Seminar in Parallelization and Program Optimization

Or Ishlach | Micheal Ghanadre

“Hands-on” Introduction to OpenMP*

* The name “OpenMP” is the property of the OpenMP Architecture Review Board.

Our OpenMP progression

OMP introduction
Parallel regions
Pi_spmd_simple.c
hello_world.c

Synchronization
False sharing, critical, atomic
Pi_spmd_final.c

Parallel loops
For, schedule, reduction
Pi_loop.c

Odds and ends
Single, sections, master, etc.

OpenMP tasks
Explicit tasks in OpenMP
Matmul.c

Our OpenMP progression

OMP introduction
Parallel regions
Pi_spmd_simple.c
hello_world.c

Synchronization
False sharing, critical, atomic
Pi_spmd_final.c

Parallel loops
For, schedule, reduction
Pi_loop.c

Odds and ends
Single, sections, master, etc.

OpenMP tasks
Explicit tasks in OpenMP
Matmul.c

OpenMP* overview:

C$OMP FLUSH

#pragma omp critical

OpenMP:An API for Writing Multithreaded Applications

- A set of **compiler directives and library routines** for parallel application programmers
- Greatly **simplifies** writing multi-threaded (MT) programs in Fortran, C and C++

#pragma omp parallel for private(A, B)

!$OMP BARRIER

C$OMP PARALLEL COPYIN(/blk/)

C$OMP DO lastprivate(XX)

Nthrds = OMP_GET_NUM_PROCS()

omp_set_lock(lck)

```
#pragma omp construct [clauses]
```

Indicates that it is as an OpenMP directive

Anatomy of an OpenMP directive

- Most of the constructs in OpenMP are compiler directives.

  #pragma omp construct [clause [clause]…]

- Function prototypes and types in the file:

  #include <omp.h> use omp_lib

```
#pragma omp construct [clauses]
```

Tells the compiler that it is a directive

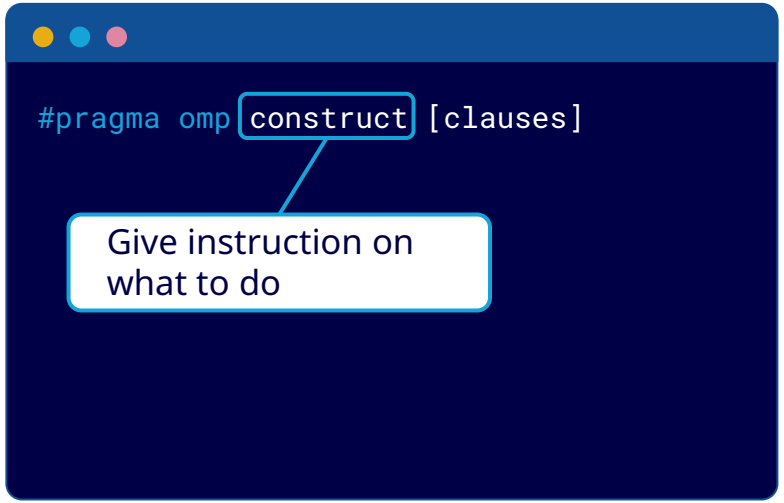
Anatomy of an OpenMP directive

- Most of the constructs in OpenMP are compiler directives.

  #pragma omp construct [clause [clause]…]

- Function prototypes and types in the file:

  #include <omp.h> use omp_lib



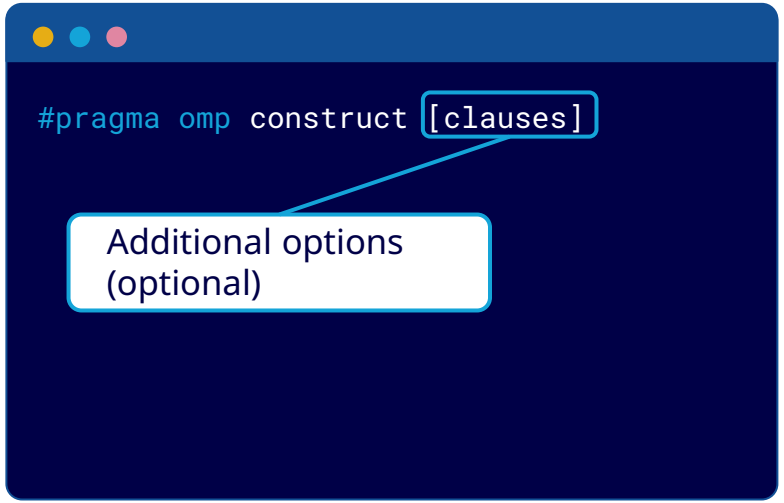
Anatomy of an OpenMP directive

- Most of the constructs in OpenMP are compiler directives.

  #pragma omp construct [clause [clause]…]

- Function prototypes and types in the file:

  #include <omp.h> use omp_lib



Anatomy of an OpenMP directive

- Most of the constructs in OpenMP are compiler directives.

  #pragma omp construct [clause [clause]…]

- Function prototypes and types in the file:

  #include <omp.h> use omp_lib

```
#pragma omp parallel [clauses]
   structured-block
```

Creates a parallel region by spawning a team of threads

The Parallel Construct

Most OpenMP* constructs apply to a **“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.

