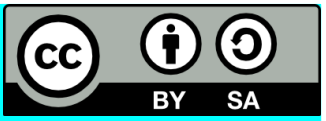
Shared-memory Programming with OpenMP

Pacheco chapter 5

Moreno Marzolla
Dip. di Informatica—Scienza e Ingegneria (DISI)
Università di Bologna

moreno.marzolla@unibo.it



Credits

- Peter Pacheco, Dept. of Computer Science, University of San Francisco http://www.cs.usfca.edu/~peter/
- Mary Hall, School of Computing, University of Utah https://www.cs.utah.edu/~mhall/
- Salvatore Orlando, DAIS, Università Ca' Foscari di Venezia, http://www.dais.unive.it/~calpar/
- Tim Mattson, Intel
- Blaise Barney, OpenMP https://computing.llnl.gov/tutorials/openMP/ (highly recommended!!)

OpenMP

- Model for shared-memory parallel programming
- Portable across shared-memory architectures
- Incremental parallelization
  - Parallelize individual computations in a program while leaving the rest of the program sequential
- Compiler based
  - Compiler generates thread programs and synchronization
- Extensions to existing programming languages (Fortran, C and C++)
  - mainly by directives (#pragma omp ...)
  - a few library routines

Most OpenMP programs use these constructs only

OpenMP pragma, function, or clause	Concepts
`#pragma omp parallel`	Parallel region, teams of threads, structured block, interleaved execution across threads
`int omp_get_thread_num()` `int omp_get_num_threads()`	Create threads with a parallel region and split up the work using the number of threads and thread ID
`double omp_get_wtime()`	Timing blocks of code
`setenv OMP_NUM_THREADS N` `export OMP_NUM_THREADS=N`	Set the default number of threads with an environment variable

`#pragma omp barrier`
`#pragma omp critical`
`#pragma omp atomic`
	Synchronization, critical sections

`#pragma omp for` `#pragma omp parallel for`	Worksharing, parallel loops
`reduction(op:list)`	Reductions of values across a team of threads
`schedule(dynamic [,chunk])` `schedule(static [,chunk])`	Loop schedules
`private(list), shared(list),` `firstprivate(list)`	Data environment
`#pragma omp master` `#pragma omp single`	Worksharing with a single thread
`#pragma omp task` `#pragma omp taskwait`	Tasks including the data environment for tasks.

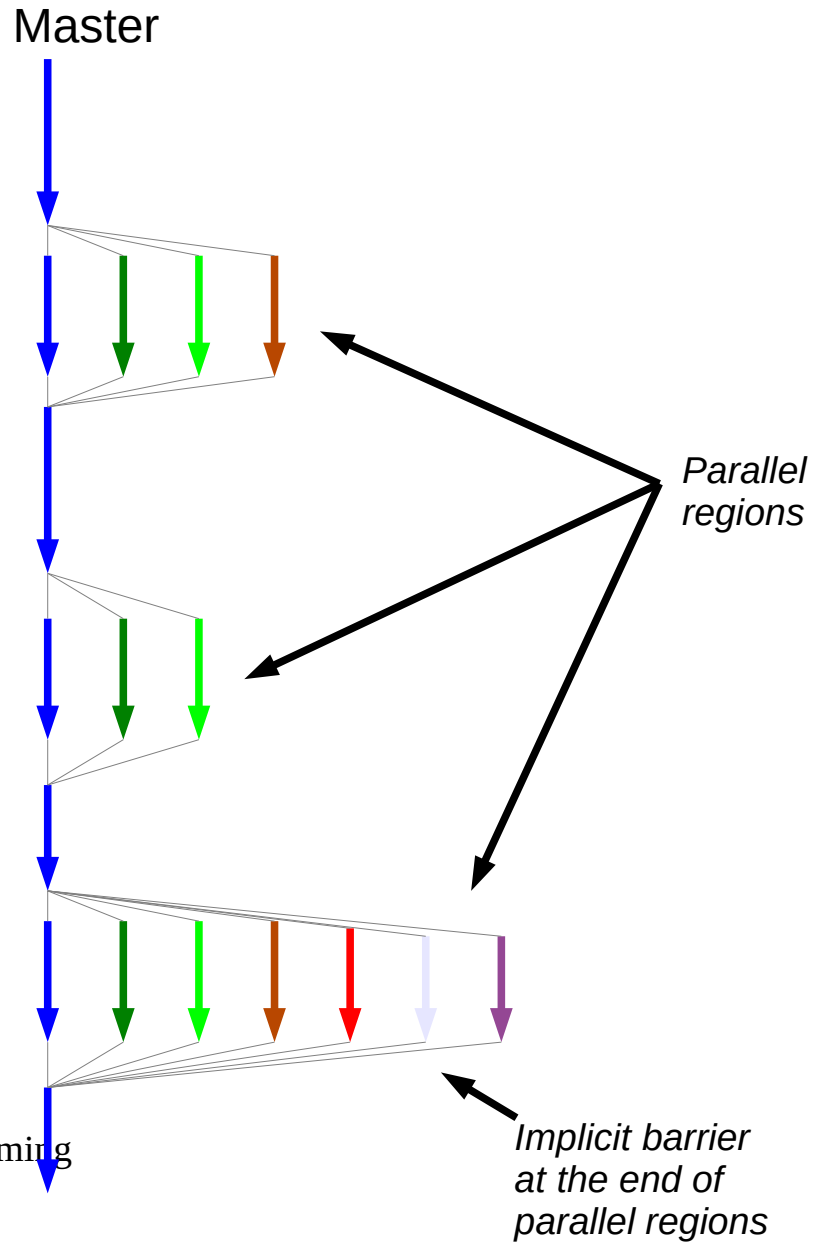
Credits: Tim Mattson

A Programmer's View of OpenMP

- OpenMP is a portable, threaded, shared-memory programming specification with "light" syntax
  - Requires compiler support (C/C++ or Fortran)
- OpenMP will:
  - Allow a programmer to separate a program into serial regions and parallel regions
  - Provide synchronization constructs
- OpenMP will **not**:
  - Parallelize automatically
  - Guarantee speedup
  - Avoid data races

OpenMP Execution Model

- Fork-join model of parallel execution
- Begin execution as a single process (master thread)
- Start of a parallel construct:
  - Master thread creates team of threads (worker threads)
- Completion of a parallel construct:
  - Threads in the team synchronize – implicit barrier
- Only the master thread continues execution



OpenMP uses Pragmas

```
#pragma omp construct [clause [clause ...]]
```

- *Pragmas* are special preprocessor directives
  - They allow behaviors that are not part of the C specification
- Compilers that don't support the *pragmas* ignore them
- Most OpenMP constructs apply to the structured block following the directive
  - Structured block: a block of one or more statements with one point of entry at the top and one point of exit at the bottom
  - Returning from inside a parallel block is not allowed

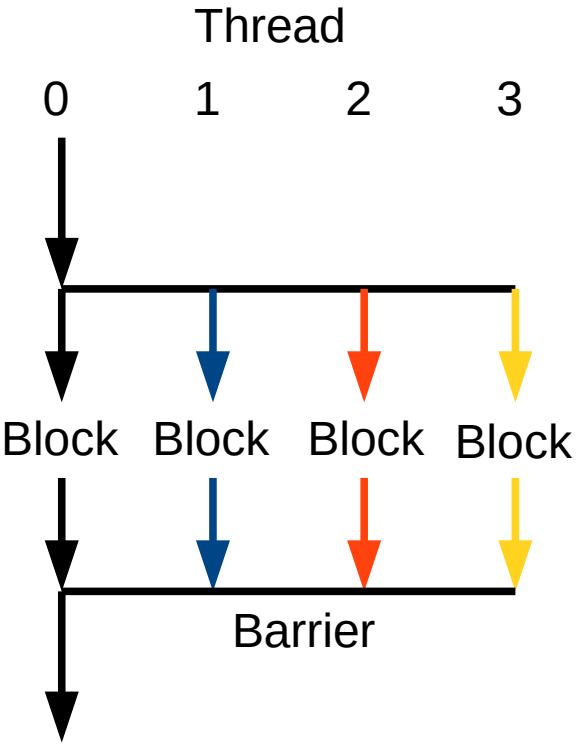
The `#pragma omp parallel` directive

- When a thread reaches a **`parallel`** directive, it creates a pool of threads and becomes the master of the team
  - The master has thread ID 0
- The default pool size is implementation-dependent
- The code of the parallel region is duplicated and all threads will execute it
- There is an implied barrier at the end of a parallel section. Only the master continues execution past this point

```
#pragma omp parallel [clause ...]
clause ::=
    if (scalar_expression) |
    private (list) |
    shared (list) |
    default (shared | none) |
    firstprivate (list) |
    reduction (operator: list) |
    copyin (list) |
    num_threads(thr)
```

“Hello, world” in OpenMP

```c
/* omp-demo0.c */
#include <stdio.h>

int main( void )
{
    #pragma omp parallel
    {
       printf("Hello, world!\n");
    }

    return 0;
}
```




```
$ gcc -fopenmp omp-demo0.c -o omp-demo0
$ ./omp-demo0
Hello, world!
Hello, world!
$ OMP_NUM_THREADS=4 ./omp-demo0
Hello, world!
Hello, world!
Hello, world!
Hello, world!
```

“Hello, world” in OpenMP

```c
/* omp-demo1.c */
#include <stdio.h>
#include <omp.h>

void say_hello( void )
{
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();
    printf("Hello from thread %d of %d\n",
           my_rank, thread_count);
}

int main( void )
{
    #pragma omp parallel
    say_hello();

    return 0;
}
```

```
$ gcc -fopenmp omp-demo1.c -o omp-demo1
$ ./omp-demo1
Hello from thread 0 of 2
Hello from thread 1 of 2
$ OMP_NUM_THREADS=4 ./omp-demo1
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
Hello from thread 3 of 4
```

```c
/* omp-demo2.c */
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void say_hello( void )
{
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();
    printf("Hello from thread %d of %d\n",
           my_rank, thread_count);
}

int main( int argc, char* argv[] )
{
    int thr = atoi( argv[1] );
    #pragma omp parallel num_threads(thr)
    say_hello();

    return 0;
}
```

```
$ gcc -fopenmp omp-demo2.c -o omp-demo2
$ ./omp-demo2 2
Hello from thread 0 of 2
Hello from thread 1 of 2
$ ./omp-demo2 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
Hello from thread 3 of 4
```

