xbf8d9a9c thread 0
```

Private Data

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

int main() {
  float a;
  a = 91000.;
  printf( "Out region: %p\n", &a);
#pragma omp parallel private(a)
  {
    printf("In region: %p thread %d\n",
      &a, omp_get_thread_num() );
  }
  return 0;
}
```

Clause

Address after parallel region?

$ gcc –fopenmp –o test test.c

$ OMP_NUM_THREADS=4
$ salloc –n 1 ./test
Out region: 0xbf887e2c
In region: 0xbf887dfc thread 0
In region: 0xb67cf2ec thread 3
In region: 0xb6fd02ec thread 2
In region: 0xb77d12ec thread 1

Initialized Private Data

- Private data are not initialized
  - Value can be anything

- Clause to initialize data with value prior to parallel region
  - `firstprivate( var1[, var2]*)`

- Same variable behavior
  - Addresses of variables are not influenced, only values

Initialized Private Data

```
#include <stdio.h>

int main() {
  float a;
  a = 91000.;
#pragma omp parallel default(none) \
    firstprivate(a)
  {
    a = a + 680.;
    printf("a = %f\n",a);
  }
  printf("After region, a = %f\n",
    a);
  return 0;
}
```

```
$ gcc –o prog -fopenmp prog.c

$ export OMP_NUM_THREADS=4
$ salloc –n 1 ./prog
a = 91680.
a = 91680.
a = 91680.
a = 91680.
After region, a = 91000.
```

Region Extent

```
/* File prog.c */
#include <omp.h>

void sub(void);

int main() {
#pragma omp parallel
  {
    sub();
  }
  return 0;
}
```

```
/* File sub.c */
#include <stdio.h>
#include <omp.h>

void sub(void) {
  int p=0;
#ifdef _OPENMP
  p = omp_in_parallel();
#endif
  printf("Parallel? d\n",
    p);
}
```

```
$ gcc –o prog -fopenmp prog.c sub.c
$ export OMP_NUM_THREADS=4
$ salloc –n 1 ./prog
Parallel? 1
Parallel? 1
Parallel? 1
Parallel? 1
```

Region Extent

- Local variables are private even inside other functions

```
int main() {
#pragma omp parallel \
  default(shared)
{
  sub();
}
return 0;
}
```

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

void sub(void) {
  int a;

  a = 91680;
  a = a +
    omp_get_thread_num();
  printf("a = %d\n", a);
}
```

```
$ salloc –n 1 ./prog
a = 91683
a = 91681
a = 91682
a = 91680
```

Arguments

- Transfer by reference acts like the original variable
  - Need to check if memory space is shared or not

```
#include <stdio.h>
int main() {
  int a, b;
  a = 91000;
#pragma omp parallel shared(a) \
   private(b)
  {
    sub(a, &b);
    printf("b = %d\n",b);
  }
  return 0;
}
```

```
#include <omp.h>

void sub(int x, int *y)
{
*y = x +
omp_get_thread_num();
}
```

```
$ salloc –n 1 ./prog
b = 91003
b = 91001
b = 91002
b = 91000
```

Static Variables

- Static variables
  - Global variables
  - Shared by every thread in the same process
- FORTRAN
  - Variables in COMMON block
  - Inside a MODULE
  - With SAVE qualifier
- C
  - Global variables (declared outside function blocks)
  - With static qualifier

Static Variables

```
#include <omp.h>
float a;

int main() {
  a = 91000;
#pragma omp parallel
  {
    sub();
  }
  return 0;
}
```

```
#include <stdio.h>
extern float a;

void sub(void) {
  float b;

