

Parallel Programming in OpenMP – part II

Outline

- ❑ Data scoping – cont'd
- ❑ Orphaning
- ❑ Tasking
- ❑ OpenMP correctness & Data Races
- ❑ Runtime library
- ❑ Scheduling
- ❑ A real world example

OpenMP Syntax

More on data scoping

OpenMP Syntax

Reminder: the “private” clause –

- ❑ declares variables private to each thread:

```
#pragma omp directive private (list)
```

- ❑ i.e. a **new** variable is declared once for each thread
- ❑ all references are replaced with references to the newly declared variable
- ❑ variables declared private are uninitialized for each thread!

OpenMP Syntax

Consequences of private(...):

```
main() {  
    ...  
    A = 10;  
  
    #pragma omp parallel  
    {  
        #pragma omp for private(i, A, B) ...  
        for(i = 0; i < n; i++) {  
            ...  
            B = A + i;    // A undefined!  
                        // unless declared firstprivate  
  
            ...  
        } /* end of omp for */  
        ...  
    } /* end of omp parallel */  
  
    C = B;                // B undefined!  
                        // unless declared lastprivate  
  
}
```

OpenMP Syntax

Solutions:

```
#pragma omp ... firstprivate(list)
```

- ❑ All variables in list are initialized with the value the original object had before entering the parallel construct.

```
#pragma omp ... lastprivate(list)
```

- ❑ The thread that executes the sequentially last iteration updates all variables in list.

OpenMP Syntax

The “threadprivate” and “copyin” clauses:

- ❑ **threadprivate(list)**: creates a private copy of global data (e.g. common blocks or global variables in modules in Fortran) for each thread
- ❑ **copyin(list)**: copies the values from the master thread into the private copies – like a ‘firstprivate’ for global variables
- ❑ subsequent modifications of list affect only the private copies – within one parallel region

OpenMP Syntax

Example 1:

```
int counter = 0;
#pragma omp threadprivate(counter)

int increment_counter()
{
    counter++;
    return(counter);
}
```

```
INTEGER FUNCTION INCREMENT_COUNTER()
COMMON/A22_COMMON/COUNTER
!$OMP THREADPRIVATE(/A22_COMMON/)
COUNTER = COUNTER +1
INCREMENT_COUNTER = COUNTER
RETURN
END FUNCTION INCREMENT_COUNTER
```


OpenMP Syntax

Example 2:

```
int  
increment_counter()  
{  
    static int counter = 0;  
    #pragma omp threadprivate(counter)  
  
    counter++;  
    return(counter);  
}
```

OpenMP Syntax

The `copyprivate(...)` clause

- ❑ copying a value out of a single region into the private data of other threads (“broadcast”)

```
#pragma omp single copyprivate(list)
{
    ...
}
```

```
!$OMP SINGLE ....
...
!$OMP END SINGLE COPYPRIVATE(LIST)
```

OpenMP Syntax

Example:

```
int x, y; /* global data */
#pragma omp threadprivate(x, y)

void use_values(int id, int a, int b) {
    printf("  TID %d: a = %d, b = %d, c = %d, d = %d\n",
          id, a, b, x, y);
}

void init(int id, int *a, int *b) {
    int r_a, r_b;

    #pragma omp single copyprivate(r_a, r_b, x, y)
    {
        scanf("%d %d %d %d", &r_a, &r_b, &x, &y);
    }

    *a = r_a; *b = r_b;
    use_values(id, *a, *b);
}

...
```

OpenMP Syntax

Example (cont'd):

```
int main(int argc, char *argv[] ) {  
  
    int tid = 0;  
    int a, b;  
  
    #pragma omp parallel private(tid,a,b)  
    {  
        #ifdef _OPENMP  
            tid = omp_get_thread_num();  
        #endif  
  
        init(tid,&a,&b);  
        printf("In main - TID %d: a = %d, b = %d,",  
              " x = %d, y = %d\n",  
              tid, a, b, x, y);  
    } /* end of omp parallel */  
  
    return(0);  
}
```

OpenMP Syntax

Example output:

```
$ OMP_NUM_THREADS=3 ./copypriv
```

```
1 2 3 4
```

```
TID 2: a = 1, b = 2, c = 3, d = 4
```

```
TID 1: a = 1, b = 2, c = 3, d = 4
```

```
TID 0: a = 1, b = 2, c = 3, d = 4
```

```
In main - TID 0: a = 1, b = 2, x = 3, y = 4
```

```
In main - TID 1: a = 1, b = 2, x = 3, y = 4
```

```
In main - TID 2: a
```

```
env OMP_NUM_THREADS=3 ./copypriv
```

```
1 2 3 4
```

```
TID 1: a = 0, b = 1, c = 3, d = 4
```

```
TID 0: a = 0, b = 0, c = 3, d = 4
```

```
TID 2: a = 1, b = 2, c = 3, d = 4
```

```
In main - TID 2: a = 1, b = 2, x = 3, y = 4
```

```
In main - TID 0: a = 0, b = 0, x = 3, y = 4
```

```
In main - TID 1: a = 0, b = 1, x = 3, y = 4
```

without copyprivate
on r_a and r_b

OpenMP Orphaning

Orphaning in OpenMP

OpenMP Orphaning

The OpenMP standard does not restrict worksharing and synchronization directives to be within the lexical extent of a parallel region. Those directives can be orphaned, i.e. they can appear outside a parallel region:

```
#pragma omp parallel
{
    :
    dowork();
    :
}
```

orphaned
worksharing
directive

```
void dowork(void) {
    :
    #pragma omp for
    for(i=0; i<N; i++) {
        :
    }
}
```

OpenMP Orphaning

- ❑ When an orphaned directive is detected within the dynamic extent of a parallel region, its behaviour is similar to the non-orphaned case.
- ❑ When an orphaned directive is detected in the sequential part of the program, it will be ignored.

```
dowork(); // serial for  
  
#pragma omp parallel  
{  
    :  
    dowork(); // parallel for  
    :  
}
```

```
void dowork(void) {  
    :  
    #pragma omp for  
    for(i=0; i<N; i++) {  
        :  
    }  
}
```


