… a bit of OpenMP – Part II

“In theory, there is no difference between theory and practice. In practice there is.”

Yogi Berra

Assigning Iterations to Threads

- The `schedule` clause of the `for` directive deals with the assignment of iterations to threads

- The general form of the `schedule` directive is

  `schedule(scheduling_class[, parameter])`

- OpenMP supports four scheduling classes: `static, dynamic, guided`, and `runtime`

Static vs. Dynamic Scheduling

- **Static scheduling**
  - Low overhead
  - May exhibit high workload imbalance
- **Dynamic scheduling**
  - Higher overhead
  - Can reduce workload imbalance

Chunks

- A chunk is a contiguous range of iterations

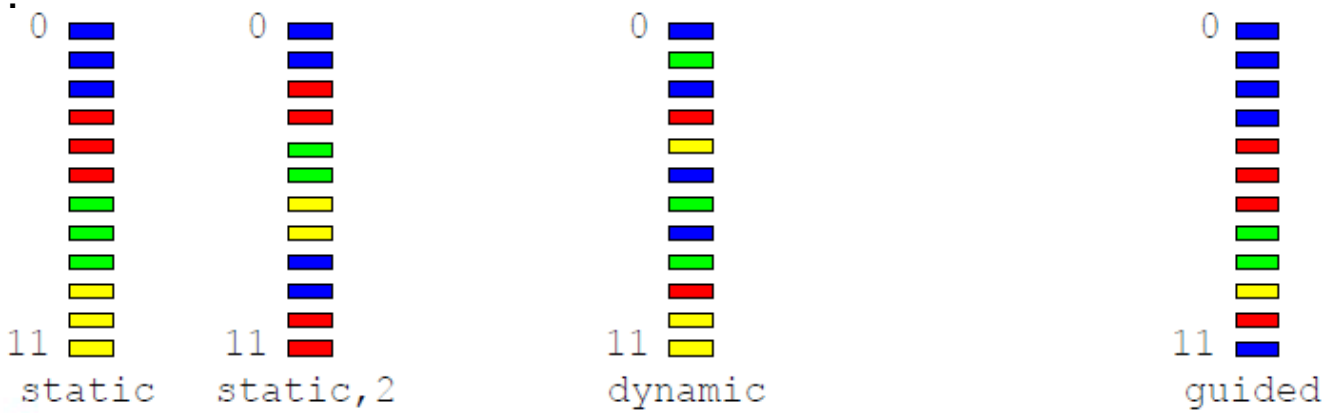
- **Increasing** the chunk size reduces overhead and may increase the cache hit rate (good!)
- **Decreasing** the chunk size allows finer balancing of workloads (good!)

`schedule static` clause

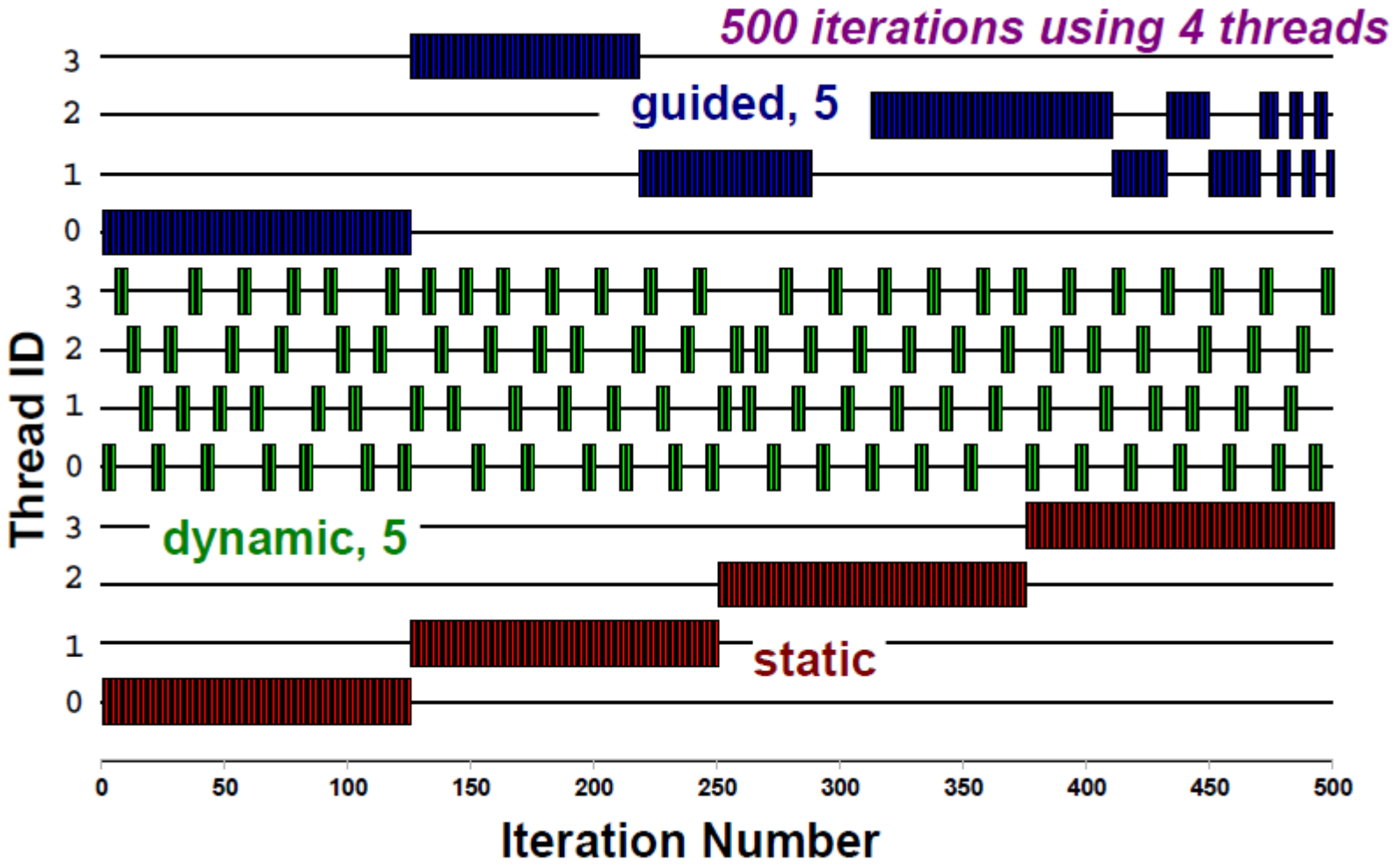
- `Schedule (static[,n])`
  - Group of **n** consecutive integers assigned once for all to threads on the basis of their id (**omp_get_thread_num**), cyclically until covering the #of total iterations
  - If **n** is not specified, iterations are divided in a group of n consecutive integer of **dimensions that are approximately equal** and assigned to threads on the basis of their id

schedule dynamic, guided and runtime clauses

- `Schedule (dynamic[,n])`
  - Set of n consecutive threads assigned to the fist thread that requires them, until all iterations are terminated
    - Each thread asks a bunch of threads and executes, asks, executes, etc
    - Partitioning is not determined a priori, depends on execution…
  - If n is not specified, value is 1
- `Schedule (guided[,n])`
  - As Dynamic, with increasing chunks
- `Schedule (runtime)`
  - The choice of the three available kinds is determined by environment variables …



Schedule : Example



When to use them?

Schedule Clause	When To Use
STATIC	Pre-determined and predictable by the programmer
DYNAMIC	Unpredictable, highly variable work per iteration
GUIDED	Special case of dynamic to reduce scheduling overhead

Least work at runtime : scheduling done at compile-time

Most work at runtime : complex scheduling logic used at run-time

Example

Dynamic Schedule (Sum of 2 arrays)

Change schedule!

