Super Computing and Distributed Systems Camp
Catay, Santander, Colombia
August 15-22, 2010

Introduction to OpenMP Programming

Robinson Rivas Suarez
robinson.rivas@ciens.ucv.ve

Universidad Central de Venezuela

Agenda

- What OpenMP stands for?
- Differences between OpenMP and MPI
- Parallel Regions
- Tags for parallel work
- Data sharing
- Explicit synchronization
- Scheduling Instructions

OpenMP at a glance

- OpenMP **IS:**
- Compiler directives **and** a library for multithread programming
- Available for Fortran and C/C++, several companies involved
- Support for parallel data model
- Incremental parallelism
- Combines serial and parallel code in the same source
- Simple: it allows to run our serial code without modificactions (almost)

OpenMP at a glance

- OpenMP **IS NOT:**
- A library for message-passing programming
- Available for any major language
- Suitable for high-scale parallelism, i.e. programming over the grid
- 100% portable: programs must be re-compiled over new architectures. OpenMP exploits architecture-dependant advantages

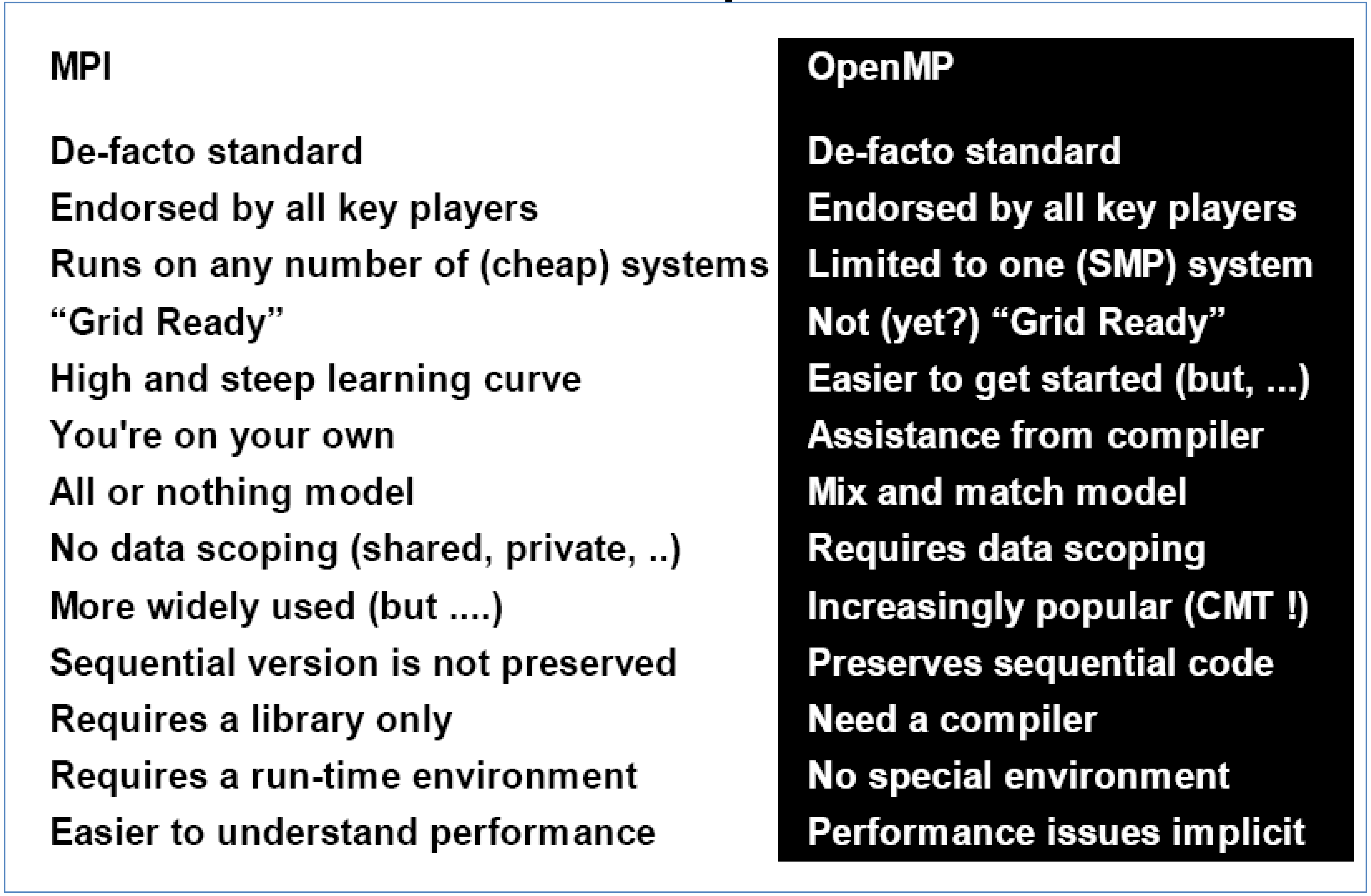
OpenMP vs MPI

- MPI uses the message passing paradigm, i.e., distributed memory. OpenMP uses fork-join model on shared memory
- MPI can exploit massive parallelism over hundreds or thousands of nodes. OpenMP uses physical access on relatively small number of cores
- Due to its nature, none of MPI nor OpenMP are deterministic.
- OpenMP have scheduling instructions. MPI doesn’t have it.

