

Shared Memory Programming with OpenMP

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

moreno.marzolla@unibo.it

Copyright © 2013, 2014, 2017–2019
Moreno Marzolla, Università di Bologna, Italy
<http://www.moreno.marzolla.name/teaching/HPC/>



This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License (CC BY-SA 4.0). To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/4.0/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.

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

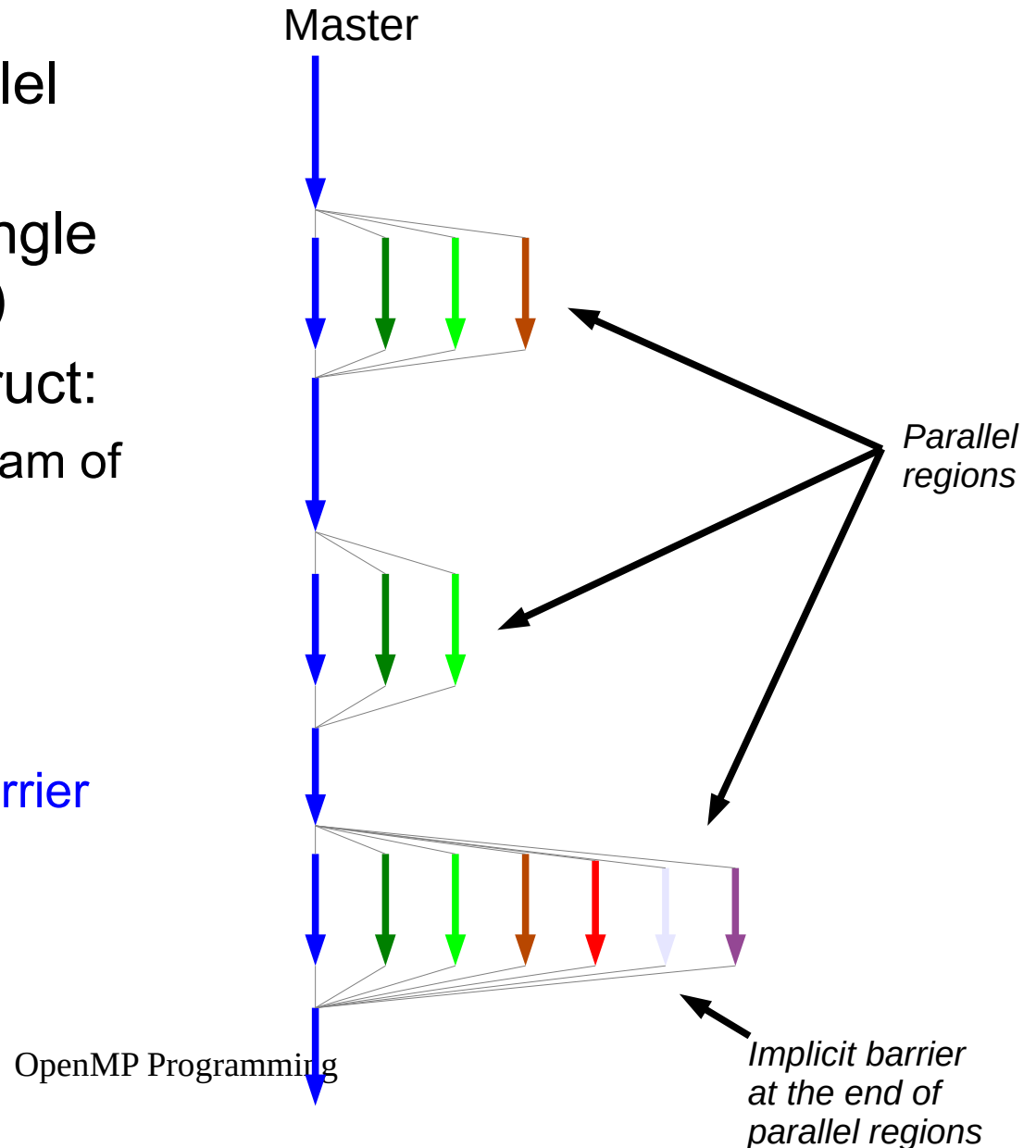
- Model for shared-memory parallel programming
- Portable across shared-memory architectures
- Incremental parallelization
 - Parallelize individual computations in a program while leaving the rest of the program sequential
- Compiler based
 - Compiler generates thread programs and synchronization
-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

