Shared Memory Programming with OpenMP

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/
- Blaise Barney, OpenMP https://computing.llnl.gov/tutorials/openMP/ (highly recommended!!)

OpenMP

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

OpenMP

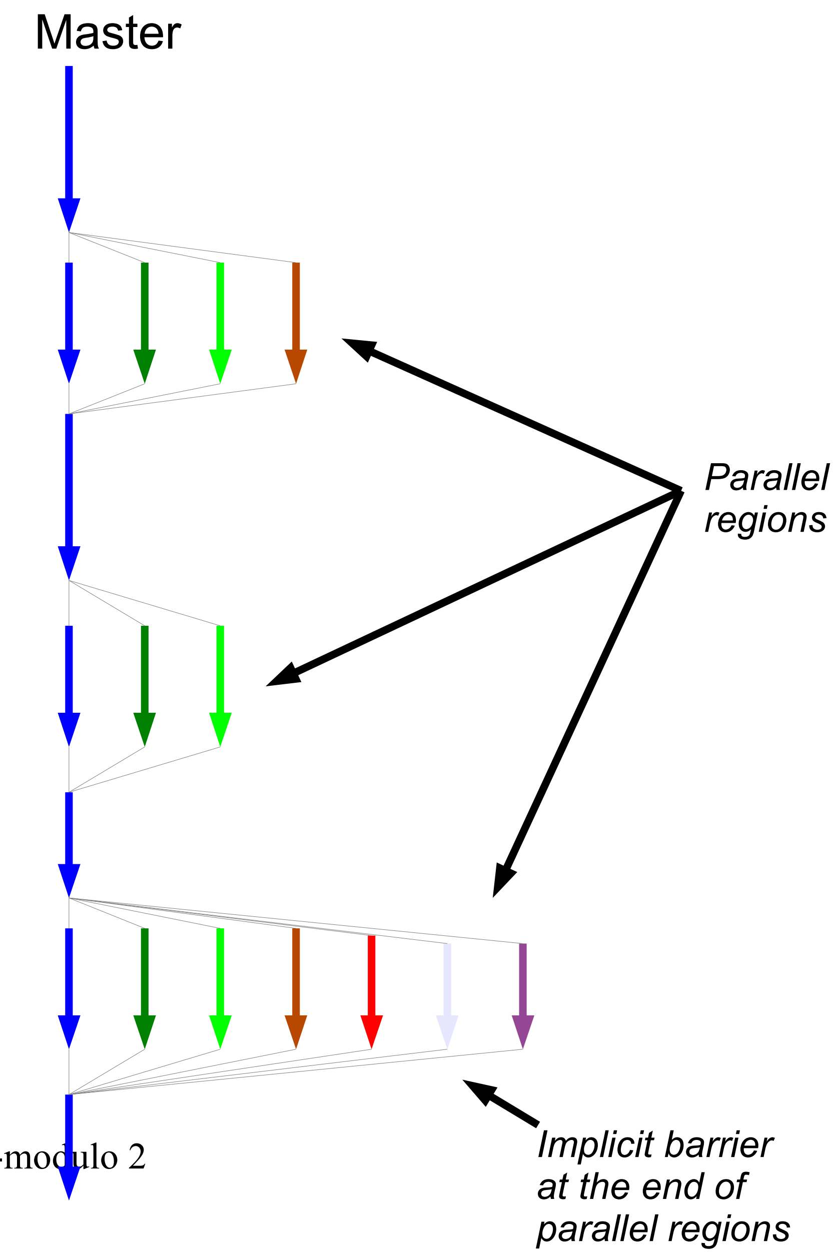
- OpenMP continues to evolve
- Initially, the API specifications were released separately for C and Fortran.
- Since 2005, they have been released together.
  - Oct 1997 Fortran 1.0
  - Oct 1998 C/C++ 1.0
  - Nov 1999 Fortran 1.1
  - Nov 2000 Fortran 2.0
  - Mar 2002 C/C++ 2.0
  - May 2005 OpenMP 2.5 ← *In these slides we will consider (a subset of) OpenMP 2.5, since it is more likely to be widely and correctly supported across compilers*
  - May 2008 OpenMP 3.0
  - Jul 2011 OpenMP 3.1
  - Jul 2013 OpenMP 4.0

