# PHY1610 - High Performance Scientific Computing with OpenMP, part 2

Ramses van Zon

March 24, 2022

## 1

## Reductions

## Dot Product

- Dot product of two vectors

$$n = \vec{x} \cdot \vec{y} = \sum_i x_i\, y_i$$

- Start from a serial implementation, then will add OpenMP

- Program tells answer, correct answer, time.

## Dot Product Code

```
// ndot_main.cc
#include <iostream>
#include <rarray>
#include "ticktock.h"
double ndot(const rarray<double,1>& x,
            const rarray<double,1>& y);
int main()
{
   int n = 20'000'000;
   rarray<double,1> x(n), y(n);
   for (int i=0; i<n; i++)
       x[i]=y[i]=i;
   double nn  = n;
   double ans = (nn-1)*nn*(2*nn-1)/6;
   TickTock tt;
   tt.tick();
   double dot = ndot(x,y);
   std::cout << "Dot product: " << dot << "\n"
             << "Exact answer: " << ans << "\n";
   tt.tock("Took");
}
```

```
// serial_ndot.cc
#include <rarray>
#include <algorithm>
double ndot(const rarray<double,1>& x,
            const rarray<double,1>& y)
{
   int n = std::min(x.size(), y.size());
   double tot=0;
   for (int i=0; i<n; i++)
      tot += x[i] * y[i];
   return tot;
}
```

# Dot Product Code

```
// ndot_main.cc
#include <iostream>
#include <rarray>
#include "ticktock.h"
double ndot(const rarray<double,1>& x,
            const rarray<double,1>& y);
int main()
{
   int n = 20'000'000;
   rarray<double,1> x(n), y(n);
   for (int i=0; i<n; i++)
       x[i]=y[i]=i;
   double nn  = n;
   double ans = (nn-1)*nn*(2*nn-1)/6;
   TickTock tt;
   tt.tick();
   double dot = ndot(x,y);
   std::cout << "Dot product: " << dot << "\n"
             << "Exact answer: " << ans << "\n";
   tt.tock("Took");
}
```

```
// serial_ndot.cc
#include <rarray>
#include <algorithm>
double ndot(const rarray<double,1>& x,
            const rarray<double,1>& y)
{
   int n = std::min(x.size(), y.size());
   double tot=0;
   for (int i=0; i<n; i++)
      tot += x[i] * y[i];
   return tot;
}
```

```
$ make serial_ndot
$ ./serial_ndot
Dot product:  2.66667e+21
Exact answer: 2.66667e+21
Took    0.1055 sec
$
```

# Towards A Parallel Dot Product

- We could clearly parallelize the loop.
- We could make `tot` shared, then all threads can add to it.

# Towards A Parallel Dot Product

- We could clearly parallelize the loop.
- We could make tot shared, then all threads can add to it.

```
// omp_ndot_race.cc
#include <rarray>
#include <algorithm>
double ndot(const rarray<double,1>& x,
            const rarray<double,1>& y) {
 int n = std::min(x.size(), y.size());
 double tot=0;
 #pragma omp parallel for default(none) shared(n,tot,x,y)
 for (int i=0; i<n; i++)
    tot += x[i] * y[i];
 return tot;
}
```

## Towards A Parallel Dot Product

- We could clearly parallelize the loop.
- We could make `tot` shared, then all threads can add to it.

```
// omp_ndot_race.cc
#include <rarray>
#include <algorithm>
double ndot(const rarray<double,1>& x,
            const rarray<double,1>& y) {
 int n = std::min(x.size(), y.size());
 double tot=0;
 #pragma omp parallel for default(none) shared(n,tot,x,y)
 for (int i=0; i<n; i++)
    tot += x[i] * y[i];
 return tot;
}
```

```
$ make omp_ndot_race
$ export OMP_NUM_THREADS=16
$ ./omp_ndot_race
Dot product:  2.64925e+20
Exact answer: 2.66667e+21
Took    0.5431 sec
$ ./omp_ndot_race
Dot product:  2.62621e+20
Exact answer: 2.66667e+21
Took    0.5383 sec
```