OpenMP syntax

Tasking (since OpenMP 3.0)

- ❑ allows parallelization of work that is generated dynamically
- ❑ provides a flexible model for irregular parallelism
- ❑ uses a “task pool” concept
- ❑ new opportunities:
 - ❑ while loops
 - ❑ recursive structures

OpenMP syntax

❑ Syntax C/C++:

```
#pragma omp task [clause]
{
    ...
}
```

❑ clause can be

- ❑ if (int_expr)
- ❑ default(shared|none)
- ❑ private(list), shared(list)
- ❑ firstprivate(list)
- ❑ untied

OpenMP syntax

❑ Syntax Fortran:

```
!$OMP task [clause]  
  
...  
!$OMP end task
```

❑ where clause can be

- ❑ if (int_expr)
- ❑ default(shared|private|firstprivate|none)
- ❑ private(list), shared(list)
- ❑ firstprivate(list)
- ❑ untied

OpenMP syntax

Tasking example I:

while loop:

```
p = lhead;
while (p != NULL)
{
    do_work(p);
    p = next(p);
}
```

parallel while loop with OpenMP tasks:

```
#pragma omp parallel
{
    #pragma omp single
    {
        p = lhead;
        while (p != NULL) {
            #pragma omp task
            {
                do_work(p);
            }
            p = next(p);
        }
    } // end of single
} // end of parallel
```

OpenMP syntax

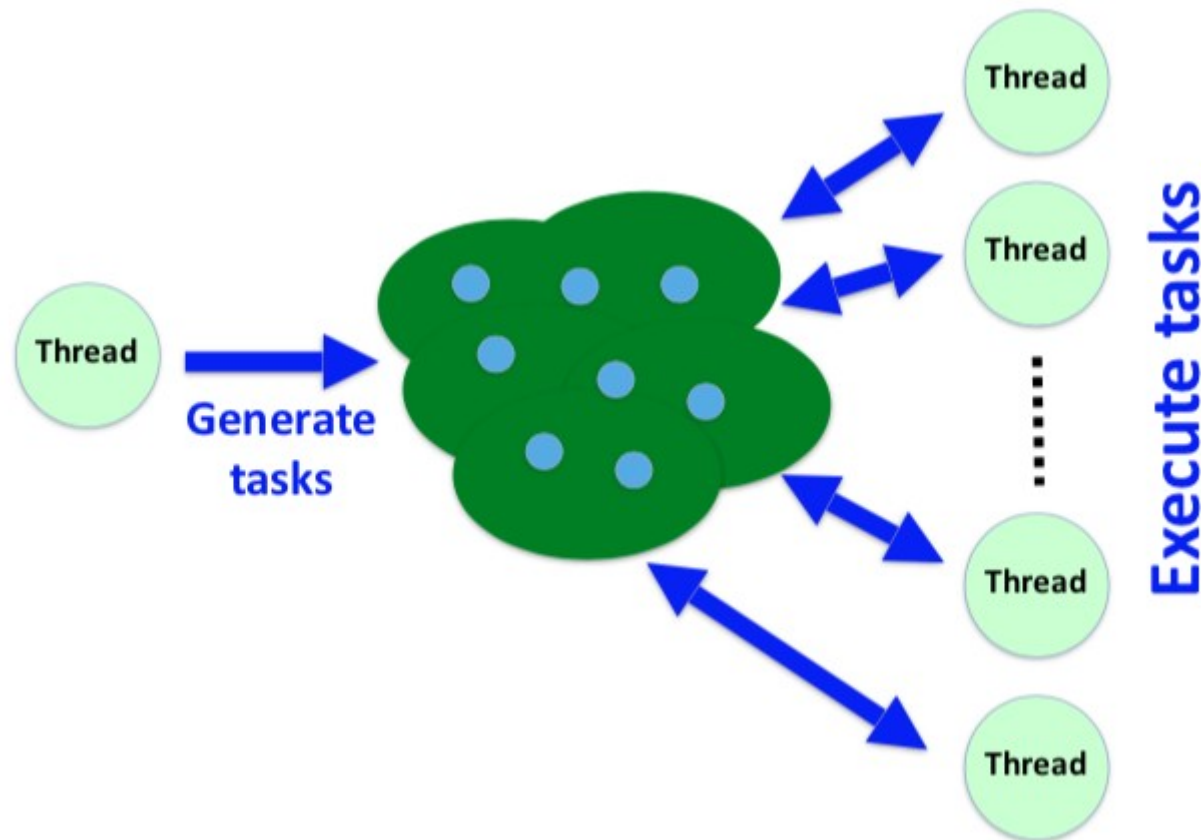
What's going on?

```
#pragma omp parallel
{
    #pragma omp single
    {
        p = lhead;
        while (p != NULL) {
            #pragma omp task
            {
                do_work(p);
            }
            p = next(p);
        }
    } // end of single
} // end of parallel
```

- ← start of parallel region
- ← one thread only, please
- ← task generation – tasks are added to the task list
- ← **all work is done here!**
- ← implicit barrier – all unfinished tasks have to be finished

OpenMP syntax

The task pool concept:



courtesy: Ruud van der Pas, Oracle

OpenMP syntax

Tasks and recursion: Fibonacci numbers

- ❑ Recursive scheme to calculate the n^{th} Fibonacci number:
 - ❑ $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$
 - ❑ stopping criterion: return 1 if $n < 2$
- ❑ Caveat: this method is not very effective, but used here to demonstrate the concept of tasking!