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define CHUNKSIZE   10
#define N       100

int main (int argc, char *argv[])
{
int nthreads, tid, i, chunk;
float a[N], b[N], c[N];

/* Some initializations */
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;
chunk = CHUNKSIZE;

#pragma omp parallel shared(a,b,c,nthreads,chunk) private(i,tid)
  {
  tid = omp_get_thread_num();
  if (tid == 0)
    {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }
  printf("Thread %d starting...\n",tid);

  #pragma omp for schedule(dynamic,chunk)
  for (i=0; i<N; i++)
    {
    c[i] = a[i] + b[i];
    printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
    }

  }  /* end of parallel section */

}
```

Hint

- Always try difference schedulings, etc
- What you think will happen, usually doesn't
- And viceversa ☺
- Example: The **collapse** clause for omp parallel for:

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(j) collapse(2)
for (i = 0; i < 4; i++)
    for (j = 0; j < 100; j++)
```

The outer loop only has four iterations. **If you have more than four threads then some will be wasted.** But when you collapse the threads will distribute among 400 iterations which is likely to be much greater than the number of threads.

Critical Sections in OpenMP

- The **critical** directive is used to execute part of a code a thread at a time
- Protects the access to **shared variables**
- Does **<u>not</u>** provide a **implicit barrier** at the end
  - Each thread enters the “critical” code part
  - Executes the instructions
  - Once terminated this part, it continues without waiting other threads

Critical Sections in OpenMP

- If sum is a shared variable, this loop can not run in parallel

```
for (i=0; i < n; i++){
      .....
   sum += a[i];
      .....
}
```

- We must use a critical region for this:

```
for (i=0; i < n; i++){
      .....                    one at a time can proceed
   sum += a[i];
      .....                    next in line, please
}
```

Atomic construct

Critical: All threads execute the code, but only one at a time:

```
#pragma omp critical [(name)]
{<code-block>}
```

There is no implied barrier on entry or exit !

Atomic: Only the loads and stores are atomic:

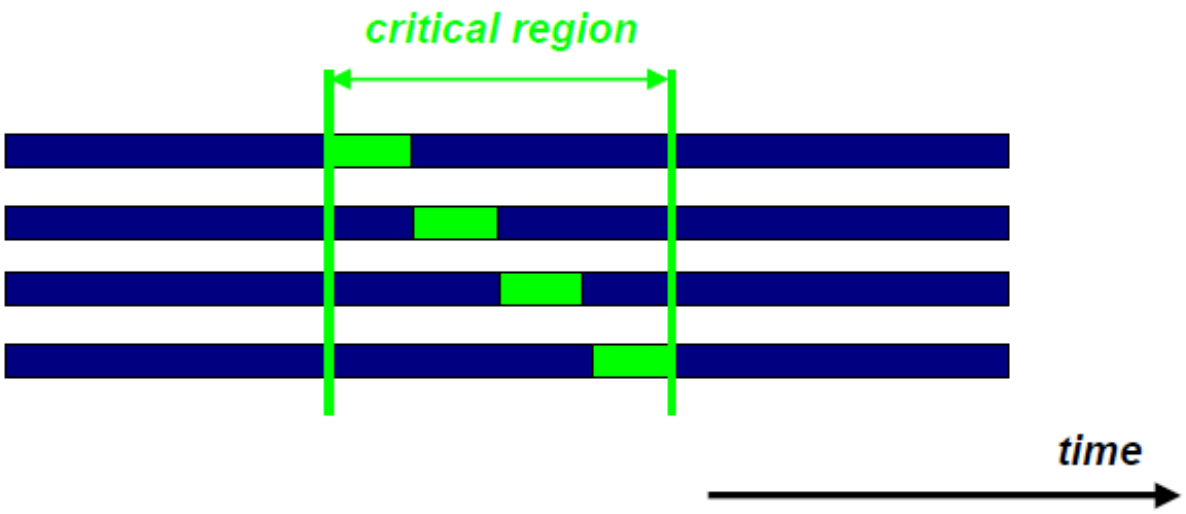
```
#pragma omp atomic
   <statement>
```

This is a lightweight, special form of a critical section

```
#pragma omp atomic
    a[indx[i]] += b[i];
```

Critical Sections in OpenMP

- Useful to avoid a race condition, or to perform I/O (but still has a random order)
- Be aware that there is a cost associated with a critical region



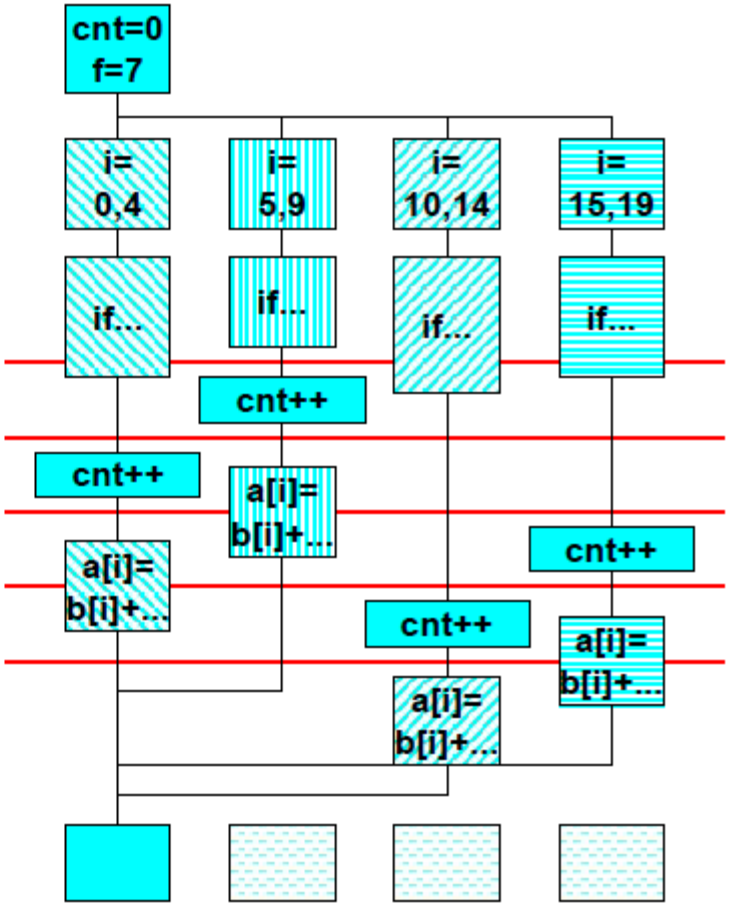
Critical Sections in OpenMP

C/C++:

`#pragma omp critical` [(*name*)] *new-line*
structured-block

A thread waits at the beginning of a critical region until no other thread in the team is executing a critical region with the same name. All unnamed `critical` directives map to the same unspecified name.

```
    cnt = 0;
    f=7;
#pragma omp parallel
{
#pragma omp for
    for (i=0; i<20; i++) {
      if (b[i] == 0) {

        #pragma omp critical
         cnt ++;

      } /* endif */
      a[i] = b[i] + f * (i+1);
    } /* end for */
} /*omp end parallel */
```