## Towards A Parallel Dot Product

- We could clearly parallelize the loop.
- We could make `tot` shared, then all threads can add to it.

```
// omp_ndot_race.cc
#include <rarray>
#include <algorithm>
double ndot(const rarray<double,1>& x,
            const rarray<double,1>& y) {
 int n = std::min(x.size(), y.size());
 double tot=0;
 #pragma omp parallel for default(none) shared(n,tot,x,y)
 for (int i=0; i<n; i++)
    tot += x[i] * y[i];
 return tot;
}
```

```
$ make omp_ndot_race
$ export OMP_NUM_THREADS=16
$ ./omp_ndot_race
Dot product:  2.64925e+20
Exact answer: 2.66667e+21
Took    0.5431 sec
$ ./omp_ndot_race
Dot product:  2.62621e+20
Exact answer: 2.66667e+21
Took    0.5383 sec
```

**Wrong answer!**

## Towards A Parallel Dot Product

- We could clearly parallelize the loop.
- We could make `tot` shared, then all threads can add to it.

```
// omp_ndot_race.cc
#include <rarray>
#include <algorithm>
double ndot(const rarray<double,1>& x,
            const rarray<double,1>& y) {
 int n = std::min(x.size(), y.size());
 double tot=0;
 #pragma omp parallel for default(none) shared(n,tot,x,y)
 for (int i=0; i<n; i++)
    tot += x[i] * y[i];
 return tot;
}
```

```
$ make omp_ndot_race
$ export OMP_NUM_THREADS=16
$ ./omp_ndot_race
Dot product:  2.64925e+20
Exact answer: 2.66667e+21
Took    0.5431 sec
$ ./omp_ndot_race
Dot product:  2.62621e+20
Exact answer: 2.66667e+21
Took    0.5383 sec
```

**Wrong answer!**

**Answer varies!**

## Towards A Parallel Dot Product

- We could clearly parallelize the loop.
- We could make tot shared, then all threads can add to it.

```
// omp_ndot_race.cc
#include <rarray>
#include <algorithm>
double ndot(const rarray<double,1>& x,
            const rarray<double,1>& y) {
 int n = std::min(x.size(), y.size());
 double tot=0;
 #pragma omp parallel for default(none) shared(n,tot,x,y)
 for (int i=0; i<n; i++)
    tot += x[i] * y[i];
 return tot;
}
```

```
$ make omp_ndot_race
$ export OMP_NUM_THREADS=16
$ ./omp_ndot_race
Dot product:  2.64925e+20
Exact answer: 2.66667e+21
Took    0.5431 sec
$ ./omp_ndot_race
Dot product:  2.62621e+20
Exact answer: 2.66667e+21
Took    0.5383 sec
```

**Wrong answer!**

**Answer varies!**

**Slower computation!**

## Our very first race condition!

- Can be very subtle, and only appear intermittently.
- Your program can have a bug but not display any symptoms for small runs!
- Primarily a problem with shared memory.
- Classical parallel bug.
- Multiple writers to some shared resource.

## Race Condition Example

Say, initially, `tot=0`, and one threads want to add 1 to it while a second thread want to add 2 at the same time.

- The correct answer for `tot` is, clearly, three.
- However, we may see any of the answers 1, 2, or 3.

How does this issue arise?

## Race Condition Example

Say, initially, `tot=0`, and one threads want to add 1 to it while a second thread want to add 2 at the same time.

- The correct answer for `tot` is, clearly, three.
- However, we may see any of the answers 1, 2, or 3.

How does this issue arise?

**Non-atomic adding and updating**

## Race Condition Example

Say, initially, `tot=0`, and one threads want to add 1 to it while a second thread want to add 2 at the same time.

- The correct answer for `tot` is, clearly, three.
- However, we may see any of the answers 1, 2, or 3.

How does this issue arise?