BUT: in real world, OpenMP and MPI does not compete!! They interact with each another to take advantages of specific architectures.

OpenMP vs MPI

A more detailed comparison



MPI	OpenMP
De-facto standard	De-facto standard
Endorsed by all key players	Endorsed by all key players
Runs on any number of (cheap) systems	Limited to one (SMP) system
“Grid Ready”	Not (yet?) “Grid Ready”
High and steep learning curve	Easier to get started (but, ...)
You're on your own	Assistance from compiler
All or nothing model	Mix and match model
No data scoping (shared, private, ..)	Requires data scoping
More widely used (but ....)	Increasingly popular (CMT !)
Sequential version is not preserved	Preserves sequential code
Requires a library only	Need a compiler
Requires a run-time environment	No special environment
Easier to understand performance	Performance issues implicit

OpenMP at a glance

```
C$OMP FLUSH
```

```
#pragma omp critical
```

```
C$OMP THREADPRIVATE(/ABC/)
```

```
call omp_test_lock(jlok)
```

```
C$OMP parallel do shared(a, b, c)
```

```
call OMP_INIT
```

```
C
```

```
C$OMP  SINGLE PRIV
```

```
ynamic”
```

```
C$OMP PARALLEL D
```

http://www.openmp.org

Current specification is about 250 pages

(C/C++ and Fortran)

```
C$OMP PARALLEL  REDUCTION (+: A, B)
```

```
C$OMP SECTIONS
```

```
#pragma omp parallel for private(A, B)
```

```
!$OMP  BARRIER
```

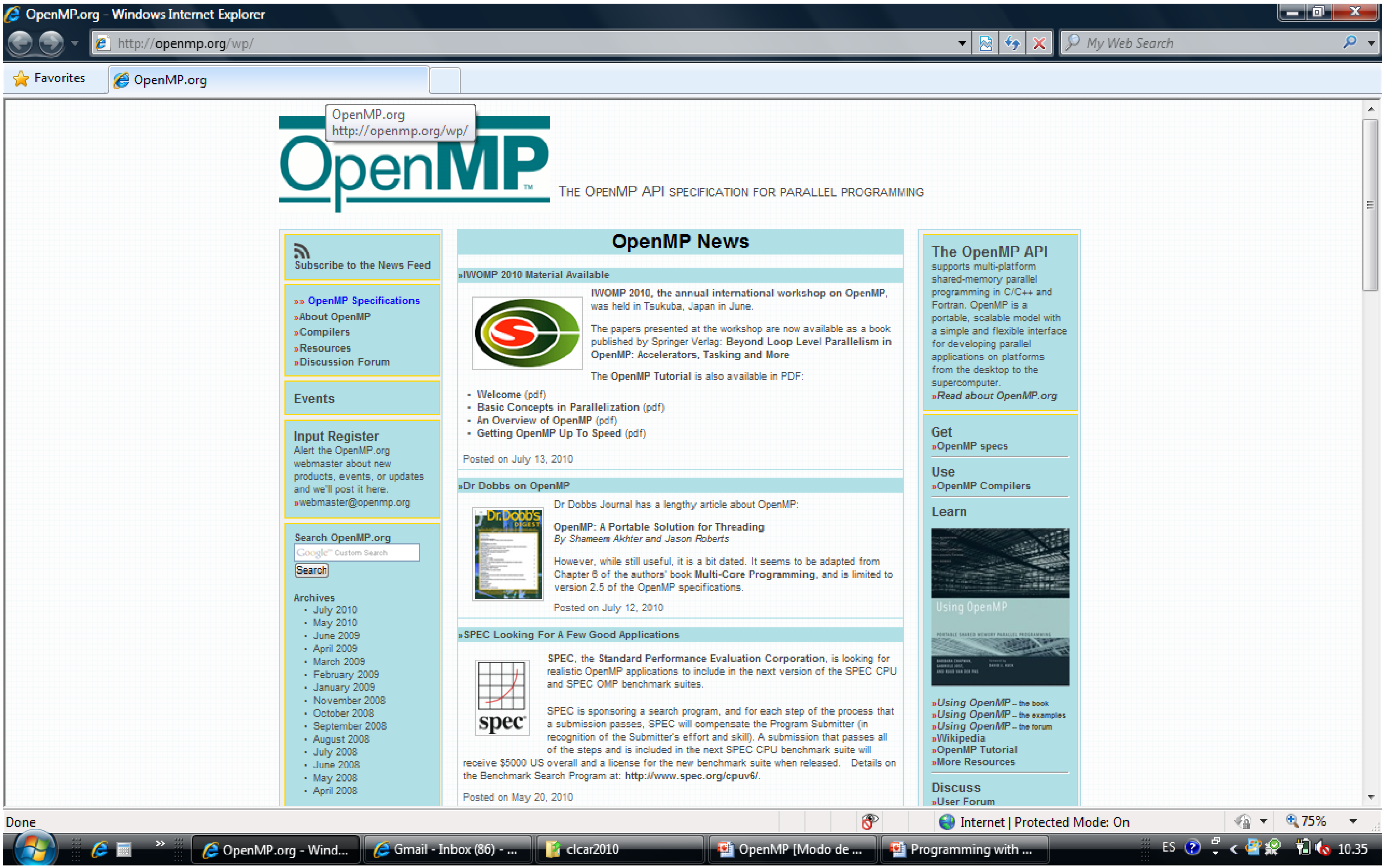
```
C$OMP PARALLEL COPYIN(/blk/)
```

```
C$OMP DO lastprivate(XX)
```

```
Nthrds = OMP_GET_NUM_PROCS()
```

```
omp_set_lock(lck)
```

