UNIVERSITÀ di VERONA

Dipartimento di INFORMATICA

PARCO Parallel Computing Lab

OpenMP

Part II

Laurea magistrale in Ingegneria e Scienze Informatiche
Laurea Magistrale in Medical Bioinformatics

Nicola Bombieri – Federico Busato

Agenda

- Threads Scheduling
  - STATIC, DYNAMIC, GUIDED, AUTO
- Other three work-sharing construct:
  - SINGLE/MASTER, SECTIONS, TASKS
- Synchronization
  - Barriers
  - Critical Region
  - Atomic updates
  - Locks
  - Reduction
- Practical concerns
  - Usage suggestions
  - Loop-carried dependencies
- Advanced conpect
  - Vectorization
- Conclusions
- Examples and exercises

Threads Scheduling (I)

STATIC SCHEDULE

- The scheduling algorithm for the for loop can be explicitly controlled through the schedule clause

```
#pragma omp for schedule(static)
```

- static is the default schedule
- static schedule makes each thread execute approximately N/P iterations for a loop of length N and P threads

```
#pragma omp for schedule(static, <chunk_size>)
```

- If chunk_size is provided, static schedule divides iterations into blocks of size chunk_size, which are assigned to threads in the team in a round-robin fashion

Threads Scheduling (II)

DYNAMIC SCHEDULE

```
#pragma omp for schedule(dynamic)
```

- Use the internal work queue to give a block of loop iterations to each thread. When a thread is finished, it retrieves the next block of loop iterations from the top of the work queue.
- By default, the chunk size is **1** but can also be specified to lessen the number of calls to the runtime library

```
#pragma omp for schedule(dynamic, <min_chunk_size>)
```

- Be careful when using this scheduling type because of the extra overhead involved.
- This is most useful when used in conjunction with the ordered clause, or when the different iterations in the loop may take different time to execute.

Threads Scheduling (III)

OTHER SCHEDULES

```
#pragma omp for schedule(guided [, <max_chunk_size>])
```

- Similar to dynamic scheduling, but the chunk size starts off large and decreases to better handle load imbalance between iterations. The optional chunk parameter specifies them minimum size chunk to use.
- By default the chunk size is approximately loop_count/number_of_threads

```
#pragma omp for schedule(auto)
```

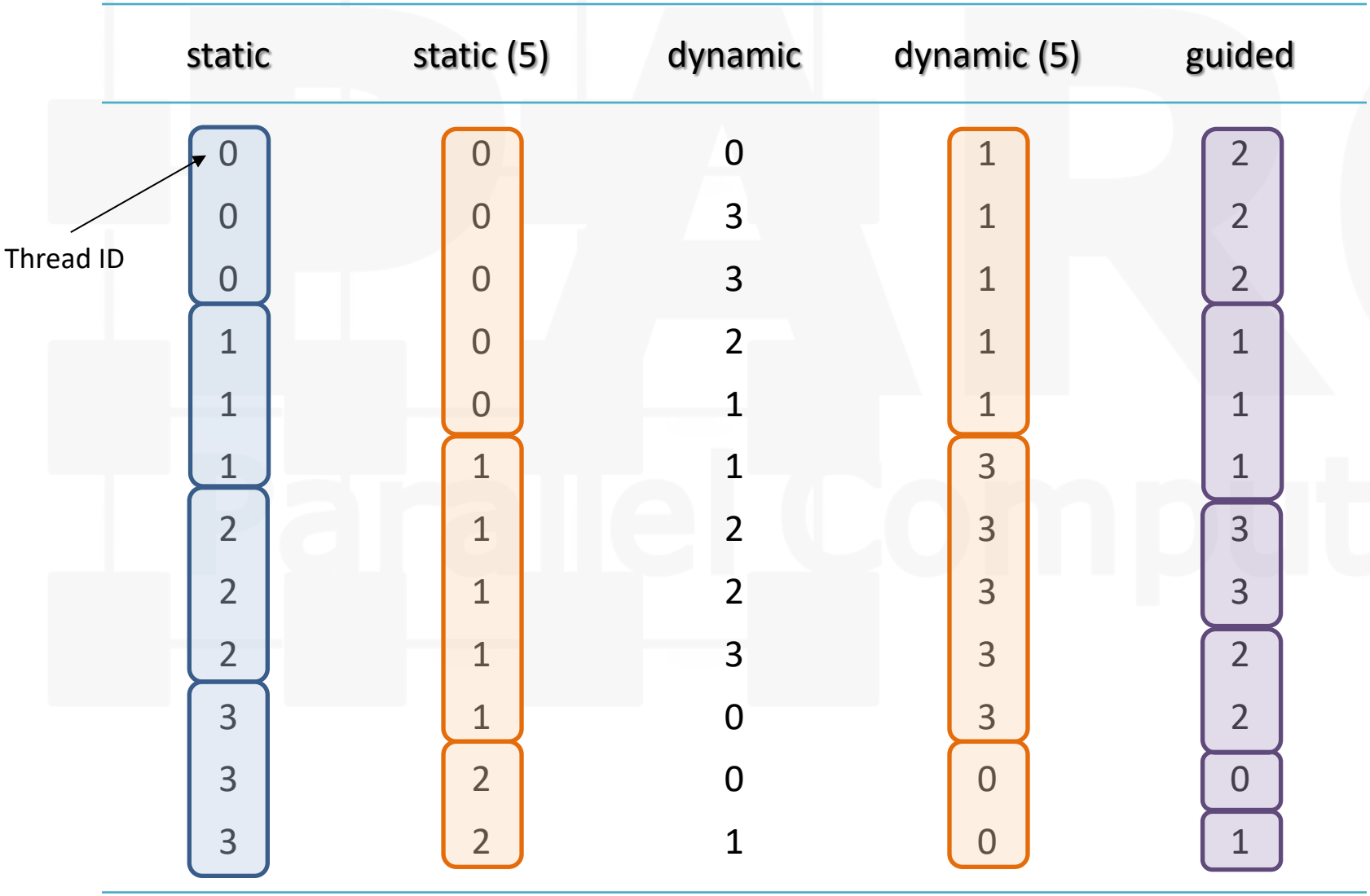
- The decision regarding scheduling is delegated to the compiler. The programmer gives the compiler the freedom to choose any possible mapping of iterations to threads in the team

```
#pragma omp for schedule(runtime)
```