It’s OK to have an exit() within the structured block.

```
#pragma omp parallel [clauses]
   structured-block
```

Optional clause

The Parallel Construct

Most OpenMP* constructs apply to a **“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.

It’s OK to have an exit() within the structured block.

```
#pragma omp parallel [clauses]
   structured-block
```

Block of code

The Parallel Construct

Most OpenMP* constructs apply to a **“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.

It’s OK to have an exit() within the structured block.

Exercise: Hello world

- Write a program that prints “hello world”.

```
#include<stdio.h>
int main()
{



    int ID = 0;
    printf(“ hello(%d) ”, ID);
    printf(“ world(%d) \n”, ID);


}
```

Exercise: Hello world

- Write a multithreaded program that prints “hello world”.

```
#include<opm.h>
#include<stdio.h>
int main()
{

  #pragma omp parallel
  {
    int ID = omp_get_thread_num();
    printf(“ hello(%d) ”, ID);
    printf(“ world(%d) \n”, ID);
  }

}
```

OpenMP include file

Parallel region with default number of threads

Runtime library function to return a thread ID.

End of the Parallel region

Sample Output:

```
hello(1) hello(0) world(1)
world(0)
hello (3) hello(2) world(3)
world(2)
```

How do threads interact?

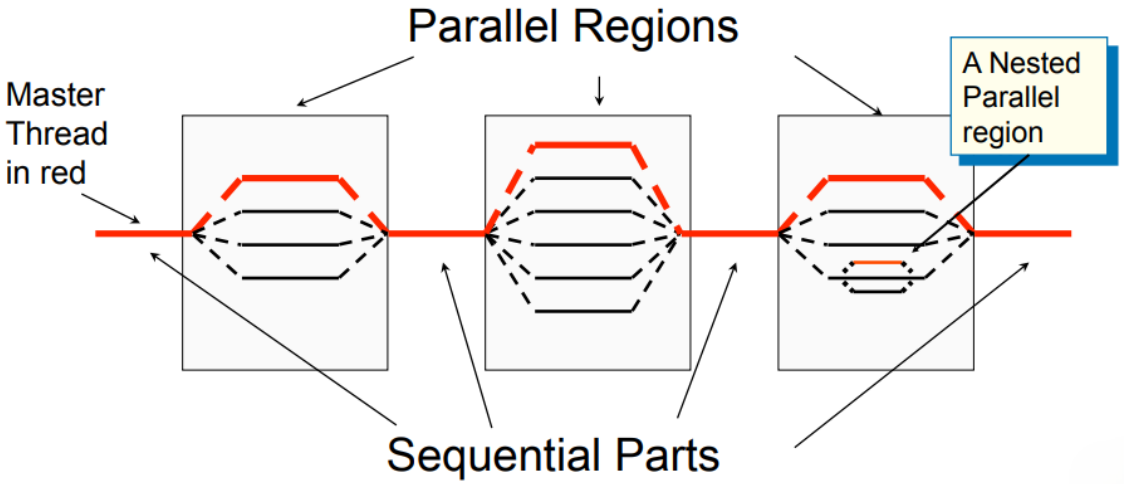
OpenMP overview:

- **OpenMP is a multi-threading, shared address model**
  - Threads communicate by sharing variables.
- **Unintended sharing of data causes race conditions:**
  - Race condition:
    when the program's outcome changes as the threads are scheduled differently.
- **To control race conditions:**
  - Use synchronization to protect data conflicts.
- **Synchronization is expensive so:**
  - Change how data is accessed to minimize the need for synchronization.

OpenMP programming model

Fork-Join Parallelism:

- Master thread spawns a team of threads as needed.
- Parallelism added incrementally until performance goals are met, i.e., the sequential program evolves into a parallel program.



Thread creation: Parallel regions

- You create threads in OpenMP* with the parallel construct.
- For example, To create a 4 thread Parallel region:

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    pooh(ID, A);
}
printf("all done\n");
```

Runtime function to request a certain number of threads

Runtime function returning a thread ID

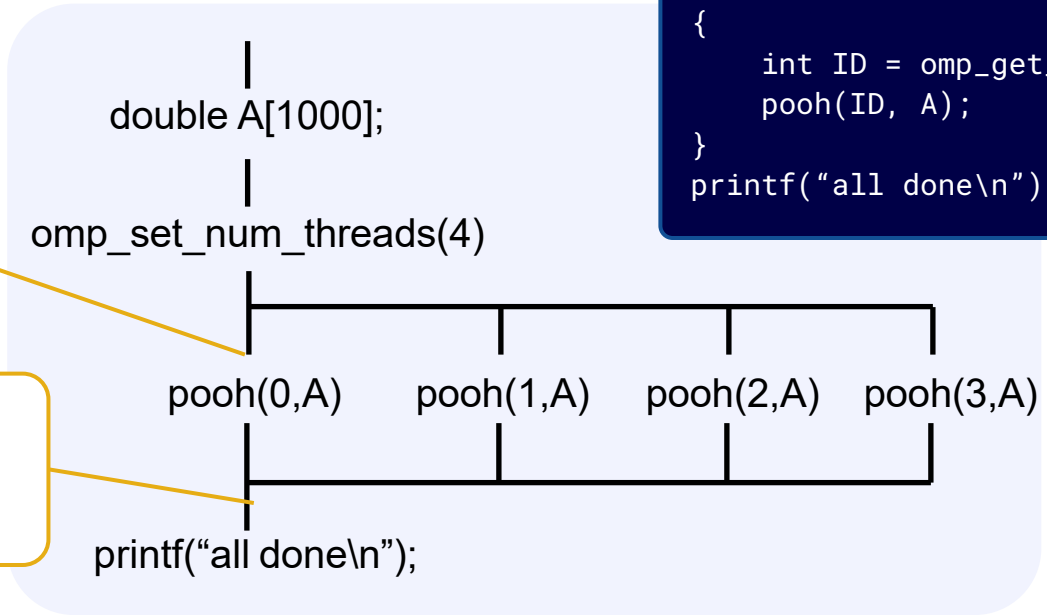
Each thread executes a copy of the code within the structured block **SPMD**

- Each thread calls **pooh(ID,A)** for ID = 0 to 3

Thread creation: Parallel regions example

- Each thread executes the same code redundantly.

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    pooh(ID, A);
}
printf("all done\n");
```