Synchronization: Atomic

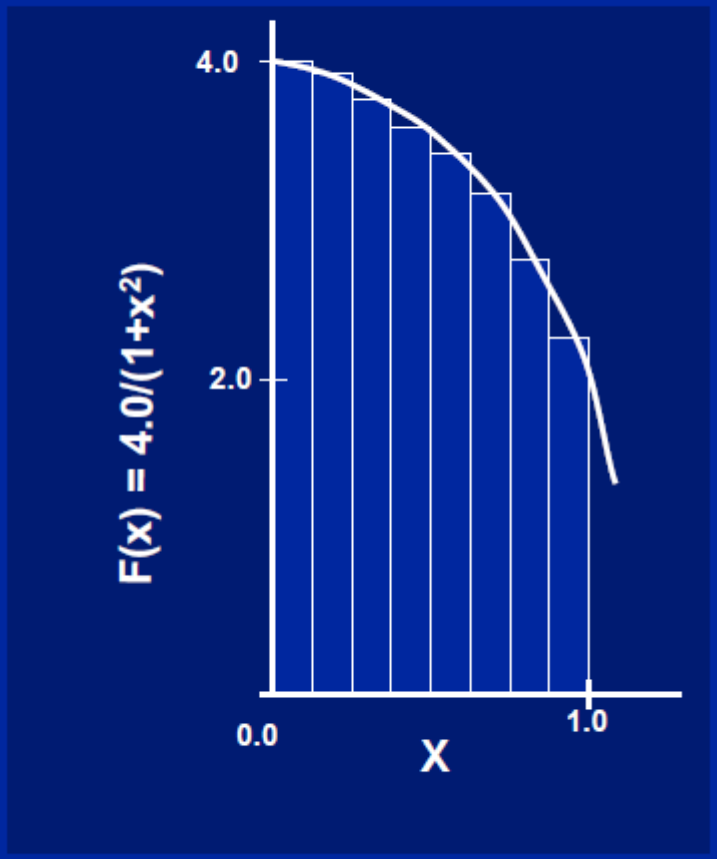
- Atomic provides mutual exclusion, but only applies to the update of a memory location (the update of X in the example)

```
#pragma omp parallel
{
        double tmp, B;
        B =  DOIT();
        tmp = big_ugly(B);
 #pragma omp atomic
          X +=  tmp;
}
```

Atomic only protects the read/update of X

π computation

Numerical Integration



Mathematically, we know that:

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

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

Serial program

```
static long num_steps = 100000;
double step;
int main ()
{       int i;   double x, pi, sum = 0.0;

        step = 1.0/(double) num_steps;

        for (i=0;i< num_steps; i++){
                x = (i+0.5)*step;
                sum = sum + 4.0/(1.0+x*x);
        }
        pi = step * sum;
}
```

Race Condition (cont.)

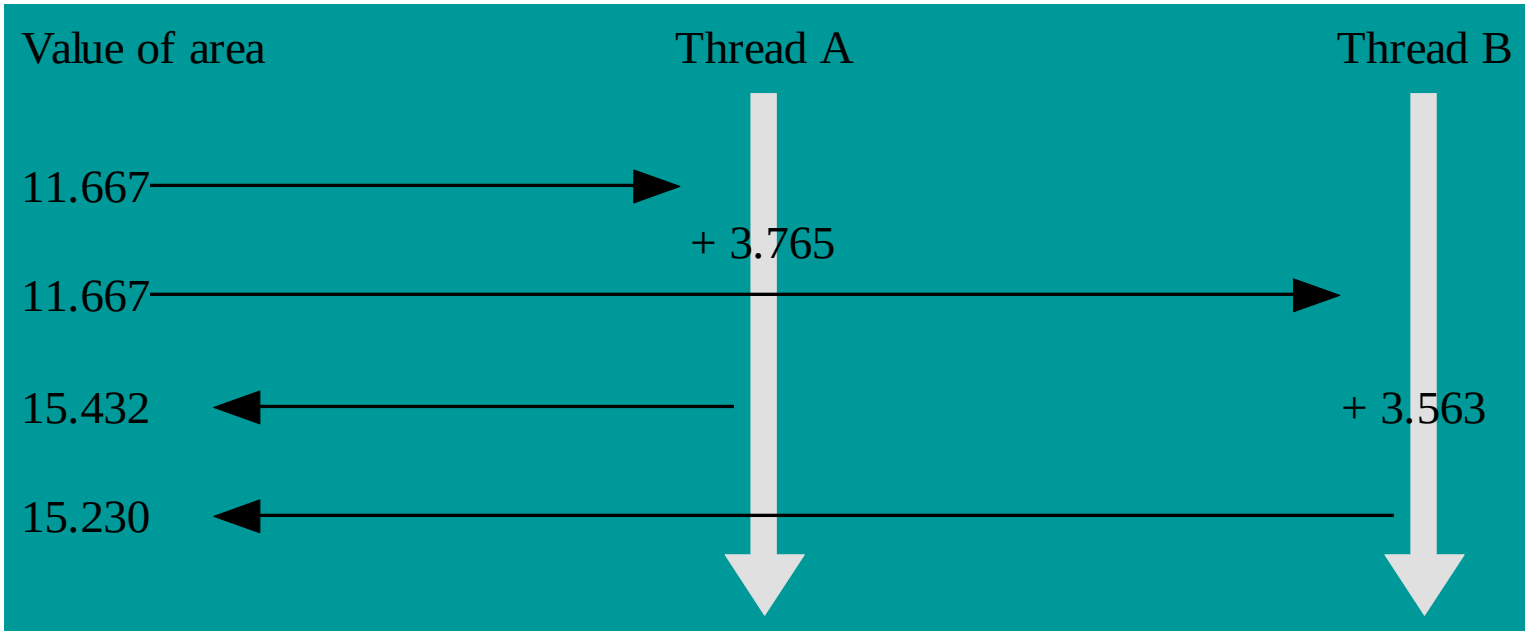
- ... we set up a **race condition** in which one process may "race ahead" of another and not see its change to shared variable **`sum`**

`sum` 15.230 **Answer should be 18.995**

Thread A 15.432 Thread B 15.230

```
sum += 4.0/(1.0 + x*x)
```

Race Condition Time Line



Example: A Simple Parallel pi Pogram

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{           int i, nthreads;  double pi, sum[NUM_THREADS];
            step = 1.0/(double) num_steps;
            omp_set_num_threads(NUM_THREADS);
   #pragma omp parallel
   {
          int i, id,nthrds;
          double x;
          id = omp_get_thread_num();
          nthrds = omp_get_num_threads();

          for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds) {
                x = (i+0.5)*step;
                sum[id] += 4.0/(1.0+x*x);
          }
   }
        for(i=0, pi=0.0; i<nthrds; i++) pi += sum[i] * step
}
```

Promote scalar to an array dimensioned by number of threads to avoid race condition.

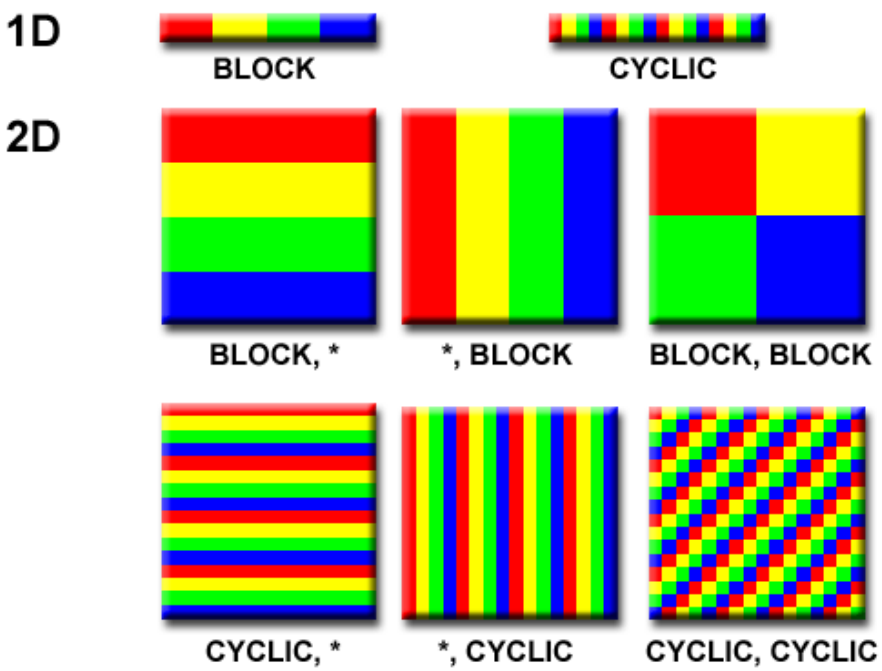
Cyclic distribution!

#pragma omp for?

This is a common trick in SPMD programs to create a cyclic distribution of loop iterations

Cyclic and Block Partioning

The previous for loop can be partitioned in cyclic or block manner



Esempio:

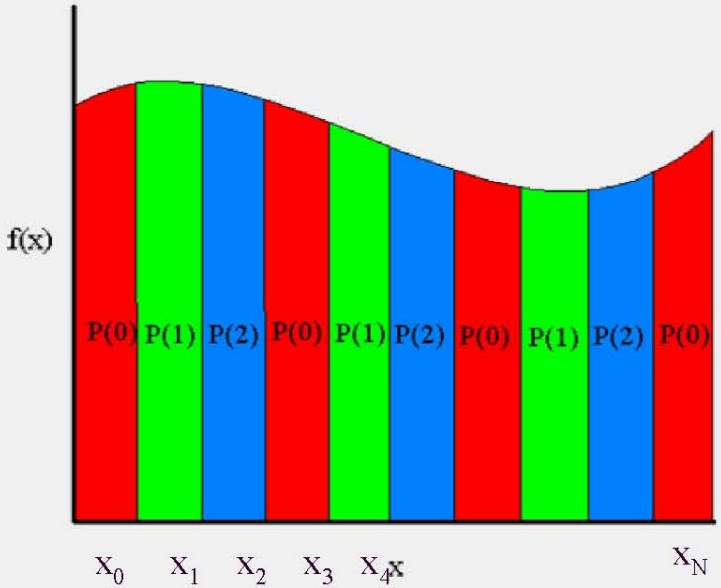
N = 10, h=0.1

Procrs: {P(0),P(1),P(2)}

P(0) -> {.05, .35, .65, .95}

P(1) -> {.15, .45, .75}

P(2) -> {.25, .55, .85}



Version 2 - Block

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{           int i, nthreads;  double pi, sum[NUM_THREADS];
            step = 1.0/(double) num_steps;
            omp_set_num_threads(NUM_THREADS);
   #pragma omp parallel
   {
          int i, id,nthrds;
          double x;
          id = omp_get_thread_num();
          nthrds = omp_get_num_threads();

          for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds) {
                x = (i+0.5)*step;
                sum[id] += 4.0/(1.0+x*x);
          }
   }
   for(i=0, pi=0.0; i<nthrds; i++) pi += sum[i] * step
}
```