- The decision regarding scheduling is deferred until run time. The schedule is implementation defined.

Threads Scheduling (IV)

- Four threads and a for loop with 12 iterations:



The SINGLE and MASTER directives (I)

```
#pragma omp single
```

- The SINGLE directive forces a statement (or a code block) to be executed by only one thread
- It is unknown which thread actually gets to execute
- All the other threads skip the statement/block and wait at an implicit barrier at the end of the construct
- There is no barrier at the beginning of the construct

```
#pragma omp master
```

- The MASTER directive is similar, except that the statement/block is to be executed by the master thread of the team
- There is no implicit barrier either on entry to or exit from the master construct

The SINGLE and MASTER directives (II)

```
#pragma omp parallel
{
  Work1();
  #pragma omp single
  { Work2(); }
  Work3();
}
```

- In a 2-cpu system, this will run Work1 twice, Work2 once and Work3 twice. There is an implied barrier at the end of the single directive, but not at the beginning of it

```
#pragma omp parallel
{
  Work1();
  #pragma omp master
  { Work2(); }
  Work3();
  #pragma omp master
  { Work4(); }
}
```

- If you have multiple MASTER blocks in a PARALLEL SECTION, you are guaranteed that they are executed by the same thread every time, and hence, the values of PRIVATE (thread-local) variables are the same

The SECTIONS Pragma (I)

- The sections directive specifies a set of code blocks which are to be distributed among the threads in a team
- It allows to indicate code portions which can run in parallel

```
#pragma omp sections
{
  { Work1() }
  #pragma omp section
  { Work2();
  Work3() }
   #pragma omp section
  { Work4() }
}
```

- This code indicates that any of the tasks Work1, Work2 + Work3 and Work4 may run in parallel, but that Work2 and Work3 must be run in sequence

The SECTIONS Pragma (II)

- Just like the FOR directive, the SECTIONS directive is a work-sharing construct, so it does not launch any new thread → it must be enclosed in a parallel region
- There is a shorthand to combine both these directives:

```
#pragma omp parallel sections
```

- The SECTION (not SECTIONS) directives may also followed by clauses to specify data sharing:
  - `private(list)`
  - `firstprivate(list)`
  - `lastprivate(list)`
  - …

The TASK Pragma (I)

OpenMP 3.0 (2008): Task Parallelism

Abstraction Task: A basic unit of parallel work

User:

Specify the tasks

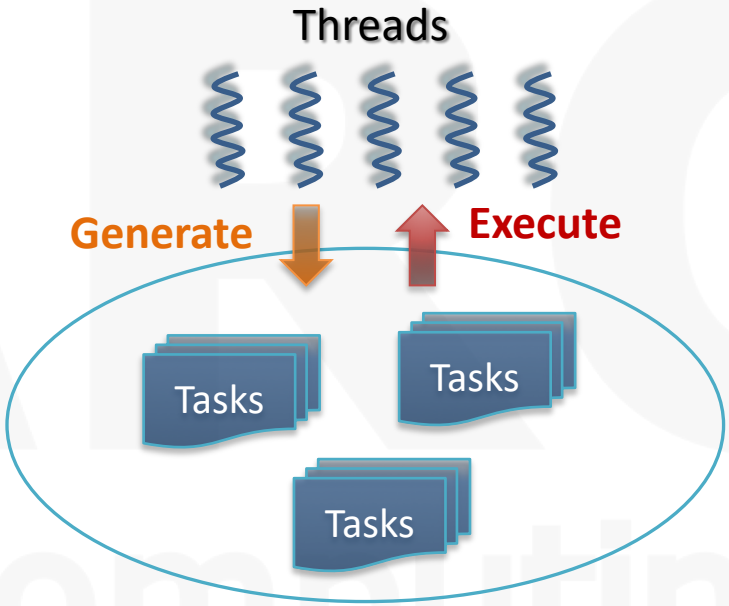
The assumption is that all tasks can be executed independently

OpenMP runtime system:

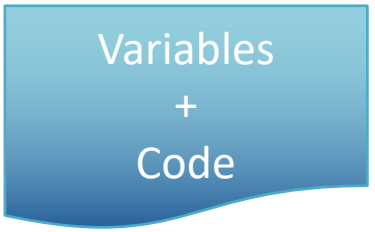
Task scheduling

Task synchronization

The only way to control the execution flow



Task:



Good for simple recursive algorithms

OpenMP 4.0: Tasking Extensions

- Support task dependences
- Hierarchy of tasks (child tasks, descendant tasks)

The TASK Pragma (II)

```
#pragma omp parallel
{
  #pragma omp single
  {
  #pragma omp task
  { work1() }
   #pragma omp task
  { work2() }
  }
}
```

- The SINGLE construct is needed so that tasks will be created by one thread only. If there was no single construct, each task would get created *num_threads* times, which might not be what one wants.

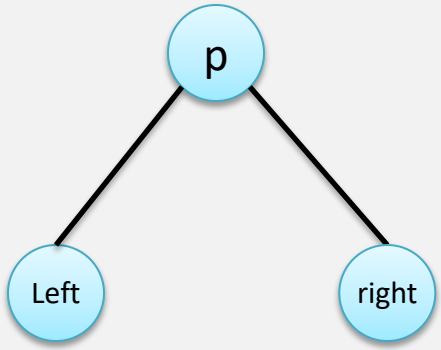
```
#pragma omp taskwait
```

- TASKWAIT works very like a barrier for tasks - it ensures that current execution flow will get paused until all queued tasks have been executed.

The TASK Pragma (III)

- In the next example, we force a postorder traversal of the tree by adding a taskwait directive.

```
void postorder_traverse(node* p)
{
  if (p->left)
      #pragma omp task
      postorder_traverse(p->left);
  if (p->right)
  #pragma omp task
      postorder_traverse(p->right);
  #pragma omp taskwait
  process(p);
}
```

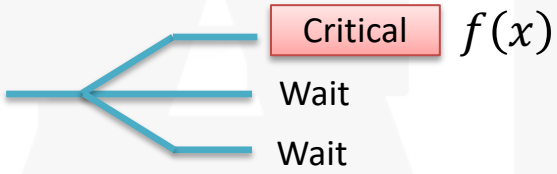


- Now, we can safely assume that the left and right sons have been executed before we process the current node