Setting the number of threads programmatically

```
/* omp-demo3.c */
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void say_hello( void )
{
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();
    printf("Hello from thread %d of %d\n",
           my_rank, thread_count);
}

int main( int argc, char* argv[] )
{
    omp_set_num_threads(4);
    #pragma omp parallel
    say_hello();

    return 0;
}
```

```
$ gcc -fopenmp omp-demo3.c -o omp-demo3
$ OMP_NUM_THREADS=8 ./omp-demo3
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
Hello from thread 3 of 4
```

Warning

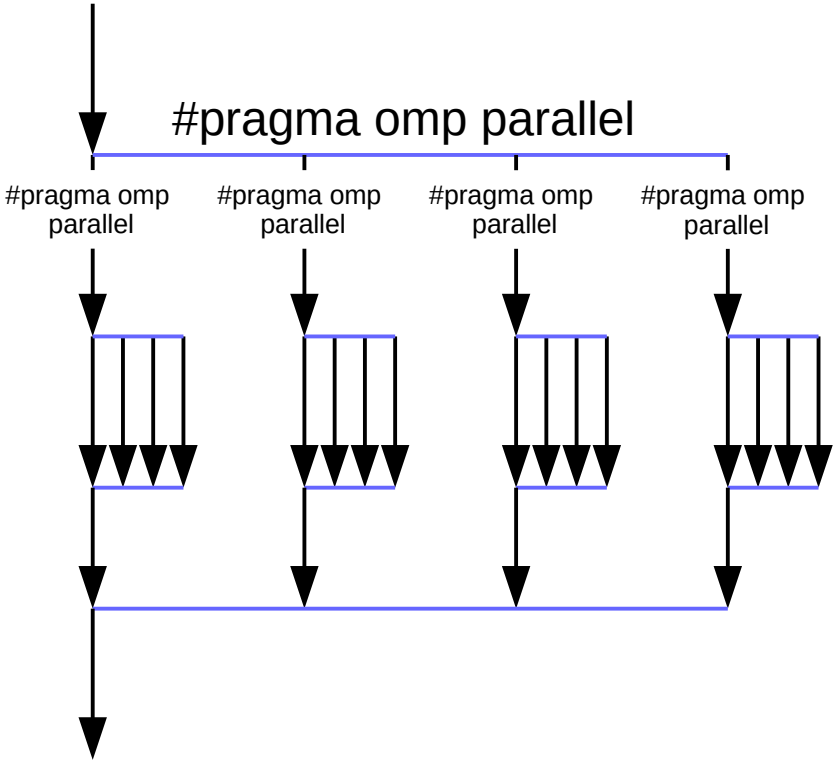
- **`omp_get_num_threads()`** returns the number of threads in the currently active pool
  - If no thread pool is active, the function returns 1
- **`omp_get_max_threads()`** returns the maximum number of threads that can be created

```
int main( int argc, char* argv[] )
{
    printf("Before par region: threads=%d, max=%d\n",
           omp_get_num_threads(), omp_get_max_threads());
#pragma omp parallel
    {
        printf("Inside par region: threads=%d, max=%d\n",
               omp_get_num_threads(), omp_get_max_threads());
    }
    return 0;
}
```

See omp-demo4.c

Nested parallelism

- It is possible to nest parallel regions
  - Nesting must be enabled by setting the environment variable **`OMP_NESTED=true`**
  - **`omp_get_num_threads()`** returns the number of the innermost thread pool a thread is part of
  - **`omp_get_thread_num()`** returns the ID of a thread in the innermost pool it is part of
- Nested parallelism could cause *oversubscription*
  - more running threads than processor cores
- See omp-nested.c

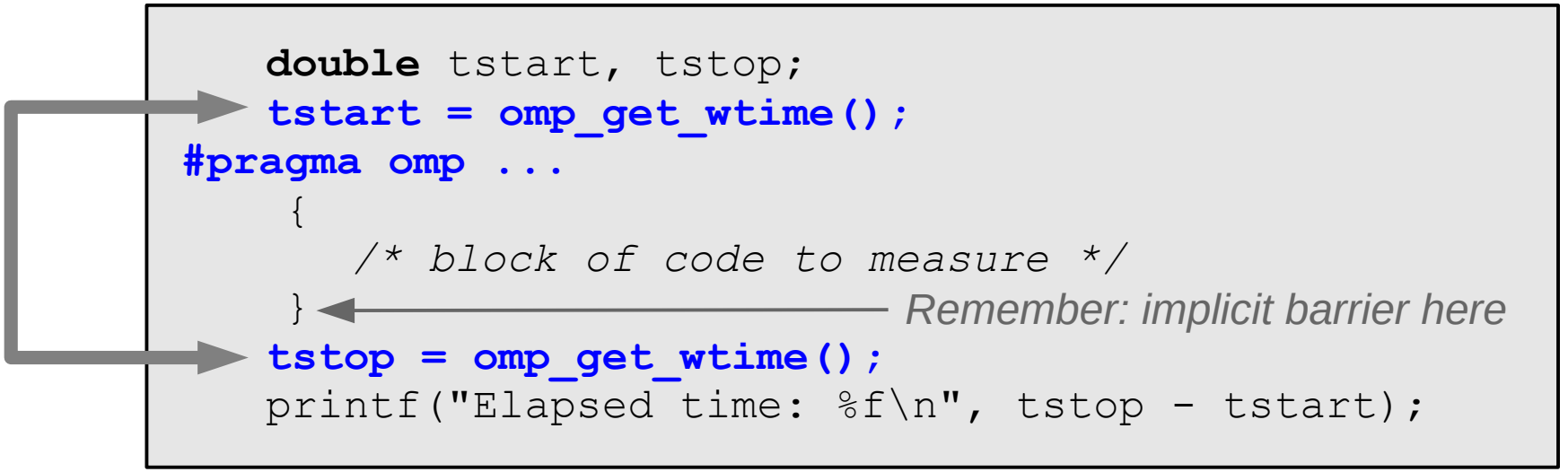


More complex example

```
int num_thr = 3
#pragma omp parallel if(num_thr>=4) num_threads(num_thr)
{
    /* parallel block */
}
```

- The “if” clause is evaluated
  - If the clause evaluates to *true*, the **`parallel`** construct is enabled with **`num_thr`** threads
  - If the clause evaluates to *false*, the **`parallel`** construct is ignored

Taking times



Scoping of variables

Scope

- In serial programming, the scope of a variable consists of the parts of a program where the variable can be used
- In OpenMP, the scope of a variable refers to the set of threads that can access the variable
- By default:
  - All variables that are visible at the beginning of a parallel block are shared across threads
  - All variables defined inside a parallel block are private to each thread

Variables scope in OpenMP

- **`shared(x)`** ◀ **Default**
  - all threads access the same memory location
- **`private(x)`**
  - each thread has its own private copy of x
  - all local instances of x **are not initialized**
  - local updates to x are **lost** when exiting the parallel region
  - the original value of x is retained at the end of the block (OpenMP ≥ 3.0 only)
- **`firstprivate(x)`**
  - each thread has its own private copy of x
  - all copies of x **are initialized with the current value of x**
  - local updates to x are **lost** when exiting the parallel region
  - the original value of x is retained at the end of the block (OpenMP ≥ 3.0 only)
- **`default(shared)`** or **`default(none)`**
  - affects all the variables not specified in other clauses
  - **`default(none)`** ensures that you **must** specify the scope of each variable used in the parallel block that the compiler can not figure out by itself (**highly recommended!!**)

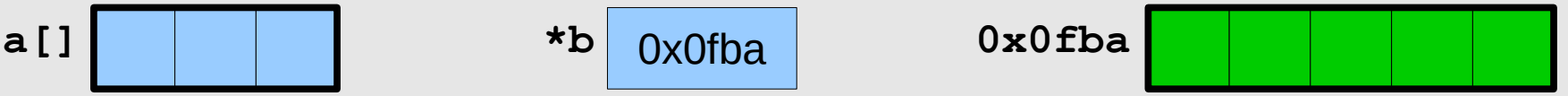
What is the output of this program?

```
/* omp-scope.c */
#include <stdio.h>

int main( void )
{
    int a=1, b=1, c=1, d=1;
#pragma omp parallel num_threads(10) \
    private(a) shared(b) firstprivate(c)
    {
        printf("Hello World!\n");
        a++;
        b++;
        c++;
        d++;
    }
    printf("a=%d\n", a);
    printf("b=%d\n", b);
    printf("c=%d\n", c);
    printf("d=%d\n", d);
    return 0;
}
```

Hint: compile with -Wall

Sharing arrays

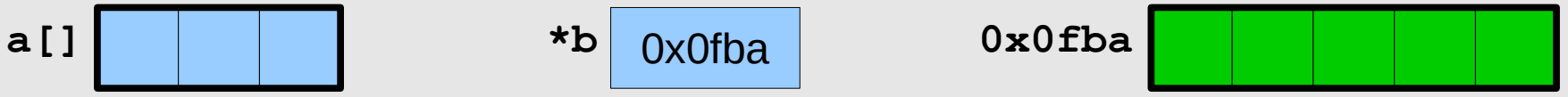
```
int a[3];
int *b = (int*)malloc(5 * sizeof(*b) );
```