Promote scalar to an array dimensioned by number of threads to avoid race condition.

Cyclic distribution!

#pragma omp for
mettendo
for(i=0; i<num_steps; i++)

SPMD: Single Program Multiple Data

- Run the same program on P processing elements (threads, here) where P can be arbitrary large
- Use the rank, an ID ranging from 0 to (P-1), to select between a set of tasks and to manage any shared data structures

This pattern is very general and has been used to support most (if not all) the algorithm strategy patterns.

MPI programs almost always use this pattern … it is probably the most commonly used pattern in the history of parallel programming.

gcc/g++ -O compiler optimizations

option	optimization level	execution time	code size	memory usage	compile time
-O0	optimization for compilation time (default)	+	+	-	-
-O1 or -O	optimization for code size and execution time	-	-	+	+
-O2	optimization more for code size and execution time	--		+	++
-O3	optimization more for code size and execution time	---		+	+++
-Os	optimization for code size		--		++
-Ofast	O3 with fast none accurate math calculations	---		+	+++

+increase ++ increase more +++increase even more
-reduce --reduce more ---reduce even more

gcc/g++ -O compiler optimizations

- By changing compiler optimizations, different execution times can be obtained for the **serial** version
- … but also for the **parallel** one!

- **Take different timings with different compiler options to evaluate speedups, etc…**

First version

- Original Serial pi program with 100000000 steps ran in 1.83 seconds.

Example: A simple Parallel pi program

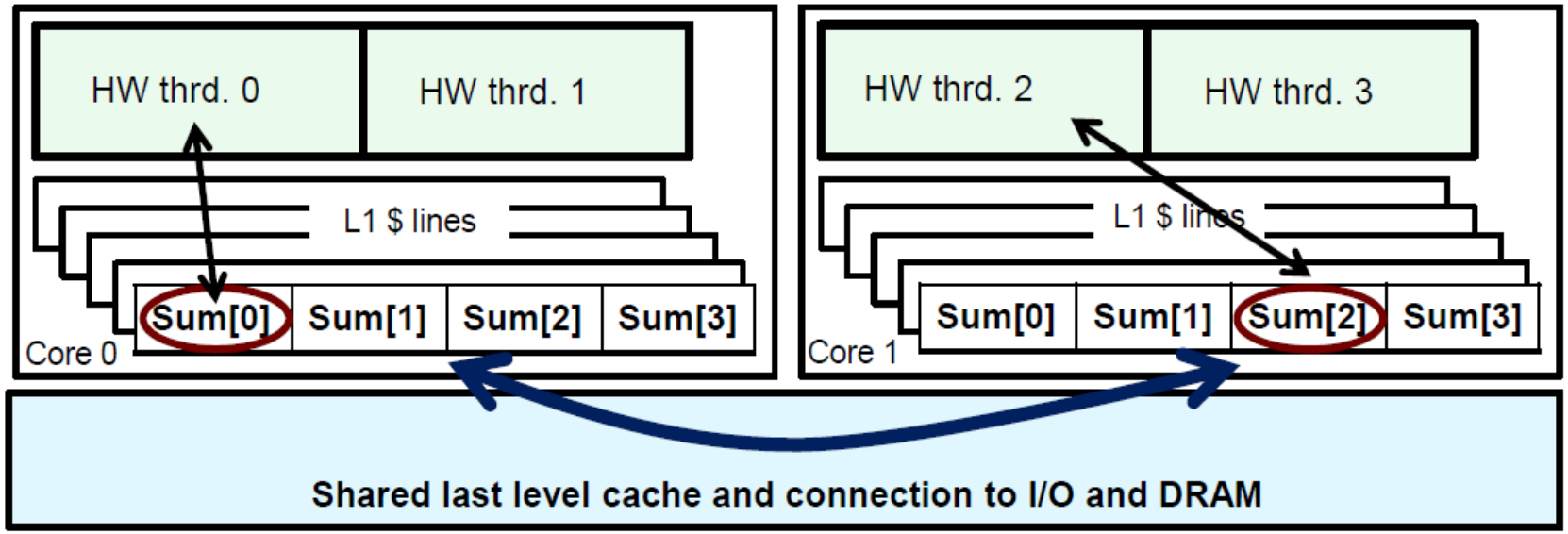
```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{           int i, nthreads;  double pi, sum[NUM_THREADS];
            step = 1.0/(double) num_steps;
            omp_set_num_threads(NUM_THREADS);
   #pragma omp parallel
   {
            int i, id,nthrds;
           double x;
           id = omp_get_thread_num();
           nthrds = omp_get_num_threads();
           if (id == 0)  nthreads = nthrds;
            for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds) {
                      x = (i+0.5)*step;
                      sum[id] += 4.0/(1.0+x*x);
            }
   }
            for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i] * step;
}
```

threads	1st SPMD
1	1.86
2	1.03
3	1.08
4	0.97

*Intel compiler (icpc) with no optimization on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.

Why such poor scaling? False sharing!

- If independent data elements happen to sit on the same cahe line, each update will cache lines to «slosh back and forth» between threads…. FALSE SHARING.