  b = a + 680.;
  printf(
    "b = %f\n",b);
}
```

```
$ export OMP_NUM_THREADS=2
$ salloc –n 1 ./prog
b = 91680
b = 91680
```

Dynamic Allocation

- Dynamic memory allocation is allowed inside parallel region
  - Deallocation is authorized as well

- On shared variable
  - Be careful to concurrent call to dynamic allocation functions

- On private variable
  - Pointer will be initialized to a private memory zone
  - If pointer value is transmitted to other threads, this zone can be concurrently accessed

Dynamic Allocation

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main(){
  int n, begin, end, rank, n_tasks, i;
  float *a;
  n=1024, n_tasks=4;
  a=(float *) malloc(n*n_tasks*sizeof(float));
  #pragma omp parallel default(none) \
     private(begin,end,rank,i) shared(a,n) \
     if(n > 512)
  {
    rank=omp_get_thread_num();
    begin=rank*n;
    end=(rank+1)*n;
    for (i=begin; i<end; i++)
       a[i] = 92291. + (float) i;
    printf("Rank: %d ; A[%.4d],...,A[%.4d] : %#.0f,...,%#.0f \n",
           rank,begin,end-1,a[begin],a[end-1]);
  }
  free(a);
  return 0;
}
```

```
$ OMP_NUM_THREADS=4 salloc –n 1 ./prog
Rank: 3 ; A[3072],...,A[4095] : 95363.,...,96386.
Rank: 0 ; A[0000],...,A[1023] : 92291.,...,93314.
Rank: 1 ; A[1024],...,A[2047] : 93315.,...,94338.
Rank: 2 ; A[2048],...,A[3071] : 94339.,...,95362.
```

OpenMP Basics

- Worksharing

Worksharing

- Launching parallel region + access to rank & number of threads
  - Enough to parallelize an application
  - But might be complicated to implement with performance
  - Example: vector addition

- OpenMP proposes worksharing directives
  - Distribute loop iterations: for
  - Distribute blocks of instructions: sections

- Worksharing implies strong constraints
  - Work that can be shared should be independent
  - Compiler and library will not check if directives are correct!
  - Worksharing constructs should be encountered by every thread of the parallel region or none
    - Same behavior as MPI collective communication (for `MPI_COMM_WORLD`)

Parallel Loop

- Distribute iteration domain of loop nest over active threads
  - Directive `for`
  - Apply on perfect loop nest
  - Need a regular for loop (no irregular loops or while loops)
  - Useful inside a parallel region (directive `for` does not launch threads)

- Iteration scheduling can be specified
  - Clause `schedule`
  - Default choice is implementation dependent!

- Scheduling policy allows better load balancing
  - May increase overhead
  - Depend on implementation

- Synchronization
  - Barrier at the end of the loop (default)
  - Can be removed with `nowait` clause

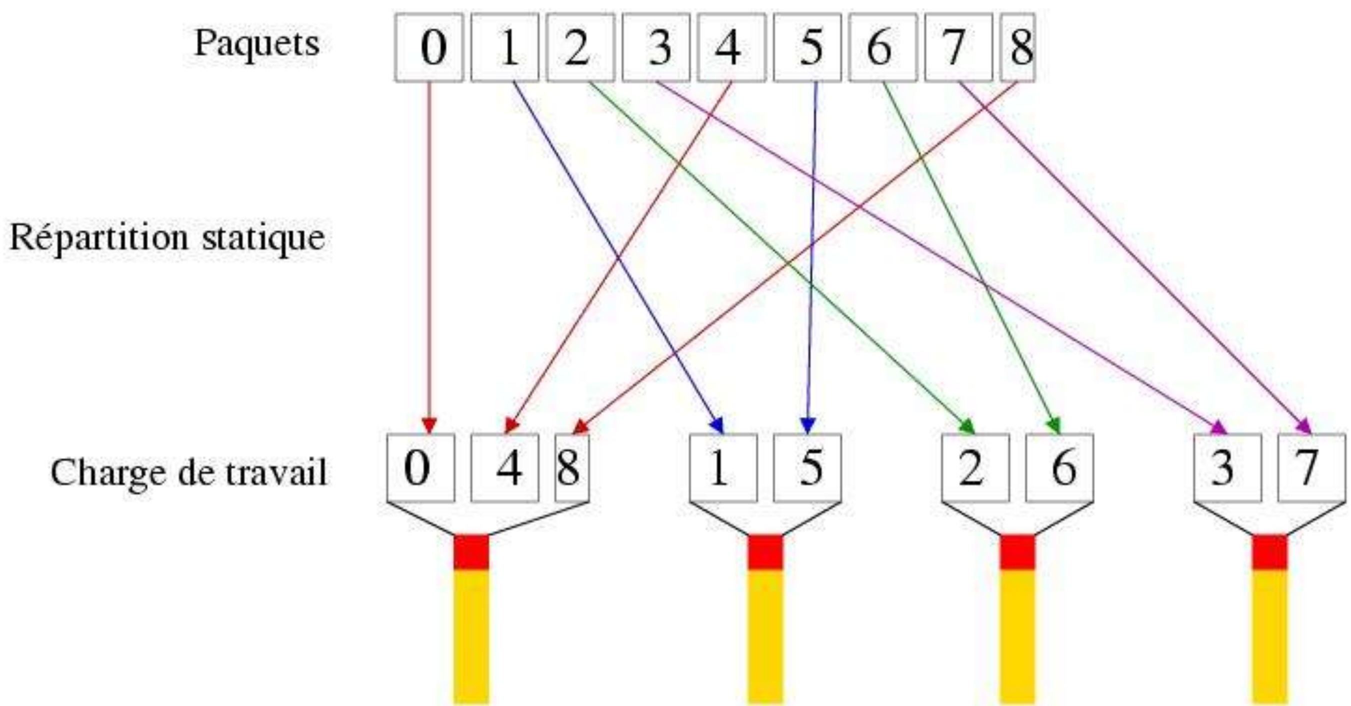
Parallel Loop

```
#include <stdio.h>

int main( int argc, char ** argv ) {
  int N ;

  N = 10 ;

#pragma omp parallel
  {
    int i ;
#pragma omp for schedule(static)
    for ( i = 0 ; i < N ; i++ ) {
      printf( "Thread %d running iteration %d\n",
          omp_get_thread_num(), i ) ;
    }
  }
  return 0 ;
}
```

```
$ gcc –fopenmp –o prog prog.c

$ salloc –n 1 ./prog
Thread 0 running iteration 0
Thread 0 running iteration 1
Thread 0 running iteration 2
Thread 3 running iteration 9
Thread 2 running iteration 6
Thread 2 running iteration 7
Thread 2 running iteration 8
Thread 1 running iteration 3
Thread 1 running iteration 4
Thread 1 running iteration 5
```