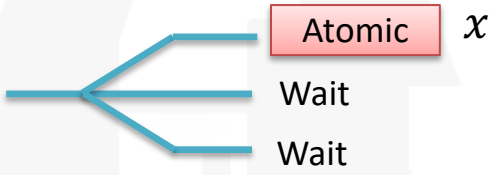
OpenMP Threads Synchronization

- Barrier

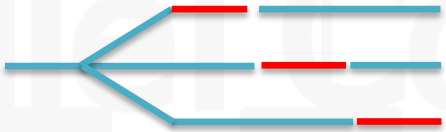
- Critical Region



- Atomic



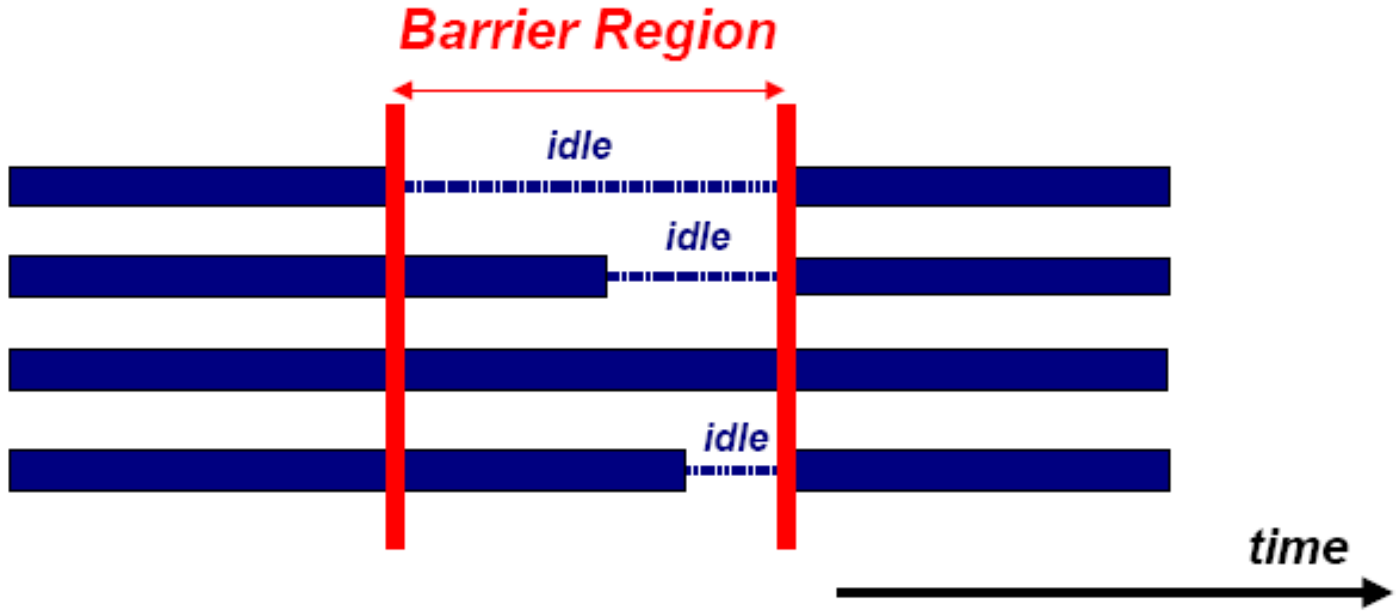
- Ordered



- Low Level Synchronization: lock, flush
- Special Case: Reduction

When to use Barriers?

- Barriers are to be used if data is updated asynchronously and data integrity is at risk
  - e.g. between code portions which read and write the same section of memory or after a timestep in a solver



- Unfortunately barriers tend to be expensive and may not scale to a large number of processors

The BARRIER Pragma

```
#pragma omp barrier
```

- The BARRIER directive specifies an explicit barrier at the point in which the construct appears
- Each thread waits at the barrier point and is allowed to continue only when all threads have reached the barrier point

```
#pragma omp parallel
{
  Work1();
  #pragma omp barrier
  Work2();
}
```

- All threads execute Work2, but not before all threads have finished executing Work1.

The NOWAIT Pragma

- There is an implicit barrier at the end of each parallel region, and at the end of each sections, for and single construct unless the NOWAIT clause is used

```
#pragma omp (sections | for | single) nowait
```

- The NOWAIT clause can only be applied to SECTIONS, FOR and SINGLE directives
- It allows to minimize synchronization between threads

The CRITICAL Pragma

```
#pragma omp critical [(<name>)]
```

- The CRITICAL directive restricts the execution of a statement or a code block to a single thread at a time
- The critical directive may contain an optional NAME that identifies its type; this name is global to the entire program
  - All critical directives without name are assumed to have the same unspecified name
  - The critical directive enforces exclusive access to the associated region with respect to all critical regions with the same name
  - A thread must wait at the beginning of a critical region until no thread is executing a critical region identified by the same name
- There is no implicit barrier either on entry to or exit from the critical region

The ATOMIC Pragma

```
#pragma omp atomic
<simple assignment>     //counter += value;
```

- The ATOMIC directive is used to ensure that a specific variable is updated atomically
  - in order to not expose it to the possibility of multiple simultaneous assignments by other threads in the team
- The ATOMIC directive can only be used in assignments involving a simple expression, such as increments, decrements and simple arithmetic operations; it does not support function calls, array indexing, overloaded operators, multiple statements
- Only the load and store of the assigned variable are atomic; the evaluation of the right-hand expression is not guaranteed to be atomic
- If you need to atomicise more complex constructs, use either the CRITICAL directive or the locks.