**Non-atomic adding and updating**

| Thread 0: add 1 | Thread 1: add 2 |
|---|---|

## Race Condition Example

Say, initially, `tot=0`, and one threads want to add 1 to it while a second thread want to add 2 at the same time.

- The correct answer for `tot` is, clearly, three.
- However, we may see any of the answers 1, 2, or 3.

How does this issue arise?

**Non-atomic adding and updating**

| Thread 0: add 1 | Thread 1: add 2 |
|---|---|
| read tot=0 to reg0 | . |

# Race Condition Example

Say, initially, `tot=0`, and one threads want to add 1 to it while a second thread want to add 2 at the same time.

- The correct answer for `tot` is, clearly, three.
- However, we may see any of the answers 1, 2, or 3.

How does this issue arise?

**Non-atomic adding and updating**

| Thread 0: add 1 | Thread 1: add 2 |
|---|---|
| read tot=0 to reg0 | . |
| reg0 = reg0+1 | read tot=0 to reg1 |

## Race Condition Example

Say, initially, `tot=0`, and one threads want to add 1 to it while a second thread want to add 2 at the same time.

- The correct answer for `tot` is, clearly, three.
- However, we may see any of the answers 1, 2, or 3.

How does this issue arise?

**Non-atomic adding and updating**

| Thread 0: add 1 | Thread 1: add 2 |
|---|---|
| read tot=0 to reg0 | . |
| reg0 = reg0+1 | read tot=0 to reg1 |
| store reg0(=1) in tot | reg1 = reg1 + 2 |

## Race Condition Example

Say, initially, `tot=0`, and one threads want to add 1 to it while a second thread want to add 2 at the same time.

- The correct answer for `tot` is, clearly, three.
- However, we may see any of the answers 1, 2, or 3.

How does this issue arise?

**Non-atomic adding and updating**

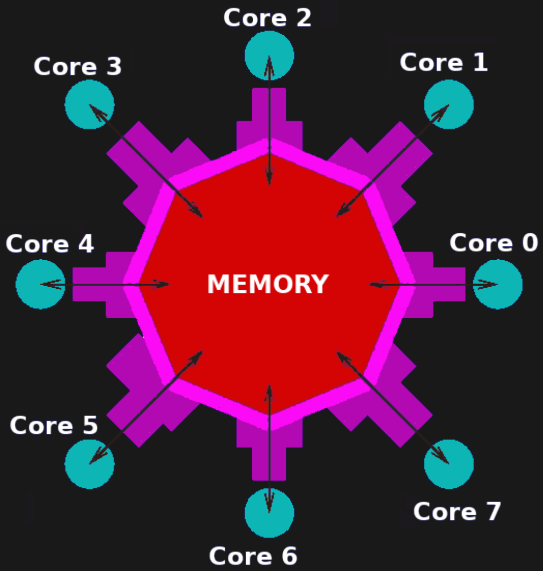
| Thread 0: add 1 | Thread 1: add 2 |
|---|---|
| read tot=0 to reg0 | . |
| reg0 = reg0+1 | read tot=0 to reg1 |
| store reg0(=1) in tot | reg1 = reg1 + 2 |
| . | store reg1(=2) in tot |

## So it's wrong, but why is it slower?

You might thing the parallel version should at least still be faster, though it may be wrong. But even that's not the case.

- Here, multiple cores repeatedly try to read, access and store the same variable in memory.
- This means the shared variable that is updated in a register, cannot stay in register: It has to be copied back to main memory, so the other threads see it correctly.
- The other threads then have to re-read the variable.
- This write-back would not be necessary if the variable was shared but not written to.

# Memory hierarchy



- Memory is layered: between registers and shared main memory there are further layers called **caches**.
- Caches are faster but more expensive and therefore smaller. They are like private memory for each core.
- Main memory is the slowest part of the memory.
- Caches are automatically kept coherent between cores.

2

## Fixing the race condition

## OpenMP critical construct

Our code get it wrong because different threads are updating the `tot` variable at the same time.