- If you promote scalars to an array to support creation of a SPDM program, the array elements are contiguous in memory and hence share cache lines… results in poor scalability
- Solution: Pad arrays so elements you see are on distinct cache lines

Padding

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define    PAD      8          // assume 64 byte L1 cache line size
#define NUM_THREADS 2
void main ()
{           int i, nthreads;  double pi, sum[NUM_THREADS][PAD];
            step = 1.0/(double) num_steps;
            omp_set_num_threads(NUM_THREADS);
   #pragma omp parallel
   {        int i, id,nthrds;
            double x;
            id = omp_get_thread_num();
            nthrds = omp_get_num_threads();

            for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds) {
                        x = (i+0.5)*step;
                        sum[id][0] += 4.0/(1.0+x*x);
            }
    }
            for(i=0, pi=0.0; i<nthrds; i++) pi += sum[i][0] * step
}
```

Anche 16 Byte!

Pad the array so each sum value is in a different cache line

Results

- Original Serial pi program with 100000000 steps ran in 1.83 seconds.

Example: eliminate False sharing by padding the sum array

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define   PAD    8          // assume 64 byte L1 cache line size
#define NUM_THREADS 2
void main ()
{           int i, nthreads;  double pi, sum[NUM_THREADS][PAD];
            step = 1.0/(double) num_steps;
            omp_set_num_threads(NUM_THREADS);
   #pragma omp parallel
   {
            int i, id,nthrds;
           double x;
           id = omp_get_thread_num();
           nthrds = omp_get_num_threads();
           if (id == 0)   nthreads = nthrds;
            for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds) {
                      x = (i+0.5)*step;
                      sum[id][0] += 4.0/(1.0+x*x);
            }
    }
            for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i][0] * step;
}
```

threads	1st SPMD	1st SPMD padded
1	1.86	1.86
2	1.03	1.01
3	1.08	0.69
4	0.97	0.53

Padding really needed?

- Padding arrays requires deep knowledge of cache architecture. Portability issues if program runs on different architecture (different cache lines sizes!)
- There has to be a better way to deal with false sharing…

Critical Sections to remove FS

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{            double  pi;          step = 1.0/(double) num_steps;
             omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
{
             int i, id,nthrds;    double x, sum;
            id = omp_get_thread_num();
            nthrds = omp_get_num_threads();
            if (id == 0)   nthreads = nthrds;


             for (i=id, sum=0.0;i< num_steps; i=i+nthreads){
                    x = (i+0.5)*step;
                    sum += 4.0/(1.0+x*x);
             }
           #pragma omp critical
                  pi += sum * step;
}
}
```

Create a scalar local to each thread to accumulate partial sums.

This time sum is private for each thread!

#pragma omp for?

No array, so no false sharing.

Sum goes "out of scope" beyond the parallel region … so you must sum it in here. Must protect summation into pi in a critical region so updates don't conflict

New results

- Original Serial pi program with 100000000 steps ran in 1.83 seconds.

Example: Using a critical section to remove impact of false sharing

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{           double  pi;          step = 1.0/(double) num_steps;
            omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
{
            int i, id,nthrds;    double x, sum;
           id = omp_get_thread_num();
           nthrds = omp_get_num_threads();
           if (id == 0)   nthreads = nthrds;
             id = omp_get_thread_num();
           nthrds = omp_get_num_threads();
            for (i=id, sum=0.0;i< num_steps; i=i+nthreads){
                      x = (i+0.5)*step;
                      sum += 4.0/(1.0+x*x);
            }
           #pragma omp critical
                  pi += sum * step;
}
}
```

threads	1st SPMD	1st SPMD padded	SPMD critical
1	1.86	1.86	1.87
2	1.03	1.01	1.00
3	1.08	0.69	0.68
4	0.97	0.53	0.53

*Intel compiler (icpc) with no optimization on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.

Careful!

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{            double  pi;          step = 1.0/(double) num_steps;
             omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
{
             int i, id,nthrds;    double x;
            id = omp_get_thread_num();
            nthrds = omp_get_num_threads();
            if (id == 0)   nthreads = nthrds;
              id = omp_get_thread_num();
            nthrds = omp_get_num_threads();
              for (i=id, sum=0.0;i< num_steps; i=i+nthreads){
                        x = (i+0.5)*step;
                       #pragma omp critical
                             pi += 4.0/(1.0+x*x);
              }
}
pi *= step;
}
```

Be careful where you put a critical section

What would happen if you put the critical section inside the loop?

Attenzione! Critical eseguito "troppe" volte!

Even `atomic` works!

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{           double  pi;          step = 1.0/(double) num_steps;
            omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
{
             int i, id,nthrds;    double x, sum;
           id = omp_get_thread_num();
           nthrds = omp_get_num_threads();
           if (id == 0)   nthreads = nthrds;
             id = omp_get_thread_num();
           nthrds = omp_get_num_threads();
             for (i=id, sum=0.0;i< num_steps; i=i+nthreads){
                        x = (i+0.5)*step;
                        sum += 4.0/(1.0+x*x);
             }
              sum = sum*step;
          #pragma atomic
                  pi += sum ;
}
```

Create a scalar local to each thread to accumulate partial sums.

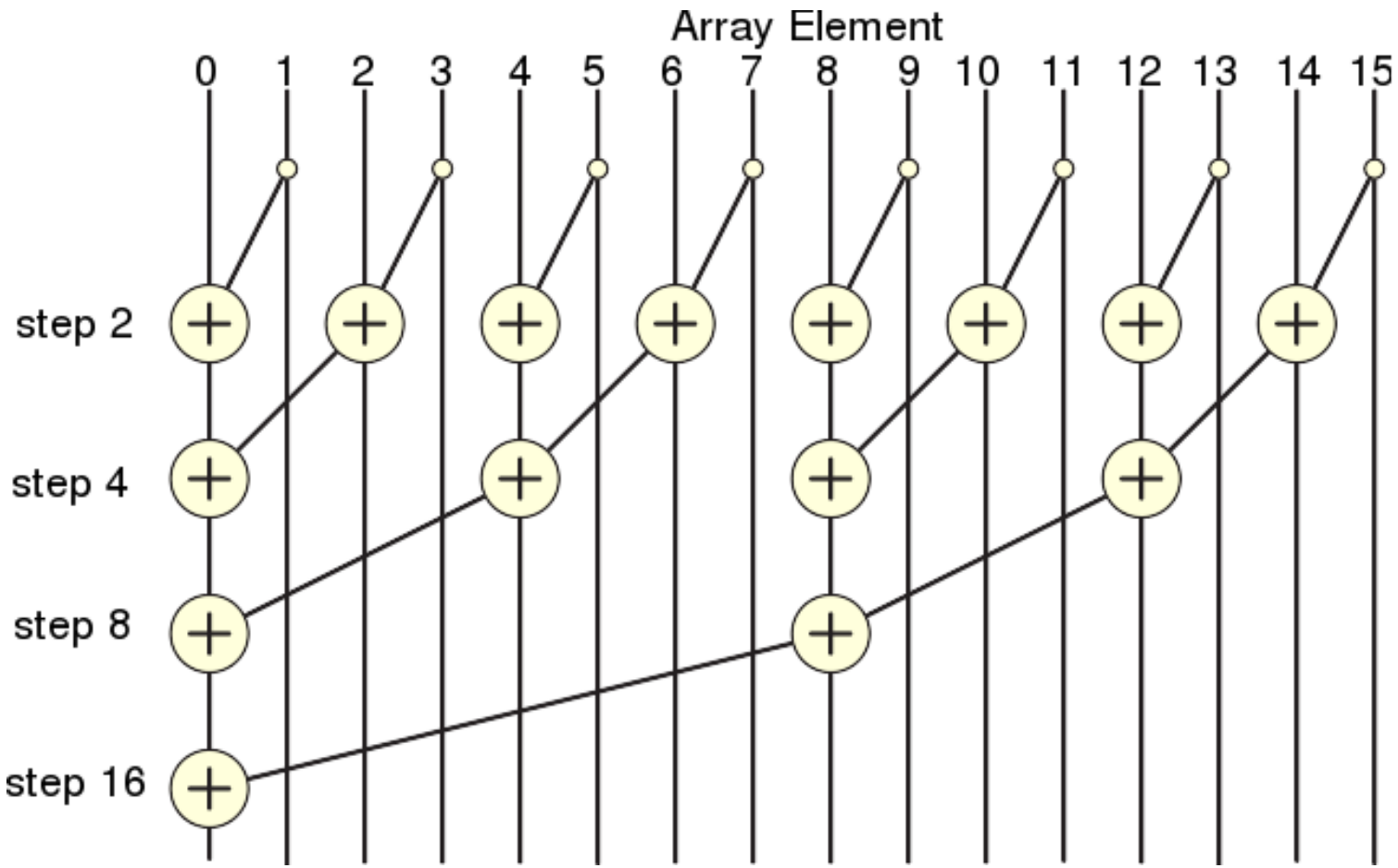
No array, so no false sharing.