Most OpenMP programs only use these items

OpenMP pragma, function, or clause	Concepts
<code>#pragma omp parallel</code>	Parallel region, teams of threads, structured block, interleaved execution across threads
<code>int omp_get_thread_num()</code> <code>int omp_get_num_threads()</code>	Create threads with a parallel region and split up the work using the number of threads and thread ID
<code>double omp_get_wtime()</code>	Timing blocks of code
<code>setenv OMP_NUM_THREADS N</code> <code>export OMP_NUM_THREADS=N</code>	Set the default number of threads with an environment variable
<code>#pragma omp barrier</code> <code>#pragma omp critical</code> <code>#pragma omp atomic</code>	Synchronization, critical sections
<code>#pragma omp for</code> <code>#pragma omp parallel for</code>	Worksharing, parallel loops
<code>reduction(op:list)</code>	Reductions of values across a team of threads
<code>schedule(dynamic [,chunk])</code> <code>schedule(static [,chunk])</code>	Loop schedules
<code>private(list), shared(list),</code> <code>firstprivate(list)</code>	Data environment
<code>#pragma omp master</code> <code>#pragma omp single</code>	Worksharing with a single thread
<code>#pragma omp task</code> <code>#pragma omp taskwait</code>	Tasks including the data environment for tasks.

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 instructions
 - 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

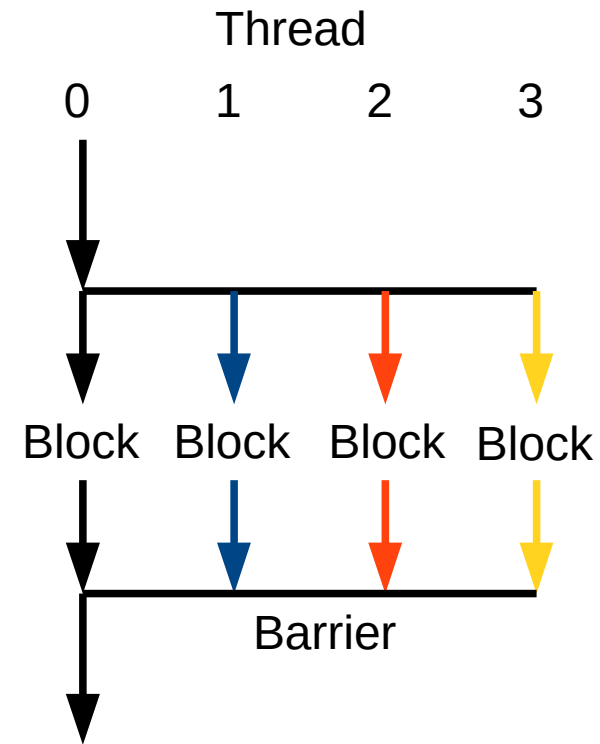
The `#pragma omp parallel` directive

- When a thread reaches a `parallel` directive, it creates a team of threads and becomes the master of the team
 - The master has thread ID 0
- 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 thread 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

```
/* 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