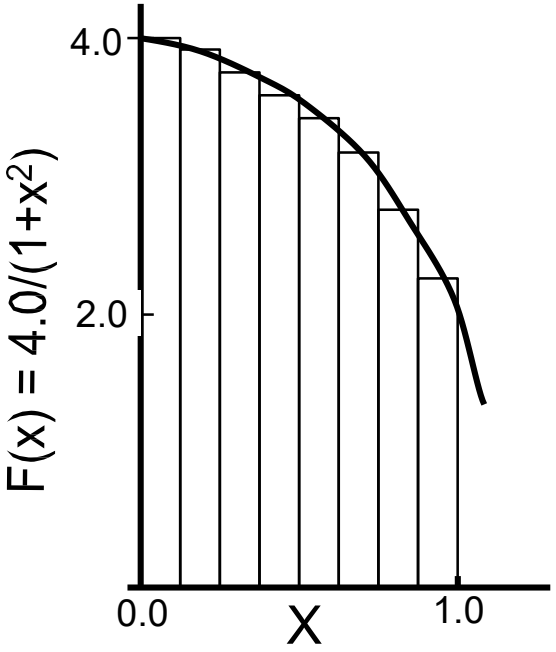


A single copy of A is shared between all threads.

Threads wait here for all threads to finish before proceeding (i.e., a barrier)

Exercise:

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.

Exercise:
Create a parallel version of the pi 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);
 }
```

In addition to a parallel construct,
you will need the runtime library routines:

#pragma omp parallel.
- int omp_get_num_threads();
- int omp_get_thread_num();
- double omp_get_wtime();
- omp_set_num_threads();

Number of threads in the team

Thread ID or rank

Time in Seconds since a fixed point in the past

Request a number of threads in the team

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;
}
```

Results*:

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

```
#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;
}
```

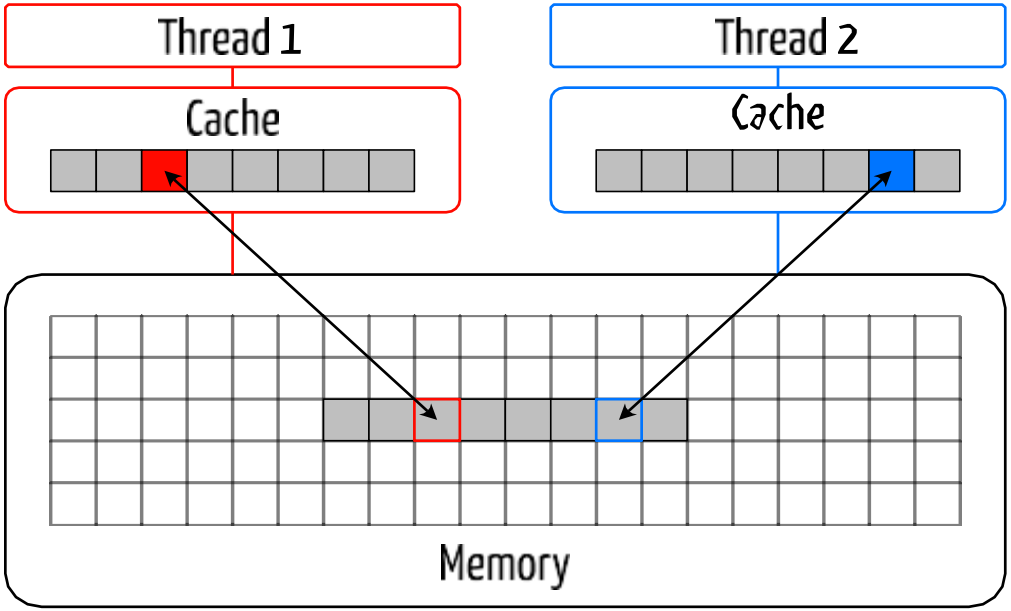
threads	1st SPMD
1	1.86
2	1.03
3	1.08
4	0.97

*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?