Sum goes "out of scope" beyond the parallel region … so you must sum it in here. Must protect summation into pi so updates don't conflict

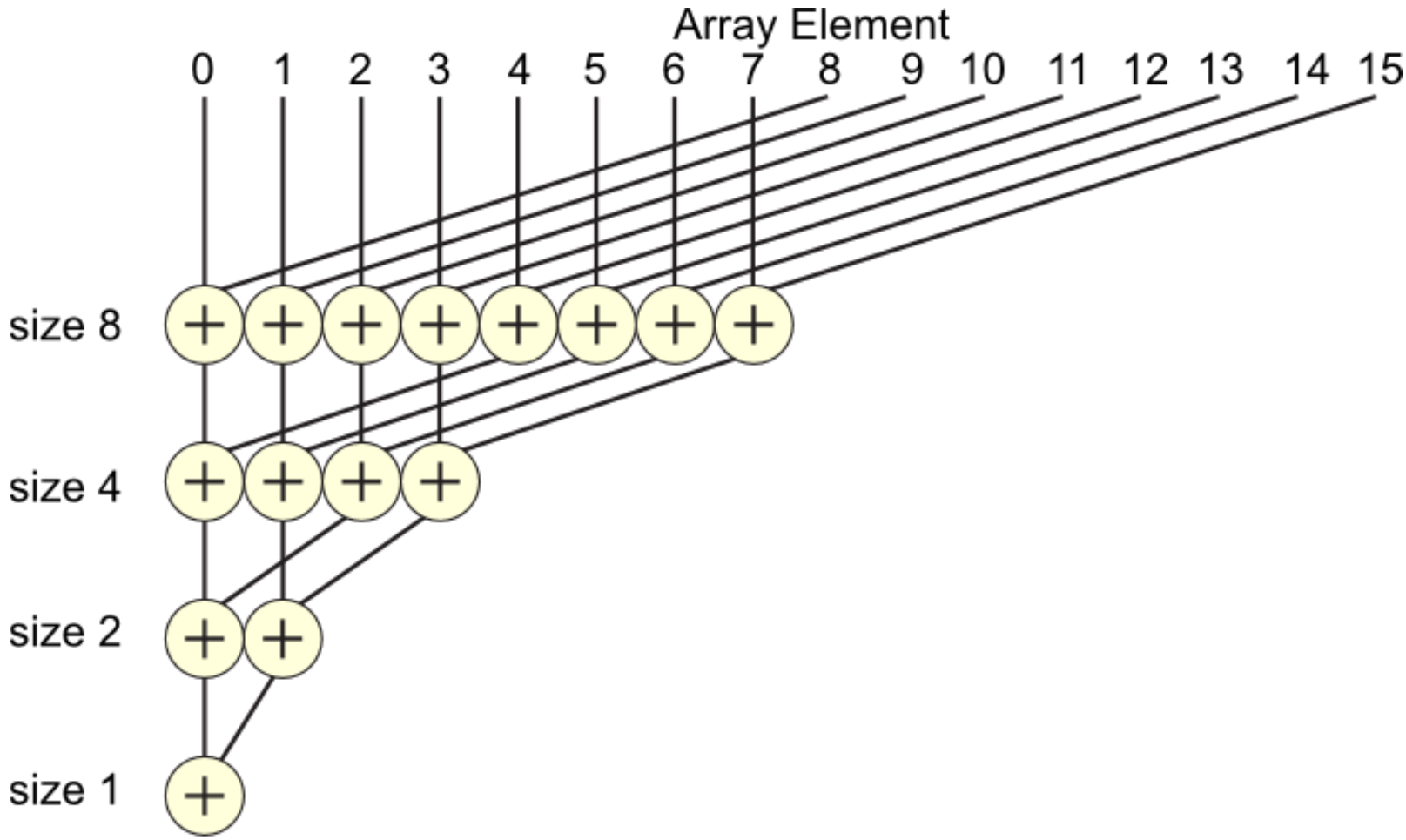
Reductions

- Reductions are so common that OpenMP provides support for them
- May add reduction clause to **`parallel for`** pragma
- Specify a **reduction operation** and **reduction variable**
- OpenMP takes care of storing partial results in private variables and combining partial results after the loop

Reductions


Array Element
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
step 2
step 4
step 8
step 16

Reductions


Array Element
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
size 8
size 4
size 2
size 1

`reduction` Clause

- The reduction clause has this syntax:
  `reduction (<op> :<variable>)`
- Operators
  - + Sum
  - * Product
  - & Bitwise and
  - | Bitwise or
  - ^ Bitwise exclusive or
  - && Logical and
  - || Logical or

`reduction` example

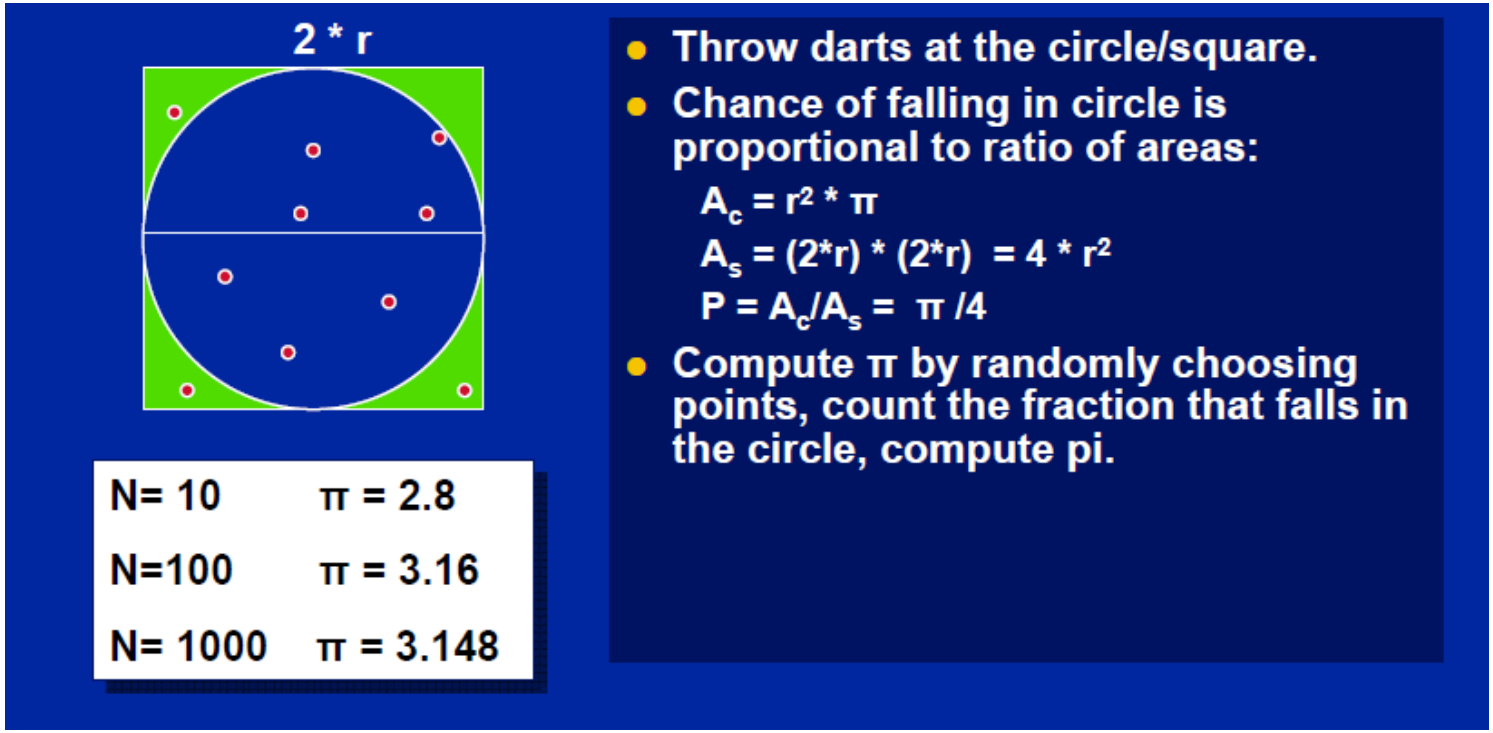
```
#include <omp.h>
#define NUM_THREADS 2
void main ()
{
    int i;
    double ZZ, func(), sum=0.0;

    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel for reduction(+:sum) private(ZZ)

    for (i=0; i< 1000; i++){
       ZZ = func(i);
       sum = sum + ZZ;
    }
}
```