The `critical` construct:

- Defines a critical region.
- Only one thread can be operating within this region at a time.
- Keeps modifications to shared resources safe.

## OpenMP critical construct

Our code get it wrong because different threads are updating the tot variable at the same time.

The critical construct:

- Defines a critical region.
- Only one thread can be operating within this region at a time.
- Keeps modifications to shared resources safe.

```
// omp_ndot_critical.cc
#include <rarray>
#include <algorithm>
double ndot(const rarray<double,1>& x,
            const rarray<double,1>& y)
{
   int n = std::min(x.size(), y.size());
   double tot=0;
   #pragma omp parallel for default(none) shared(n,tot,x,y)
   for (int i=0; i<n; i++)
      #pragma omp critical
      tot += x[i] * y[i];
   return tot;
}
```

## Critical Construct Timing

```
// omp_ndot_critical.cc
#include <rarray>
#include <algorithm>
double ndot(const rarray<double,1>& x,
            const rarray<double,1>& y)
{
   int n = std::min(x.size(), y.size());
   double tot=0;
   #pragma omp parallel for default(none) shared(n,tot,x,y)
   for (int i=0; i<n; i++)
      #pragma omp critical
      tot += x[i] * y[i];
   return tot;
}
```

# Critical Construct Timing

```
// omp_ndot_critical.cc
#include <rarray>
#include <algorithm>
double ndot(const rarray<double,1>& x,
            const rarray<double,1>& y)
{
   int n = std::min(x.size(), y.size());
   double tot=0;
   #pragma omp parallel for default(none) shared(n,tot,x,y)
   for (int i=0; i<n; i++)
      #pragma omp critical
      tot += x[i] * y[i];
   return tot;
}
```

```
$ make omp_ndot_critical
$ export OMP_NUM_THREADS=16
$ ./omp_ndot_critical
Dot product:  2.66667e+21
Exact answer: 2.66667e+21
Took 4.6697 sec
```

**Correct, but 44× slower than serial version!**

## OpenMP atomic construct

- Most hardware has support for atomic instructions (indivisible so cannot get interrupted)
- Small subset, but load/add/store usually in it.
- Not as general as critical
- Much lower overhead.
- `#pragma omp atomic [read|write|update|capture]`

```
// omp_ndot_atomic.cc
#include <rarray>
#include <algorithm>
double ndot(const rarray<double,1>& x,
            const rarray<double,1>& y)
{
   int n = std::min(x.size(), y.size());
   double tot=0;
   #pragma omp parallel for default(none) shared(n,tot,x,y)
   for (int i=0; i<n; i++)
      #pragma omp atomic update
      tot += x[i] * y[i];
   return tot;
}
```

## Atomic Construct Timing

```
// omp_ndot_atomic.cc
#include <rarray>
#include <algorithm>
double ndot(const rarray<double,1>& x,
            const rarray<double,1>& y)
{
   int n = std::min(x.size(), y.size());
   double tot=0;
   #pragma omp parallel for default(none) shared(n,tot,x,y)
   for (int i=0; i<n; i++)
      #pragma omp atomic update
      tot += x[i] * y[i];
   return tot;
}
```

## Atomic Construct Timing

```
// omp_ndot_atomic.cc
#include <rarray>
#include <algorithm>
double ndot(const rarray<double,1>& x,
            const rarray<double,1>& y)
{
   int n = std::min(x.size(), y.size());
   double tot=0;
   #pragma omp parallel for default(none) shared(n,tot,x,y)
   for (int i=0; i<n; i++)
      #pragma omp atomic update
      tot += x[i] * y[i];
   return tot;
}
```

```
$ make omp_ndot_atomic
$ export OMP_NUM_THREADS=16
$ ./omp_ndot_atomic
Dot product:  2.66667e+21
Exact answer: 2.66667e+21
Took 2.177 sec
```

About twice faster than critical, but still not great.

## Local Sums

The issue we have not resolved is that we're still updating tot, which causes copies to main memory at every iteration.