```
/* 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
```

```
/* 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

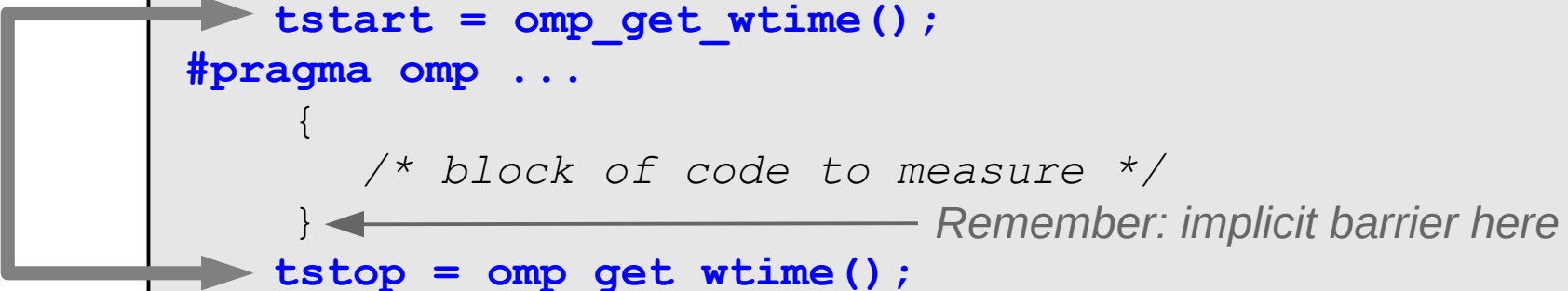
- `omp_get_num_threads()` returns the number of threads in the **currently active** pool
 - If no thread pool is active, the function returns 1
- `omp_get_max_threads()` returns the **maximum number of threads** that can be created

```
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](#)

Taking times

```
double tstart, tstop;
tstart = omp_get_wtime();
#pragma omp ...
{
    /* block of code to measure */
} ← Remember: implicit barrier here
tstop = omp_get_wtime();
printf("Elapsed time: %f\n", tstop - tstart);
```




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?

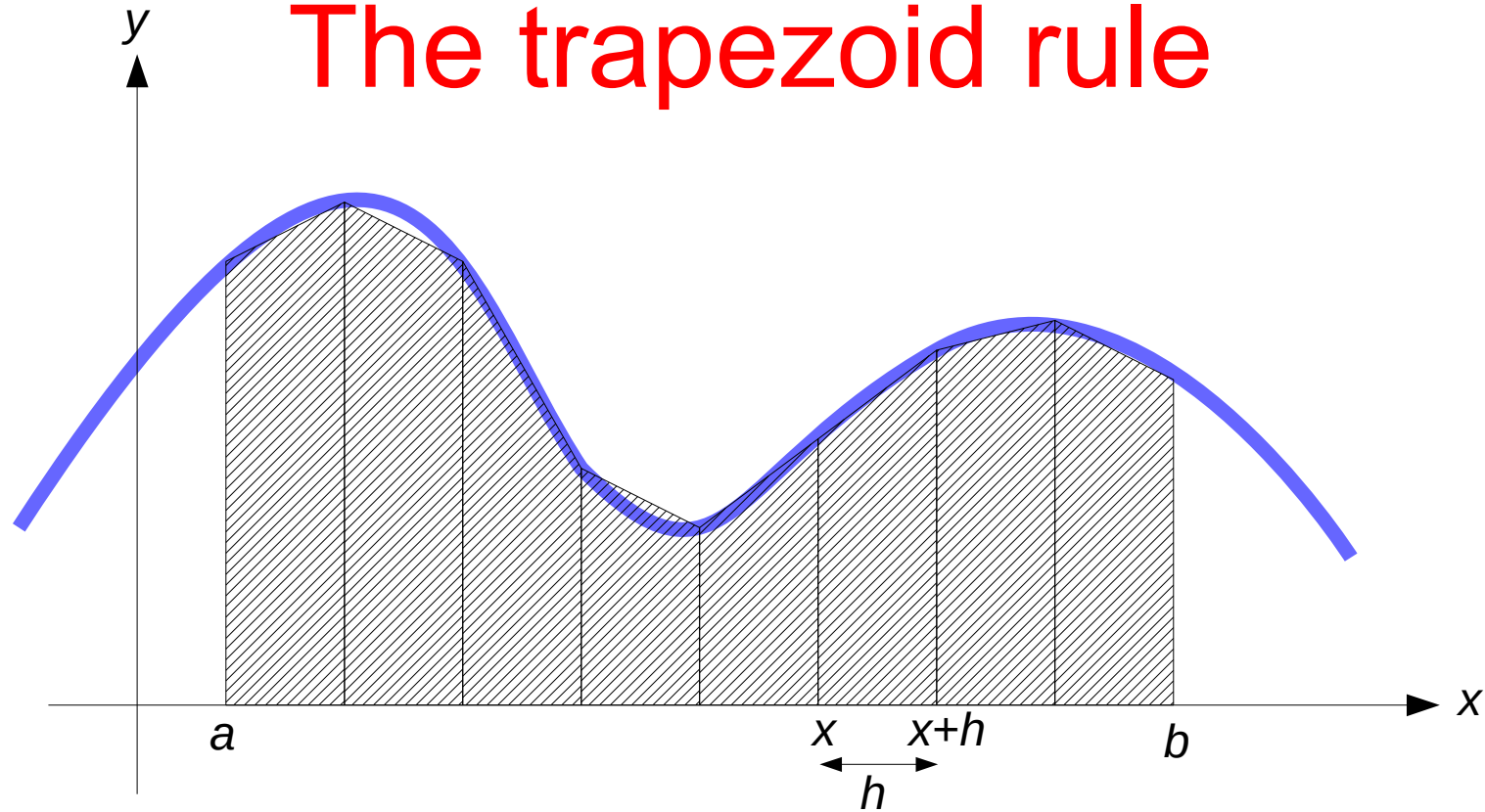
```
/* omp-scope.c */
#include <stdio.h>