OpenMP at a glance



OpenMP.org - Windows Internet Explorer

http://openmp.org/wp/

THE OPENMP API SPECIFICATION FOR PARALLEL PROGRAMMING

Subscribe to the News Feed

» OpenMP Specifications
»About OpenMP
»Compilers
»Resources
»Discussion Forum

Events

Input Register
Alert the OpenMP.org webmaster about new products, events, or updates and we'll post it here.
»webmaster@openmp.org

Search OpenMP.org

Archives
- July 2010
- May 2010
- June 2009
- April 2009
- March 2009
- February 2009
- January 2009
- November 2008
- October 2008
- September 2008
- August 2008
- July 2008
- June 2008
- May 2008
- April 2008

OpenMP News

»IWOMP 2010 Material Available

IWOMP 2010, the annual international workshop on OpenMP, was held in Tsukuba, Japan in June.

The papers presented at the workshop are now available as a book published by Springer Verlag: Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More

The OpenMP Tutorial is also available in PDF:

- Welcome (pdf)
- Basic Concepts in Parallelization (pdf)
- An Overview of OpenMP (pdf)
- Getting OpenMP Up To Speed (pdf)

Posted on July 13, 2010

»Dr Dobbs on OpenMP

Dr Dobbs Journal has a lengthy article about OpenMP:

OpenMP: A Portable Solution for Threading
By Shameem Akhter and Jason Roberts

However, while still useful, it is a bit dated. It seems to be adapted from Chapter 6 of the authors' book Multi-Core Programming, and is limited to version 2.5 of the OpenMP specifications.

Posted on July 12, 2010

»SPEC Looking For A Few Good Applications

SPEC, the Standard Performance Evaluation Corporation, is looking for realistic OpenMP applications to include in the next version of the SPEC CPU and SPEC OMP benchmark suites.

SPEC is sponsoring a search program, and for each step of the process that a submission passes, SPEC will compensate the Submitter (in recognition of the Submitter's effort and skill). A submission that passes all of the steps and is included in the next SPEC CPU benchmark suite will receive $5000 US overall and a license for the new benchmark suite when released. Details on the Benchmark Search Program at: http://www.spec.org/cpuv6/.

Posted on May 20, 2010

The OpenMP API
supports multi-platform shared-memory parallel programming in C/C++ and Fortran. OpenMP is a portable, scalable model with a simple and flexible interface for developing parallel applications on platforms from the desktop to the supercomputer.
»Read about OpenMP.org

Get
»OpenMP specs

Use
»OpenMP Compilers

Learn

»Using OpenMP – the book
»Using OpenMP – the examples
»Using OpenMP – the forum
»Wikipedia
»OpenMP Tutorial
»More Resources