LOCK Routines (I)

- Locks provide greater flexibility over critical regions and atomic updates; they can be used as a way to express a local mutex
- A lock is a variable having omp_lock_t as type, and must be accessed only through the following routines:

omp_init_lock()	initializes the lock
omp_set_lock()	attempts to set the lock; if the lock is already set by another thread, it will wait until the lock is no longer set, and then sets it
omp_unset_lock()	unsets the lock (it should be called by the same thread that set the lock)
omp_test_lock()	attempts to set the lock; if the lock is already set by another thread, it returns 0 (but does not suspend execution), otherwise it sets it and returns 1
omp_destroy_lock()	destroys the lock

Lock Routines (II)

- Simple example:

```
omp_lock_t lock;
omp_init_lock(&lock);
#pragma omp parallel
  {
    …
    if (omp_test_lock(&lock) {
        omp_set_lock(&lock);
        Work1();
        omp_unset_lock(&lock);
    }
    …
  }
  omp_destroy_lock(&lock);
```

The FLUSH Pragma

```
#pragma omp flush(<var1>, <var2>, … )
```

- Even when variables used by threads are supposed to be SHARED, the compiler may take liberties and optimize them as register variables. This can skew concurrent observations of the variable.
- The FLUSH directive can be used to ensure that the value observed in one thread is also the value observed by other threads.

```
            a = 0, b = 0
// first threads              // second thread
b = 1                         a = 1
#pragma omp flush(a,b)           #pragma omp flush(a,b)
if (a == 0)                   if (b == 0)
  // Critical Section              // Critical Section
```

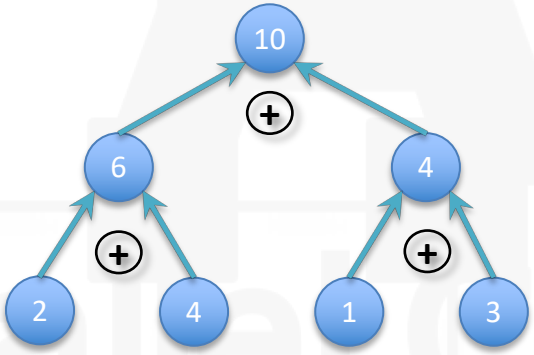
- In this example, it is enforced that at the time either of a or b is accessed, practically ensuring that not both of the two threads enter the critical section.

Note: It is still possible that neither of them can enter it

The REDUCTION clause (I)

```
#pragma omp parallel reduction(<op> : <var1>, <var2>, … )
```

- The **reduction** clause allows to accumulate a shared variable without the atomic clause



- Reduction require Binary, Associative Operator:
  - +, *, -, &, |, ^, &&, ||, max, min

Require OpenMP v3.1 (gcc v4.7)

- It will often produce faster executing code than by using the ATOMIC clause

The REDUCTION clause (II)

```
int sum = 0;
#pragma omp parallel for reduction(+:sum)
{
  for(int n=2; n<=number; ++n)
  sum += n;        // remember: A[i] += n not possible!
}
```

- At the beginning of the parallel region, a private copy of each variable in the list is created for each thread and properly initialized (according to the op operator)
- At the end of the parallel region, each variable in the list is updated by combining its original value with the final value of the corresponding private copies using the op operator
- The REDUCTION clause has the same restriction as the ATOMIC directive concerning the expression involved

OpenMP usage suggestions

- Use static scheduling in for-loops (dynamic scheduling involves too much overhead), unless iterations are “heavy” and/or unbalanced
- Keep number of lock to a minimum (if you have to use them, make sure you unset them)
- Keep number of barriers to a minimum (use the NOWAIT clause whenever an implicit barrier is not required)
- Minimize critical regions code size, in order to reduce potential wait times
- Since overheads are associated with starting and terminating a parallel region, maximize parallel regions code size and minimize their number
- Carefully consider which variables have to be shared and which ones could be made private

Loop-carried dependencies

- When parallelizing loops, try to make loop iterations independent
  - So that they can be executed safely in any order without loop-carried dependencies

```
int i, j;
j = 5;
for (i = 0; i < MAX; ++i) {
    j += 2;
    ...
}
```

Loop-carried dependence

```
int i;
for (i = 0; i < MAX; ++i) {
    int j = 5 + i * 2;
    ...
}
```

The SIMD Pragma

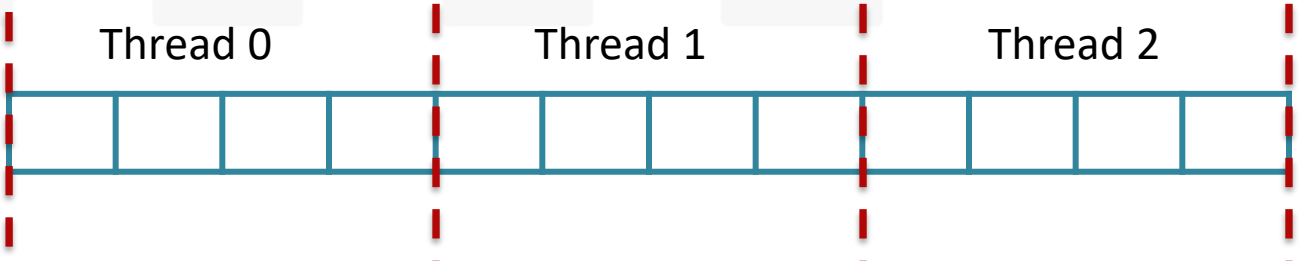
OpenMP 4.0 (July 2013) - Vectorization

Single Precision Floating Point width vs. Speedup


```
#pragma omp simd
void f (int* A, int* B, int* C,...) {
    for ( int i = 0; i < N; i++ )
        C[i] = A[i] + B[i]
```

Compile

```
void f (vec4* A, vec4* B, ...) {
    for ( int i = ThID; i < N; i += |Th| )
        C[i] = A[i] + B[i]
```



Conclusions

- OpenMP is a very good choice for writing shared memory applications
  - Ease of programming (just adding a few lines to the original source code is required)
  - Hides a lot of implementation details to the user
- But it is effective for small-scale parallelism
  - Up to 16 processors get decent performance
- Not much useful for large-scale parallelism
  - Difficult to get good parallel efficiency for any significant number of processors

Examples

- To compile a program containing OpenMP directives you have to use gcc/g++ and add the –fopenmp flag:

```
g++ –fopenmp source.cpp -o out.x
```

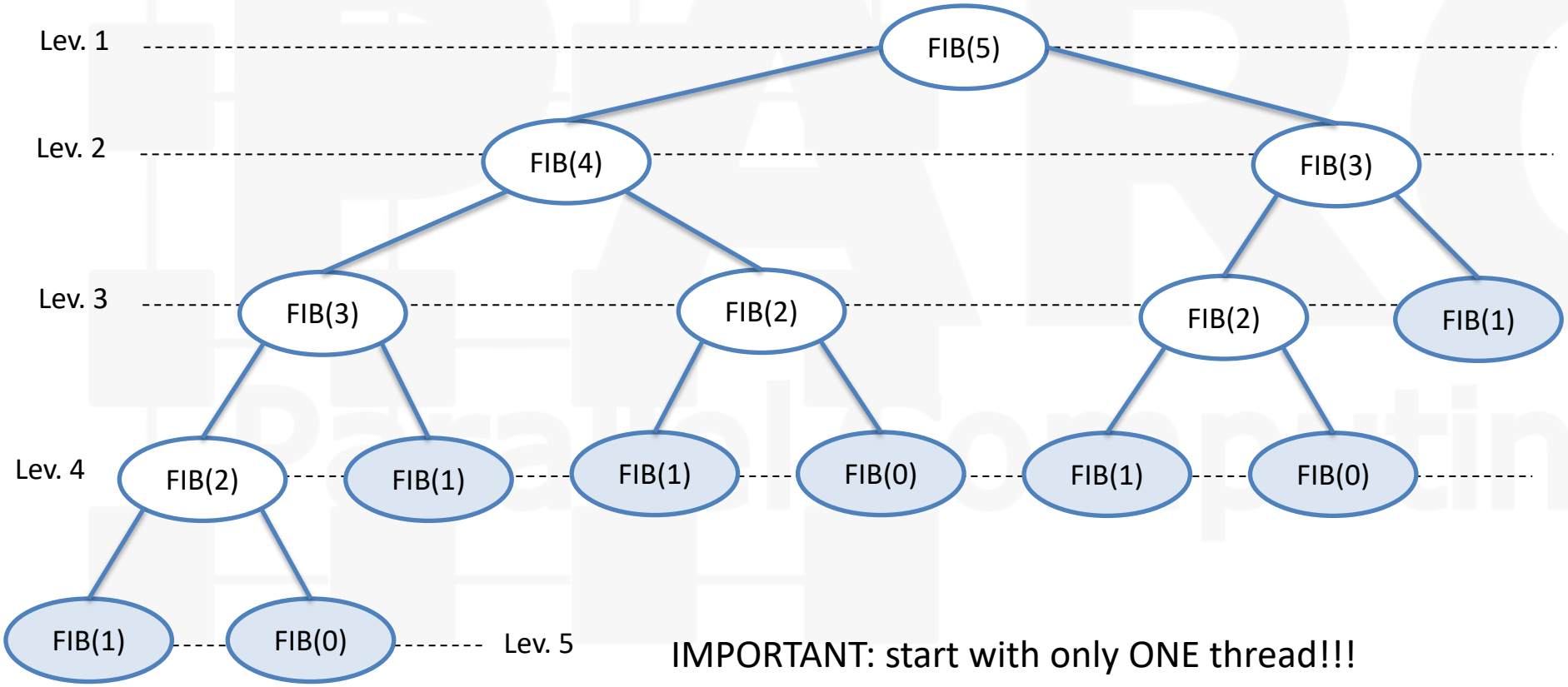
- **Examples:**
  - Test Scheduling: STATIC, DYNAMIC, GUIDED, AUTO
  - Test SINGLE/MASTER, SECTIONS
  - Solve the FIND problem
  - ATOMIC/REDUCTION/ CRITICAL/LOCKS in the sum problem

  WHAT IS THE FASTEST IMPLEMENTATION?

- **Exercises:**
  - Task -> Fibonacci
  - Task -> Quicksort
  - Synchronization -> Producer/Consumer

Exercises (I)

TASKS - FIBONACCI

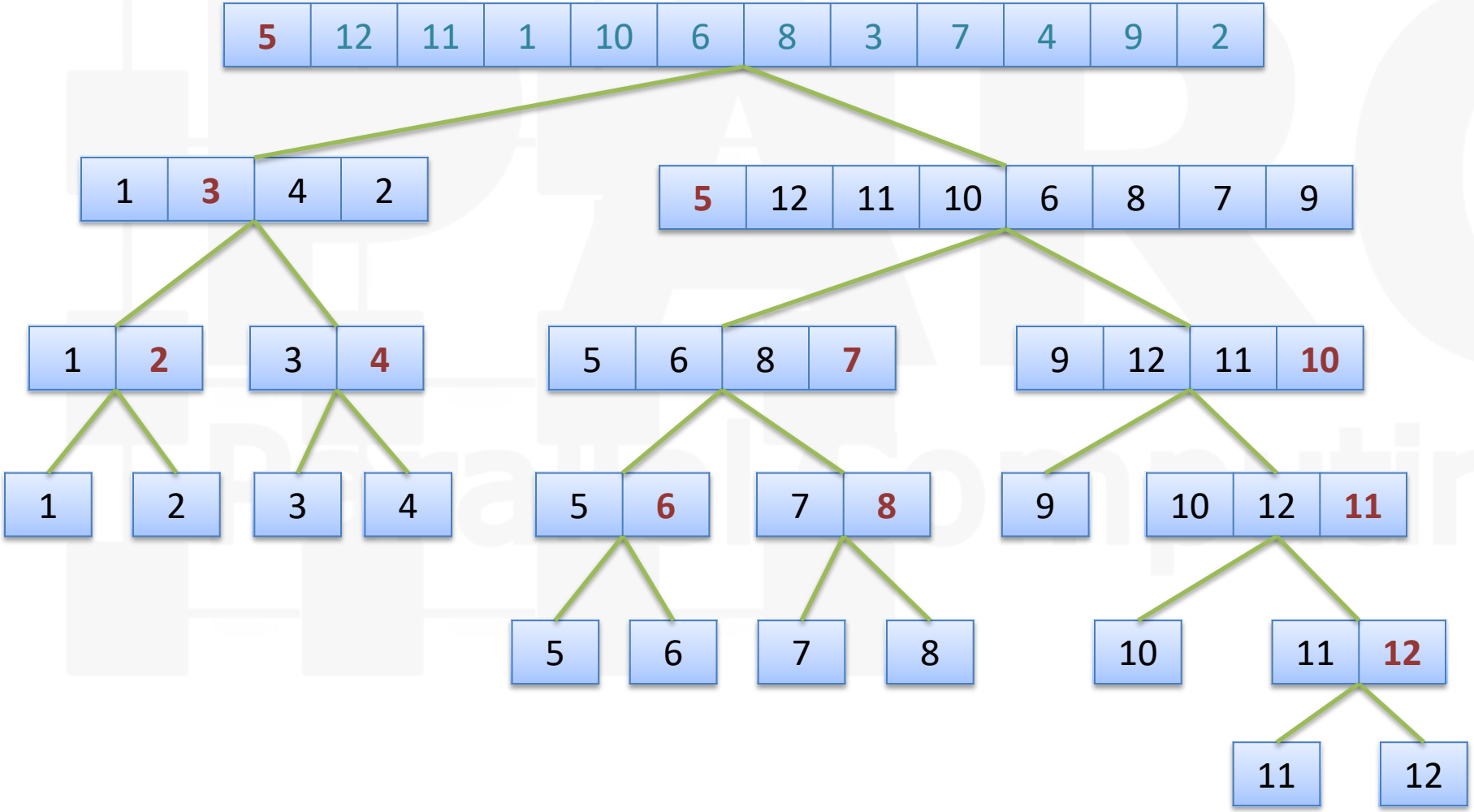


IMPORTANT: start with only ONE thread!!!

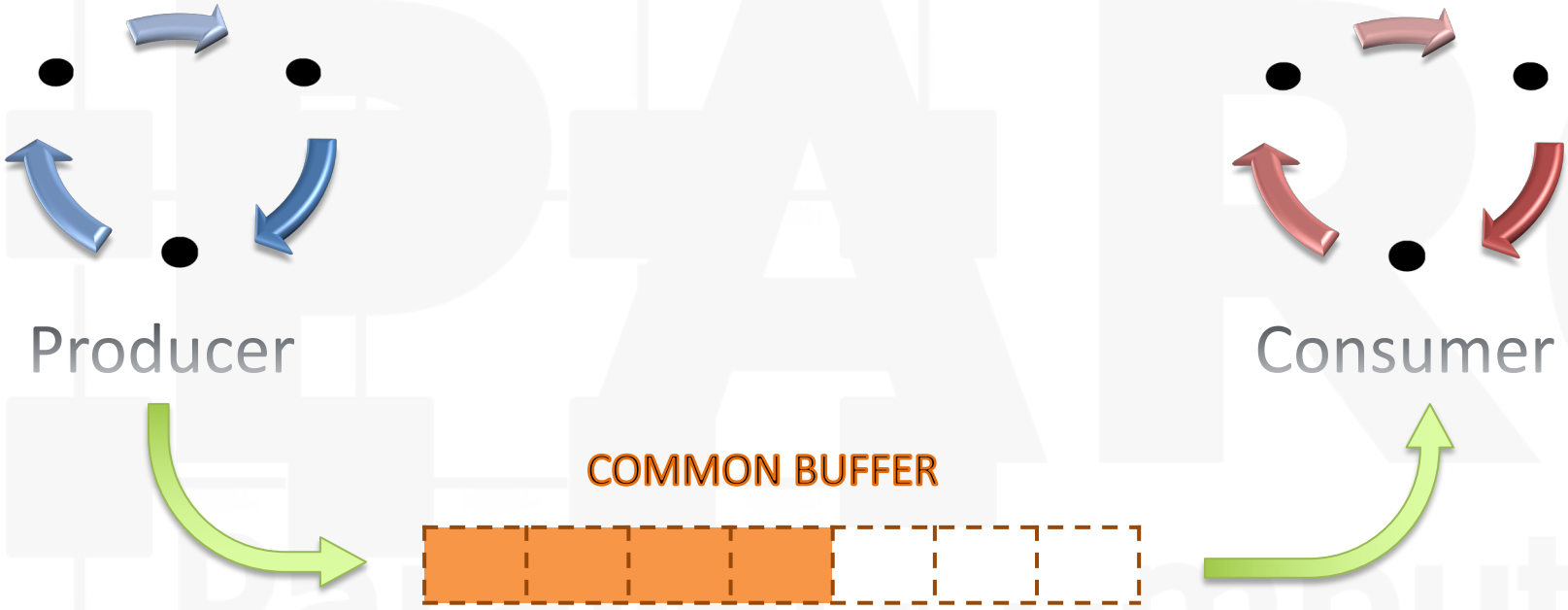
Is the parallel version worth it? When?

Exercises (II)

TASKS - QUICKSORT



Exercises (III)
Producer - Consumer



- Can be implemented with:
  - Critical region
  - Locks (mutex)
  - Atomic?
  - Barrier?
  - Only shared variable?

References

- Official OpenMP Specification (v4.0.0)
  www.openmp.org/mp-documents/OpenMP4.0.0.pdf
- OpenMP in Visual C++
  https://msdn.microsoft.com/en-us/library/tt15eb9t.aspx
- Guide into OpenMP: Easy multithreading programming for C++
  http://bisqwit.iki.fi/story/howto/openmp/
- A “Hands-on” Introduction to OpenMP
  http://openmp.org/mp-documents/omp-hands-on-SC08.pdf
- 32 OpenMP traps for C++ developers
  https://software.intel.com/en-us/articles/32-openmp-traps-for-c-developers