OpenMP syntax

The sequential code:

```
int
main(int argc, char* argv[]) {
    [...]
    fib(input);
    [...]
}
```

```
int
fib(int n) {

    int x, y;

    if (n < 2) return n;

    x = fib(n - 1);
    y = fib(n - 2);

    return(x + y);
}
```


OpenMP syntax

OpenMP version of fib() with tasks:

```
int
fib(int n) {

    int x, y;

    if (n < 2) return n;

    #pragma omp task shared(x)
    x = fib(n - 1);
    #pragma omp task shared(y)
    y = fib(n - 2);

    #pragma omp taskwait
    return(x + y);

}
```

note the special
scoping rules!

generate two tasks,
calling fib() recursively

task synchronization -
to get the right results

OpenMP syntax

Scoping rules with tasks:

- ❑ Static and global variables are shared
- ❑ Local (aka automatic) variables are private
- ❑ Orphaned task variables are firstprivate
- ❑ Non-orphaned task variables inherit the shared attribute
- ❑ (Local) Task variables are firstprivate, unless declared shared
- ❑ Thus, we have to declare x and y as shared

OpenMP syntax

Task synchronization:

- ❑ `#pragma omp taskwait`
- ❑ suspends the encountering task, until all child tasks are completed
- ❑ direct children only, not descendants
- ❑ needed here, to make sure that x and y are still exist when we take the sum.

OpenMP syntax

OpenMP version of main() with tasks:

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

    #pragma omp parallel
    {
        #pragma omp single
        {
            fib(input);
        }
        // end of omp parallel

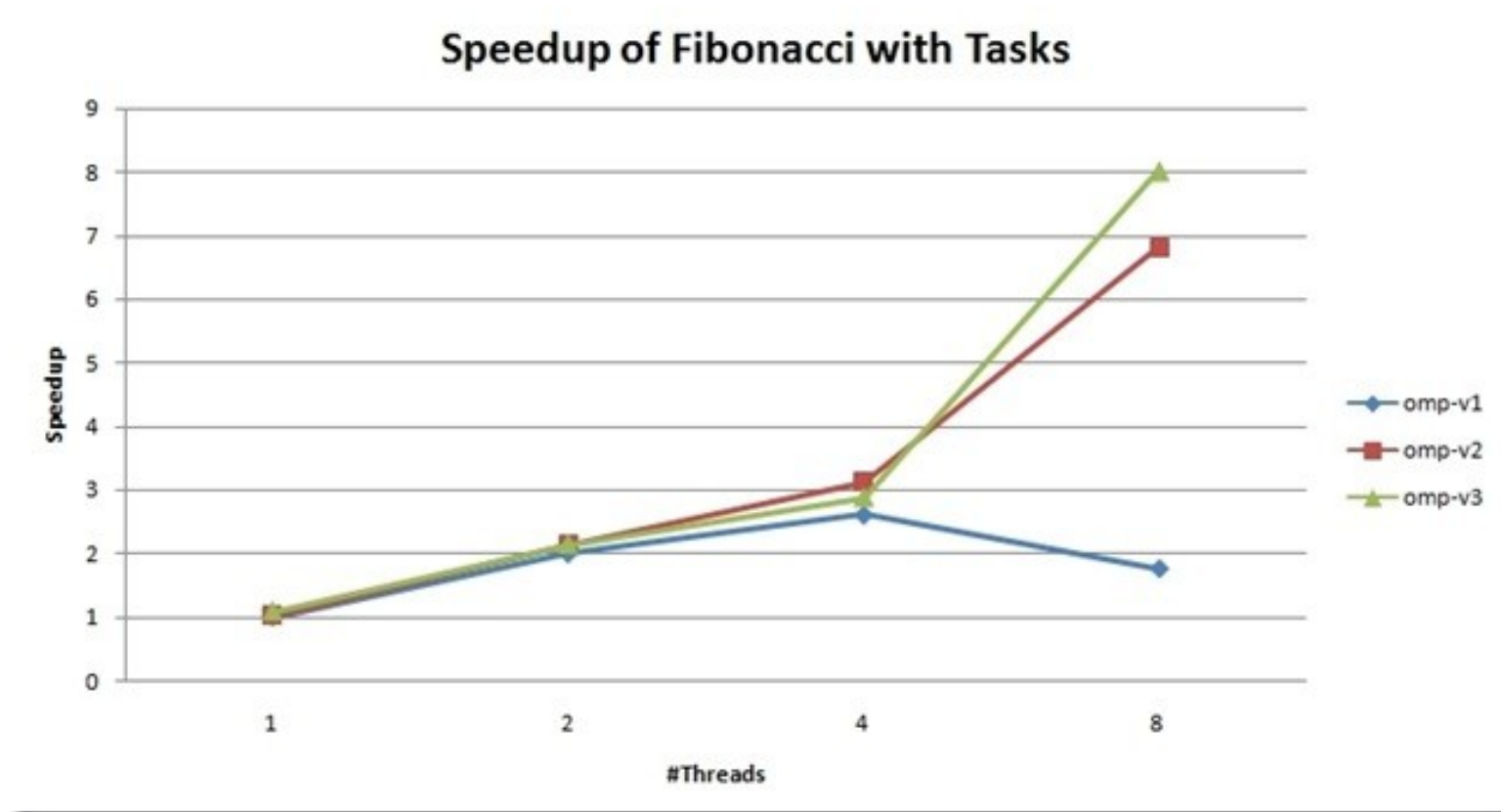
        [...]
    }
}
```

start of parallel region -
team of worker threads

task generation by one
thread, only!

OpenMP syntax

Results of the Fibonacci program



courtesy: Chr. Terboven, RWTH Aachen

OpenMP syntax

Notes on the Fibonacci speedup results:

- ❑ The simple OpenMP version (omp-v1) doesn't scale – as expected – due to the large amount of tasks generated
- ❑ Improvement 1 (omp-v2):
 - ❑ add an if-clause to the tasks:
`#pragma omp task if(n>=30) shared(...)`
 - ❑ improves the speed-up, but still not perfect
- ❑ Improvement 2 (omp-v3): (see next slide)

OpenMP syntax

version omp-v3 of fib() with tasks:

```
int
fib(int n) {

    int x, y;

    if (n < 2) return n;
    if (n < 30) {
        return(fib(n-1) + fib(n-2));
    }

    #pragma omp task shared(x)
    x = fib(n - 1);
    #pragma omp task shared(y)
    y = fib(n - 2);