What if we accumulated tot for each core, and sum them up later?

```
double ndot(const rarray<double,1>& x,
            const rarray<double,1>& y)
{
 int n = std::min(x.size(), y.size());
 double tot=0;
 #pragma omp parallel default(none) shared(n,tot,x,y)
 {
   double localtot=0;
   #pragma omp for
   for (int i=0; i<n; i++)
     localtot += x[i] * y[i];
   #pragma omp atomic update
   tot += localtot;
 }
 return tot;
}
```

## Local Sums

The issue we have not resolved is that we're still updating `tot`, which causes copies to main memory at every iteration.

What if we accumulated `tot` for each core, and sum them up later?

```
double ndot(const rarray<double,1>& x,
            const rarray<double,1>& y)
{
 int n = std::min(x.size(), y.size());
 double tot=0;
 #pragma omp parallel default(none) shared(n,tot,x,y)
 {
   double localtot=0;
   #pragma omp for
   for (int i=0; i<n; i++)
     localtot += x[i] * y[i];
   #pragma omp atomic update
   tot += localtot;
 }
 return tot;
}
```

```
$ export OMP_NUM_THREADS=16
$ ./omp_ndot_local
Dot product:  2.66667e+21
Exact answer: 2.66667e+21
Took 0.01715 sec
```

Correct answer, 6x faster!

## OpenMP Reduction Operations

- What we did is quite common, taking a bunch of data and summing it to one value: **reduction**

## OpenMP Reduction Operations

- What we did is quite common, taking a bunch of data and summing it to one value: **reduction**
- OpenMP supports this using **reduction variables**.

## OpenMP Reduction Operations

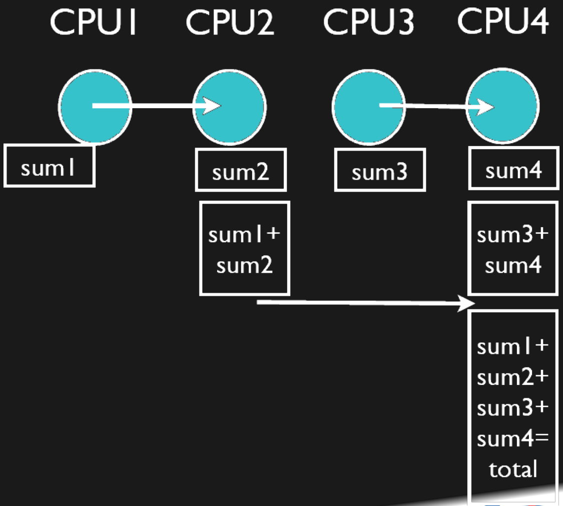
- What we did is quite common, taking a bunch of data and summing it to one value: **reduction**
- OpenMP supports this using **reduction variables**.
- When declaring a variables as reduction variables, private copies are made (much as for private variables), which are combined at the end of a parallel region through some operation (+, *, min, max).

## OpenMP Reduction Operations

- What we did is quite common, taking a bunch of data and summing it to one value: **reduction**
- OpenMP supports this using **reduction variables**.
- When declaring a variables as reduction variables, private copies are made (much as for private variables), which are combined at the end of a parallel region through some operation (+, *, min, max).
- `omp_ndot_reduction.cc`

## OpenMP Reduction Operations

- What we did is quite common, taking a bunch of data and summing it to one value: **reduction**
- OpenMP supports this using **reduction variables**.
- When declaring a variables as reduction variables, private copies are made (much as for private variables), which are combined at the end of a parallel region through some operation (+, *, min, max).
- `omp_ndot_reduction.cc`



## Reduction Timing

```
// omp_ndot_reduction.cc
#include <rarray>
#include <algorithm>
double ndot(const rarray<double,1>& x,
            const rarray<double,1>& y)
{
  int n = std::min(x.size(), y.size());
  double tot=0;
  #pragma omp for default(none) shared(n,x,y) reduction(+:tot)
  for (int i=0; i<n; i++)
    tot += x[i] * y[i];
  return tot;
}
```