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

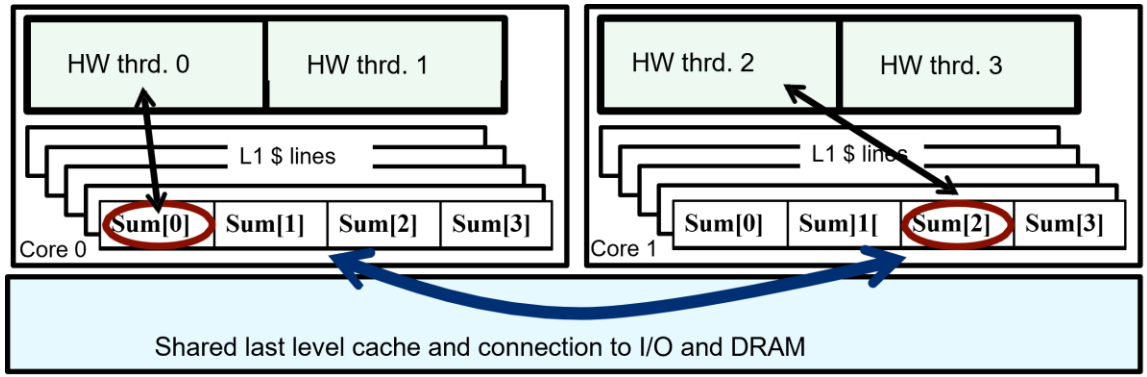
... This is called **“false sharing”.**



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 …
Results in poor scalability.

Solution:
Pad arrays so elements you use are on distinct cache lines.



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;
}
```

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

Results*: pi program padded accumulator

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

```
#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

*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.

Our OpenMP progression

OMP introduction
Parallel regions, False sharing
hello_world.c
Pi_spmd_simple.c

Synchronization
critical, atomic
Pi_spmd_final.c

Parallel loops
For, schedule, reduction
Pi_loop.c

Odds and ends
Single, sections, master, etc.

OpenMP tasks
Explicit tasks in OpenMP
Matmul.c

Synchronization

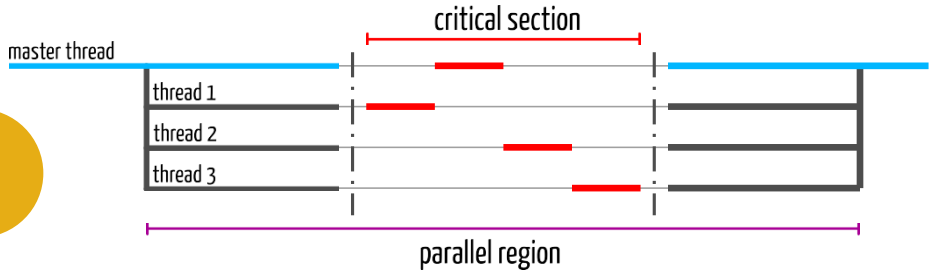
Synchronization is used to impose order constraints and to protect access to shared data

- **High level synchronization:**
  - critical
  - atomic
  - barrier
- **Low level synchronization:**
  - flush
  - locks (both simple and nested)

Synchronization: critical

```
float res;
#pragma omp parallel
{
   float B;
   int i, id, nthrds;
   id = omp_get_thread_num();
   nthrds = omp_get_num_threads();

   for(i=id;i<niters;i+=nthrds)
   {
       B = big_job(i);
       #pragma omp critical
        res += consume (B);
   }
}
```

Threads wait their turn – only one at a time calls consume()



Mutual exclusion:

Only one thread at a time can enter a **critical** region.

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 ()
{
   int nthreads; double pi=0.0; 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+nt
         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

Results*: pi program critical section

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

```
#include <omp.h>
static long num_steps = 100000; double step;
#define NUM_THREADS 2
void main ()
{
   int nthreads; double pi=0.0; step = 1.0/(double) n
   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+nthrds) {
         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.

Synchronization: atomic

```
#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

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

Example:

Using an atomic section to remove impact of false sharing

```
#include <omp.h>
static long num_steps = 100000; double step;
#define NUM_THREADS 2
void main ()
{
   int nthreads; double pi=0.0; 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+nt
         x = (i+0.5)*step;
         sum += 4.0/(1.0+x*x);
      }
      #pragma omp atomic
       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

Atomic vs. Critical

```
Safely increasing the value of count in parallel can be done
either by using an atomic or a critical directive
```

#pragma omp atomic
count++;

- An atomic operation has much lower overhead but the set of possible operations is restricted
- It can take advantage of hardware support for atomic operations

#pragma omp critical
count++;

- A critical section can surround any arbitrary block of code
- There is a significant overhead when a thread enters and exits the critical section

Our OpenMP progression

OMP introduction
Parallel regions
Pi_spmd_simple.c
hello_world.c

Synchronization
False sharing, critical, atomic
Pi_spmd_final.c

Parallel loops
For, schedule, reduction
Pi_loop.c

Odds and ends
Single, sections, master, etc.

OpenMP tasks
Explicit tasks in OpenMP
Matmul.c

A parallel construct by itself creates an SPMD or "Single Program Multiple Data" program

... i.e., each thread redundantly executes the same code.

SPMD vs. worksharing

How do you split up pathways through the code between threads within a team?

– Worksharing constructs

- Loop construct
- Sections/section constructs

– Task constructs

The loop worksharing constructs

The loop worksharing construct splits up loop iterations among the threads in a team

```
#pragma omp parallel
{
  #pragma omp for
   for (i=0;i<N;i++) {
     NEAT_STUFF(I);
   }
}
```

The variable ‘i’ is made private by default.
You could do this explicitly with a “private(i)” clause

Loop worksharing constructs: A motivating example

Sequential code