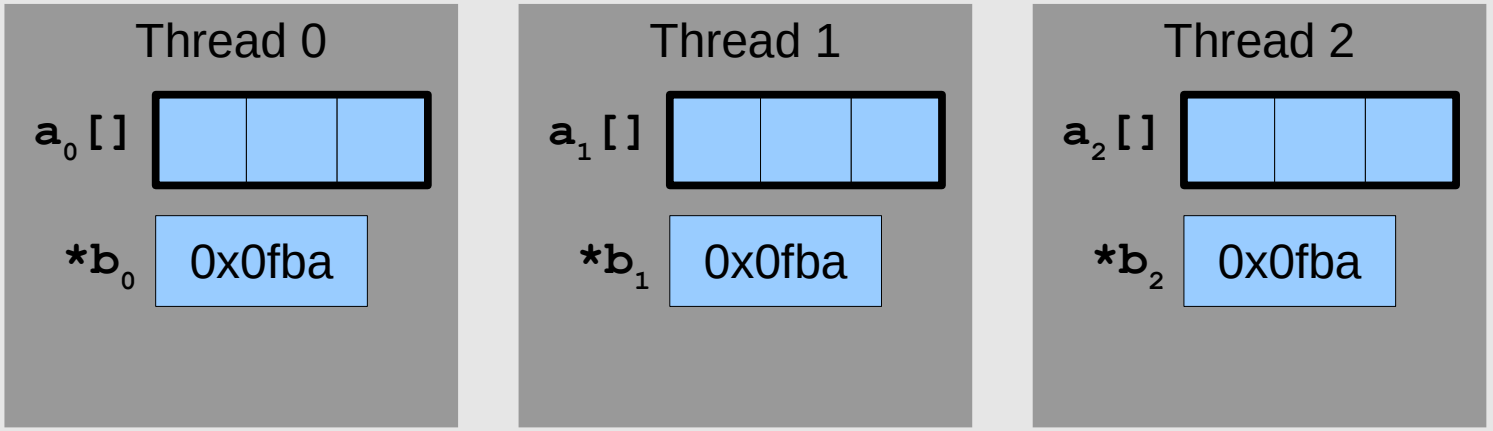

```
#pragma omp parallel num_threads(3) private(a, b)
{




}
```

Sharing arrays

```
int a[3];
int *b = (int*)malloc(5 * sizeof(*b) );
```




```
#pragma omp parallel num_threads(3) private(a, b)
{
}
```



GCC annoys us

- When using **`default(none)`**
  - In GCC < 9.x, **`const`** variables are predetermined shared
  - In GCC ≥ 9.x, you must explicitly specify the visibility of every variable
- See: https://www.gnu.org/software/gcc/gcc-9/porting_to.html

GCC annoys us

- GCC < 9.x: OK
- GCC ≥ 9.x:
  error: ‘foo’ not specified in enclosing ‘parallel’

```
const int foo = 1;
#pragma omp parallel default(none)
{
    int baz = 0;
    baz += foo;
}
```

See omp-bug1.c

GCC annoys us

- GCC < 9.x:
  'foo' is predetermined 'shared' for 'shared'
- GCC ≥ 9.x: OK

```
const int foo = 1;
#pragma omp parallel default(none) shared(foo)
{
    int baz = 0;
    baz += foo;
}
```

See omp-bug2.c

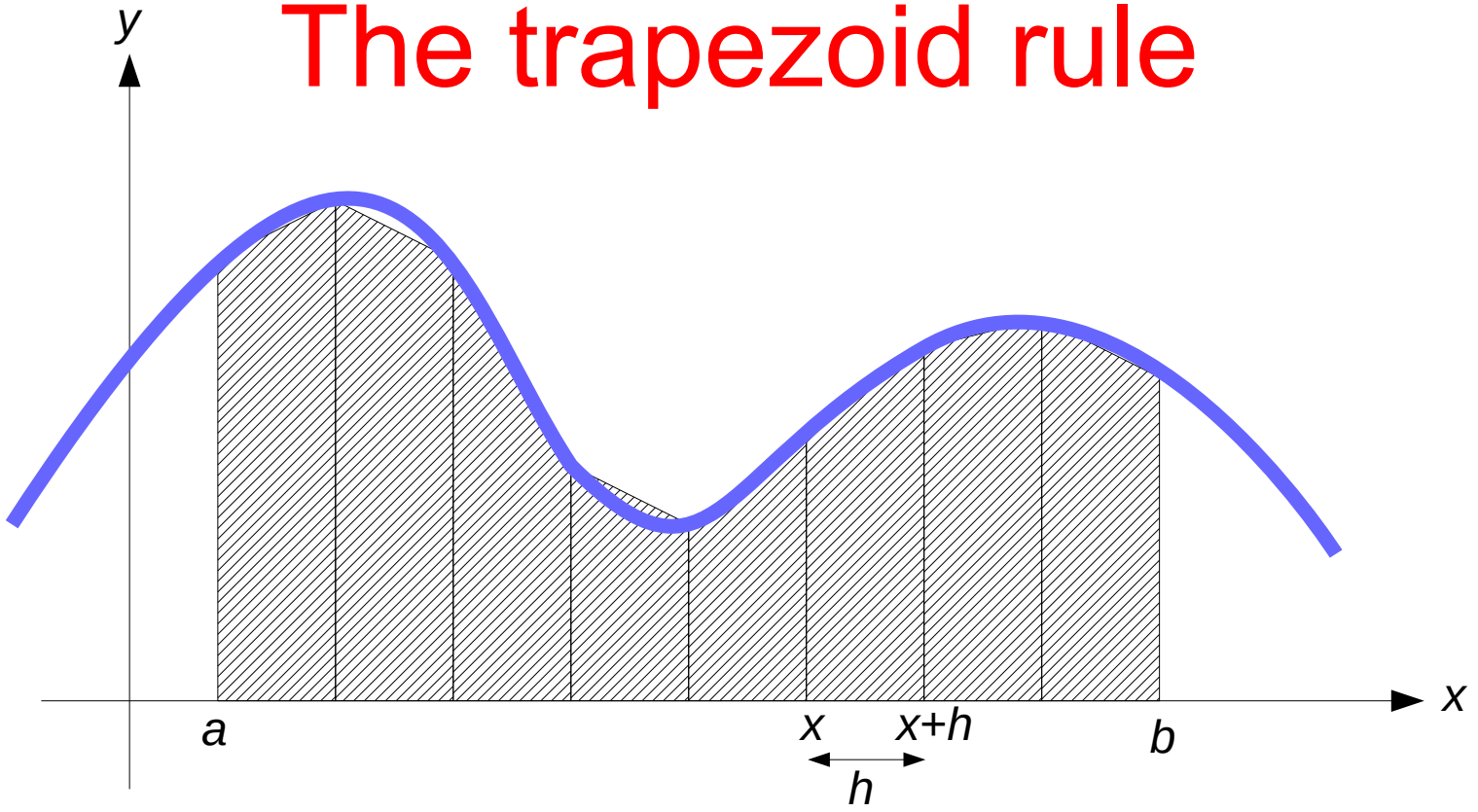
GCC annoys us

- GCC < 9.x: OK
- GCC ≥ 9.x: OK

```
const int foo = 1;
#if __GNUC__ < 9
#pragma omp parallel default(none)
#else
#pragma omp parallel default(none) shared(foo)
#endif
{
    int baz = 0;
    baz += foo;
}
```

Example: the trapezoid rule

The trapezoid rule




```
/* Serial trapezoid rule */
h = (b-a)/n;
result = 0;
x = a;
for (i=0; i<n; i++) {
    result += h*(f(x) + f(x+h))/2;
    x += h;
}
return result;
```

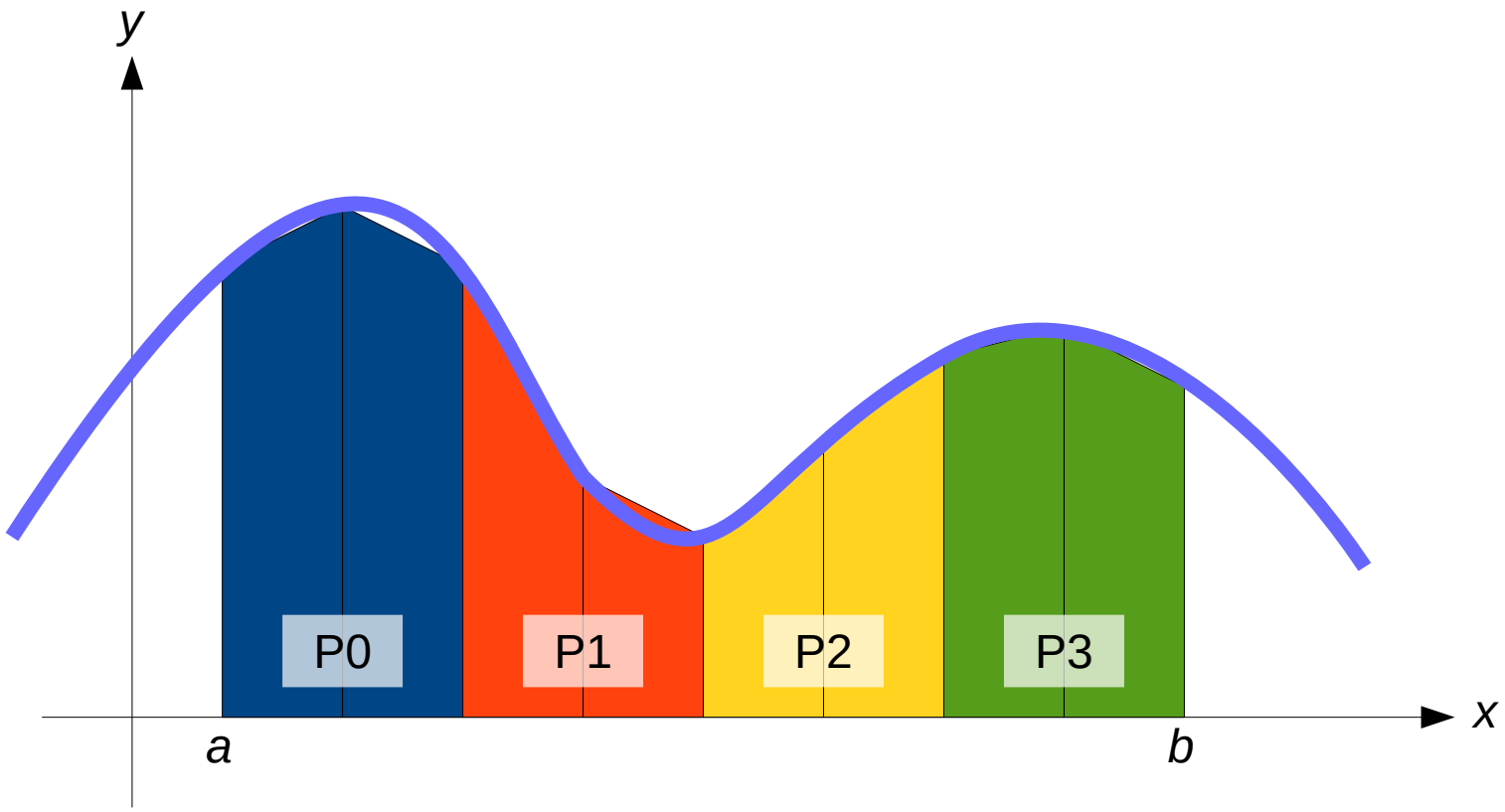
See trap.c

A first OpenMP version

- Two types of tasks
  - computation of the areas of individual trapezoids
  - adding the areas of trapezoids
- Areas can be computed independently
  - *embarrassingly parallel* problem
- We assume that there are more trapezoids than OpenMP threads (*n* >> *P*)

```
/* Serial trapezoid rule */
h = (b-a)/n;
result = 0;
x = a;
for (i=0; i<n; i++) {
    result += h*(f(a) + f(a+h))/2;
    x += h;
}
return result;
```