## Reduction Timing

```
// omp_ndot_reduction.cc
#include <rarray>
#include <algorithm>
double ndot(const rarray<double,1>& x,
            const rarray<double,1>& y)
{
  int n = std::min(x.size(), y.size());
  double tot=0;
  #pragma omp for default(none) shared(n,x,y) reduction(+:tot)
  for (int i=0; i<n; i++)
    tot += x[i] * y[i];
  return tot;
}
```

```
$ make omp_ndot_reduction
$ export OMP_NUM_THREADS=8
$ ./omp_ndot_reduction
Dot product:  2.66667e+21
Exact answer: 2.66667e+21
Took 0.01691 sec
$
```

## Reduction Timing

```cpp
// omp_ndot_reduction.cc
#include <rarray>
#include <algorithm>
double ndot(const rarray<double,1>& x,
            const rarray<double,1>& y)
{
  int n = std::min(x.size(), y.size());
  double tot=0;
  #pragma omp for default(none) shared(n,x,y) reduction(+:tot)
  for (int i=0; i<n; i++)
    tot += x[i] * y[i];
  return tot;
}
```

```
$ make omp_ndot_reduction
$ export OMP_NUM_THREADS=8
$ ./omp_ndot_reduction
Dot product:  2.66667e+21
Exact answer: 2.66667e+21
Took 0.01691 sec
$
```

**Correct, same timing as local sums, but simpler code.**

3

## Load Balancing

# Load Balancing in OpenMP

- So far every iteration of the loop had the same amount of work.
- Not always the case.
- Sometimes cannot predict beforehand how unbalanced the problem is

OpenMP has work sharing constructs that allow you do statically or dynamically balance the load.

## Example: Mandelbrot Set

- Let $a$ be a parameter in the quadratic map:

$$b_{n+1} = b_n^2 + a$$

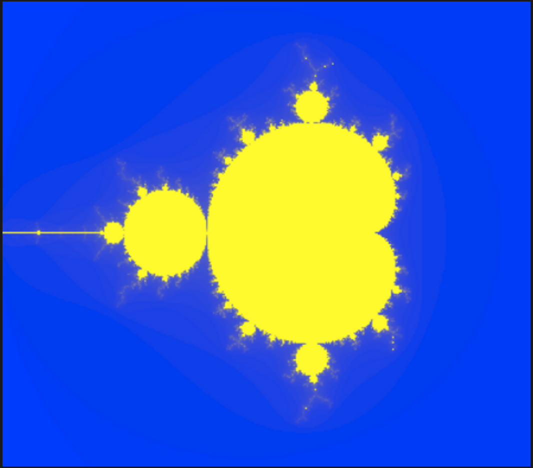
  Depending on $a$, points $b$ can escape to infinite, or not, as $n \to \infty$.

- The mandelbrot set is the boundary between the set of $a$ values which allow $b_n$ to escape, and the set of values that do not.
- Note: if $\|b_n\| > 2$, $b$ will escape.

Calculation:

- Iterate for each point $a$ in square, starting from $b_0 = 0$, and see if $\|b_n\| > 2$.
- if `n<nmax`, then blue, else yellow.

On the outside points diverge quickly.
Inside points: lots of work.



## Mandelbrot Code Overview

```
// iterations for each point
int how_many_iter(std::complex<double> a, int maxiter);
// compute iterations for each point in a rectangle
rmatrix<int> make_mandel_map(double xmin, double xmax, double ymin,
                             double ymax, int npix, int maxiter)
// display specific stuff
char display_map(const rmatrix<int>&,float,double,double&,double&,double&,double&);
void my_pgctab(float,float,float,float,float,float,int);
// driver routine
int main();
```

Compile and run:

```
$ make mandel mandel-parallel
$ ./mandel
     5.06 sec
...
$ export OMP_NUM_THREADS=16
$ ./mandel-parallel
     1.366 sec
```