A Programmer’s View of OpenMP

- OpenMP is a portable, threaded, shared-memory programming specification with “light” syntax
  - Exact behavior depends on OpenMP implementation!
  - Requires compiler support (C/C++ or Fortran)
- OpenMP will:
  - Allow a programmer to separate a program into serial regions and parallel regions, rather than concurrently-executing threads.
  - Hide stack management
  - Provide synchronization constructs
- OpenMP will not:
  - Parallelize automatically
  - Guarantee speedup
  - Provide freedom from 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 master thread continues execution



OpenMP uses Pragmas

`#pragma omp …`

- Pragmas are special preprocessor instructions.
- Typically added to a system to allow behaviors that aren’t part of the basic C specification.
- Compilers that don’t support the pragmas ignore them.
- Interpretation of OpenMP pragmas :
  - they modify the statement immediately following the pragma
  - this could be a compound statement such as a loop

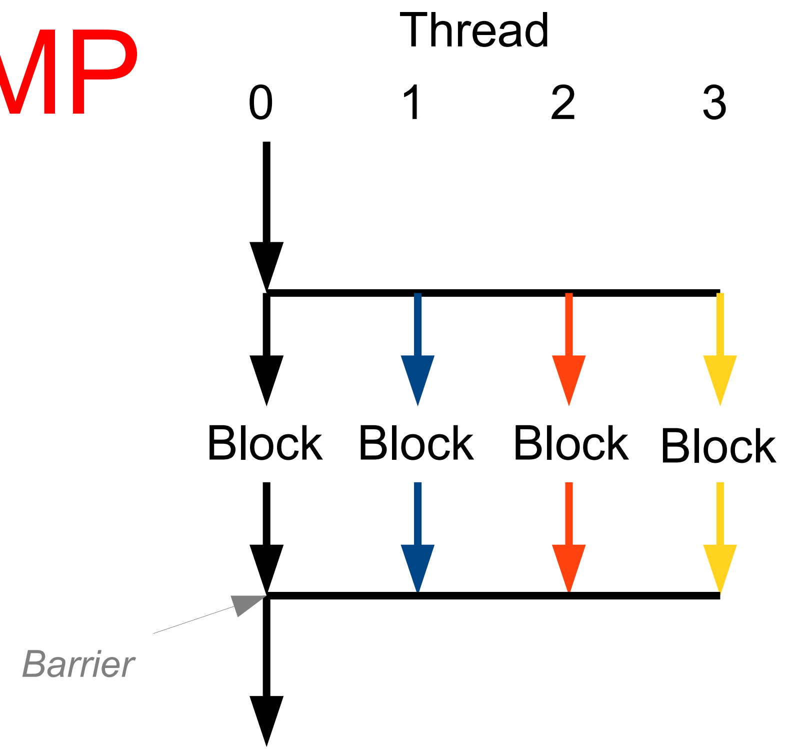
The `#pragma omp parallel` directive

- When a thread reaches a parallel directive, it creates a team of threads and becomes the master of the team. The master has thread number 0.
- Starting from the beginning of this parallel region, the code is duplicated and all threads will execute that code.
- 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

```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
```

In case the compiler doesn't support OpenMP

```
#ifdef _OPENMP
#include <omp.h>
#endif

/* … */

#ifdef _OPENMP
   int my_rank = omp_get_thread_num ( );
   int thread_count = omp_get_num_threads ( );
#else
   int my_rank = 0;
   int thread_count = 1;
#endif
```

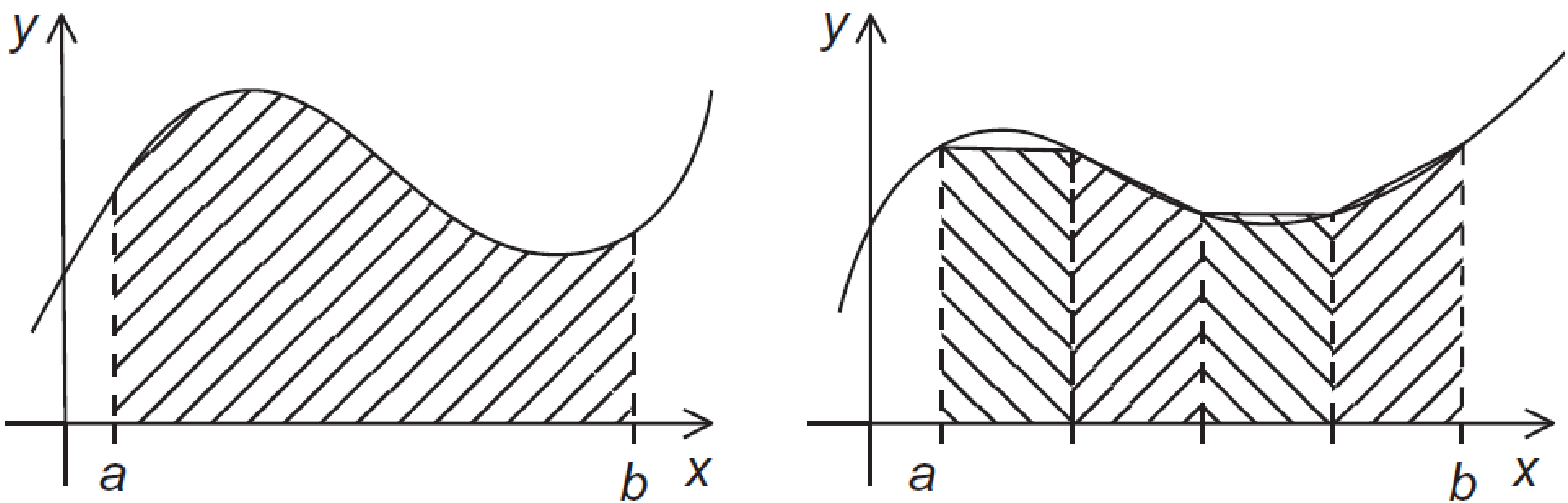
More complex example

```
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