Static Scheduling

- Static scheduling splits iteration domain into chunk with equal size (when possible)
  - By default: one chunk of max size
  - When specified, chunk size is constant and chunks are distributed in round robin fashion



Static Scheduling

```
#include <stdio.h>

int main( int argc, char ** argv ) {
  int N ;

  N = 10 ;

#pragma omp parallel
  {
    int i ;
#pragma omp for schedule(static,1)
    for ( i = 0 ; i < N ; i++ ) {
      printf( "Thread %d on iteration %d\n",
          omp_get_thread_num(), i ) ;
    }
  }
  return 0 ;
}
```

```
$ gcc –fopenmp –o prog prog.c

$ salloc –n 1 ./prog
Thread 0 on iteration 0
Thread 0 on iteration 4
Thread 0 on iteration 8
Thread 3 on iteration 3
Thread 3 on iteration 7
Thread 2 on iteration 2
Thread 2 on iteration 6
Thread 1 on iteration 1
Thread 1 on iteration 5
Thread 1 on iteration 9
```

Static Scheduling

```
#include <stdio.h>

int main( int argc, char ** argv ) {
  int N ;

  N = 10 ;

#pragma omp parallel
  {
    int i ;
#pragma omp for schedule(static,2)
    for ( i = 0 ; i < N ; i++ ) {
      printf( "Thread %d on iteration %d\n",
          omp_get_thread_num(), i ) ;
    }
  }
  return 0 ;
}
```

```
$ gcc –fopenmp –o prog prog.c

$ salloc –n 1 ./prog
Thread 0 on iteration 0
Thread 0 on iteration 1
Thread 0 on iteration 8
Thread 0 on iteration 9
Thread 3 on iteration 6
Thread 3 on iteration 7
Thread 1 on iteration 2
Thread 1 on iteration 3
Thread 2 on iteration 4
Thread 2 on iteration 5
```

Schedule Clause

- Scheduling policy can be controlled at runtime
  - Need to specific `schedule(runtime)` for target loops
  - Environment variable `OMP_SCHEDULE`
  - Function call `omp_set_schedule(…)`