π-finding Code with Reduction Clause

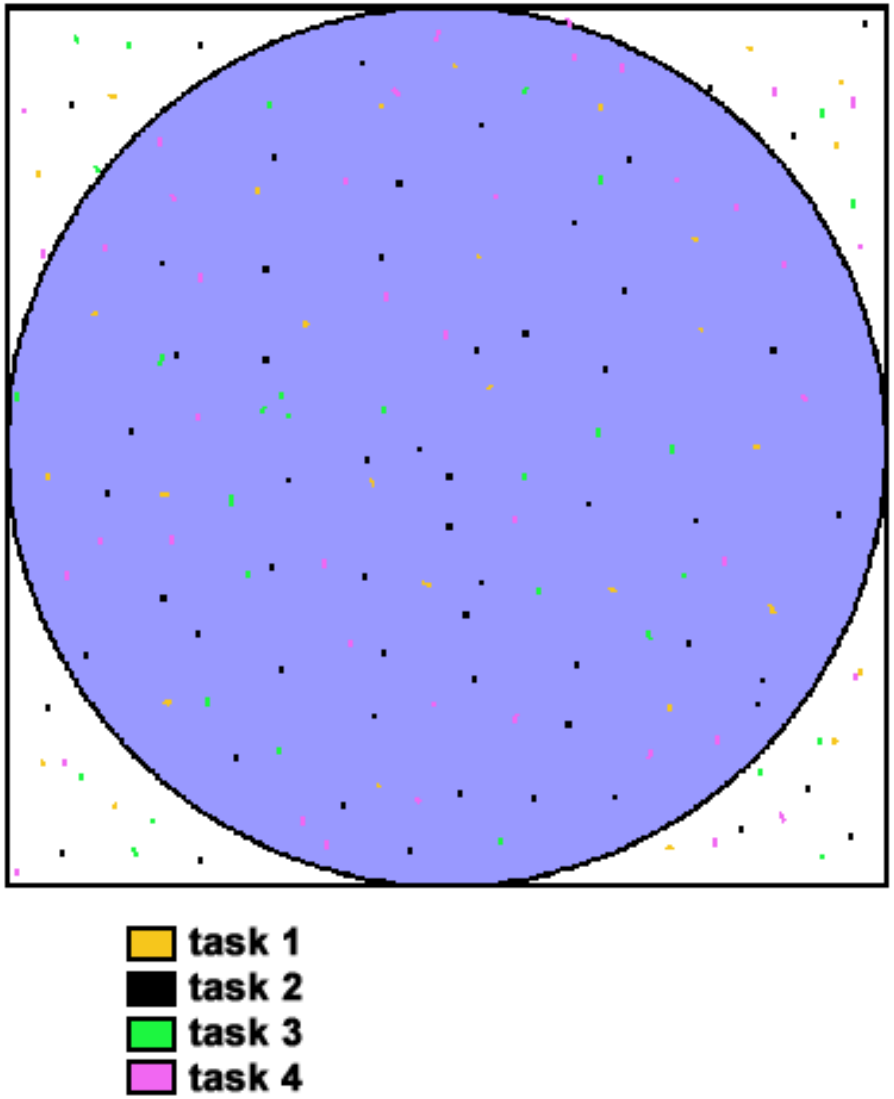
```
double area, pi, x;
int i, n;
...
area = 0.0;
#pragma omp parallel for \
       private(x) reduction(+:area)
for (i = 0; i < n; i++) {
   x = (i + 0.5)/n;
   area += 4.0/(1.0 + x*x);
}
pi = area / n;
```

… other method for computing π

- **Monte Carlo Calculations**: Using Random numbers to solve **<u>tough</u>** problems
  - Sample a problem domain to estimate areas, compute probabilities, find optimal values, etc
  - Example: Computing pi with a digital dart board:



Be careful to the random seed!



Use rand_r() in C, better than rand() !

Embarrassingly parallel

Embarrassingly parallel: the parallelism is so easy its embarrassing.

Add two lines and you have a parallel program.

```
#include "omp.h"
static long num_trials = 10000;
int main ()
{
  long i;     long Ncirc = 0;      double pi, x, y;
  double r = 1.0;   // radius of circle. Side of squrare is 2*r
  seed(0,-r, r);  // The circle and square are centered at the origin
  #pragma omp parallel for private (x, y) reduction (+:Ncirc)
  for(i=0;i<num_trials; i++)
  {
    x = random();        y = random();
    if ( x*x + y*y) <= r*r)   Ncirc++;
   }

   pi = 4.0 * ((double)Ncirc/(double)num_trials);
   printf("\n %d trials, pi is %f \n",num_trials, pi);
}
```

Use rand_r() in C, better than rand() !

Improving performances #1

In general:

- Too many fork/joins can lower performance
- Inverting loops may help performance if
  - Parallelism is in inner loop
  - After inversion, the outer loop can be made parallel
  - Inversion does not significantly lower cache hit rate

Improving performances #2

In general:

- If loop has too few iterations, fork/join overhead is greater than time savings from parallel execution
- The **`if`** clause instructs compiler to insert code that determines at run-time whether loop should be executed in parallel; e.g.,

```
#pragma omp parallel for if(n > 5000)
```

Improving performances #3

In general:

- We can use **`schedule`** clause to specify how iterations of a loop should be allocated to threads
- Static schedule: all iterations allocated to threads before any iterations executed
- Dynamic schedule: only some iterations allocated to threads at beginning of loop's execution. Remaining iterations allocated to threads that complete their assigned iterations.

Homework 2 ☺

- Hello world!
- Compute π
  - With critical section, padding, parallel for, etc
  - With Reduction
  - Monte Carlo
  - Schedule static, dynamic, etc (try, try, try!)
  - What you expect is probably not verified!
- Always get timings !!!

Addendum

The SPMD pattern

- The most common approach for parallel algorithms is the SPMD or Single Program Multiple Data pattern.
- Each thread runs the same program (Single Program), but using the thread ID, they operate on different data (Multiple Data) or take slightly different paths through the code.
- In OpenMP this means:
  - A parallel region "near the top of the code".
  - Pick up thread ID and num_threads.
  - Use them to split up loops and select different blocks of data to work on.