Example: the trapezoid rule

The trapezoid rule



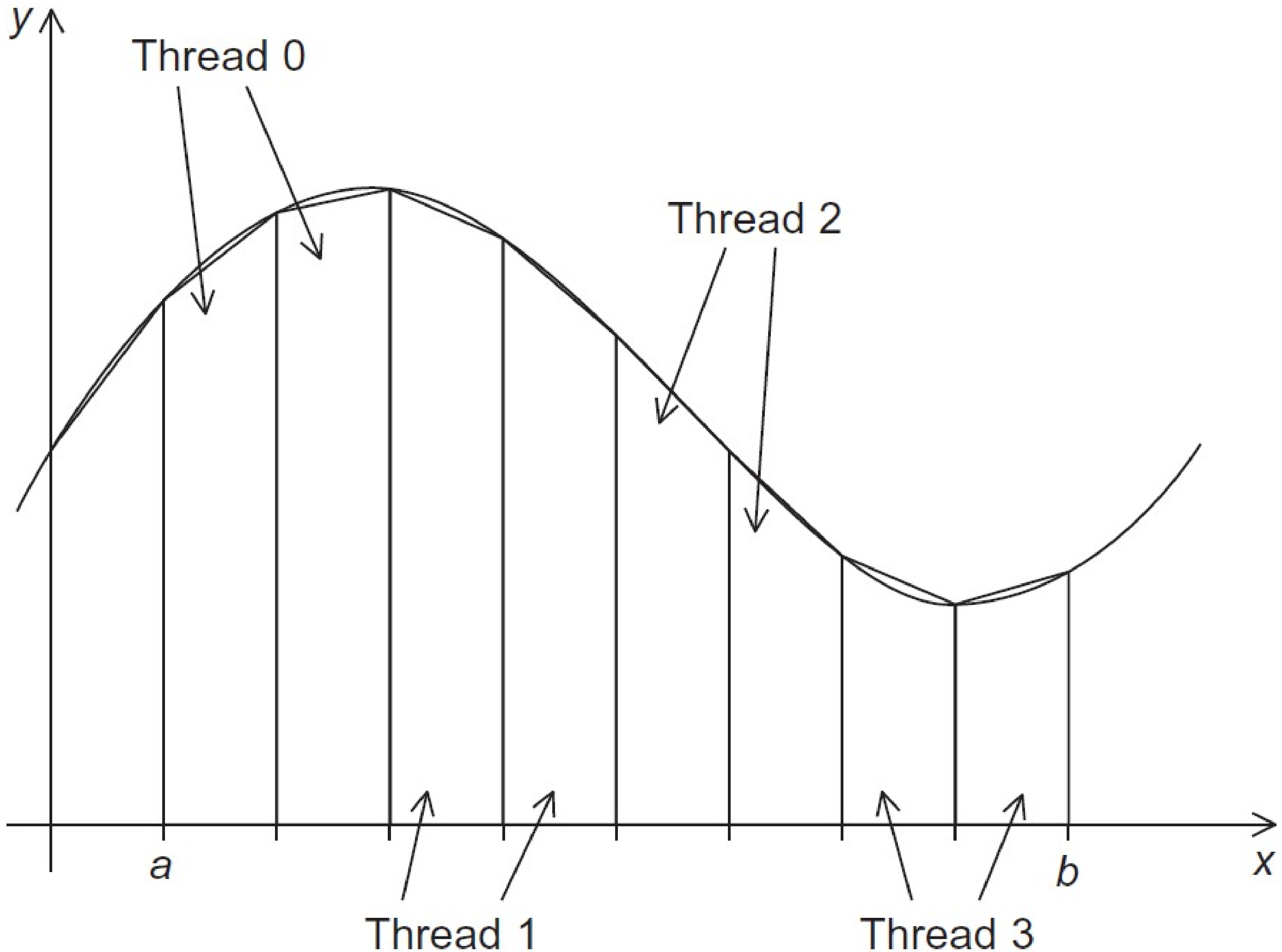

```
/* Serial trapezoid rule */
/* Input: a, b, n */
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
x_i = a + h;
for (i=1; i<n; i++) {
    approx += f(x_i);
    x_i += h;
}
return h * approx;
```

See trap_serial.c

A First OpenMP Version

- We identify two types of tasks:
  - computation of the areas of individual trapezoids, and
  - adding the areas of trapezoids.
- Areas can be computed independently (no communication needed)
- We assume that there are more trapezoids than cores

Assignment of trapezoids to threads



A First OpenMP Version of the Trapezoid Rule

- See omp_trap0.c

Scope

- In serial programming, the scope of a variable consists of those parts of a program in which the variable can be used.

- In OpenMP, the scope of a variable refers to the set of threads that can access the variable in a parallel block.

Scope in OpenMP

- `private(x)`
  - each thread has his own copy of x; **x is not initialized**
- `shared(x)` **(DEFAULT)**
  - every thread accesses the same memory location
- `firstprivate(x)`
  - each thread has his own copy of x; **x is initialized with the current value of x before the various threads start**
- `default(shared)` or `default(none)`
  - affects all the variables not specified in other clauses. default(none) can be used to check whether you consider all the variables

The reduction clause

- The `#pragma omp critical` directive ensures that only one thread at the time executes the next block

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

See trap_omp0.c

`#pragma omp atomic` *can also be used: it protects access to a shared variable, while* `#pragma omp critical` *defines a (general) critical section, that may be an arbitrary block of code*

- The code above is obviously inefficient, since it forces all threads to serialize during the update
- Solution: the `reduction` clause

Reduction operators

- A reduction operator is a binary operation (such as addition or multiplication).
- A reduction is a computation that repeatedly applies the same reduction operator to a sequence of operands in order to get a single result.
- All of the intermediate results of the operation should be stored in the same variable: the reduction variable.

The `reduction` clause

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

+, *, |, ^, &, |, &&, ||, in principle also subtraction, but in practice do NOT use it since 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_trap1.c

The `parallel for` directive

- Forks a team of threads to execute the for loop that follows
- The system parallelizes the for loop by dividing the iterations of the loop among the threads.

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

The loop variable is `private` *by default, so we don't need to do anything special*

See omp_trap2.c

Legal forms for parallelizable *for* statements

$$
\textbf{for}\left(\begin{array}{lclcl}
 & & & & \texttt{index++} \\
 & & & & \texttt{++index} \\
 & & \texttt{index < end} & & \texttt{index--} \\
 & & \texttt{index <= end} & & \texttt{--index} \\
\texttt{index = start} & ; & \texttt{index >= end} & ; & \texttt{index += incr} \\
 & & \texttt{index > end} & & \texttt{index -= incr} \\
 & & & & \texttt{index = index + incr} \\
 & & & & \texttt{index = incr + index} \\
 & & & & \texttt{index = index - incr}
\end{array}\right)
$$

Caveats

- The 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, then incr must have integer type.
- The expressions start, end, and incr must not change during execution of the loop.
- During execution of the loop, the variable index can only be modified by the "increment expression" in the for statement.

Data Dependencies
Estimating π

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

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 data dependency

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

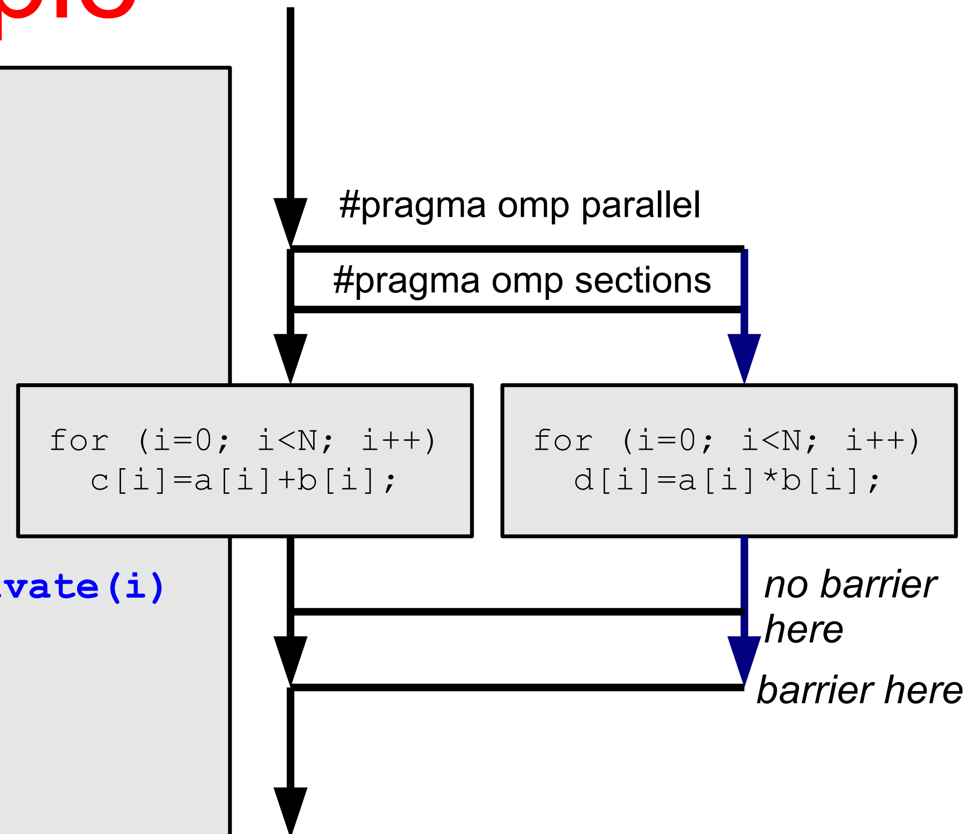
Ensures factor has private scope.

Work-sharing construct: the `sections` directive

- It specifies that the enclosed section(s) of code are to be divided among the threads in the team.
- Independent `section` directives are nested within a `sections` directive
  - Each `section` is executed once by one thread
  - Different `section` may be executed by different threads.
  - It is possible for a thread to execute more than one `section` if it is quick enough and the implementation permits such.
- There is an implied barrier at the end of a `sections` directive, unless the `nowait` clause is used.

Example

```c
#define N     1000
int main(void)
{
    int i;
    float a[N], b[N], c[N], d[N];
    /* Some initializations */
    for (i=0; i < N; i++) {
        a[i] = i * 1.5;
        b[i] = i + 22.35;
    }
#pragma omp parallel shared(a,b,c,d) private(i)
    {
    #pragma omp sections nowait
    {
        #pragma omp section
        for (i=0; i < N; i++)
            c[i] = a[i] + b[i];
        #pragma omp section
        for (i=0; i < N; i++)
            d[i] = a[i] * b[i];
        }  /* end of sections */
    }  /* end of parallel section */
    return 0;
}
```



Scheduling loops

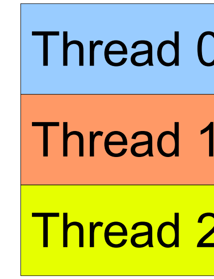
Example

```
/* mandelbrot.c */

...

int main( int argc, char *argv[] )
{
    int x, y;

    gfx_open( xsize, ysize, "Mandelbrot Set");
#pragma omp parallel for private(x,y) 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;
}
```

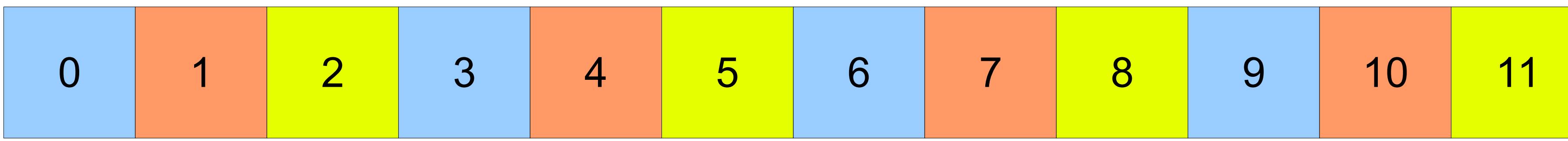
`schedule(type, chunksize)`

- `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

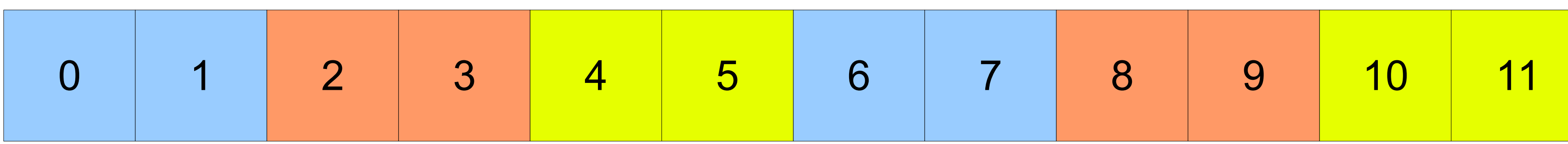
Example



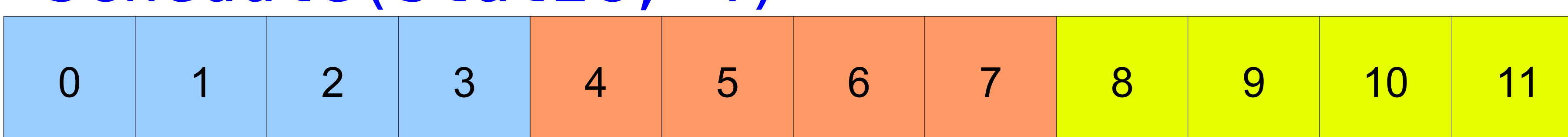
- Twelve iterations 0, 1, … 11 and three threads
- `schedule(static, 1)`



- `schedule(static, 2)`



- `schedule(static, 4)` Default 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
- *Q: Which considerations should we follow in order to choose between static and dynamic scheduling in OpenMP?*

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 private(x,y) collapse(2)
    for ( y = 0; y < ysize; y++ ) {
        for ( x = 0; x < xsize; x++ ) {
            drawpixel( x, y );
        }
    }
```

Taking times

```
double tstart, tend;
tstart = omp_wtime();
#pragma omp parallel for private(x,y) collapse(2)
for ( y = 0; y < ysize; y++ ) {
    for ( x = 0; x < xsize; x++ ) {
        drawpixel( x, y );
    }
}    Remember: implicit barrier here
tend = omp_wtime();
printf("Elapsed time: %f", tend-tstart);
```

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.

References

- An excellent tutorial on OpenMP is available at: https://computing.llnl.gov/tutorials/openMP/