```
for(i=0;i<N;i++){
  a[i] = a[i] + b[i];
}
```

OpenMP parallel region

```
#pragma omp parallel
{
  int id, i, Nthrds, istart, iend;
  id = omp_get_thread_num();
  Nthrds = omp_get_num_threads();
  istart = id * N / Nthrds;
  iend = (id+1) * N / Nthrds;
  if (id == Nthrds-1)iend = N;
  for(i=istart;i<iend;i++){
    a[i] = a[i] + b[i];
  }
}
```

OpenMP parallel region and a worksharing for construct

```
#pragma omp parallel
#pragma omp for
  for(i=0;i<N;i++){
    a[i] = a[i] + b[i];
  }
```

Loop Scheduling

Loop scheduling, specify how iterations of a loop are divided into contiguous non-empty subsets (chunks), and how these chunks are distributed to the threads. Changing the loop scheduling is possible to use the schedule clause.

```
#pragma omp for schedule(kind,chunk)
 fof-toop
```

Where the value of kind can be static , dynamic , guided or runtime.

The default scheduling is static.

The optional chunk may have different behavior depending on the scheduling.

Static Loop Scheduling

With static loop scheduling, iterations are divided into chunks and the chunks are assigned to the threads.

Each chunk contains the same number of iterations, except for the chunk that contains the last iteration, which may have fewer iterations.

```
#pragma omp for schedule(static)
 fof-toop
```

Dynamic Loop Scheduling

With dynamic loop scheduling, the iterations are distributed to threads in chunks. Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be distributed.

```
#pragma omp for schedule(dynamic)
 fof-toop
```

Guided Loop Scheduling

The guided loop scheduling is similar to the dynamic scheduling, except that the size of each chunk is proportional to the number of unassigned iterations, decreasing to one.

```
#pragma omp for schedule(guided)
 fof-toop
```

Loop work-sharing constructs:

`The schedule clause`

Schedule Clause	One thread
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

omp_sched.c

Combined parallel/worksharing construct

OpenMP shortcut:
Put the “parallel” and the worksharing directive on the same line

```
double res[MAX];
int i;
#pragma omp parallel
{
  #pragma omp for
  for (i=0;i< MAX; i++) {
    res[i] = huge();
  }
}
```

```
double res[MAX];
int i;
#pragma omp parallel for
  for (i=0;i< MAX; i++) {
    res[i] = huge();
  }
```

These are equivalent

Working with loops

Basic approach

- Find compute intensive loops
- Make the loop iterations independent ...

So they can safely execute in any order without loop-carried dependencies

- Place the appropriate OpenMP directive and test

```
int i, j, A[MAX]; j = 5;
for (i=0;i< MAX; i++)
{
  j +=2;
  A[i] = big(j);
}
```

Remove loop carried dependence

```
int i, A[MAX];
#pragma omp parallel for
for (i=0;i< MAX; i++)
{
   int j = 5 + 2*(i+1);
   A[i] = big(j);
}
```

Note:
loop index “i” is private by default

Nested loops

- For perfectly nested rectangular loops we can parallelize multiple loops in the nest with the collapse clause:

```
#pragma omp parallel for collapse(2)
for (int i=0; i<N; i++) {
  for (int j=0; j<M; j++){
   .....
  }
}
```

Number of loops to be parallelized, counting from the outside

- Will form a single loop of length NxM and then parallelize that.
- Useful if N is O(no. of threads) so parallelizing the outer loop makes balancing the load difficult.

Reduction

- How do we handle this case?

```
double ave=0.0, A[MAX];
int i;
for (i=0;i< MAX; i++)
{
    ave + = A[i];
}
ave = ave/MAX;
```

- We are combining values into a single accumulation variable (ave) … there is a true dependence between loop iterations that can't be trivially removed
- This is a very common situation … it is called a "reduction".
- Support for reduction operations is included in most parallel programming environments.

OpenMP: Reduction operands/initial-values

Many different associative operands can be used with reduction:
Initial values are the ones that make sense mathematically.

Operator	Initial value
+	0
×	1
-	0
min	Largest pos. number
max	Most neg. number

C/C++ only	
Operator	Initial value
&	~0
|	0
^	0
&&	1
||	0

```
double ave=0.0, A[MAX]; int i;
#pragma omp parallel for reduction (+:ave)
for (i=0;i< MAX; i++){
    ave + = A[i];
}
ave = ave/MAX;
```

Example:

Pi with a loop and a reduction

```
#include <omp.h>
static long num_steps = 100000; double step;
void main ()
{
  int i; double x, pi, sum = 0.0;
  step = 1.0/(double) num_steps;
  #pragma omp parallel
  {
    double x;
    #pragma omp for reduction(+:sum)
      for (i=0;i< num_steps; i++ ){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
      }
   }
  pi = step * sum;
}
```

Create a team of threads … without a parallel construct, you'll never have more than one thread

Create a scalar local to each thread to hold value of x at the center of each interval

Break up loop iterations and assign them to threads … setting up a reduction into sum. Note … the loop index is local to a thread by default.

Results*: pi with a loop and a reduction

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

```
#include <omp.h>
static long num_steps = 100000; double step;
void main ()
{
  int i; double x, pi, sum = 0.0;
  step = 1.0/(double) num_steps;
  #pragma omp parallel
  {
    double x;
    #pragma omp for reduction(+:sum)
      for (i=0;i< num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
      }
   }
  pi = step * sum;
}
```

threads	1st SPMD	1st SPMD padded	SPMD critical	PI Loop
1	1.86	1.86	1.87	1.91
2	1.03	1.01	1.00	1.02
3	1.08	0.69	0.68	0.80
4	0.97	0.53	0.53	0.68

*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.

Our OpenMP progression

OMP introduction
Parallel regions
Pi_spmd_simple.c
hello_world.c

Synchronization
False sharing, critical, atomic
Pi_spmd_final.c

Parallel loops
For, schedule, reduction
Pi_loop.c

Odds and ends
Single, sections, master, etc.

OpenMP tasks
Explicit tasks in OpenMP
Matmul.c

Synchronization: Barrier

Barrier:

Each thread waits until all threads arrive.

```
double A[big], B[big], C[big];
#pragma omp parallel
{
  int id=omp_get_thread_num();
  A[id] = big_calc1(id);
 #pragma omp barrier
 #pragma omp for
  for(i=0;i<N;i++){C[i]=big_calc3(i,A);}
 #pragma omp for nowait
  for(i=0;i<N;i++){ B[i]=big_calc2(C, i);}
  A[id] = big_calc4(id);
}
```

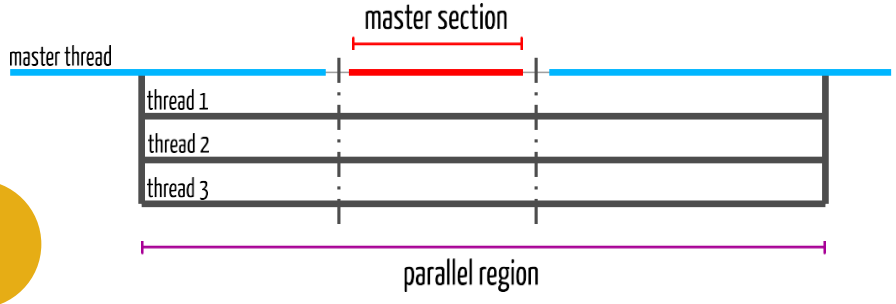
implicit barrier at the end of a for worksharing construct

no implicit barrier due to nowait

implicit barrier at the end of a parallel region

Master construct

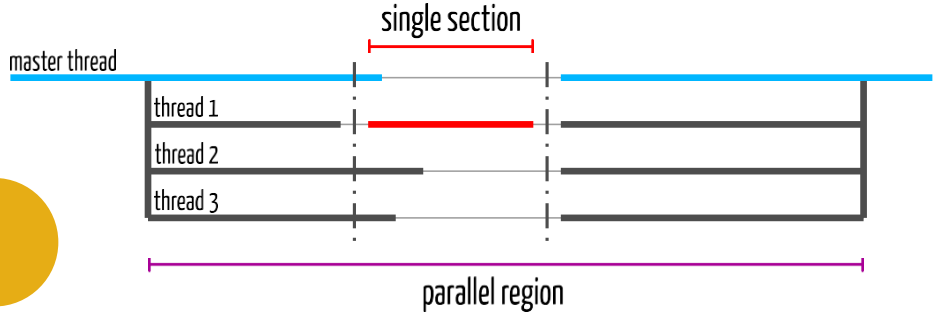
```
#pragma omp parallel
{
  do_many_things();
  #pragma omp master
  { exchange_boundaries(); }

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



- The master construct denotes a structured block that is only executed by the master thread.
- The other threads just skip it (no synchronization is implied).

Single construct

```
#pragma omp parallel
{
  do_many_things();
  #pragma omp single
   {  exchange_boundaries();  }
      do_many_other_things();
}
```



- The single construct denotes a block of code that is executed by only one thread (not necessarily the master thread).
- A barrier is implied at the end of the single block (can remove the barrier with a **nowait** clause).

Sections worksharing construct

```
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section X_calculation();
    #pragma omp section y_calculation();
    #pragma omp section z_calculation();
  }
}
```

By default,
there is a barrier at the end
of the “omp sections”.
Use the “nowait” clause to
turn off the barrier.

- The Sections worksharing construct gives a different structured block to each thread.

Our OpenMP progression

OMP introduction
Parallel regions
Pi_spmd_simple.c
hello_world.c

Synchronization
False sharing, critical, atomic
Pi_spmd_final.c

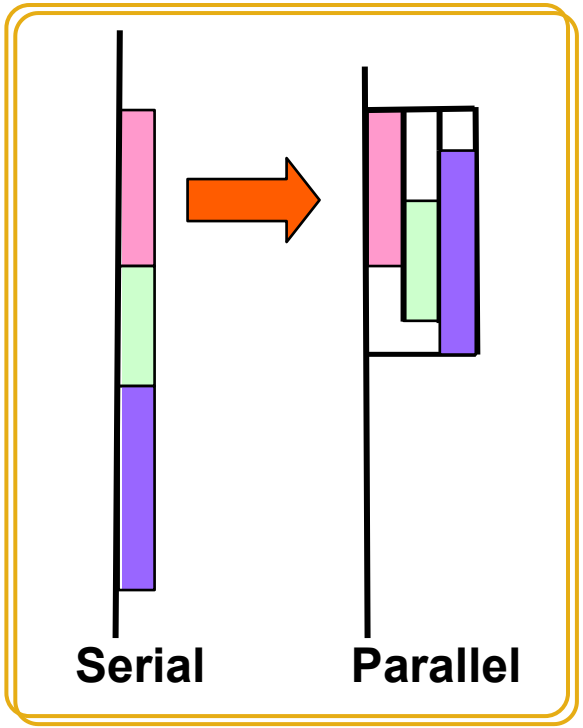
Parallel loops
For, schedule, reduction
Pi_loop.c

Odds and ends
Single, sections, master, etc.

OpenMP tasks
Explicit tasks in OpenMP
Matmul.c

OpenMP Tasks

- Tasks are independent units of work.
- Tasks are composed of:
  - **code** to execute
  - **data** environment
  - **internal control variables** (ICV)
- Threads perform the work of each task.
- The runtime system decides when tasks are executed
  - Tasks may be deferred
  - Tasks may be executed immediately



How tasks work

- The task construct defines a section of code

```
#pragma omp task
{
    ...some code
}
```

- Inside a parallel region,
  a thread encountering a task construct will package up the task for execution
- Some thread in the parallel region will execute the task at some point in the future
- Tasks can be nested: i.e., a task may itself generate tasks

When are tasks guaranteed to complete

- Tasks are guaranteed to be complete at thread barriers:
  `#pragma omp barrier`
- or task barriers
  `#pragma omp taskwait`

```
#pragma omp parallel
{
    #pragma omp task
    foo();
    #pragma omp barrier
    #pragma omp single
    {
      #pragma omp task
      bar();
    }
}
```

Multiple foo tasks created here – one for each thread

All foo tasks guaranteed to be completed here

One bar task created here

bar task guaranteed to be completed here

Exercise:

Strassen's Matrix Multiplication

Reminder:

Basic Matrix Multiplication

Suppose we want to multiply two matrices of size N x N:

C = A x B

$$\begin{vmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{vmatrix} = \begin{vmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{vmatrix} \begin{vmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{vmatrix}$$

$C_{11} = a_{11}b_{11} + a_{12}b_{21}$ $\quad$ $C_{21} = a_{21}b_{11} + a_{22}b_{21}$

$C_{12} = a_{11}b_{12} + a_{12}b_{22}$ $\quad$ $C_{22} = a_{21}b_{12} + a_{22}b_{22}$

2x2 matrix multiplication can be accomplished in 8 multiplication. **($2^{\log_2 8} = 2^3$)**

Reminder:

Basic Matrix Multiplication

- Algorithm

```
void matrix_mult ()
{
  for (i = 1; i <= N; i++)
  {
      for (j = 1; j <= N; j++) compute Ci,j;
  }
}
```

- Time analysis:

$$C_{i,j} = \sum_{k=1}^{N} a_{i,k}\, b_{k,j}$$

Thus

$$T(N) = \sum_{i=1}^{N} \sum_{j=1}^{N} \sum_{k=1}^{N} c = cN^3 = \mathbf{O}(N^3)$$

Exercise:

Strassen's Matrix Multiplication

- Strassen showed that 2x2 matrix multiplication can be accomplished in 7 multiplication and 18 additions or subtractions.

$$(2^{\log_2 7} = 2^{2.807})$$

- This reduce can be done by Divide and Conquer Approach.

Divide-and-Conquer

- Divide-and conquer is a general algorithm design paradigm:
  - Divide: divide the input data $\boldsymbol{S}$ in two or more disjoint subsets $\boldsymbol{S}_1$, $\boldsymbol{S}_2$, …
  - Recur: solve the subproblems recursively
  - Conquer: combine the solutions for $\boldsymbol{S}_1$, $\boldsymbol{S}_2$, …, into a solution for $\boldsymbol{S}$
- The base case for the recursion are subproblems of constant size
- Analysis can be done using recurrence equations

Divide and Conquer Matrix Multiply

$$A \times B = R$$

$$\begin{bmatrix} A_0 & A_1 \\ A_2 & A_3 \end{bmatrix} \times \begin{bmatrix} B_0 & B_1 \\ B_2 & B_3 \end{bmatrix} = \begin{bmatrix} \mathbf{A_0 \times B_0 + A_1 \times B_2} & \mathbf{A_0 \times B_1 + A_1 \times B_3} \\ \mathbf{A_2 \times B_0 + A_3 \times B_2} & \mathbf{A_2 \times B_1 + A_3 \times B_3} \end{bmatrix}$$

- Divide matrices into sub-matrices: A_0 , A_1, A_2 etc
- Use blocked matrix multiply equations
- Recursively multiply sub-matrices

Divide and Conquer Matrix Multiply

$$A \times B = R$$

$$\boxed{a_0} \times \boxed{b_0} = \boxed{a_0 \times b_0}$$

- Terminate recursion with a simple base case

Strassens's Matrix Multiplication

$$\begin{vmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{vmatrix} = \begin{vmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{vmatrix} \begin{vmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{vmatrix}$$

$P_1 = (A_{11} + A_{22})(B_{11} + B_{22})$
$P_2 = (A_{21} + A_{22}) * B_{11}$
$P_3 = A_{11} * (B_{12} - B_{22})$
$P_4 = A_{22} * (B_{21} - B_{11})$
$P_5 = (A_{11} + A_{12}) * B_{22}$
$P_6 = (A_{21} - A_{11}) * (B_{11} + B_{12})$
$P_7 = (A_{12} - A_{22}) * (B_{21} + B_{22})$