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

Example: the trapezoid rule

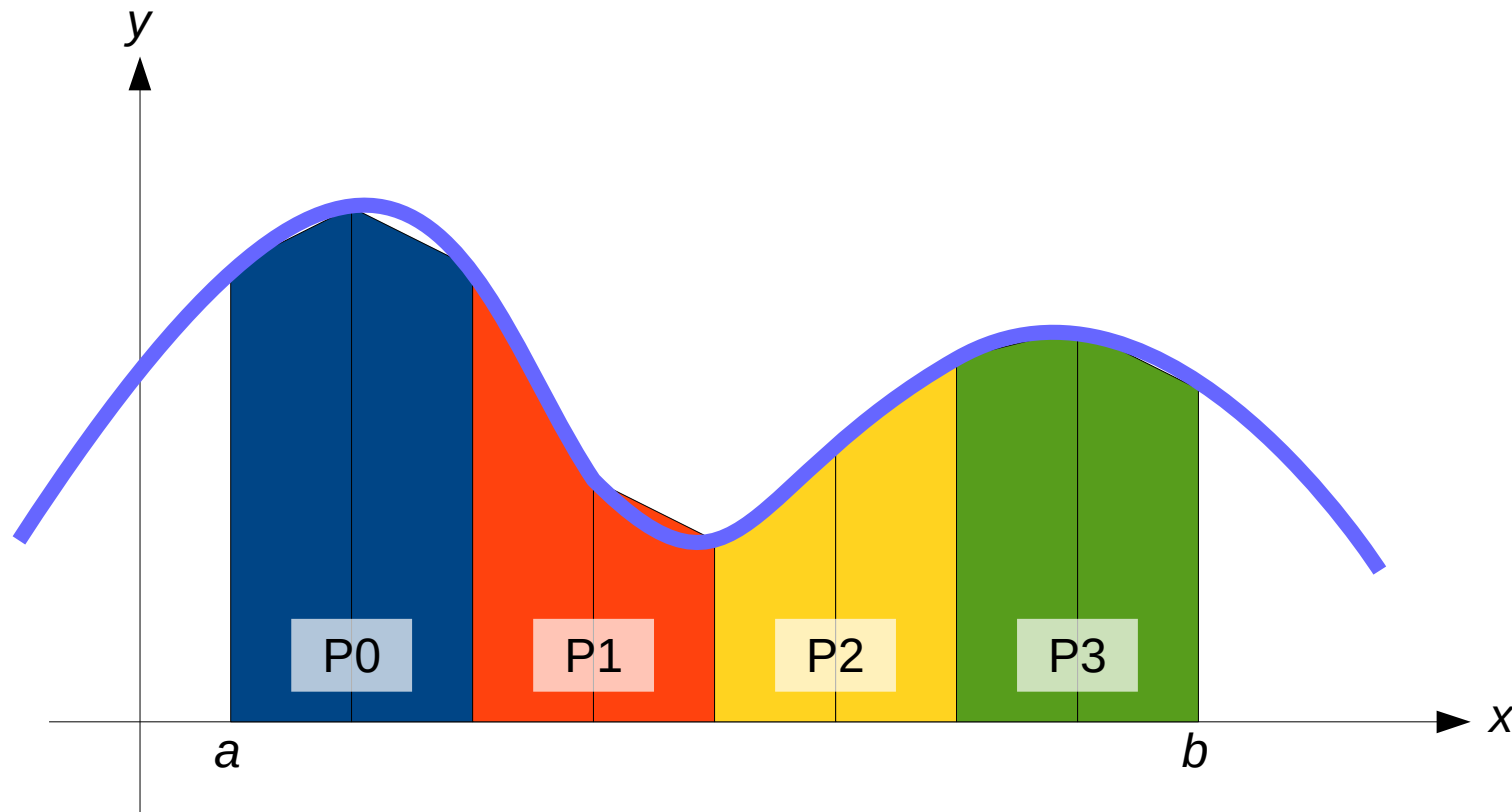
The trapezoid rule



```
/* 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

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 +, *, |, ^, &, |, &&, ||, (in principle also subtraction, but the OpenMP specification does not guarantee the result to be uniquely determined)

```
#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 \diamond is associative if $(a \diamond b) \diamond c = a \diamond (b \diamond c)$
- A **reduction** is a computation that repeatedly applies the same reduction operator to a sequence of operands to get a single result
 - \diamond -reduce(x_0, x_1, \dots, x_{n-1}) = $x_0 \diamond x_1 \diamond \dots \diamond x_{n-1}$

How the **reduction** clause works

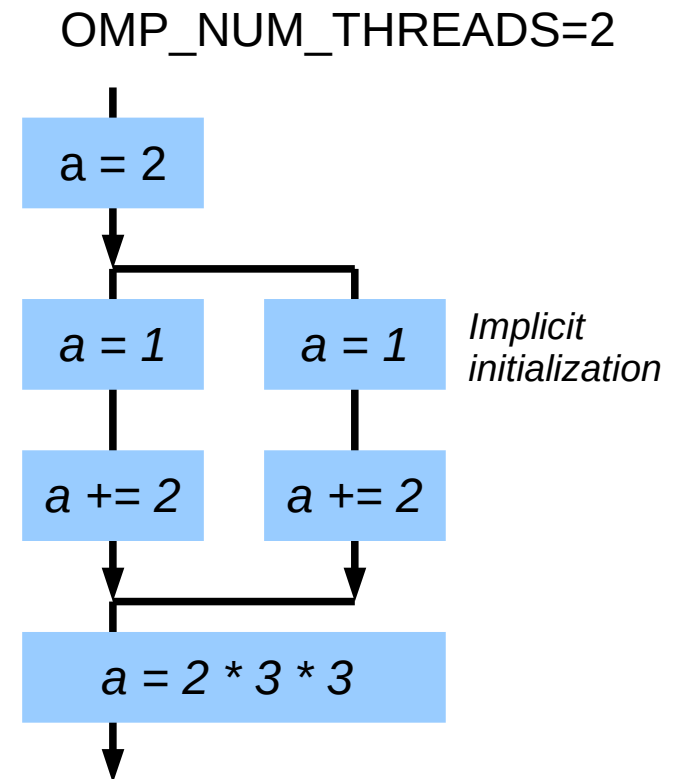
- One **private** copy of the reduction variable is created for each thread
- Each private copy is initialized with the neutral element of the reduction operator (e.g., 1 for *****, 0 for **+**)
- Each thread executes the parallel region
- When all threads finish, the reduction operator is applied to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

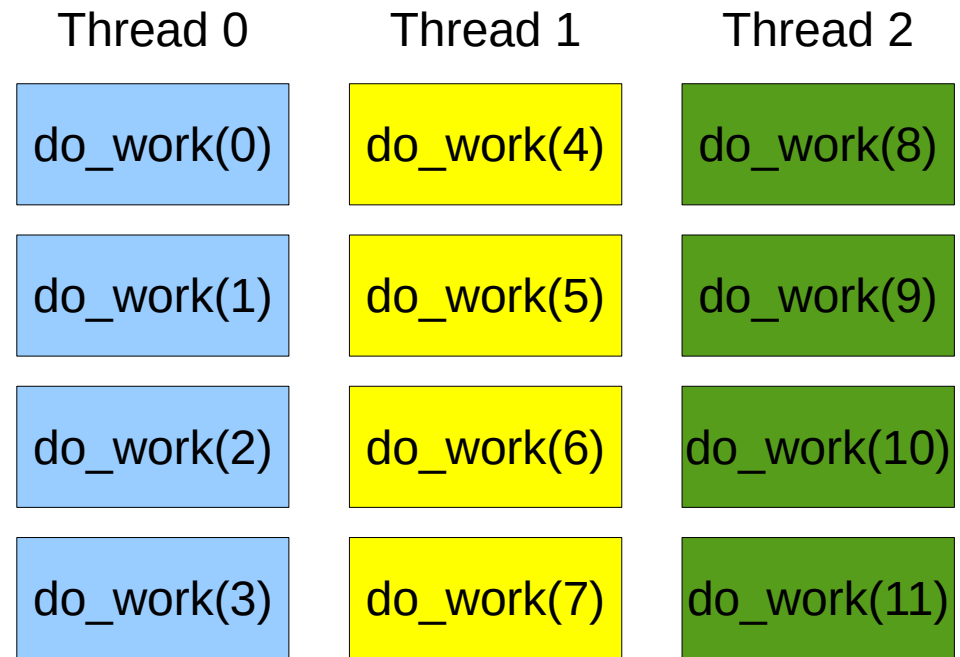
← *OpenMP 3.1
and later*

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`)

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

*The loop variable
(in this case, "i") is
private by default*



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 -= 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

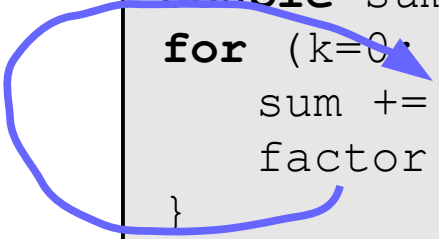
Data dependencies

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

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

*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?

- We will devote a whole lecture to this problem
 - Stay tuned...
- 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 */  
}
```

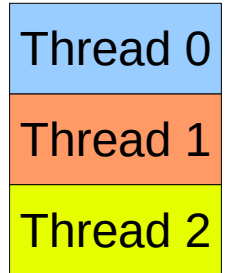
Scheduling loops

```
/* omp-mandelbrot.c */  
  
...  
  
int main( int argc, char *argv[] )  
{  
    int x, y;  
  
    gfx_open( xsize, ysize, "Mandelbrot Set");  
    #pragma omp parallel for private(x) schedule(dynamic,64)  
    for ( y = 0; y < ysize; y++ ) {  
        for ( x = 0; x < xsize; x++ ) {  
            drawpixel( x, y );  
        }  
    }  
    printf("Click to finish\n");  
    gfx_wait();  
    return 0;  
}
```

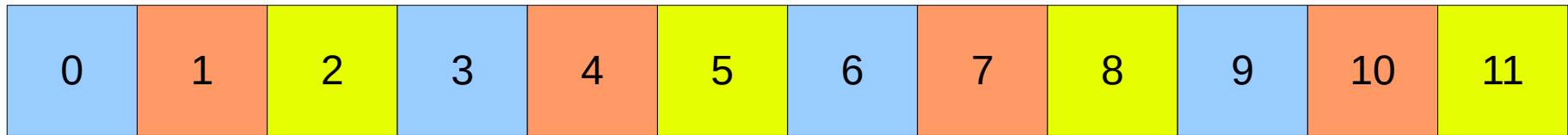
`schedule (type, chunksize)`

- `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

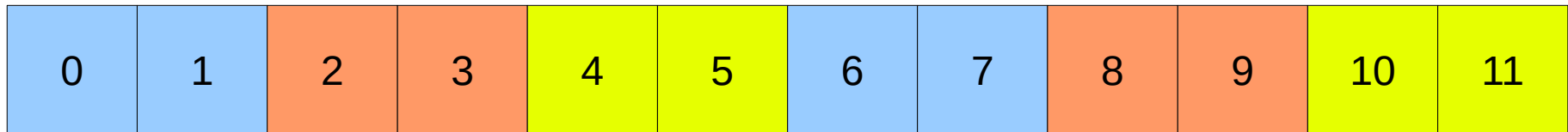
Example



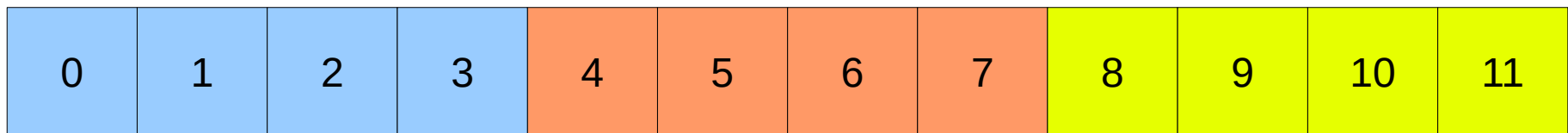