Discuss
»User Forum

Done | Internet | Protected Mode: On | 75%

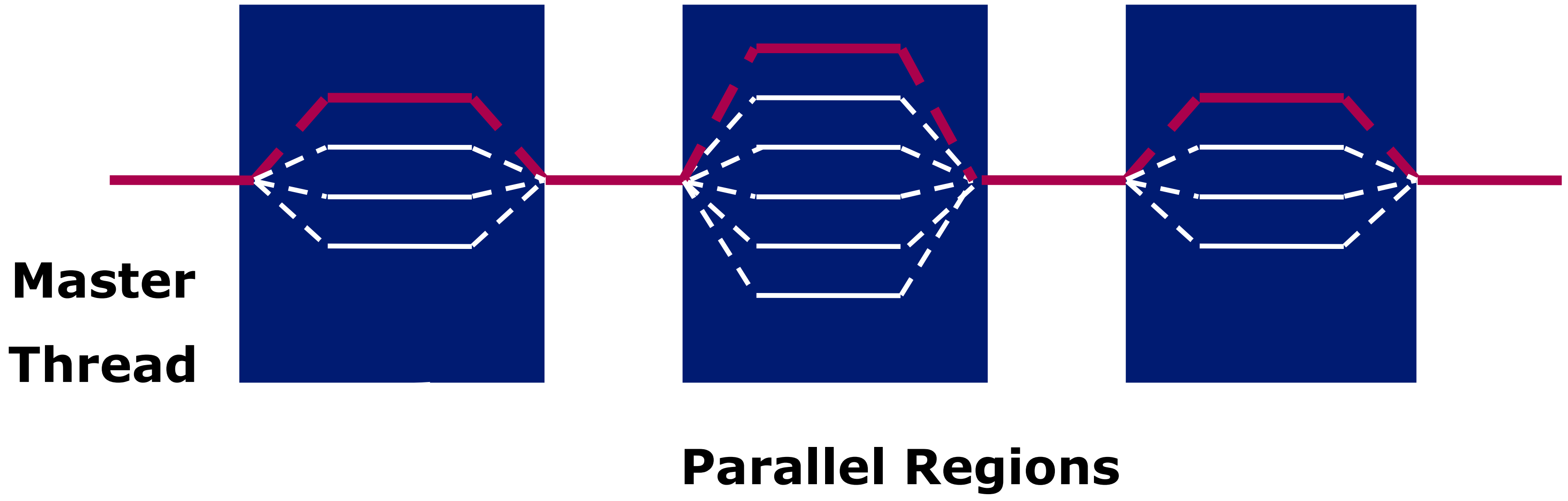
OpenMP architecture

- fork-join model
- Construction blocks for parallel execution
- Construction blocks for data scope management
- Construction blocks for synchronization
- API (Application Program Interface) for programming tuning

OpenMP architecture

fork-join model:

- **Master thread divides itself in *sub-threads* as it is needed**
- **Incremental parallelism: sequential code becomes parallel depending on problem's conditions**



OpenMP syntax

- Most of OpenMP blocks are really compiler directives. In C/C++ they are called *pragmas*. Syntax is:

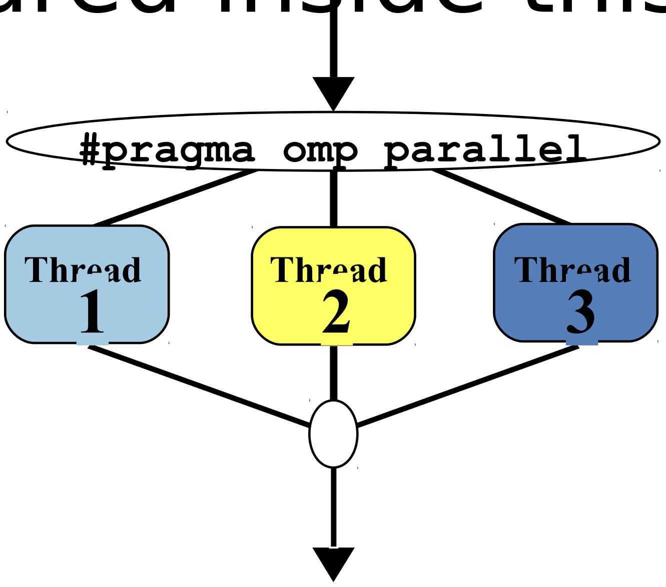
#pragma omp ***construct [clause [clause]…]***

Parallel regions

- Pragma parallel defines a parallel region on an structured code
- Threads created using this prgama, synchronizes at the end of the block
- By default, data is shared inside this region

C/C++ :

```
#pragma omp parallel
        {
                code
        }
```



How many threads?

- The environment variable defines how many threads we will create.

```
set OMP_NUM_THREADS=4
```

- There is not default for this. In most systems, # of threads = # of cores. However, you can define more threads than physical cores.
  - Intel® compilers uses this standard

Example

Use the pragma to parallelize this code

```
int main()
{
    hello();
}

int hello()
{
    int i;

    for(i=0;i<10;i++)
    {
        printf("Hello World!!\n");
        sleep(1)
    }
}
```

Solution

```
int main()
{
#pragma omp parallel
    hello();
}

int hello()
{
    int i;

    for(i=0;i<10;i++)
    {
        printf("Hello world!!\n");
        sleep(1)
    }
}
```

Parallel for

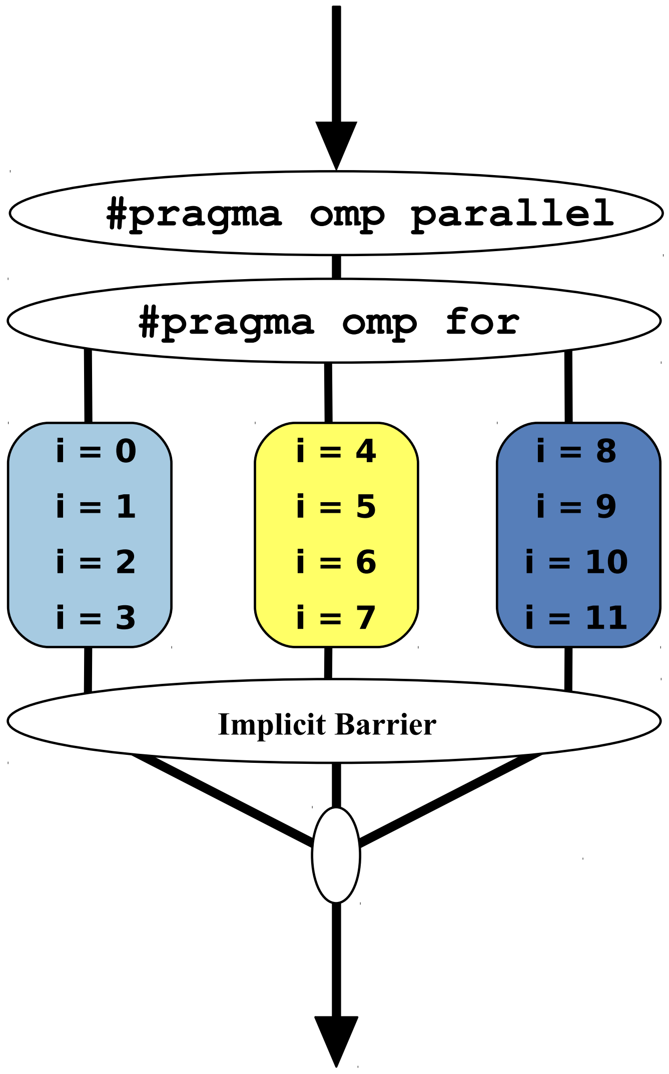
```
#pragma omp parallel
#pragma omp for
  for (i=0; i<N; i++){
  Do_Work(i);
  }
```