## Computationally most demanding functions

```
rmatrix<int> make_mandel_map(double xmin, double xmax,
                             double ymin, double ymax,
                             int npix, int maxiter) {
  rmatrix<int> mymap(npix,npix);

  for (int i=0; i<npix; i++)
    for (int j=0; j<npix; j++) {
      double x = ((double)i)/((double)npix)*(xmax-xmin)+xmin;
      double y = ((double)j)/((double)npix)*(ymax-ymin)+ymin;
      std::complex<double> a(x,y);
      mymap[i][j] = how_many_iter(a,maxiter);
    }
  return mymap;
}
```

## Computationally most demanding functions

```
rmatrix<int> make_mandel_map(double xmin, double xmax,
                             double ymin, double ymax,
                             int npix, int maxiter) {
  rmatrix<int> mymap(npix,npix);

  for (int i=0; i<npix; i++)
    for (int j=0; j<npix; j++) {
      double x = ((double)i)/((double)npix)*(xmax-xmin)+xmin;
      double y = ((double)j)/((double)npix)*(ymax-ymin)+ymin;
      std::complex<double> a(x,y);
      mymap[i][j] = how_many_iter(a,maxiter);
    }
  return mymap;
}
```

```
int how_many_iter(std::complex<double> a, int maxiter) {
   std::complex<double> b = a;
   for (int i=0; i<maxiter; i++) {
      if (std::norm(b) > 4) return i;
      b = b*b + a;
   }
   return maxiter;
}
```

## Computationally most demanding functions

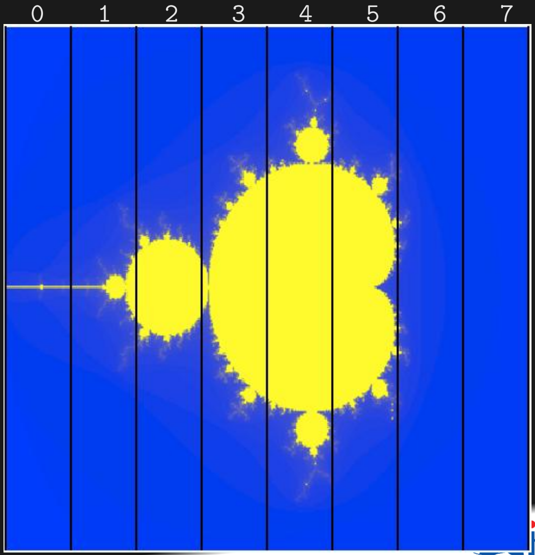
```
rmatrix<int> make_mandel_map(double xmin, double xmax,
                              double ymin, double ymax,
                              int npix, int maxiter) {
  rmatrix<int> mymap(npix,npix);
  #pragma omp parallel for default(none) shared(mymap,xmin,xmax,ymin,ymax,npix,maxiter)
  for (int i=0; i<npix; i++)
    for (int j=0; j<npix; j++) {
      double x = ((double)i)/((double)npix)*(xmax-xmin)+xmin;
      double y = ((double)j)/((double)npix)*(ymax-ymin)+ymin;
      std::complex<double> a(x,y);
      mymap[i][j] = how_many_iter(a,maxiter);
    }
  return mymap;
}
```

```
int how_many_iter(std::complex<double> a, int maxiter) {
   std::complex<double> b = a;
   for (int i=0; i<maxiter; i++) {
      if (std::norm(b) > 4) return i;
      b = b*b + a;
   }
   return maxiter;
}
```

## First Try OpenMP Mandelbrot

- Default work sharing breaks N iterations into N/nthreads chunks and assigns them to threads.
- But threads 0, 1, 6 and 7 will be done and sitting idle while threads 2, 3, 4 and 5 work on the rest
- Inefficient use of resources.

| | |
|---|---|
| Serial | 5.060s |
| Nthreads=16 | 1.336s |
| Speedup | 3.8x |
| Efficiency | 24% |