$C_{11} = P_1 + P_4 - P_5 + P_7$
$C_{12} = P_3 + P_5$
$C_{21} = P_2 + P_4$
$C_{22} = P_1 + P_3 - P_2 + P_6$

Comparison

$$C_{11} = P_1 + P_4 - P_5 + P_7$$
$$= (A_{11} + A_{22})(B_{11} + B_{22}) + A_{22} * (B_{21} - B_{11}) - (A_{11} + A_{12}) * B_{22} +$$
$$(A_{12} - A_{22}) * (B_{21} + B_{22})$$
$$= A_{11} B_{11} + A_{11} B_{22} + A_{22} B_{11} + A_{22} B_{22} + A_{22} B_{21} - A_{22} B_{11} -$$
$$A_{11} B_{22} - A_{12} B_{22} + A_{12} B_{21} + A_{12} B_{22} - A_{22} B_{21} - A_{22} B_{22}$$
$$= A_{11} B_{11} + A_{12} B_{21}$$

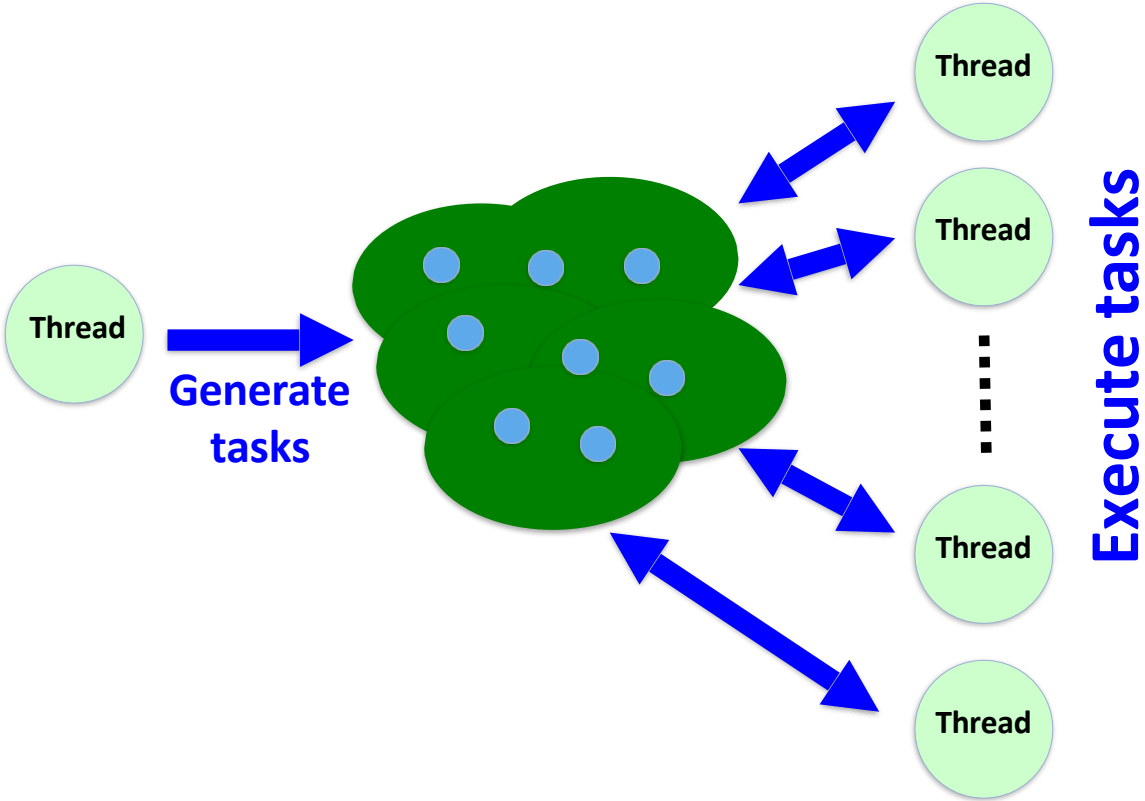
Strassen Algorithm

```
void matmul(int *A, int *B, int *R, int n) {
  if (n == 1) {
(*R) += (*A) * (*B);
  } else {
matmul(A, B, R, n/4);
matmul(A, B+(n/4), R+(n/4), n/4);
matmul(A+2*(n/4), B, R+2*(n/4), n/4);
matmul(A+2*(n/4), B+(n/4), R+3*(n/4), n/4);
matmul(A+(n/4), B+2*(n/4), R, n/4);
matmul(A+(n/4), B+3*(n/4), R+(n/4), n/4);
matmul(A+3*(n/4), B+2*(n/4), R+2*(n/4), n/4);
matmul(A+3*(n/4), B+3*(n/4), R+3*(n/4), n/4);
  }
```

We will use openMP tasks

Divide matrices in sub-matrices and recursively multiply sub-matrices

Strassens's Matrix Multiplication



Time Analysis

$$T(1) = 1 \qquad (\text{assume } N = 2^k)$$

$$T(N) = 7T(N/2)$$

$$T(N) = 7^k T(N/2^k) = 7^k$$

$$T(N) = 7^{\log N} = N^{\log 7} = N^{2.81}$$

Seminar in Parallelization and Program Optimization

Or Ishlach | Micheal Ghanadre

“Hands-on” Introduction to OpenMP*

Thank You for Your Attention!