- Divide iterations among processors
- Must be inside the parallel region
- Must precede the `for` clause

Parallel for

```
#pragma omp parallel
#pragma omp for
   for(i = 0; i < 12; i++)
      c[i] = a[i] + b[i]
```

- Each thread asigned with a number of iterations.
- Iterations are asigned with round-robin policy
- Programmer must deal with possible side effects
- There is an implicit barrier at the end of threads



Parallel for

- This two codes are equivalent

```
#pragma omp parallel
{
    #pragma omp for
    for (i=0; i< MAX; i++)
        { res[i] = huge();
        }
}
```

```
#pragma omp parallel for
    for (i=0; i< MAX; i++)
        { res[i] = huge();
        }
```

Data sharing

- OpenMP uses shared memory as default model
- Most of variables are shared by default
- Global variables are always shared
- User can modify the behavior of variables, except for global ones

Data sharing

- Some exceptions apply to data sharing:
  - Local variables of functions called from parallel regions are private
  - Variables defined inside parallel blocks are private
  - Index variables of for statements are by default private

C/C+: the first variable of the **`for`** after the **`#pragma omp for`** is private

Data sharing

- Default status can be modified

```
default (shared | none)
```

- Data scope attributes

```
shared(varname,…)
private(varname,…)
```

Data scope

- In case of private variables, compiler assigns one variable per thread
- Thread variables are language and compiler dependant, thus, inicialization, dafault values and space depends on compiler spec

Example

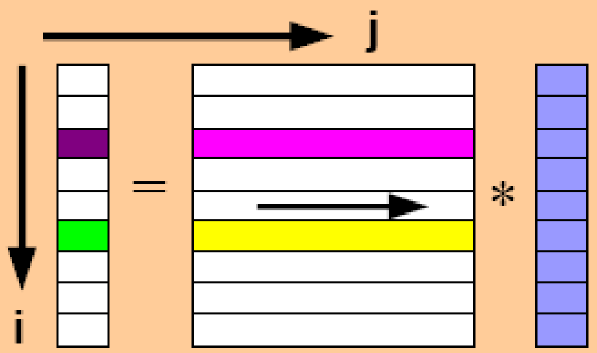
- Given a=[1,2,3,4,5,6], b=[2,4,6,8,10,12] and N=6

```
void* work(float* c, int N)
      { float x, y; int i;
#pragma omp parallel for private(x,y)
        for(i=0; i<N; i++)
           {x = a[i]; y = b[i];
                 c[i] = x + y;
           }
      }
```

- What values do this code generates if Processors are 2, 3?
- What happens if x,y are NOT private?

Example

```
#pragma omp parallel for default(none) \
            private(i,j,sum) shared(m,n,a,b,c)
for (i=0; i<m; i++)
{
   sum = 0.0;
   for (j=0; j<n; j++)
     sum += b[i][j]*c[j];
   a[i] = sum;
 }
```



TID = 0

```
for (i=0,1,2,3,4)
 i = 0
   sum =    b[i=0][j]*c[j]
   a[0] = sum
 i = 1
   sum =    b[i=1][j]*c[j]
   a[1] = sum
```

TID = 1

```
for (i=5,6,7,8,9)
 i = 5
   sum =    b[i=5][j]*c[j]
   a[5] = sum
 i = 6
   sum =    b[i=6][j]*c[j]
   a[6] = sum
```

Example

- Inner product: do this code work?

```
float dot_prod(float* a, float* b, int N)
{
  float sum = 0.0;
#pragma omp parallel for shared(sum)
   for(int i=0; i<N; i++) {
     sum += a[i] * b[i];
   }
  return sum;
}
```

Critical regions

- In many cases, public access to variables is dangerous, because different threads can modify erroneously the values. In this case, it is necessary to define a *critical region* for those values that need to be *protected*
- OpenMP provides a pragma to define critical regions

```
#pragma omp critical [(lock_name)]
```