- Twelve iterations 0, 1, ... 11 and three threads
- `schedule (static, 1)`



- `schedule (static, 2)`



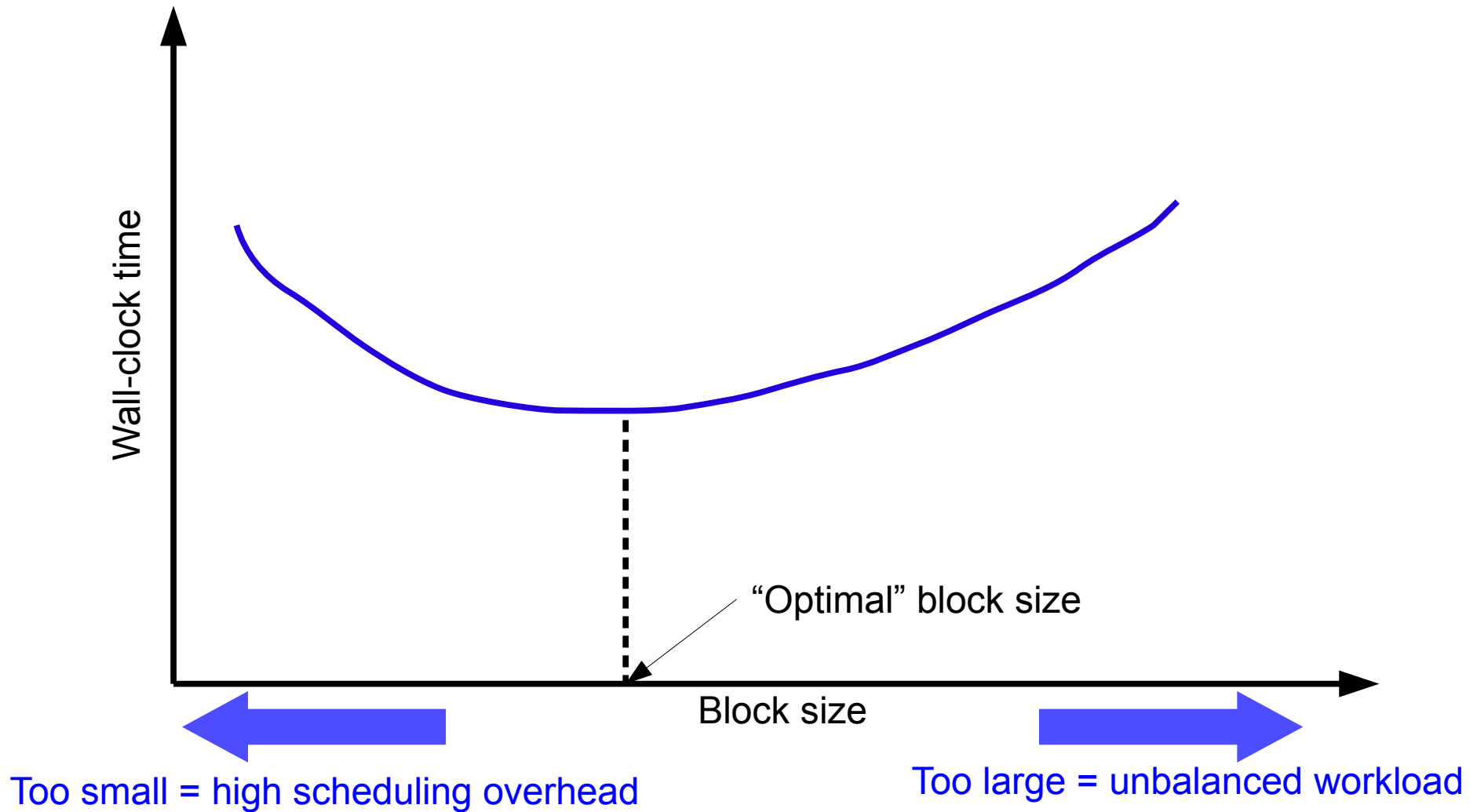
- `schedule (static, 4)` ← Default chunksize in this case



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

Choosing a block size



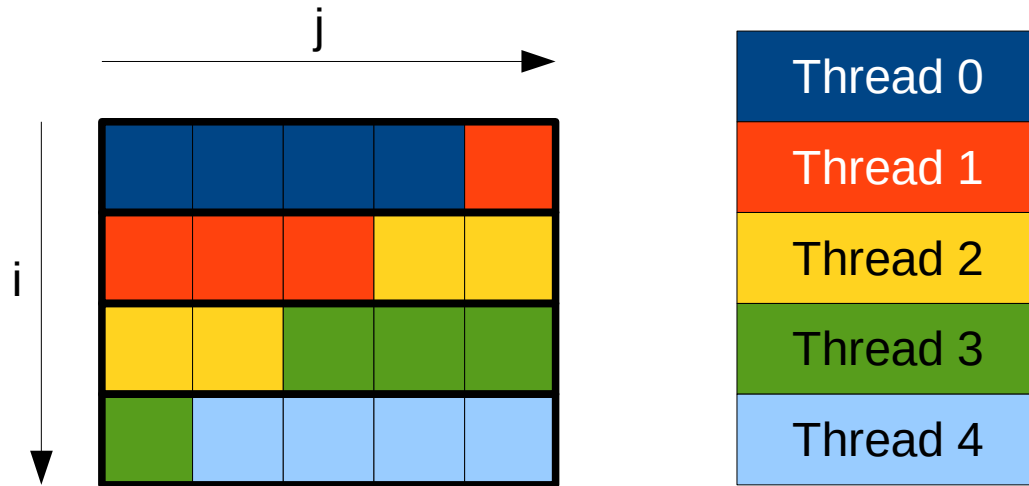
The collapse directive

- 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);
    }
}
```

See [omp-mandelbrot.c](#)

Spot the bug

```
int i, j;
...
#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];
    }
}
```