Assigning trapezoids to threads



A first OpenMP version

- Split the *n* intervals across OpenMP threads
- Thread *t* stores its result in **`partial_result[t]`**
- The master sums all partial results
- See omp-trap0.c
  - Try adding "default(none)" to the **`omp parallel`** clause

A second OpenMP version

- Split the *n* intervals across OpenMP threads
- Thread *t*...
  - ...stores its result in a local variable **`partial_result`**
  - ...updates the global result
- **`omp atomic`**
  - Protects updates to a *shared variable*
  - Updates must be of the form "read-update-write", e.g., **`var += x`**
- **`omp critical`**
  - Protects access to a *critical section*, which may consist of arbitrary instructions
  - All threads will eventually execute the critical section; however, only one thread at a time can be inside the critical block
- **`critical`** protects code; **`atomic`** protects memory locations
- See omp-trap1.c

The `atomic` directive

- The `omp atomic` directive ensures that only one thread at the time updates a shared variable

```
#pragma omp parallel
{
    double partial_result = trap(a, b, n);
#pragma omp atomic
    result += partial_result;
}
```

- The code above forces all threads to serialize during the update of the shared variable
  - This is not a real problem, since each thread will update the shared variable exactly once
- We can also use the `reduction` clause

The **reduction** clause

- **reduction(<op> : <variable>)**

can be one of +, -, *, |, ^, &, |, &&, || (subtraction is handled in a slightly different way to ensure that the result is what is expected)

```
#pragma omp parallel reduction(+:result)
{
    double partial_result = trap(a, b, n);
    result += partial_result;
}
```

See omp-trap2.c

Reduction operators

- A reduction operator is a binary associative operator such as addition or multiplication
  - An operator ◊ is associative if (a ◊ b) ◊ c = a ◊ (b ◊ c)
- A reduction is a computation that repeatedly applies the same reduction operator to a sequence of operands to get a single result
  - ◊-reduce(x_0, x_1, … x_{n-1}) = x_0 ◊ x_1 ◊ … ◊ x_{n-1}

How the `reduction` clause works

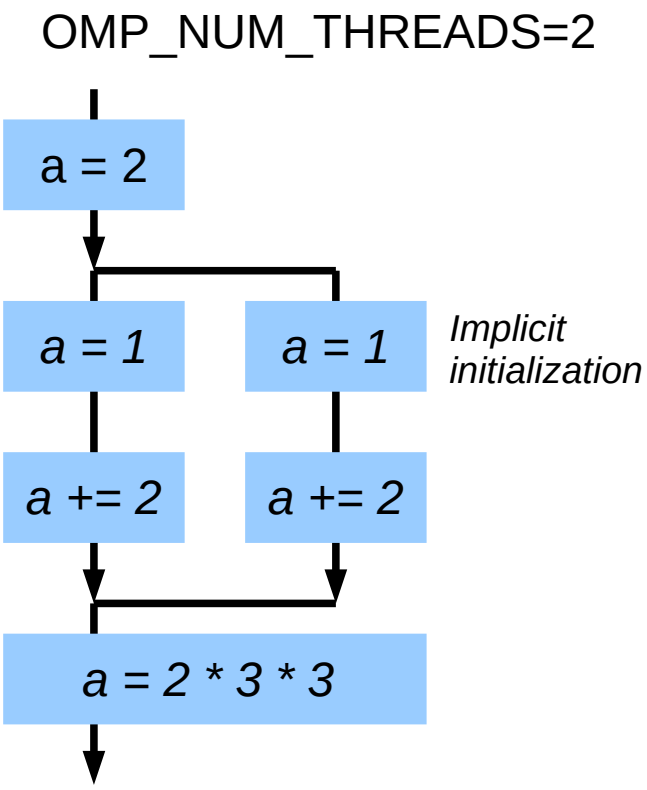
- One **`private`** copy of the reduction variable is created for each thread
- Each private copy is initialized with the neutral element of the reduction operator (e.g., 1 for *****, 0 for **+**)
- Each thread executes the parallel region
- When all threads finish, the reduction operator is applied to the *last* value of each local reduction variable, *and* the value the reduction variable had *before* the parallel region

```
/* omp-reduction.c */
#include <stdio.h>
int main( void )
{
    int a = 2;
#pragma omp parallel reduction(*:a)
    {
        a += 2;
    }
    printf("%d\n", a);
    return 0;
}
```

```
$ OMP_NUM_THREADS=1 ./omp-reduction
6
$ OMP_NUM_THREADS=2 ./omp-reduction
18
$ OMP_NUM_THREADS=4 ./omp-reduction
162
```

How the **reduction** clause works

```
/* omp-reduction.c */
#include <stdio.h>
int main( void )
{
    int a = 2;
#pragma omp parallel reduction(*:a)
    {
/* implicit initialization a = 1 */
        a += 2;
    }
    printf("%d\n",a);
    return 0;
}
```



Some valid reduction operators

Operator	Initial value
+	0
*	1
-	0
min	largest positive number
max	most negative number
&	~0
|	0
^	0
&&	0
||	1

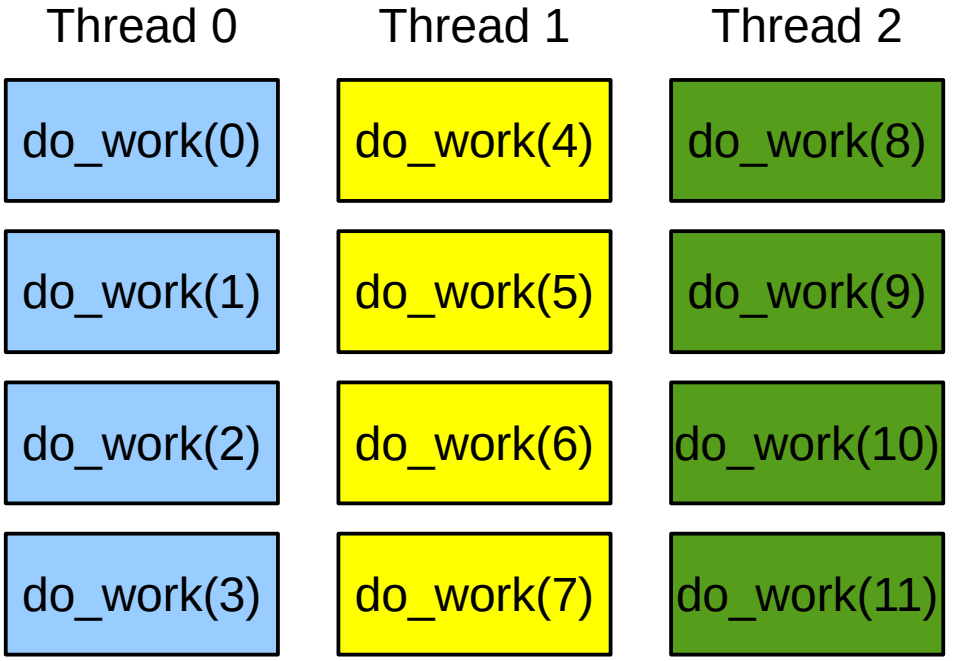
OpenMP 3.1 and later

The `omp for` directive

- The **`omp for`** directive is used inside a parallel block
- Loop iterations are assigned to the threads of the current team (the ones created with **`omp parallel`**)
  - The loop variable is made **private** by default

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

Implied barriers here



The `parallel for` directive

- The **`parallel`** and **`for`** directives can be collapsed in a single **`parallel for`**

```
double trap( double a, double b, int n )
{
    double result = 0;
    const double h = (b-a)/n;
    int i;
#pragma omp parallel for reduction(+:result)
    for ( i = 0; i<n-1; i++ ) {
       result += h*(f(a+i*h) + f(a+(i+1)*h))/2;
    }
    return result;
}
```

See omp-trap3.c

Legal forms for parallelizable *for* statements

```
for ( index = start ; index < end  ; index++
                      index <= end    ++index
                      index > end     index--
                      index >= end    --index
                                      index += incr
                                      index -= incr
                                      index = index + incr
                                      index = incr + index
                                      index = index - incr )
```

- Variable index must have integer or pointer type (e.g., it can't be a float)
- The expressions start, end, and incr must have a compatible type. For example, if index is a pointer, incr must have integer type
- The expressions start, end, and incr must not change during execution of the loop
- Variable index can only be modified by the "increment expression" in the "for" statement

Data dependencies

- It is not possible to use a **`parallel for`** directive if data dependencies are present
- Example: computation of PI

$$\pi = 4\left\{1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots\right\} = 4\sum_{k=0}^{+\infty}\frac{(-1)^k}{2k+1}$$

Loop Carried Dependency

```
double factor = 1.0;
double sum = 0.0;
for (k=0; k<n; k++) {
    sum += factor/(2*k + 1);
    factor = -factor;
}
pi_approx = 4.0 * sum;
```

Removing the data dependency

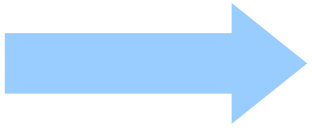
```
double factor;
double sum = 0.0;
#pragma omp parallel for private(factor) reduction(+:sum)
for (k=0; k<n; k++) {
    if ( k % 2 == 0 ) {
       factor = 1.0;
    } else {
       factor = -1.0;
    }
    sum += factor/(2*k + 1);
}
pi_approx = 4.0 * sum;
```

factor *must have private scope*

Can my loop be parallelized?