Example

- Inner product revisited

```
float dot_prod(float* a, float* b, int N)
{
  float sum = 0.0;
#pragma omp parallel for shared(sum)
   for(int i=0; i<N; i++) {
#pragma omp critical
     sum += a[i] * b[i];
   }
  return sum;
}
```

Critical regions

- In a critical region, threads waits its turn to execute the line(s) of code defined in the inner block.
- Programmer can assign a name to each critical region. This could lead to better performance when code is executed

Example

```
float R1, R2;
#pragma omp parallel
{ float A, B;
#pragma omp for
    for(int i=0; i<niters; i++){
    B = big_job(i);
#pragma omp critical(R1_lock)
      consum (B, &R1);
    A = bigger_job(i);
#pragma omp critical (R2_lock)
      consum (A, &R2);
  }
}
```

Critical regions

- Critical regions must be used carefully. In the worst case, a bad usage of `critical` pragma could lead to a serial execution
- Not every code works well with critical. Imagine

```
for(int i=0; i<N; i++) {
     sum = a[i];
#pragma omp critical
     sum += a[i] * b[i];
   }
```

Reduction

- Usually, critical pragma is not the best solution due to bottlenecks. *Reduction* is a better alternative.
- Reduction allows programmers to put in a single variable the join result of a series of calculations.
- This permits threads to use private variables, and *reduce* them to a shared variable at the end of the execution

Reduction

- Reduction pragma:
  `reduction (`*`op`*` : `*`list`*`)`
- Variables in “*list*” must be in shared mode inside the parallel region
- When reduction code begins, each thread **makes a copy** of the variables, and initializes them depending on the operator **`op`**
- Once threads finalize its execution, they puts the final value using **`op`** operator in a single shared copy of the variable.

Reduction

- In this example, each thread has their own copy of **`sum`**
- All the copies of **`sum`** are added together at the end of the computation, in a single “global” variable in the master thread

```
#pragma omp parallel for reduction(+:sum)
   for(i=0; i<N; i++)
      { sum += a[i] * b[i];
      }
```

Reduction

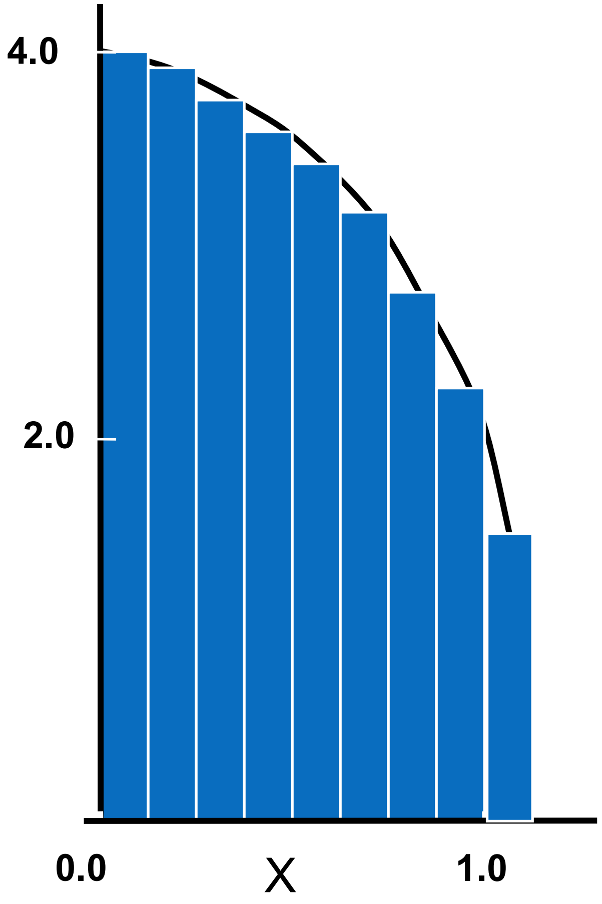
- OpenMP defines a series of both commutative and associative operators
- Initial values for variables are assigned depending on *neutral* erat

Op	Initial value
+	0
*	1
-	0
^	0

Op	Initial value
&	~0
|	0
&&	1
||	0

Example

- A known problem!!!



Given $f(x) = \frac{4.0}{(1+x^2)}$

then $\pi = \int_0^1 \frac{4.0}{(1+x^2)} dx$

Numeric Integration

```
static long num_steps=100000; double step, pi;

void main()
{   int i;
    double x, sum = 0.0;
    step = 1.0/(double) num_steps;
    for (i=0; i< num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0 + x*x);
    }
    pi = step * sum;
    printf("Pi = %f\n",pi);
}
```

Example

- Parallelize this code thinking on:
  - Variables to share
  - Variables to reduce

```
static long num_steps=100000;
double step, pi;

void main()
{   int i;
    double x, sum = 0.0;

    step = 1.0/(double) num_steps;
    for (i=0; i< num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0 + x*x);
    }
    pi = step * sum;
    printf("Pi = %f\n",pi);
}
```

Scheduling

- Some times, the amount of time used for iterations is not uniform for all cases. For instance, random values or I/O problems can affect the time used by each processor
- So, it is good idea to have a way to decide *how to assign* the values of iteration indexes to different processes
- Example: think in a gas simulation. None of the particles spend the same time to calculate its energy

Scheduling

- **`schedule`** clause defines how to assign the values to processors

`schedule(static [,chunk])`

Each thread is assigned with the same “chunk” size values, using round-robin policy

`schedule(dynamic[,chunk])`

Each thread takes “chunk” values to iterate. After processing those values, the thread takes more “chunk” values

`schedule(guided[,chunk])`

Dynamic planning, beginning with the bigger “chunk”.