    #pragma omp taskwait
    return(x + y);
}
```

OpenMP syntax

Some notes on tasking:

- ❑ tasks allow to exploit parallelism with OpenMP, that hasn't been possible before
- ❑ this makes OpenMP a more powerful parallel programming API
 - ❑ goes beyond the scope in this course
- ❑ a nice intro about tasking in OpenMP:

<https://www.openmp.org/events/webinar-how-to-get-openmp-tasks-to-do-your-work/>

OpenMP: Error detection

Tools to check your OpenMP code

OpenMP compile-time checks

- ❑ Most compiler check for OpenMP syntax
 - ❑ area of constant improvement, but quality is very much compiler dependent
- ❑ Oracle Studio compilers can do more extensive checks at compile time:
 - ❑ -xloopinfo – info on loops
 - ❑ -xvpara – compile time warnings on ...
 - ❑ ... scoping problems
 - ❑ ... possible data races
- ❑ Problem: limited to OpenMP 4.0, no newer constructs supported – but still nice as a check during code development

OpenMP: Data Race Detection

Oracle Studio has a data race detection tool:

- ❑ Thread Analyzer (`tha`)
- ❑ Quick usage guide:
 - ❑ compile and link with `-xinstrument=datarace`
 - ❑ run it: `collect -r on a.out`
 - ❑ view results (GUI): `tha tha.1.er`
 - ❑ or CLI: `er_print -races tha.1.er`
- ❑ Caveat: limited to OpenMP 4.0 and earlier

OpenMP: Data Race Detection

Example:

```
int main(int argc, char *argv[]) {  
  
    int i, total = 0, N = 2000000;  
    int primes[N];  
  
    #pragma omp parallel for  
  
    for( i = 2; i < N; i++ ) {  
        if ( is_prime(i) ) {  
            primes[total] = i;  
            total++;  
        }  
    }  
  
    printf("# of prime numbers between 2 and %d: %d\n",  
          N, total);  
    return(0);  
}
```

OpenMP: Data Race Detection

Example (cont'd): compile and run

```
$ suncc -g -fast -o prime prime.c
$ ptime ./prime
# of prime numbers between 2 and 2000000: 148933

real          10.862
user          10.483
sys           0.056

$ suncc -g -fast -xopenmp -xloopinfo -o prime prime.c
$ "prime.c", line 7: PARALLELIZED, user pragma used
$ OMP_NUM_THREADS=4 ptime ./prime
# of prime numbers between 2 and 2000000: 148310

real          3.662    <--- speed-up: 2.9x
user          10.494
sys           0.055
```

OpenMP: Data Race Detection

Example (cont'd): run, run, ... and use collect

```
$ OMP_NUM_THREADS=4 ./prime
# of prime numbers between 2 and 2000000: 148310

$ OMP_NUM_THREADS=4 ./prime
# of prime numbers between 2 and 2000000: 148328

$ suncc -g -fast -xopenmp -xinstrument=datarace \
  -o prime prime.c

$ OMP_NUM_THREADS=4 collect -r on ./prime
Creating experiment database tha.1.er ...
# of prime numbers between 2 and 2000000: 125581

$
```

OpenMP: Data Race Detection

Example (cont'd): analyze the collect data

```
$ er_print -races tha.1.er
Total Races:  2 Experiment:  tha.1.er

Race #1, Vaddr: (Multiple Addresses)
    Access 1: Write, main -- MP doall from line 6 ...
                line 10 in "prime.c"
    Access 2: Write, main -- MP doall from line 6 ...
                line 10 in "prime.c"
    Total Traces: 1

Race #2, Vaddr: 0xffbfecb4
    Access 1: Write, main -- MP doall from line 6 ...
                line 11 in "prime.c"
    Access 2: Write, main -- MP doall from line 6 ...
                line 11 in "prime.c"
    Total Traces: 1
```

OpenMP: Data Race Detection

Example (cont'd): analyze the collect data

```
$ tha tha.1.er
```

The screenshot displays the Sun Studio Analyzer interface for analyzing a data race. The main window is titled "Sun Studio Analyzer [tha.1.er] <@bohr>". The "Races" tab is active, showing a list of race accesses. The "Dual Source" view is also active, displaying the source code with highlighted race regions. The "Race Details" panel on the right shows the summary and details for a selected race.

Races List:

Race Accesses	Source File: ./prime2.c	Object File: ./prime2	Load Object: <prime2>
2	30. if (is_prime(i)) {		
2	31.		
2	32. primes[total] = i;		
2	33. total++;		
2	34. }		
2	35. }		
0	36.		
0	37. printf("# of prime numbers between 2 and %d: %d\n",N, total);		

Race Details Panel:

Summary | **Race Details**

Data for Selected Race

Id: Race #1

Vaddr: (Multiple Addresses)

Access 1

Type: Write

main -- MP doall from line 28 [_\$d1A28.main] + 0x0

Access 2

Type: Write

main -- MP doall from line 28 [_\$d1A28.main] + 0x0

OpenMP: Data Race Detection

Example (cont'd): fix the bug

```
int main(int argc, char *argv[]) {  
  
    int i, total = 0, N = 2000000;  
    int primes[N];  
    #pragma omp parallel for  
  
    for( i = 2; i < N; i++ ) {  
        if ( is_prime(i) ) {  
            #pragma omp critical  
            { primes[total] = i;  
              total++;  
            }  
        }  
    }  
    printf("# of prime numbers between 2 and %d: %d\n",  
           N, total);  
    return(0);  
}
```

OpenMP: Data Race Detection

Example (cont'd): check – and recompile

```
$ suncc -g -fast -xopenmp -xinstrument=datarace \  
-o prime prime.c
```

```
$ OMP_NUM_THREADS=4 collect -r on ./prime  
Creating experiment database tha.2.er ...  
# of prime numbers between 2 and 2000000: 148933
```

```
$ er_print -races tha.2.er  
Total Races: 0 Experiment: tha.2.er
```

```
$ cc -g -fast -xopenmp -o prime prime.c  
$ OMP_NUM_THREADS=4 ptime ./prime  
# of prime numbers between 2 and 2000000: 148933
```