## Scheduling constructs in OpenMP

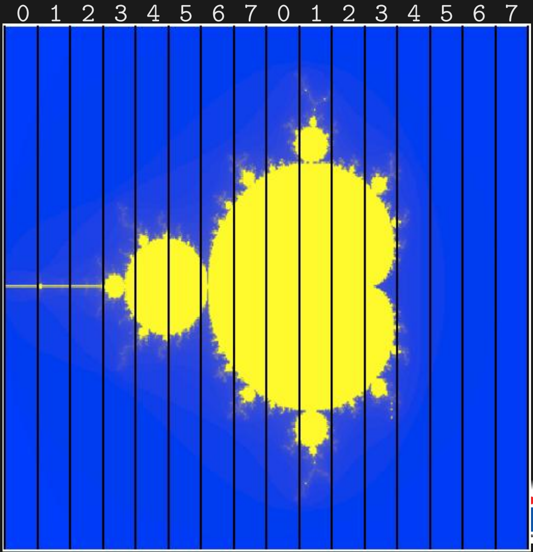
- Default: each thread gets a big consecutive chunk of the loop. Often better to give each thread many smaller interleaved chunks.
- Can add `schedule` clause to `omp for` to change work sharing.
- We can decide either at compile-time (static schedule) or run-time (dynamic schedule) how work will be split.
- `#pragma omp parallel for schedule(static, m)` gives m consecutive loop elements to each thread instead of a big chunk.
- With `schedule(dynamic, m)`, each thread will work through m loop elements, then go to the OpenMP run-time system and ask for more.
- Load balancing (possibly) better with dynamic, but larger overhead than with static.

## Second Try OpenMP Mandelbrot

```
#pragma omp parallel for schedule(static,25)
```

- Can change the chunk size different from ∼ N/nthreads
- In this case, more columns – work distributed a bit better.
- Now, for instance, thread 7 gets both a big work chunk and a little one.

| | |
|---|---:|
| Serial | 5.060s |
| Nthreads=16 | 0.7693s |
| Speedup | 6.6x |
| Efficiency | 41% |

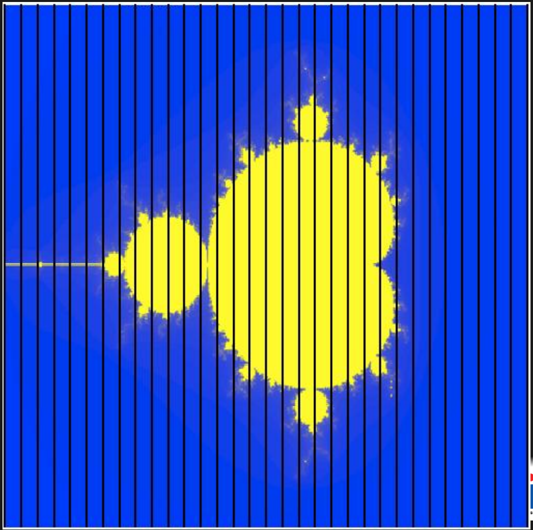


## Third Try: Schedule dynamic

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

- Break up into many pieces and hand them to threads when they are ready.
- Dynamic scheduling.
- Increases overhead, decreases idling threads.
- Can also choose chunk size.

| | |
|---|---|
| Serial | 5.060s |
| Nthreads=16 | 0.7686s |
| Speedup | 6.6x |
| Efficiency | 41% |



## Tuning

- `schedule(static)` or `schedule(dynamic)` are good starting points.
- To get best performance in badly imbalanced problems, may have to play with chunk size; depends on your problem, hardware, and compiler.

| static,1 | dynamic,1 |
|---|---|
| 0.4347s | 0.4121s |
| 11.6x | 12.3x |
| 72% | 77% |

## More. . .

There are many more features to OpenMP not discussed here.

- Collapsed loops
- Tasks
- Tasks with dependencies
- Nested OpenMP parallelism
- Locks
- SIMD
- Thread affinities
- Compute devices (e.g. NVIDIA/AMD graphics cards, Intel Xeon Phi)