- Loop dependence analysis
- A quick-and-dirty test: run the loop *backwards*
  - If the program is still correct, the loop *might* be parallelizable
  - Not 100% reliable, but works most of the time

```
for (i=0; i<n; i++) {
    /* loop body */
}
```

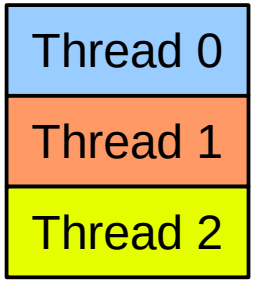


```
for (i=n-1; i>=0; i--) {
    /* loop body */
}
```

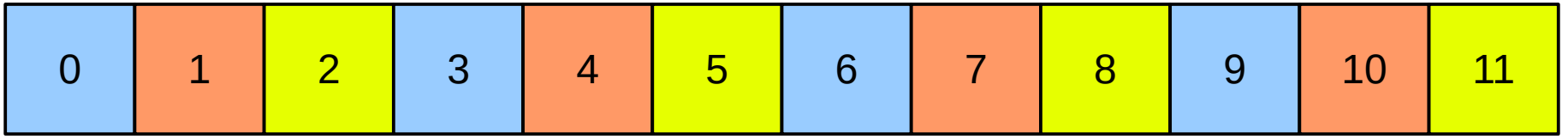
schedule(*type, chunksize*)

- **type** can be:
  - **static**: the iterations can be assigned to the threads before the loop is executed. If **chunksize** is not specified, iterations are evenly divided contiguously among threads
  - **dynamic** or **guided**: iterations are assigned to threads while the loop is executing. Default **chunksize** is 1
  - **auto**: the compiler and/or the run-time system determines the schedule
  - **runtime**: the schedule is determined at run-time using the **OMP_SCHEDULE** environment variable (e.g., **export OMP_SCHEDULE="static,1"**)
- Default schedule type is implementation dependent
  - GCC seems to use **static** by default

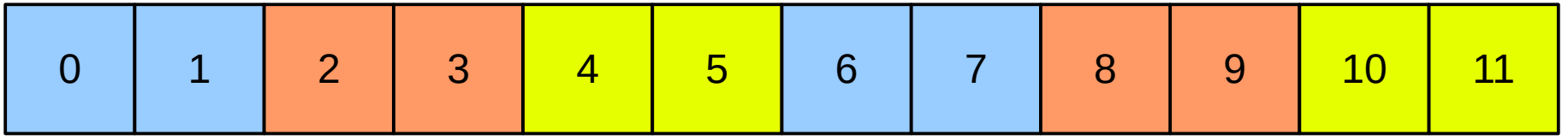
Example



- Twelve iterations 0, 1, … 11 and three threads
- **`schedule(static, 1)`**

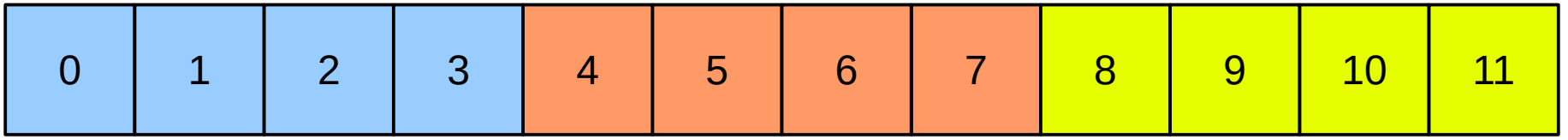


- **`schedule(static, 2)`**



- **`schedule(static, 4)`**

Default chunksize in this case



The Dynamic/Guided Schedule Types

- The iterations are broken up into chunks of **`chunksize`** consecutive iterations
  - However, in a **`guided`** schedule, as chunks are completed the size of the new chunks decreases
- Each thread executes a chunk, and when a thread finishes a chunk, it requests another one from the run-time system
  - Master/Worker paradigm

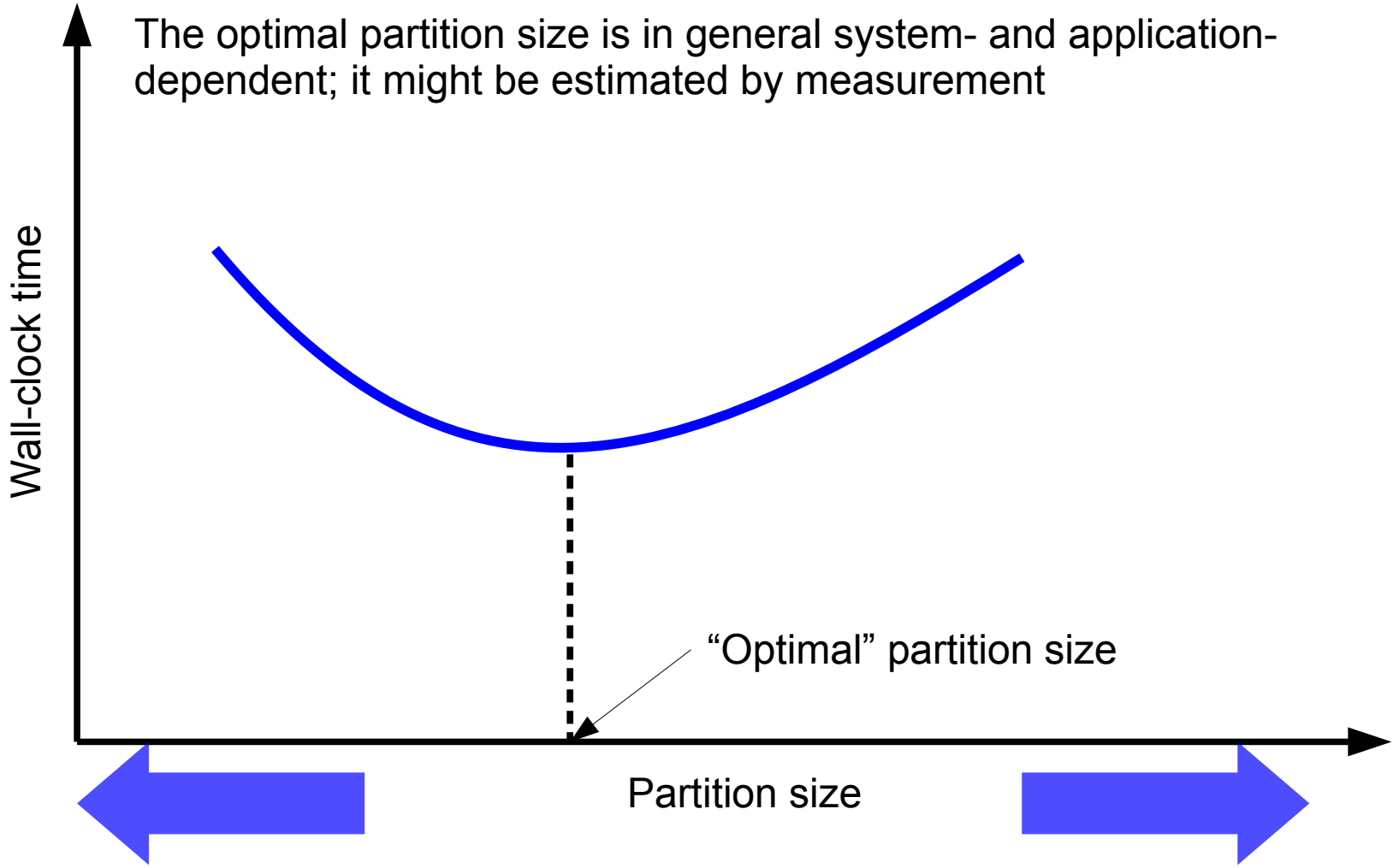
Choosing a schedule clause

Schedule clause	When to use	Note
`static`	Pre-determined and predictable work per iteration	Least work at runtime: scheduling done at compile-time
`dynamic`	Unpredictable, highly variable work per iteration	Most work at runtime: complex scheduling logic used at run-time

Credits: Tim Mattson

Choosing the partition size

The optimal partition size is in general system- and application-dependent; it might be estimated by measurement



Too small = higher scheduling overhead

Too large = unbalanced workload

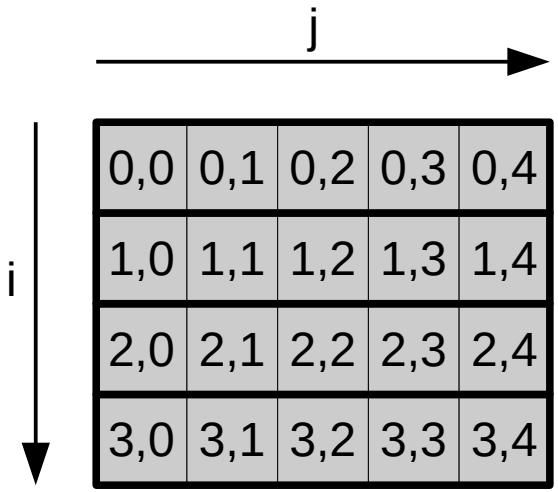
The `collapse` directive

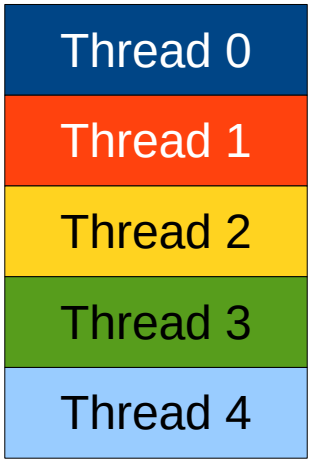
- Specifies how many loops in a nested loop should be collapsed into one large iteration space and divided according to the schedule clause

```
#pragma omp parallel for collapse(2)
    for ( y = 0; y < ysize; y++ ) {
        for ( x = 0; x < xsize; x++ ) {
            drawpixel( x, y );
        }
    }
```