Scheduling

- When to use scheduling?

Schedule	Use it when…
STATIC	Each iteration is suposed to spent the same time
DYNAMIC	Unpredictable, threads have non deterministic behavior
GUIDED	Same as dynamic, but more efficient scheduler

Scheduling

- Example of **`schedule`** clause

```
#pragma omp parallel for schedule (static, 8)
    for( int i = start; i <= end; i += 2 )
    {
       if ( TestForPrime(i) )  gPrimesFound++;
    }
```

- In this case, every thread has 8 values to search for. Note that programmer must know the size of both chunk an iterations. Portions are distributed statically

Scheduling

- Example of **`schedule`** clause

```
#pragma omp parallel for schedule (dynamic, 5)
    for( int i = start; i <= end; i += 2 )
    {
       if ( TestForPrime(i) )  gPrimesFound++;
    }
```

- In this case, every thread have chunks of size 5 for the first time. Then, new executions takes portions of 5 values once they finish partial executions

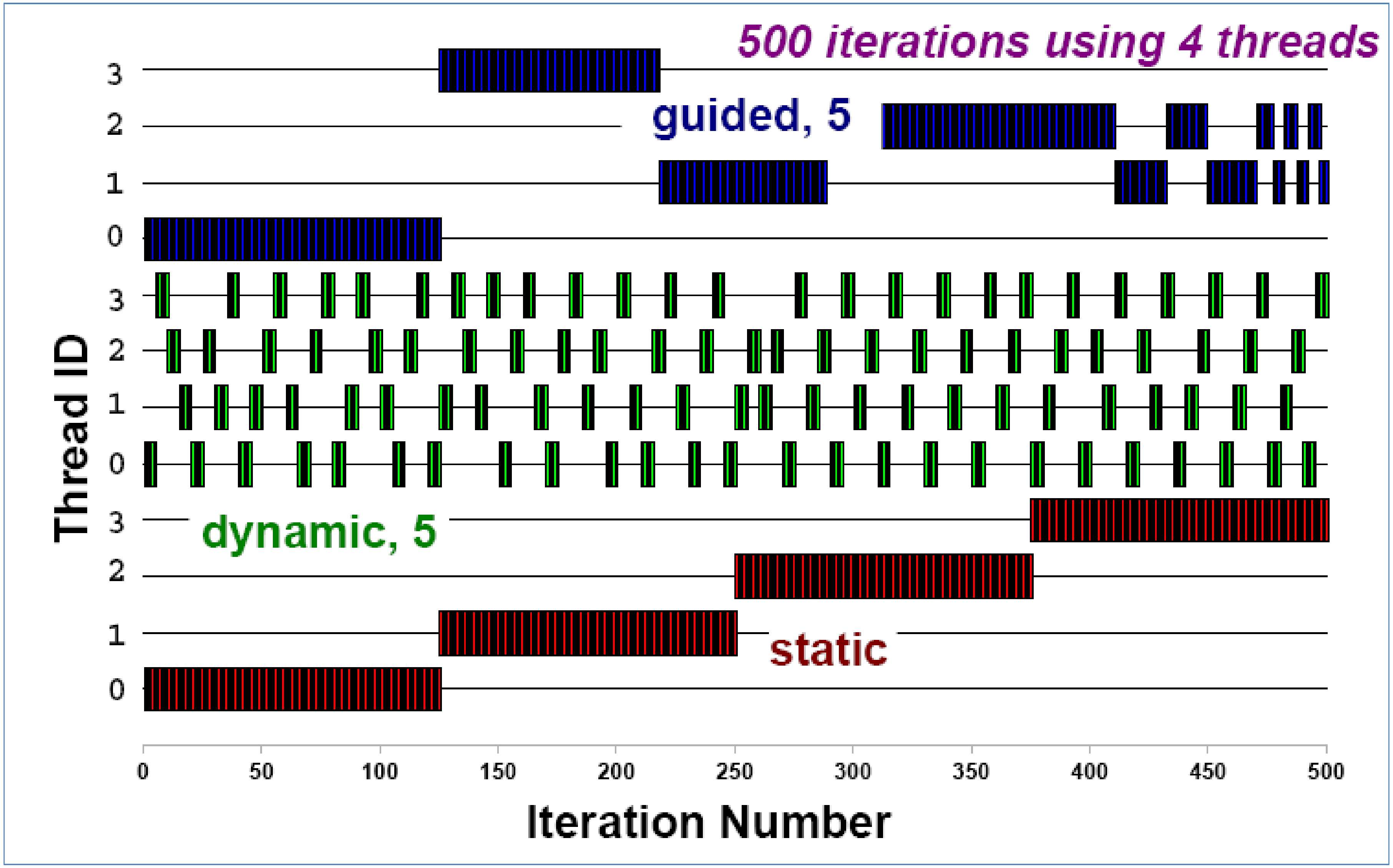
Scheduling

- Example of **`schedule`** clause

```
#pragma omp parallel for schedule (guided, 8)
    for( int i = start; i <= end; i += 2 )
    {
       if ( TestForPrime(i) )  gPrimesFound++;
    }
```

- In this case, compiler decides the size of chunks and this size decreases exponentially. It is often more efficient than dynamic behavior

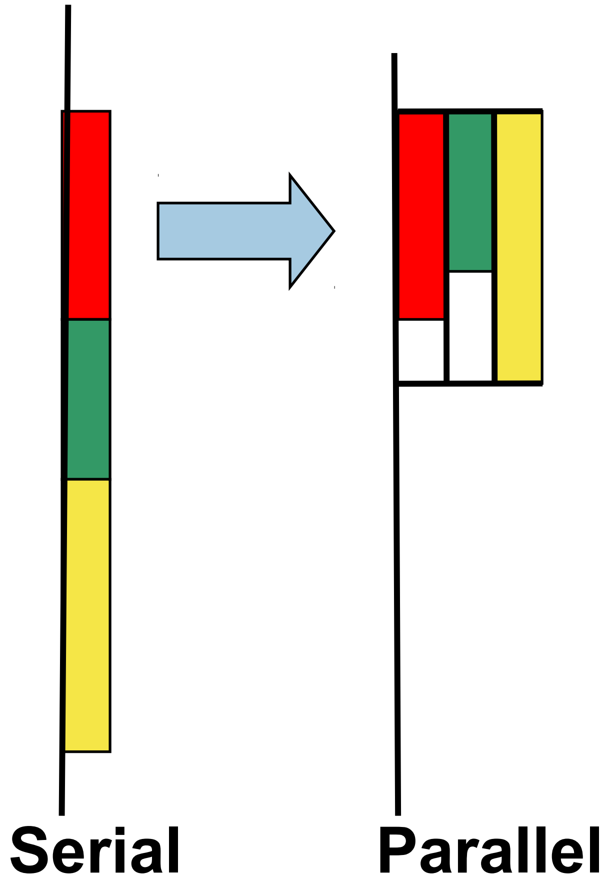
Scheduling


500 iterations using 4 threads
guided, 5
dynamic, 5
static
Thread ID
3
2
1
0
3
2
1
0
3
2
1
0
0
50
100
150
200
250
300
350
400
450
500
Iteration Number

Parallel sections

- Not only iterations can be parallelized. Also, independent sections of code can be defined as parallel with **`sections`** clause

```
#pragma omp parallel sections
{
    #pragma omp section
    phase1();
    #pragma omp section
    phase2();
    #pragma omp section
    phase3();
}
```



“single” clause

- Defines a section inside a parallel code that must be executed by only one thread
- It is not defined which thread will execute the section
- At the end there is an implicit barrier

```
#pragma omp parallel
{
   DoManyThings();
#pragma omp single
   {
     ExchangeBoundaries();
   }  // threads wait here for single
   DoManyMoreThings();
}
```

“master” clause

- Indicates a section that must be executed specifically by the master thread
- There is not an implicit barrier at the end

```
#pragma omp parallel
{
   DoManyThings();
#pragma omp master
   { // if not master, then skip to next stmt
     ExchangeBoundaries();
   }
   DoManyMoreThings();
}
```

barriers

- Programmer can define explicit barriers, so threads must wait until all threads finish their execution

```
#pragma omp parallel shared (A, B, C)
{
     DoSomeWork(A,B);
     printf(“Processed A into B\n”);
#pragma omp barrier
     DoSomeWork(B,C);
     printf(“Processed B into C\n”);
}
```

barriers

- Some OpenMP clauses uses implicit barriers

```
parallel
for
single
```

- However, this barriers could lead to performance problems
- If it is safe enough, you can use the `nowait` clause

barriers

- What do this example do?

```
#pragma omp parallel
{
#pragma omp for schedule(dynamic,1) nowait
 for(int i=0; i<n; i++)
   a[i] = bigFunc1(i);

#pragma omp for schedule(dynamic,1)
 for(int j=0; j<m; j++)
   b[j] = bigFunc2(j);
}
```

API

- It is often useful to know _who I am_ and _how many we are_ in terms of threads
- MPI users use this information to decide what parts of code must be executed by each thread. In OpenMP, API have instructions to give this information
- In this cases, there must be a header include

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

API

- To obtain the **`id`** of the thread inside a parallel section (equivalent to MPI_comm_rank)

```
int omp_get_thread_num(void);
```

- To obtain the total number of threads in an execution (equivalent to MPI_comm_size)

```
int omp_get_num_threads(void);
```

Challenge

- Write a program that uses OpenMP to find:
  - The average value of a series of real numbers
  - The maximum and minimum of a series of real numbers
  - A solution for matrix product in C