Exercise 2: A simple SPMD pi program

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{           int i, nthreads;  double pi, sum[NUM_THREADS];
            step = 1.0/(double) num_steps;
            omp_set_num_threads(NUM_THREADS);
   #pragma omp parallel
   {
           int i, id,nthrds;
           double x;
           id = omp_get_thread_num();
           nthrds = omp_get_num_threads();
           if (id == 0)   nthreads = nthrds;
           for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds) {
                    x = (i+0.5)*step;
                    sum[id] += 4.0/(1.0+x*x);
           }
    }
           for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i] * step;
}
```

Promote scalar to an array dimensioned by number of threads to avoid race condition.

Only one thread should copy the number of threads to the global value to make sure multiple threads writing to the same address don't conflict.

This is a common trick in SPMD programs to create a cyclic distribution of loop iterations

False sharing

- If independent data elements happen to sit on the same cache line, each update will cause the cache lines to "slosh back and forth" between threads.
  - This is called "false sharing".
- If you promote scalars to an array to support creation of an SPMD program, the array elements are contiguous in memory and hence share cache lines.
  - Result … poor scalability
- Solution:
  - When updates to an item are frequent, work with local copies of data instead of an array indexed by the thread ID.
  - Pad arrays so elements you use are on distinct cache lines.

Exercise 3: SPMD Pi without false sharing

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{           double  pi;        step = 1.0/(double) num_steps;
            omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
{
           int i, id,nthrds;    double x, sum;
            id = omp_get_thread_num();
            nthrds = omp_get_num_threads();
            if (id == 0)   nthreads = nthrds;
            id = omp_get_thread_num();
            nthrds = omp_get_num_threads();
            for (i=id, sum=0.0;i< num_steps; i=i+nthreads){
                      x = (i+0.5)*step;
                      sum += 4.0/(1.0+x*x);
            }
           #pragma omp critical
                  pi += sum * step;
}
}
```

Create a scalar local to each thread to accumulate partial sums.

No array, so no false sharing.

Sum goes "out of scope" beyond the parallel region … so you must sum it in here. Must protect summation into pi in a critical region so updates don't conflict

Code Examples

(to compile ...?)

```
gcc -fopenmp hello.c -o hello
```

… and

```
willy@willy-vb: export OMP_NUM_THREADS=4
```

Getting timings in Linux

```
minnie@sv: time ./a.out
real        0m0.083s
user        0m0.123s
sys         0m0.004s
```

`real =` wall clock time, that is the time from start to finish of the call

`user =` is the amount of CPU time spent in user-mode code (**outside the kernel**) *within* the process.

`sys =` is the amount of CPU time spent **in the kernel** within the process

1) user + sys gives the idea of the overall time taken by the process

2) If the process has multiple threads, (**user + sys**) could potentially exceed the wall clock time reported by **real**

Ciao Mondo
Hello World

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

int main (int argc, char *argv[])
{
int nthreads, tid;

/* Fork a team of threads giving them their own copies of variables */
#pragma omp parallel private(nthreads, tid)
  {

  /* Obtain thread number */
  tid = omp_get_thread_num();
  printf("Hello World from thread = %d\n", tid);

  /* Only master thread does this */
  if (tid == 0)
    {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }

  }  /* All threads join master thread and disband */

}
```

Get Info!

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

int main (int argc, char *argv[])
{
int nthreads, tid, procs, maxt, inpar, dynamic, nested;

/* Start parallel region */
#pragma omp parallel private(nthreads, tid)
  {

  /* Obtain thread number */
  tid = omp_get_thread_num();

  /* Only master thread does this */
  if (tid == 0)
    {
    printf("Thread %d getting environment info...\n", tid);

    /* Get environment information */
    procs = omp_get_num_procs();
    nthreads = omp_get_num_threads();
    maxt = omp_get_max_threads();
    inpar = omp_in_parallel();
    dynamic = omp_get_dynamic();
    nested = omp_get_nested();

    /* Print environment information */
    printf("Number of processors = %d\n", procs);
    printf("Number of threads = %d\n", nthreads);
    printf("Max threads = %d\n", maxt);
    printf("In parallel? = %d\n", inpar);
    printf("Dynamic threads enabled? = %d\n", dynamic);
    printf("Nested parallelism supported? = %d\n", nested);

    }

  }  /* Done */

}
```

Array sum

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define CHUNKSIZE   10
#define N       100

int main (int argc, char *argv[])
{
int nthreads, tid, i, chunk;
float a[N], b[N], c[N];

/* Some initializations */
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;
chunk = CHUNKSIZE;

#pragma omp parallel shared(a,b,c,nthreads,chunk) private(i,tid)
  {
  tid = omp_get_thread_num();
  if (tid == 0)
    {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }
  printf("Thread %d starting...\n",tid);

  #pragma omp for schedule(dynamic,chunk)
  for (i=0; i<N; i++)
    {
    c[i] = a[i] + b[i];
    printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
    }

  }  /* end of parallel section */

}
```

Dynamic Schedule … change!!!

Functional Decomposition

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define N     50

int main (int argc, char *argv[])
{
int i, nthreads, tid;
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;
  c[i] = d[i] = 0.0;
  }
```

```
#pragma omp parallel shared(a,b,c,d,nthreads) private(i,tid)
  {
  tid = omp_get_thread_num();
  if (tid == 0)
    {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }
  printf("Thread %d starting...\n",tid);

  #pragma omp sections nowait
    {
    #pragma omp section
      {
      printf("Thread %d doing section 1\n",tid);
      for (i=0; i<N; i++)
        {
        c[i] = a[i] + b[i];
        printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
        }
      }

    #pragma omp section
      {
      printf("Thread %d doing section 2\n",tid);
      for (i=0; i<N; i++)
        {
        d[i] = a[i] * b[i];
        printf("Thread %d: d[%d]= %f\n",tid,i,d[i]);
        }
      }

    }  /* end of sections */

    printf("Thread %d done.\n",tid);

  }  /* end of parallel section */

}
```

Critical Section: Scalar product

```c
/*
   OpenMP example program which computes the dot product of two arrays
   (that is sum(a[i]*b[i]) ) using explicit synchronization with a cri
   Compile with gcc -fopenmp omp_critical.c -o omp_critical
*/

#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

#define N 100

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

  double a[N], b[N];
  double localsum, sum = 0.0;
  int i, tid, nthreads;

#pragma omp parallel shared(a,b,sum) private(i, localsum)
  {
    /* Get thread number */
    tid = omp_get_thread_num();

    /* Only master thread does this */
    if (tid == 0) {
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d\n", nthreads);
    }

    /* Initialization */
#pragma omp for
    for (i=0; i < N; i++)
      a[i] = b[i] = (double)i;

    localsum = 0.0;

    /* Compute the local sums of all products */
#pragma omp for
    for (i=0; i < N; i++)
      localsum = localsum + (a[i] * b[i]);

#pragma omp critical
    sum = sum+localsum;

  }  /* End of parallel region */

  printf("   Sum = %2.1f\n",sum);
  exit(0);
}
```

```
/
* FILE: omp_reduction.c
* DESCRIPTION:
*   OpenMP Example - Combined Parallel Loop Reduction - C/C++ Version
*   This example demonstrates a sum reduction within a combined parallel loop
*   construct.  Notice that default data element scoping is assumed - there
*   are no clauses specifying shared or private variables.  OpenMP will
*   automatically make loop index variables private within team threads, and
*   global variables shared.
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
int   i, n;
float a[100], b[100], sum;

/* Some initializations */
n = 100;
for (i=0; i < n; i++)
  a[i] = b[i] = i * 1.0;
sum = 0.0;

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

printf("   Sum = %f\n",sum);

}
```

Reduction:
Scalar product

Quicksort! (base)

```
void QuickSort (int numList[],  int nLower, int nUpper)
   {
   if (nLower < nUpper)
      {
      // create partitions
      int nSplit = Partition (numList, nLower, nUpper);
      #pragma omp parallel sections
      {
         #pragma omp section
         QuickSort (numList, nLower, nSplit - 1);

         #pragma omp section
         QuickSort (numList, nSplit + 1, nUpper);
      }
   }
}
```

Homework 3 ☺

- Hello world!
- Compute π
  - With critical section, padding, parallel for, etc
  - With Reduction
  - Monte Carlo
  - Schedule static, dynamic, etc (try, try, try!)
- Always get timings !!!
- For the strong-hearted: Quicksort

Homework 4 ☺

- OpenMP implementation of the examples (Heat Equation? Try!)

- Conway's Game of Life!
  - Golly software
  - http://en.wikipedia.org/wiki/Conway's_Game_of_Life
  - https://www.youtube.com/watch?v=My8AsV7bA94
  - http://pmav.eu/stuff/javascript-game-of-life-v3.1.1/

- Try different schedulings and always get timings !!!