collapse(2) makes x and y private by default

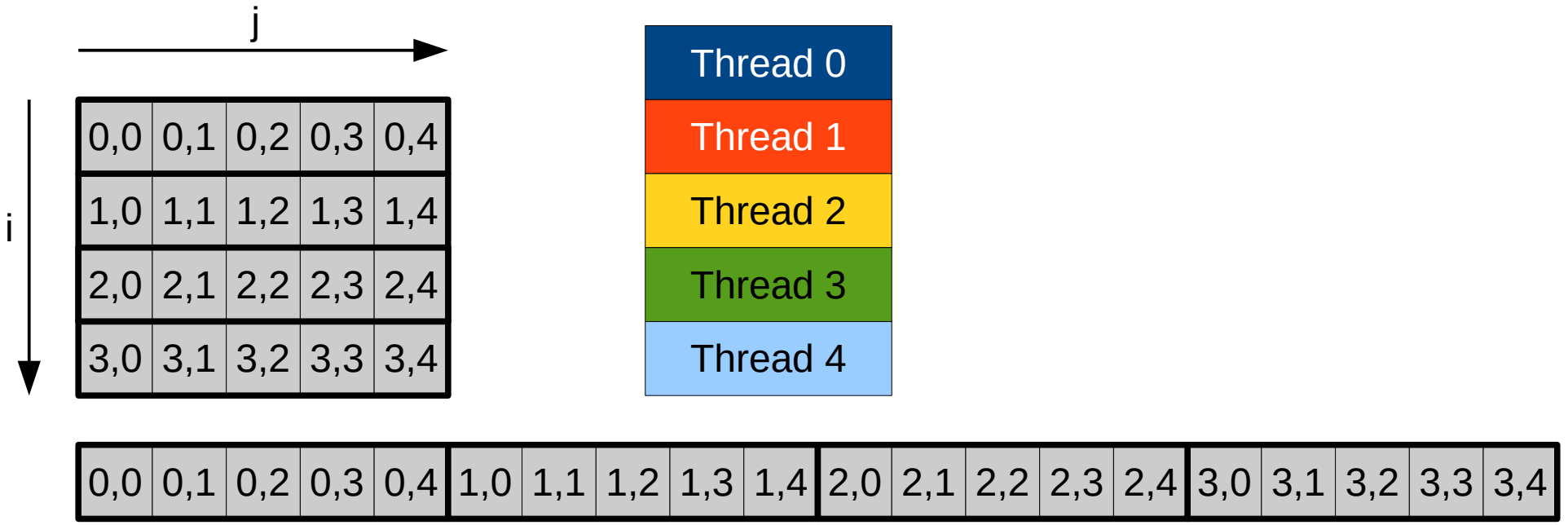
How `collapse` works





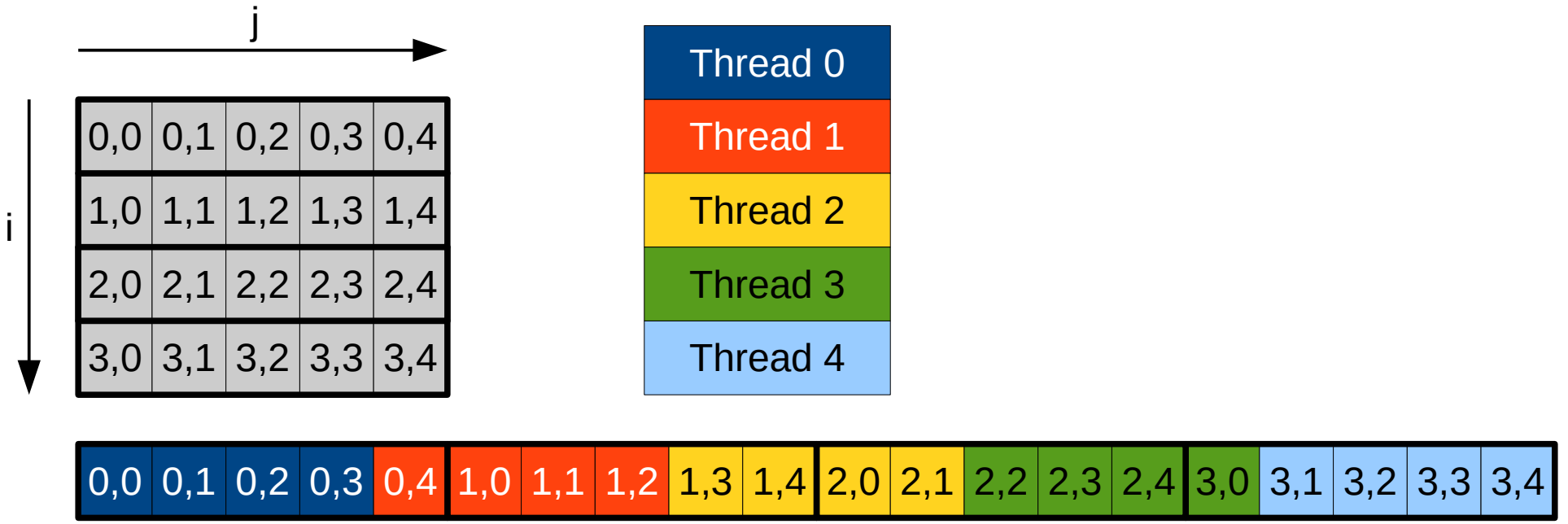

```
#pragma omp parallel for num_threads(5) collapse(2)
for (i=0; i<4; i++) {
    for (j=0; j<5; j++) {
        do_work(i,j);
    }
}
```

How `collapse` works



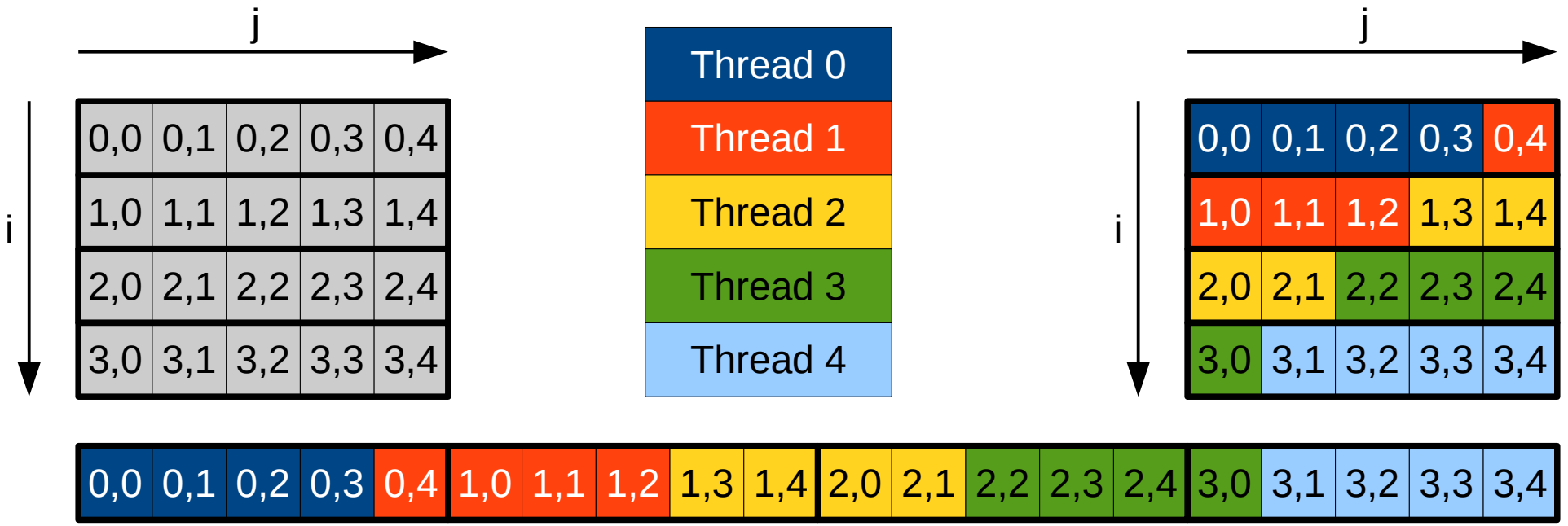

```
#pragma omp parallel for num_threads(5) collapse(2)
for (i=0; i<4; i++) {
    for (j=0; j<5; j++) {
        do_work(i,j);
    }
}
```

How `collapse` works




```
#pragma omp parallel for num_threads(5) collapse(2)
for (i=0; i<4; i++) {
    for (j=0; j<5; j++) {
        do_work(i,j);
    }
}
```

How `collapse` works




```
#pragma omp parallel for num_threads(5) collapse(2)
for (i=0; i<4; i++) {
    for (j=0; j<5; j++) {
        do_work(i,j);
    }
}
```

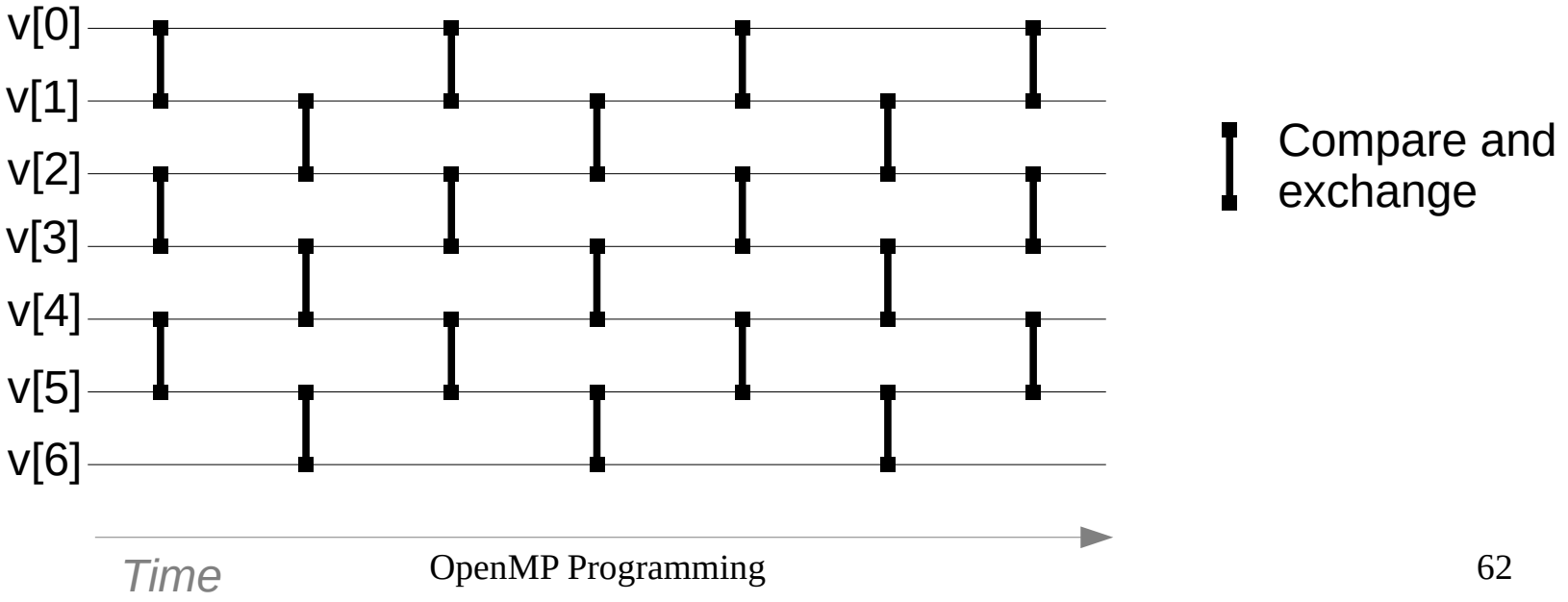
Spot the bug

```
int i, j, n, m, temp;
...
#pragma omp parallel for private(temp)
for (i=0; i<n; i++){
    for (j=0; j<m; j++){
        temp = b[i]*c[j];
        a[i][j] = temp * temp + d[i];
    }
}
```