```
real          3.561  
user          10.393  
sys           0.051
```

OpenMP: Data Race Detection

- ❑ Some notes:
 - ❑ Oracle Studio does only support OpenMP up to version 4.0
 - ❑ most of the features we use here are supported, but not all
 - ❑ thus, using this tool will not work for everything
 - ❑ e.g. “doacross” loops

OpenMP Scheduling

Controlling the scheduling
of OpenMP threads

OpenMP Scheduling

Load balancing:

- ❑ Important aspect of performance
- ❑ Especially for less regular workloads, e.g.
 - ❑ transposing a matrix
 - ❑ multiplications of triangular matrices
 - ❑ parallel searches in a linked list
- ❑ The **schedule** clause provides different iteration scheduling algorithms for loops

OpenMP Scheduling

The “schedule” clause:

```
#pragma omp for schedule(static[,chunk])
```

```
#pragma omp for schedule(dynamic[,chunk])
```

```
#pragma omp for schedule(guided[,chunk])
```

```
#pragma omp for schedule(auto) – new in 3.0
```

```
#pragma omp for schedule(runtime)
```

- ❑ If there is no schedule clause, the default is static.

OpenMP Scheduling

```
#pragma omp for schedule(static[, chunk])
```

Static schedule:

- ❑ Iterations are divided into pieces of size chunk and then **statically** assigned to the threads.
- ❑ If chunk is not defined, the work (N) is equally divided among the number of threads (P), i.e. $\text{chunk} = N/P$.

OpenMP Scheduling

```
#pragma omp for schedule(dynamic[,chunk])
```

Dynamic schedule:

- ❑ Iterations are divided into pieces of size chunk and then **dynamically** assigned to the threads – i.e. when a thread has finished one chunk, it is assigned a new one.
- ❑ The default chunk size is 1.

OpenMP Scheduling

```
#pragma omp for schedule(guided[, chunk])
```

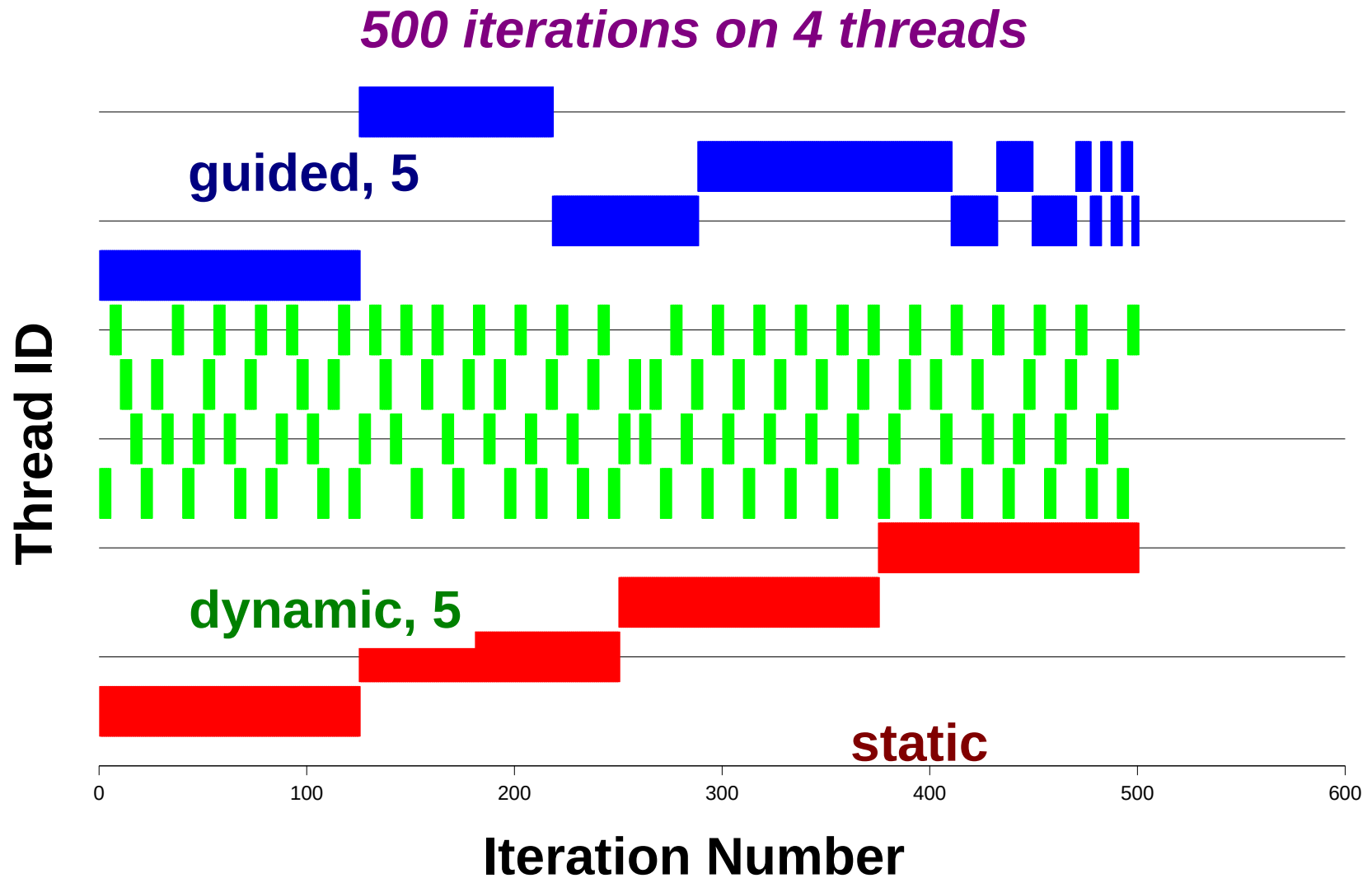
Guided schedule:

- ❑ The chunk size is exponentially reduced with each chunk that gets **dynamically** assigned to the threads; chunk defines the minimum number of iterations to assign each time.

$$\text{chunk} = \text{unass_iter} / (\text{weight} * \text{n_thr})$$

- ❑ The default minimum chunk size is 1.

OpenMP Scheduling



OpenMP Scheduling