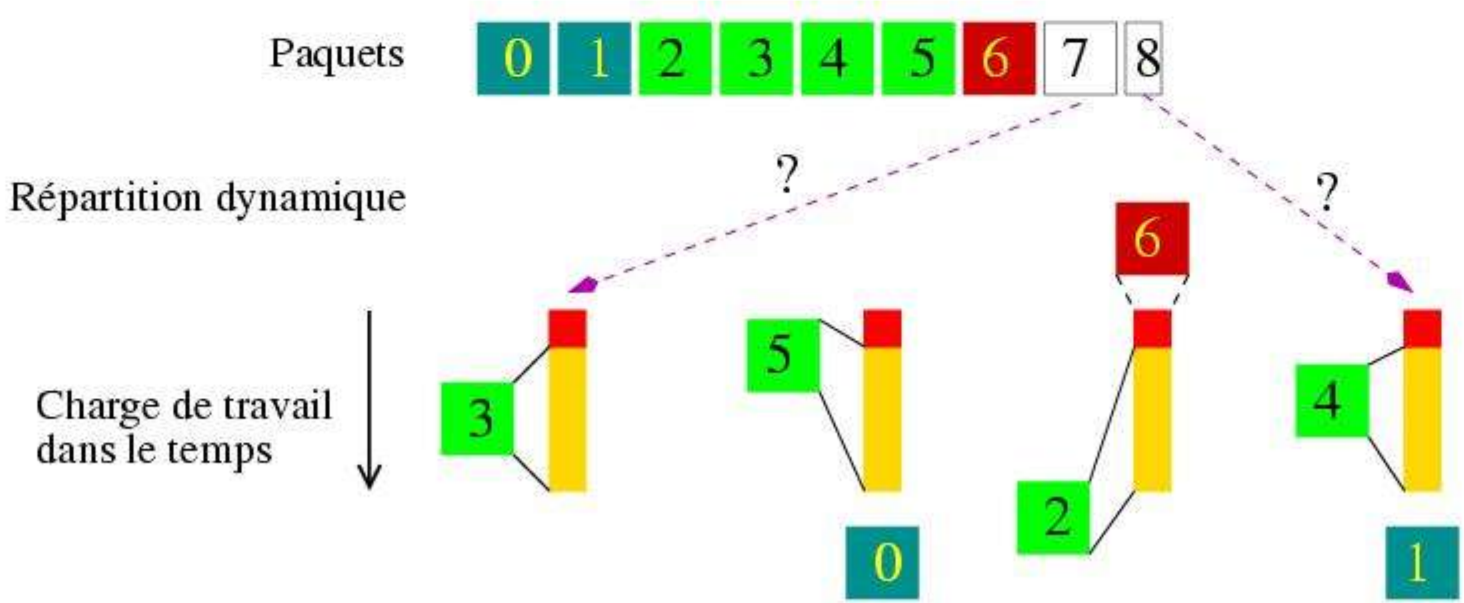
- Scheduling policy and chunk size influence performance
  - Depends on the loop
    - Iteration domain
    - Body size
  - Depends on the target architecture

Dynamic Scheduling

- `schedule(dynamic)`

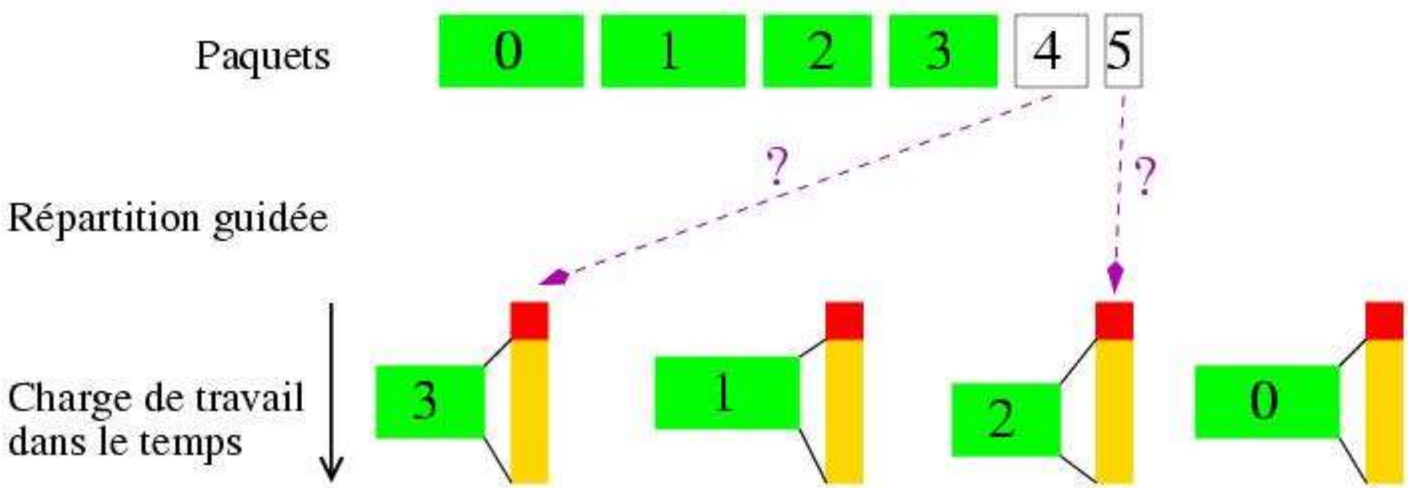
- Iterations are divided into chunks
  - Default chunk size is 1

- Chunks are distributed on-demand to threads
  - As-if behavior



Guided Clause

- Dynamic scheduling may have a large overhead (because of chunk size)
  - Large chunk → less scheduling overhead but less load balancing
  - Small chunk → Load balancing but large scheduling overhead
- Possible solution
  - Schedule(guided)
- Iterations are packed into chunks with variable size
  - Start with large chunks
  - Finish will small chunks (minimal size can be specified)



Reduction

- OpenMP Reduction
  - Associative operation applied on shared variable

- Available operations
  - Arithmetic : +, -, * ;
  - Logic : .AND., .OR., .EQV., .NEQV. ;
  - Function : MAX, MIN, IAND, IOR, IEOR.

- Each thread automatically computes partial result
  - Global result is computed with implementation-dependent strategy
  - GCC: sequential aggregation
  - INTEL: tree-based aggregation

Reduction

```
#include <stdio.h>
#define N 5
int main()
{
  int i, s=0, p=1, r=1;

  #pragma omp parallel
  {
    #pragma omp for reduction(+:s) reduction(*:p,r)
    for (i=0; i<N; i++) {
      s = s + 1;
      p = p * 2;
      r = r * 3;
    }
  }
  printf("s = %d ; p = %d ; r = %d\n",s,p,r);
  return 0;
}
```

```
$ gcc –o prog –fopenmp prog.c

$ export OMP_NUM_THREADS=4
$ salloc –n 1 ./prog
s = 5 ; p = 32 ; r = 243
```

Data Flow in Loops

- Loop construct allows clauses for data flow

- Private
  - Scoping: local declaration
  - Initial value: unknown

- Firstprivate
  - Scoping: same as private
  - Initial value: value before the loop

- Lastprivate
  - Scoping: same as private
  - Initial value: value after last iteration of the loop

Combined Regions

- Some directives may be merged together

- Examples:
  - #pragma omp parallel
    - #pragma omp for
  - ➔
  - #pragma omp parallel for

- Clauses
  - Union of possible clauses for both directives

- End of directive implies global synchronization (end of parallel region)

Loop Nest

- Loop directive may apply on loop nest
  - Clause `collapse(int)`
  - Restrictions: perfect loop nest
- Argument
  - Nest depth (i.e., number of loops that will be distributed over threads)
  - Available since OpenMP 3.0

Synchronization

- Introduction

Synchronizations

- Threads launched by a parallel region have no constraints
  - Threads start at the beginning of parallel region
  - They stop at the end of the parallel region

- But synchronization mechanisms might be used
  1. To ensure control checkpointing (global barrier)
  2. To be sure that some pieces of code are executed by only one thread at a time (mutual exclusion).
  3. To synchronize two tasks (locks).

Synchronization

- Barrier

Barrier

- Synchronization of control flow between all threads inside the same team

- Implicit barrier
  - (Almost) every construct involve an implicit barrier at the end
  - Synchronize all threads from the same team

- Explicit barrier
  - Directive `barrier`

```
#pragma omp barrier
```

Synchronization

- Atomic Operations

Atomicity

```
int i = 0 ;

int main(int argc,
     char ** argv ) {
  i += argc ;
  printf( “%d\n”, i ) ;
}
```

```
movl %esp, %ebp
movl i, %eax
addl 8(%ebp), %eax
```

Atomicity in OpenMP

- Construct for atomicity
- Ensure that following statement is executed atomically

```
#pragma omp atomic
```

- Following statement can be of specified form (usually variable update with arithmetic operation)

Atomicity in OpenMP

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

int main() {
    int c, r;

    c = 91680;
    #pragma omp parallel private(r)
    {
        r = omp_get_thread_num();

        #pragma omp atomic
          c++;

        printf("R: %d ; C: %d\n", r, c );
    }
    printf("After region, c: %d\n", c);
    return 0;
}
```

```
$ gcc –o prog –fopenmp prog.c

$ export OMP_NUM_THREADS=4
$ salloc –n 1 ./prog
R: 1 ; C: 91683
R: 0 ; C: 91681
R: 2 ; C: 91684
R: 3 ; C: 91682
After region, c: 91684
```

Atomicity in OpenMP

- Atomic statement
  - x=x~(op)~exp ;
  - x=exp~(op)~x ;
  - x=f(x,exp) ;
  - x=f(exp,x) ;

- `(op)` should be
  - +, --, *, /,
  - .AND., .OR., .EQV., .NEQV..

- `f` is a function
  - MAX, MIN, IAND, IOR, IEOR.

- `exp` expression that does not depend on x

Synchronization

- Critical Region

Critical region

- Extension of atomic operation to a region (i.e., block of statements)
  - Threads execute the inner block one at a time
  - Dynamic extent

- Directive followed by a block

```
#pragma omp critical
```

- Possibility to create named critical region
  - By default, region is anonymous
  - It applies to every region with the same name (including anonymous)

- Atomic operation and critical region have different performance…

Critical region

```
#include <stdio.h>

int main()
{
    int s, p;

    s = 0, p = 1;
    #pragma omp parallel
    {
        #pragma omp critical
        {
            s++;
            p*=2;
        }
    }
        printf("Sum & product: %d, %d\n", s, p);
    return 0;
}
```

Synchronization

- Exclusive Execution

Exclusive Execution

- Mechanism to allow only one thread to execute a target block
  - Block executed only once
- OpenMP proposes 2 directives
  - `#pragma omp master`
  - `#pragma omp single`
- Main goal is the same but definition, code generation and runtime behavior are different…

Master Construct

- Define a block of instructions that will be executed only by master thread

- No clauses
  - Directive is followed by a block of statements

- No synchronization at the entrance or at the end of the master block

Master Construct

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

int main()
{
    int r;
    float a;

    #pragma omp parallel private(a,r)
    {
        a = 91680.;

        #pragma omp master
        {
            a = -91680.;
        }

        r = omp_get_thread_num();
        printf("R: %d ; A: %f\n",r,a);
    }
    return 0;
}
```

Single construct

- Define a block of instructions to be executed only by one thread
  - Generalized master
  - `#pragma omp single`

- No constraints on which thread will execute the block
  - Maybe the first one

- Implicit barrier at the end of single construct
  - Single construct can be seen as collective operation!
  - Clause `nowait` to remove this barrier

Single construct

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

int main()
{
    int rang;
    float a;

    #pragma omp parallel private(a,rang)
    {
        a = 91680.;

        #pragma omp single
        {
            a = -91680.;
        }

        rang=omp_get_thread_num();
        printf("Rang : %d ; A vaut : %f\n",rang,a);
    }
    return 0;
}
```

Synchronization

- Locks

Lock Definition

- Locks: object for mutual exclusion
  - Allow shared-data protection
  - Enable critical sections
- States
  - locked (owned by a thread)
  - unlocked (no ownership)
- Actions
  - Acquire the lock (lock)
  - Release the lock (unlock)
- State update
  - Lock on locked status ➔ wait for release
  - Lock on unlocked status ➔ update status to locked
  - Unlock on unlocked status ➔ nothing (not recommended)
  - Unlock on locked status ➔ release

2-bit FSA

Nested/Recursive: Allow same owner multiple times

OpenMP Locks

- Opaque type for locks
  - `omp_lock_t`
- Lock initialization
  - `void omp_init_lock( omp_lock_t *lock);`
- Lock destruction
  - `void omp_destroy_lock( omp_lock_t *lock);`

Initialization

- Acquire a lock
  - `void omp_set_lock( omp_lock_t *lock);`
- Release a lock
  - `void omp_unset_lock( omp_lock_t *lock);`
- Try to acquire a lock
  - `int omp_test_lock( omp_lock_t *lock);`

Actions

OpenMP Nested Locks

- Extension to nested locks
- Opaque type for locks
  - `omp_nest_lock_t`
- Lock initialization
  - `void omp_init_nest_lock( omp_nest_lock_t *lock);`
- Lock destruction
  - `void omp_destroy_nest_lock( omp_lock_t *lock);`

Initialization

- Acquire a lock
  - `void omp_set_nest_lock( omp_nest_lock_t *lock);`
- Release a lock
  - `void omp_unset_nest_lock( omp_nest_lock_t *lock);`
- Try to acquire a lock
  - `int omp_test_nest_lock( omp_nest_lock_t *lock);`

Actions

OpenMP Lock Example

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

int main()
{
  int n ;
  int c ;
  omp_lock_t l ;
  omp_init_lock( &l ) ;

#pragma omp parallel
{
  #pragma omp single
  {
    n = omp_get_num_threads() ;
  }
  omp_set_lock(&l) ;

  c++;

  omp_unset_lock(&l) ;
}

omp_destroy_lock( &l ) ;

printf( "Number of threads
function:%d
  count:%d\n", n, c ) ;

return 0 ;
}
```

Protect variable c

Generalized way to perform atomic operations and critical sections

Lock Internals

- Example of lock implementation through a *mutex*
- Main structure + slot for thread queue

```
typedef struct slot_s {
  thread_t *thread;
  struct slot_s *next;
} slot_t;

typedef struct {
  /* Counter for waiting threads */
  volatile int nb_threads;
  /* List of blocked threads */
  volatile slot_t *list_first;
  volatile slot_t *list_last;
  /* Spinlock to control accesses to internal variables */
  spinlock_t lock;
} mutex_t;
```

Need a lock to check read/write accesses to the mutex structure!

Mutex Internals

- Function to acquire the lock

```
void mutex_lock (mutex_t * m)
{
    slot_t slot;
    spinlock (&(m→lock));
    if (m→nb_thread == 0) {
        m→nb_thread = 1;
        spinunlock (&(m→lock));
    } else {
        slot.thread = thread_self ();
        enqueue (&slot, m);
        thread_self ()→status = blocked;
        register_spinunlock (&(m→lock));
        yield ();
    }
}
```

`nb_thread` can be safely checked because of spinlock

Call to main scheduler because current thread is blocked

Mutex Internals

- Function to release the lock

```
void mutex_unlock (mutex_t * m)
{
    slot_t *slot;
    spinlock (&(m→lock));
    if (m→list_first != NULL) {
        slot = dequeue (m);
        wake (slot→thread);
    } else {
        m→nb_thread = 0;
    }
    spinunlock (&(m→lock));
}
```

`list_first` can be safely checked because of spinlock

Thread releasing the lock will wake up the next thread waiting for the mutex

Mutex Internals

- Function to test lock status

```
int mutex_trylock (mutex_t * m)
{
    slot_t slot;
    spinlock (&(m→lock));
    if (m→nb_thread == 0) {
        m→nb_thread = 1;
        spinunlock (&(m→lock));
        return 0;
    }
    spinunlock (&(m→lock));
    return 1;
}
```

Recursive Mutex Internals

- Function to acquire the lock

```
void mutex_lock (mutex_t * m) {
    slot_t slot;
    spinlock (&(m→lock));
    if (m→nb_thread == 0) {
    m→nb_thread = 1;
    m→owner = thread_self ();
    spinunlock (&(m→lock));
    } else {
        if (m→owner == thread_self ()) {
            m→step++;
            spinunlock (&(m→lock));
        } else {
            slot.thread = thread_self ();
            enqueue (&slot, m);
            thread_self ()→status = blocked;
            register_spinunlock (&(m→lock));
            yield ();
        }
    }
}
```

Recursive Mutex Internals

- Function to release the lock

```
void mutex_unlock (mutex_t * m) {
    slot_t *slot;
    spinlock (&(m→lock));
    if (m→step == 1) {
        m→step--;
        if (m→list_first != NULL) {
            slot    = dequeue (m);
            wake (slot→thread);
        } else {
            m→nb_thread = 0;
        }
    } else {
        m→step--;
    }
    spinunlock (&(m→lock));
}
```

Recursive Mutex Internals

- Function to test lock status

```
int mutex_trylock (mutex_t * m){
    slot_t slot;
    spinlock (&(m→lock));
    if (m→nb_thread == 0) {
        m→nb_thread = 1;
        m→owner = thread_self ();
        spinunlock (&(m→lock));
        return 0;
    } else {
        if (m→owner == thread_self ()) {
            m→step++;
            spinunlock (&(m→lock));
            return 0;
        }
    }
    spinunlock (&(m→lock));
    return 1;
}
```