Example: Odd-Even Transposition Sort

Serial Odd-Even Transposition Sort

- Variant of bubble sort
- Compare all (even, odd) pairs of adjacent elements, and exchange them if in the wrong order
- Then compare all (odd, even) pairs, exchanging if necessary; repeat the step above



Serial Odd-Even Transposition Sort

- Variant of bubble sort
- Compare all (even, odd) pairs of adjacent elements, and exchange them if in the wrong order
- Then compare all (odd, even) pairs, exchanging if necessary; repeat the step above

```
for (phase = 0; phase < n; phase++) {
    if ( phase % 2 == 0 ) {
        for (i=0; i<n-1; i+=2) {
            if (v[i] > v[i+1]) swap( &v[i], &v[i+1] );
        }
    } else {
        for (i=1; i<n-1; i+=2) {
            if (v[i] > v[i+1]) swap( &v[i], &v[i+1] );
        }
    }
}
```

First OpenMP Odd-Even Sort

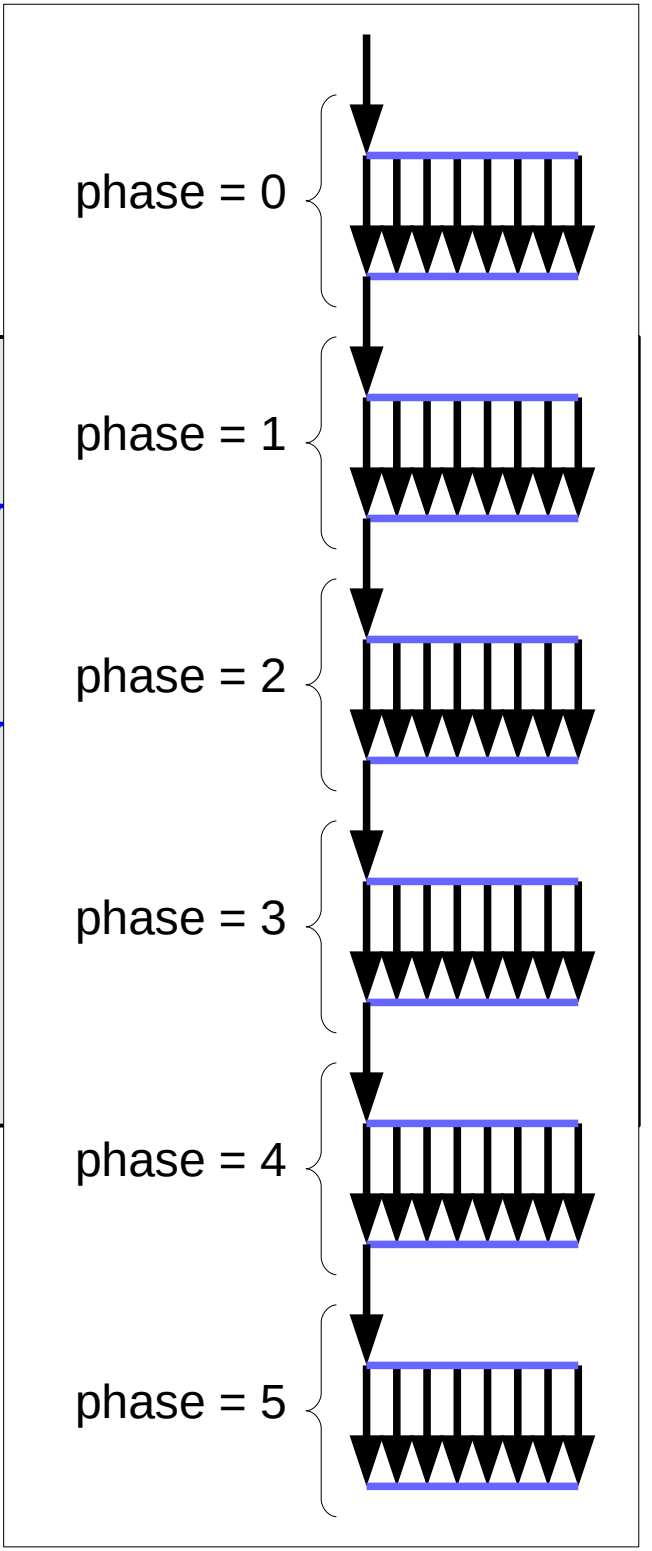
```
for (phase = 0; phase < n; phase++) {
    if ( phase % 2 == 0 ) {
#pragma omp parallel for default(none) shared(v,n)
        for (i=0; i<n-1; i+=2) {
            if (v[i] > v[i+1]) swap( &v[i], &v[i+1] );
        }
    } else {
#pragma omp parallel for default(none) shared(v,n)
        for (i=1; i<n-1; i+=2) {
            if (v[i] > v[i+1]) swap( &v[i], &v[i+1] );
        }
    }
}
```

First OpenMP Odd-Even Sort

```
for (phase = 0; phase < n; phase++) {
    if ( phase % 2 == 0 ) {
#pragma omp parallel for default(none) shared(v,n)
        for (i=0; i<n-1; i+=2) {
            if (v[i] > v[i+1]) swap( &v[i], &v[i+1] );
        }
    } else {
#pragma omp parallel for default(none) shared(v,n)
        for (i=1; i<n-1; i+=2) {
            if (v[i] > v[i+1]) swap( &v[i], &v[i+1] );
        }
    }
}
```

First OpenMP Odd-Even

```
for (phase = 0; phase < n; phase++) {
    if ( phase % 2 == 0 ) {
#pragma omp parallel for default(none) shared(v
            if (v[i] > v[i+1]) swap( &v[i], &v[i+
        }
    } else {
#pragma omp parallel for default(none) shared(v
        for (i=1; i<n-1; i+=2) {
            if (v[i] > v[i+1]) swap( &v[i], &v[i+
        }
    }
}
```



First OpenMP Odd-Even Sort

- The pool of threads is being created/destroyed at each **`omp parallel for`** region
- This *may* produce some overhead, depending on the OpenMP implementation
- You can nest **`omp for`** inside **`omp parallel`** to recycle the threads from the same pool

Second OpenMP Odd-Even Sort

```
#pragma omp parallel default(none) shared(v,n) private(phase)
for (phase = 0; phase < n; phase++) {
    if ( phase % 2 == 0 ) {
#pragma omp for
        for (i=0; i<n-1; i+=2) {
            if (v[i] > v[i+1]) swap( &v[i], &v[i+1] );
        }
    } else {
#pragma omp for
        for (i=1; i<n-1; i+=2 ) {
            if (v[i] > v[i+1]) swap( &v[i], &v[i+1] );
        }
    }
}
```

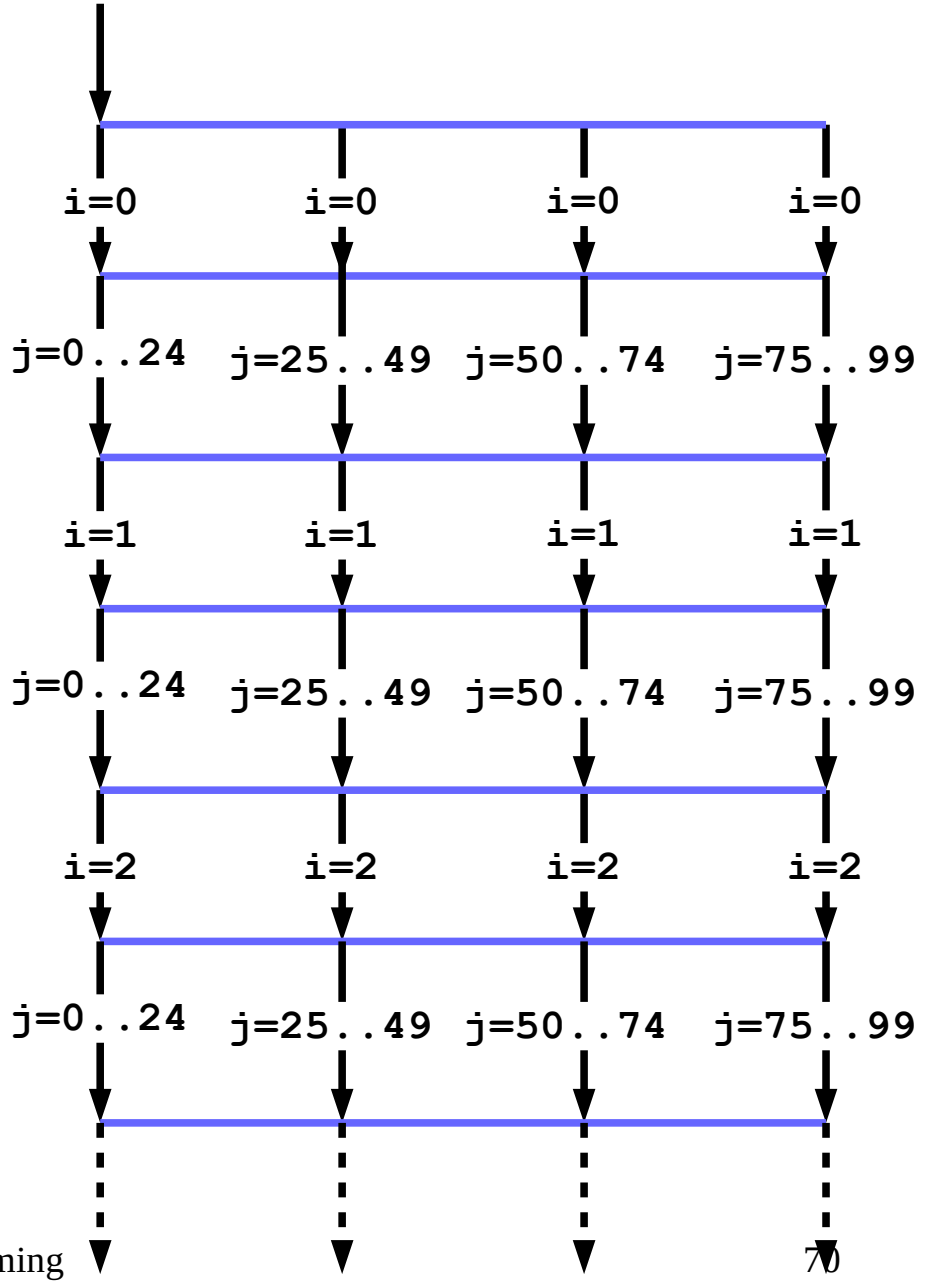
Create thread pool

`#pragma omp for` uses the threads from the pool created with `#pragma omp parallel`

See omp-odd-even.c

How it works

```
#pragma omp parallel num_threads(4) \
   default(none) private(i)
for (i = 0; i < 20; i++) {
#pragma omp for
    for (j=0; j < 100; j++) {
        do_work(i, j);
    }
}
```

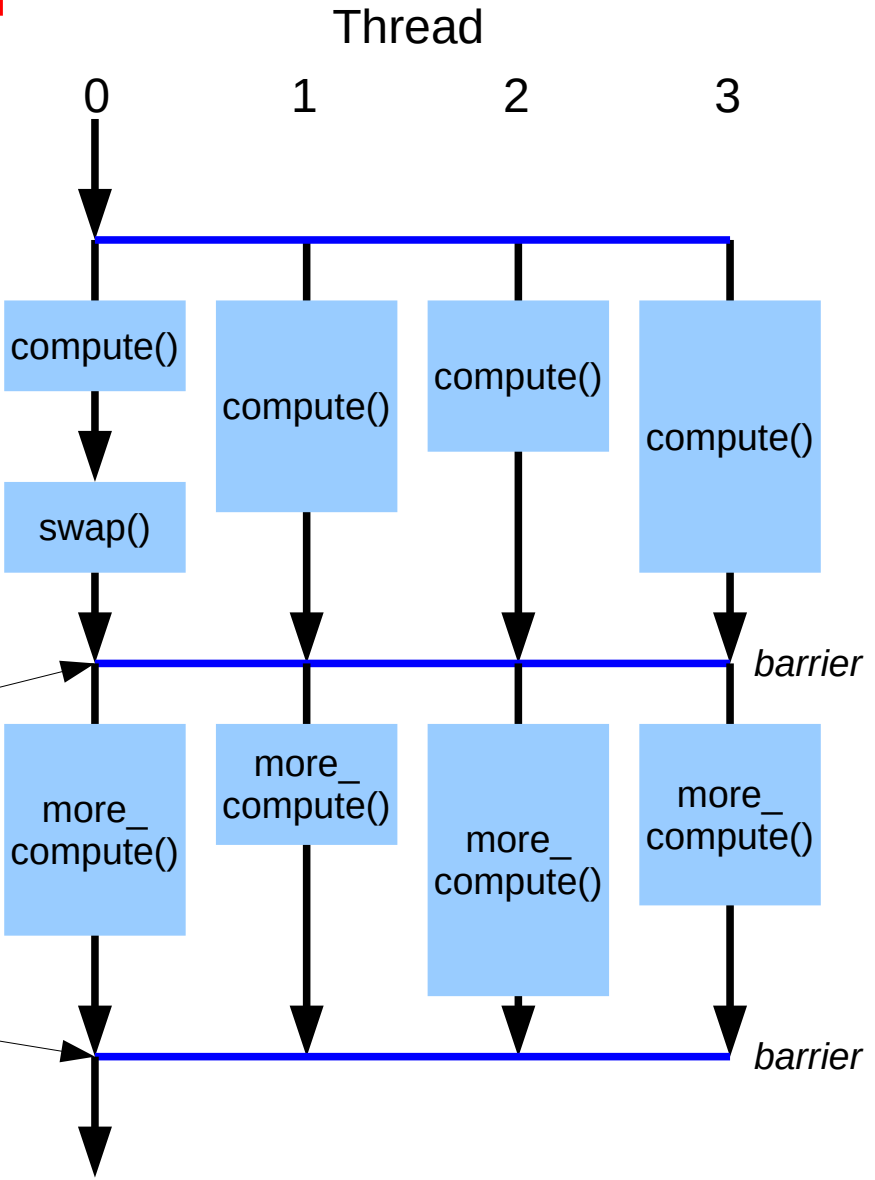


OpenMP synchronization

- **`#pragma omp barrier`**
  - All threads in the currently active team must reach this point before they are allowed to proceed
- **`#pragma omp master`**
  - Marks a parallel region which is executed by the master only (the thread with rank = 0); other threads just skip the region
  - There is no implicit barrier at the end of the block
- **`#pragma omp single`**
  - Marks a parallel region which is executed once by the first thread reaching it, whichever it is
  - A barrier is implied at the end of the block

Example

```
#pragma omp parallel
{
    compute();

#pragma omp master
    swap();

#pragma omp barrier
    more_compute();
}
```



Work-sharing constructs

OpenMP tasking constructs

- Not all programs have simple loops that OpenMP can parallelize
- Example: linked list traversal
  - Each node of the linked list is "processed" independently from other nodes

```
p = head;
while (p) {
    processwork(p);
    p = p->next;
}
```

- OpenMP **`parallel for`** works only for loops where the iteration count can be known in advance at runtime

Credits: Tim Mattson

What are OpenMP tasks?

- Tasks are independent units of work
- Tasks are composed of:
  - code to execute
  - data to compute with
- Threads are assigned to perform the work of each task
  - The thread that encounters the task construct may execute the task immediately
  - The threads may defer execution until later
- Tasks can be nested: a task may generate other tasks

Credits: Tim Mattson

#pragma omp task

```
#pragma omp parallel
{
    #pragma omp single
    {
        #pragma omp task
        fred();
        #pragma omp task
        barney();
        #pragma omp task
        wilma();
    }
}
```

Create a pool of threads

Only one thread creates the tasks (may also use #pragma omp master)

Tasks executed by some threads in some order

All tasks complete before this barrier is released

Credits: Tim Mattson

Data scoping with tasks

- You can specify the scope of variables within a **`#pragma omp task`** directive
- Variables can be shared, private or firstprivate
  - If a variable is shared on a task construct, the references to it inside the construct are to the storage with that name at the point where the task was encountered
  - If a variable is private on a task construct, the references to it inside the construct are to new uninitialized storage that is created when the task is executed
  - If a variable is firstprivate on a construct, the references to it inside the construct are to new storage that is created and initialized with the value of the existing storage of that name when the task is encountered

Credits: Tim Mattson

Data scoping with tasks

- The behavior you want for tasks is usually firstprivate, because the task may not be executed until later (and variables may have gone out of scope)
  - Variables that are private when the task construct is encountered are firstprivate by default

```
#pragma omp parallel shared(A) private(B)
{
    ...
    #pragma omp task
    {
        int C;
        compute(A, B, C);
    }
}
```

A is shared
B is firstprivate
C is private

Credits: Tim Mattson

Linked list traversal with tasks

```
#pragma omp parallel
{
#pragma omp single
    {
        p=head;
        while (p) {
#pragma omp task firstprivate(p)
            processwork(p);
            p = p->next;
        }
    }
}
```

Creates a task with its own copy of "p" initialized to the value of "p" when the task is defined

Credits: Tim Mattson

When/Where are tasks completed?

- At thread barriers (explicit or implicit)
  - e.g., at the end of a **`#pragma omp parallel`** block
  - applies to all tasks generated in the current parallel region up to the barrier
- At **`taskwait`** directive
  - wait until all tasks defined in the current task (not in the descendants!) have completed.
  - **`#pragma omp taskwait`**
  - The code executed by a thread in a parallel region is considered a task here

Credits: Tim Mattson

Example: parallel Fibonacci with tasks

```
#include <stdio.h>
int fib( int n )
{
    int n1, n2;
    if (n < 2) {
        return 1;
    } else {
        n1 = fib(n-1);
        n2 = fib(n-2);
        return n1 + n2;
    }
}

int main( int argc, char* argv[] )
{
    int n = 10, res;
    res = fib(n);
    printf("fib(%d)=%d\n", n, res);
    return 0;
}
```

- $F_n = F_{n-1} + F_{n-2}$
  - $F_0 = F_1 = 1$
- Inefficient algorithm $O(2^n)$

Credits: Tim Mattson

Example: parallel Fibonacci with tasks

```
#include <stdio.h>
int fib( int n )
{
    int n1, n2;
    if (n < 2) {
        return 1;
    } else {
#pragma omp task shared(n1)
        n1 = fib(n-1);
#pragma omp task shared(n2)
        n2 = fib(n-2);
#pragma omp taskwait
        return n1 + n2;
    }
}

int main( int argc, char* argv[] )
{
    int n = 10, res;
#pragma omp parallel
#pragma omp master
    res = fib(n);
    printf("fib(%d)=%d\n", n, res);
    return 0;
}
```

- Binary tree of tasks
- A task cannot complete until all tasks below it in the tree are complete
  - enforced with **`taskwait`**
- **`n1`**, **`n2`** are private to the current task
  - because they are local to a function called in a parallel block
  - So they would be firstprivate
- Must be shared on child tasks because their value must be stored at their parent

Credits: Tim Mattson

Concluding Remarks

- OpenMP is a standard for programming shared-memory systems.
  - Uses both special functions and preprocessor directives called *pragmas*.
- OpenMP programs start multiple threads
  - By default most systems use a block-partitioning of the iterations in a parallelized for loop
  - OpenMP offers a variety of scheduling options.
- In OpenMP the scope of a variable is the collection of threads to which the variable is accessible.
- A reduction is a computation that repeatedly applies the same reduction operator to a sequence of operands in order to get a single result.