```
#pragma omp for schedule(runtime)
```

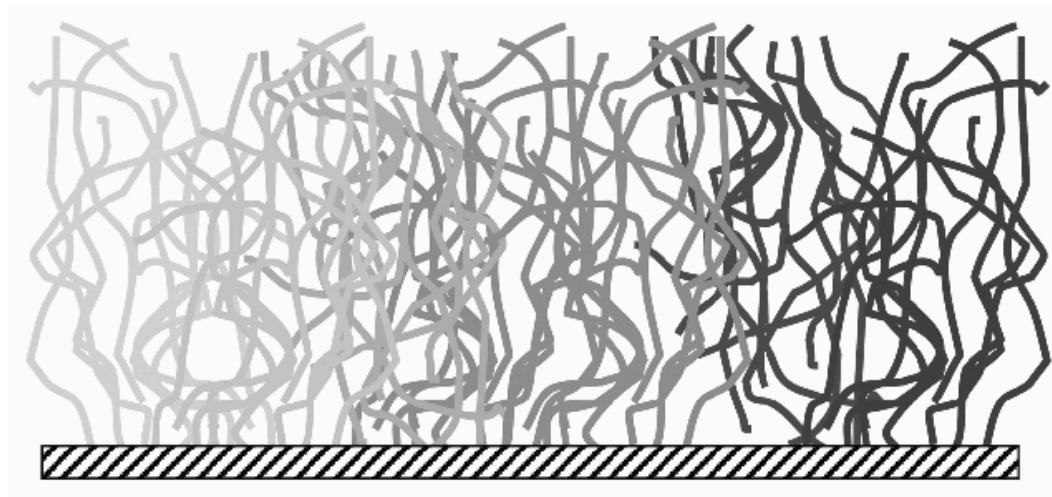
Runtime schedule:

- ❑ The schedule is detected at runtime from the setting of the OMP_SCHEDULE environment variable.
- ❑ Syntax: OMP_SCHEDULE=type,chunk

A real world example: Molecular Dynamics simulation

Example: MD simulation

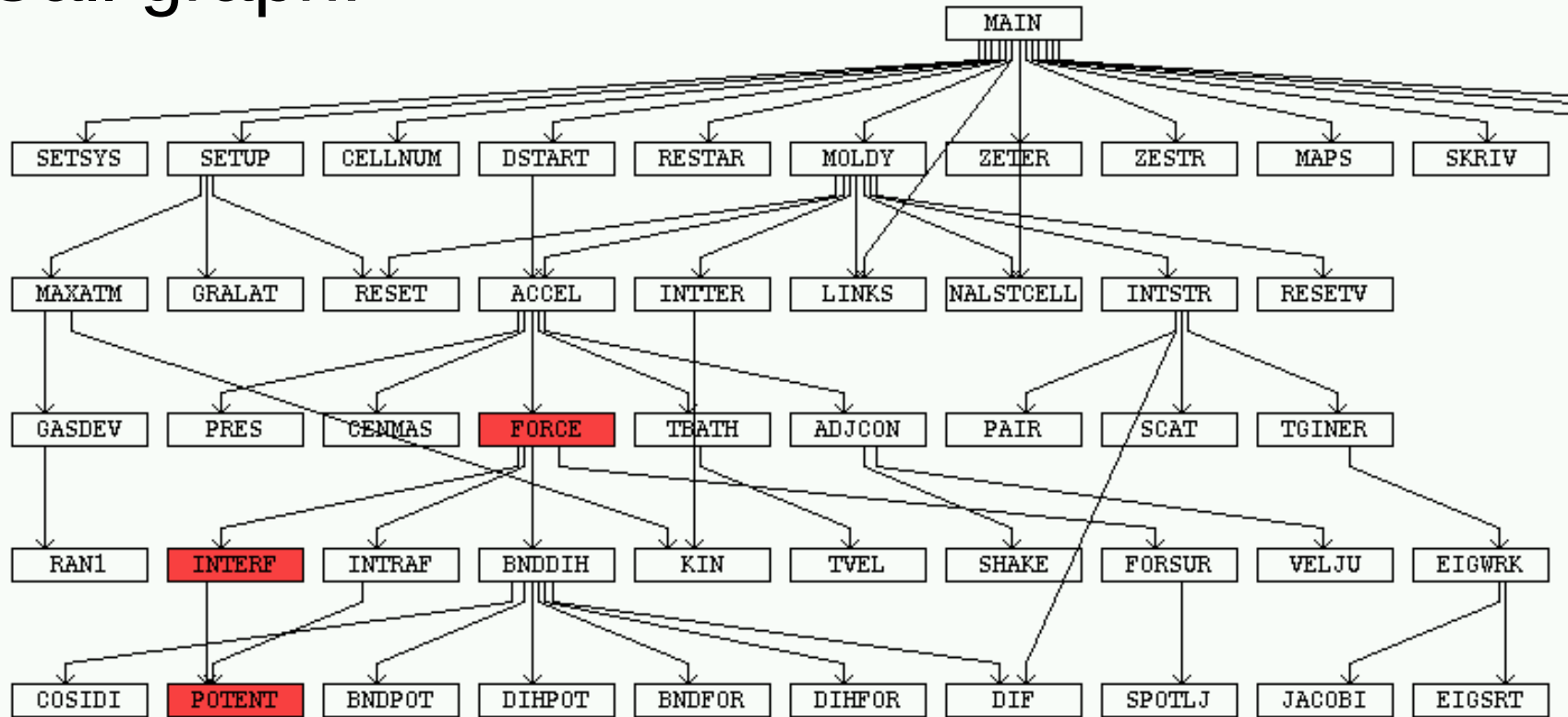
- ❑ Molecular Dynamics simulation of long carbon molecules on a surface:



- ❑ 7200+ lines of Fortran 77 code
- ❑ GOTOs, COMMON blocks, ...
- ❑ one source file

Example: MD simulation

Call graph:



more than 80% of the runtime are spent in the red part of the call graph

Example: MD simulation

- ❑ The loop to be parallelized contains a call to another subroutine.
- ❑ Data is passed the old Fortran style via COMMON blocks
- ❑ First try: Inserted one PARALLEL DO pragma in the code, using autoscoping, i.e. a feature of the Oracle Studio compiler
- ❑ The compiler generated a parallel version!

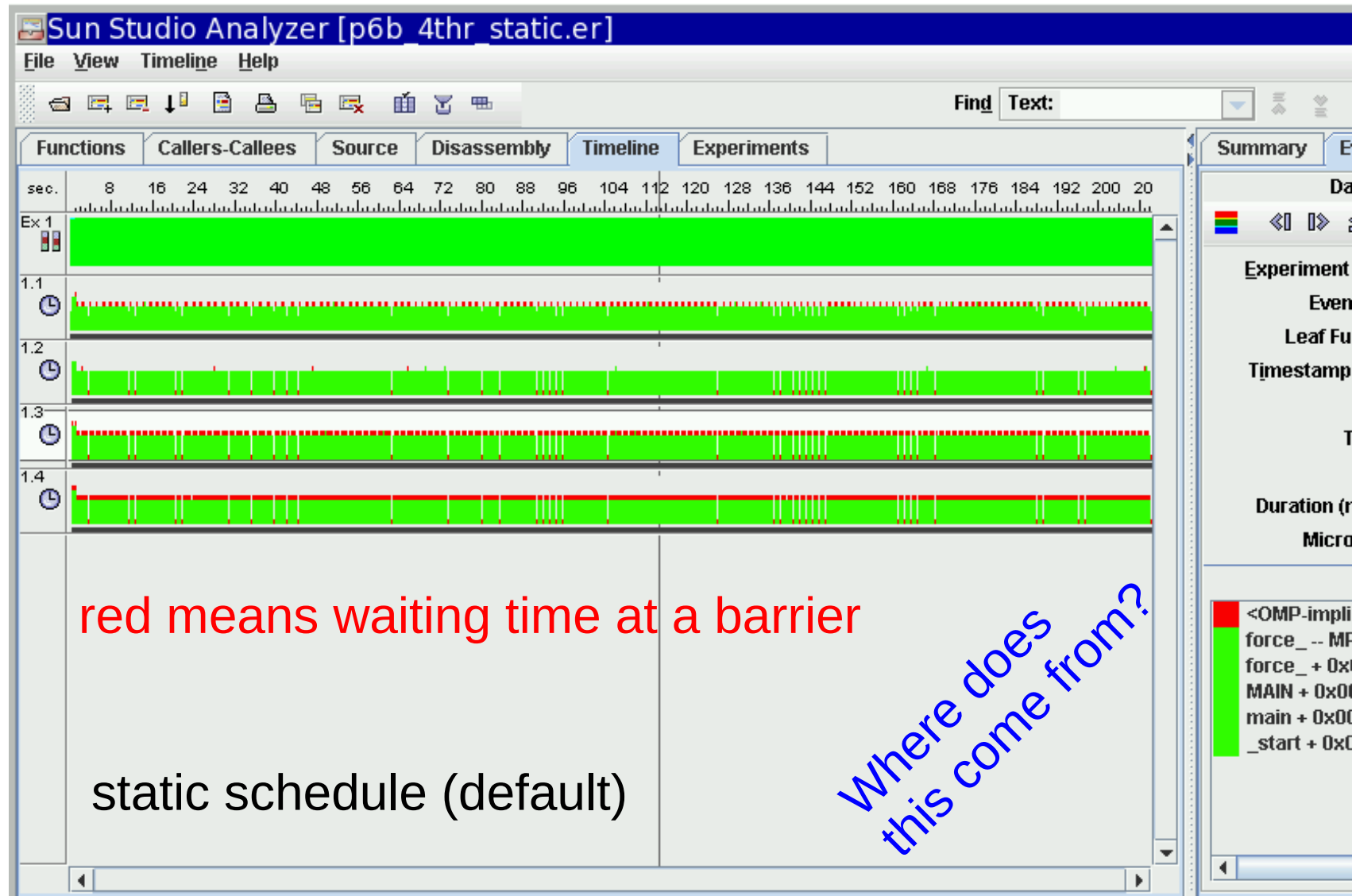
This took us by surprise!

Example: MD simulation

- ❑ First test runs:
 - ❑ It didn't scale ...
 - ❑ The results were dependent on the number of threads ...
- ❑ Thread analyzer revealed data races in two variables inside the called subroutine.
- ❑ Fix: Added additional scoping for those variables in the OpenMP pragma!
- ❑ This solved the data race problem.

Example: MD simulation

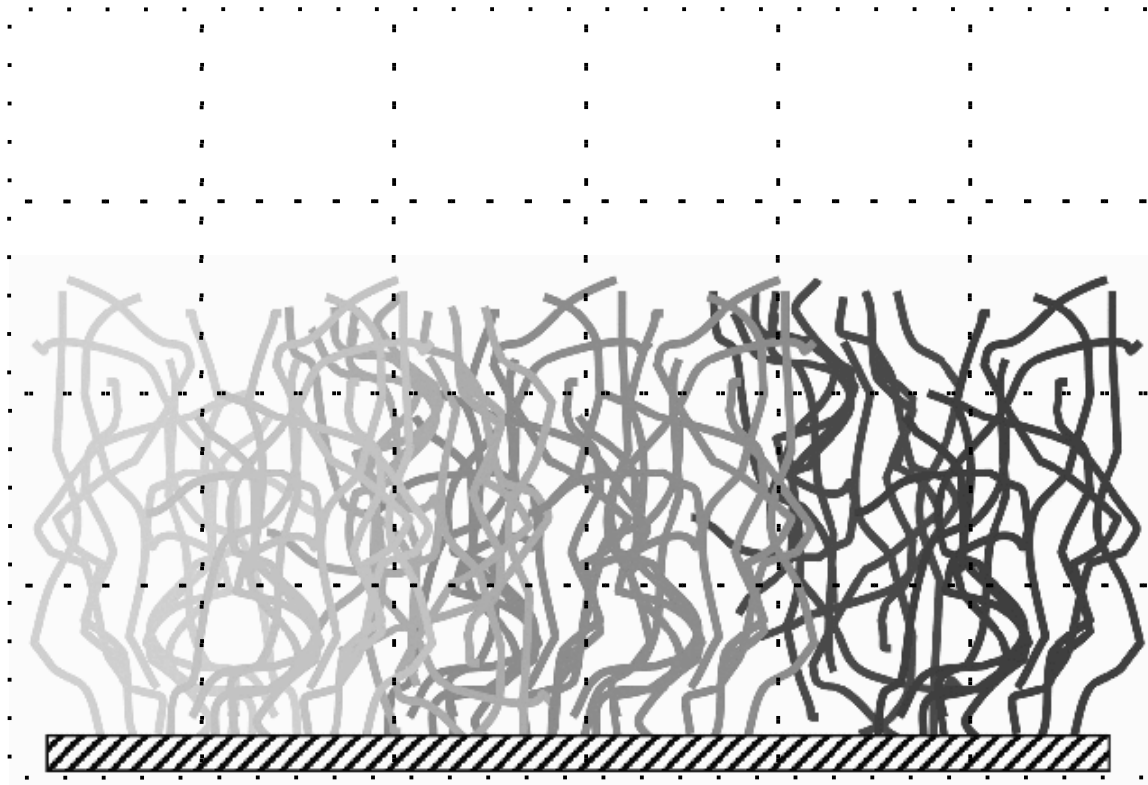
Analysis of the scaling problem:



Example: MD simulation

The simulation box:

seen from the side



← thread 4

← thread 3

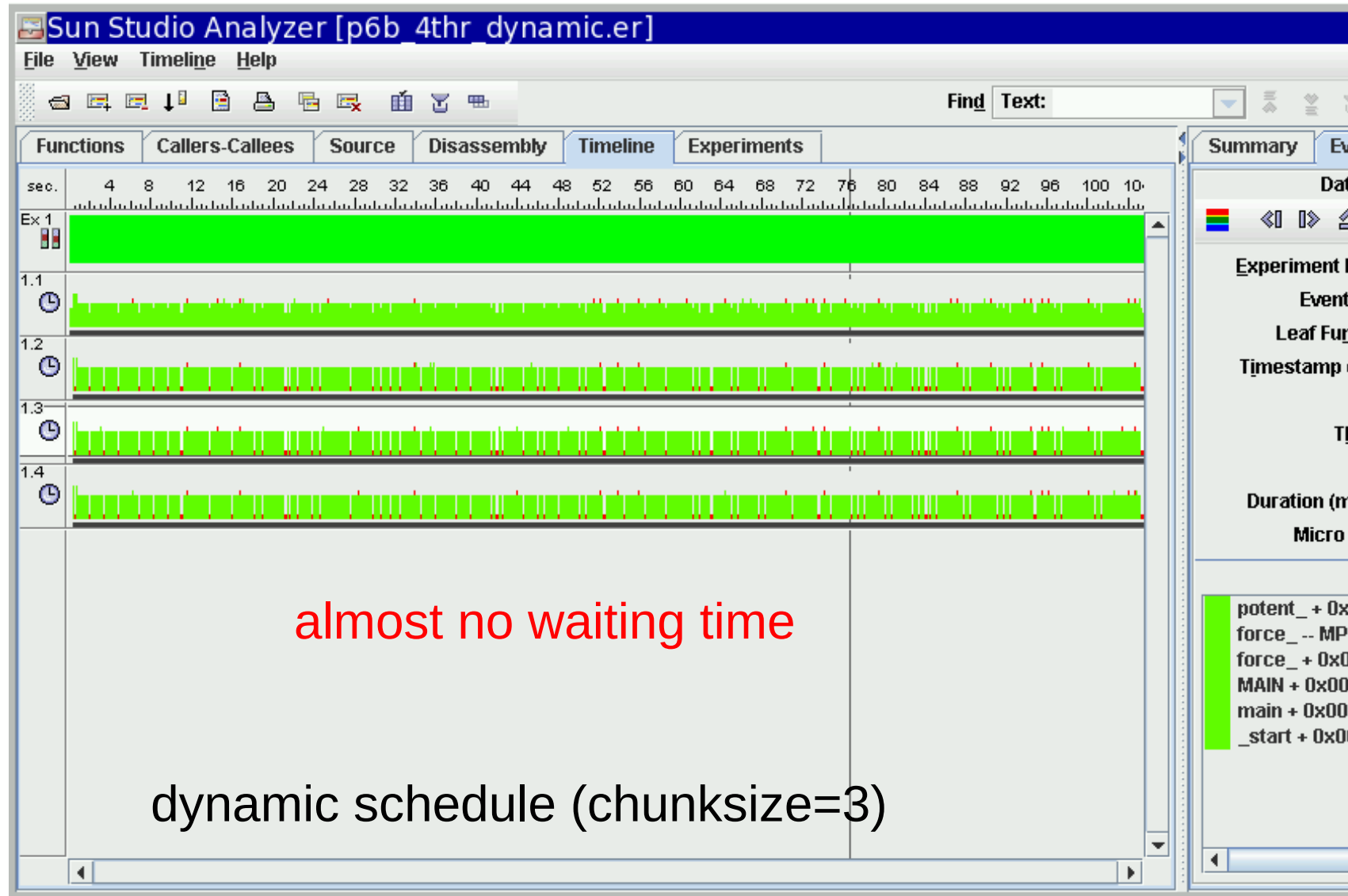
← thread 2

← thread 1

subdivision into smaller cells

Example: MD simulation

Adapted the schedule:



Example: MD simulation

Speed-up results:

