Dr. Cristian Mateos Diaz

Computación Paralela y Distribuida - 2022

Why OpenMP?

- Prominent user: UCAR (University Corporation for Atmospheric Research)
  - Non-profit consortium connecting +100 universities
  - Training and promotion of climate change research
  - Manages the NCAR (National Center for Atmospheric Research)
  - Sponsored by NSF (non-medical research)
  - NCAR/UCAR is to atmospheric and related sciences what NASA is to astronomy

NCAR Cheyenne Supercomputer

- 145,152 Intel Xeon processor cores
- 4,032 dual-socket nodes (36 cores/node)
- (36 cores/node)
- 313 TB of total memory
- 6 login nodes
- ping-pong latency < 1 µs

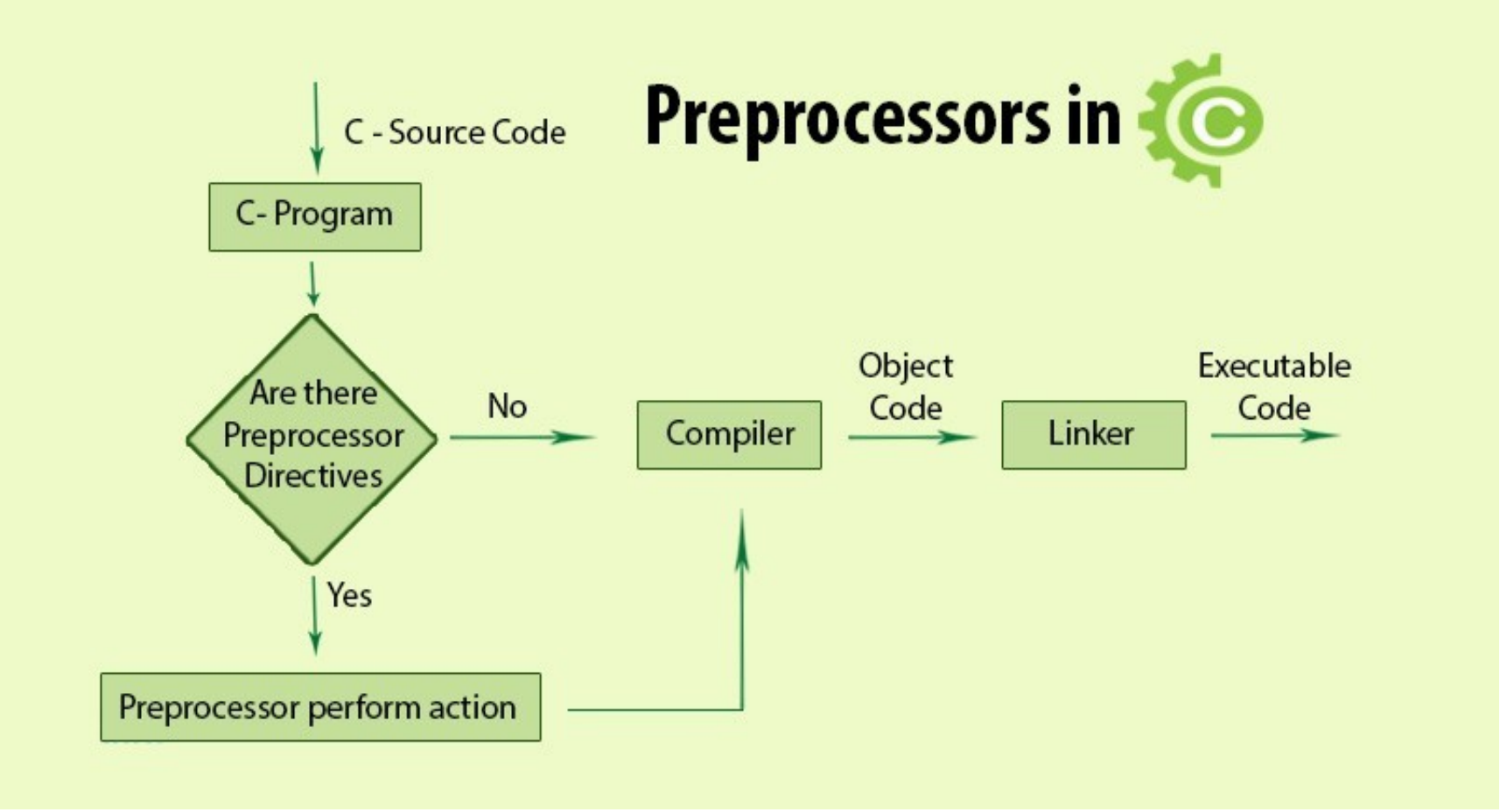
- https://www2.cisl.ucar.edu/resources/computational-systems/cheyenne/running-jobs/hyper-threading-cheyenne

OpenMP overview

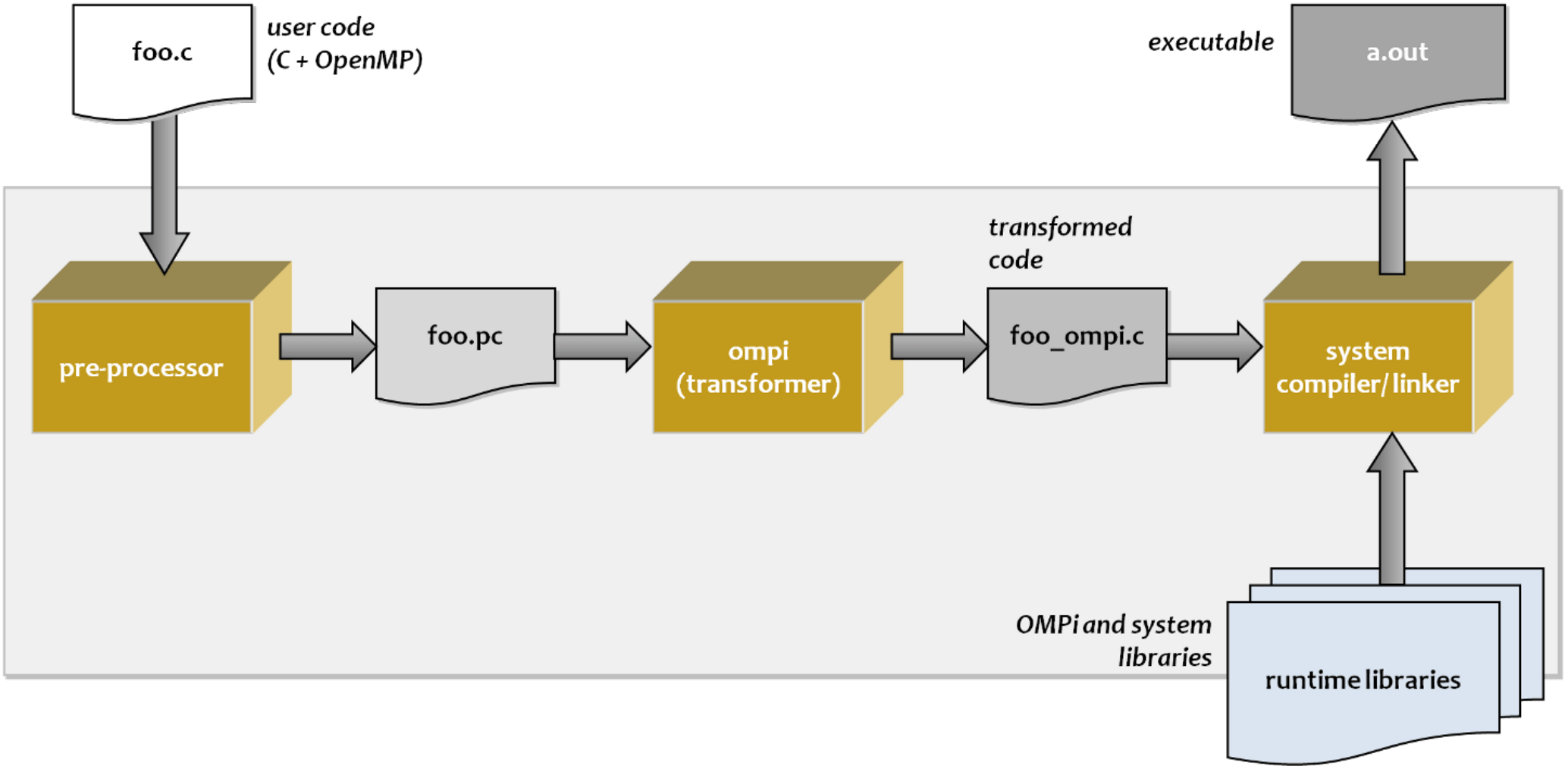
- OpenMP: An API to write multi-threaded applications
- A set of compiler directives and library routines for parallel application programmers
- Simplifies writing multi-threaded programs (MT) in several languages
- Standardizes last 20+ of SMP practice

Compiler directives: Recap

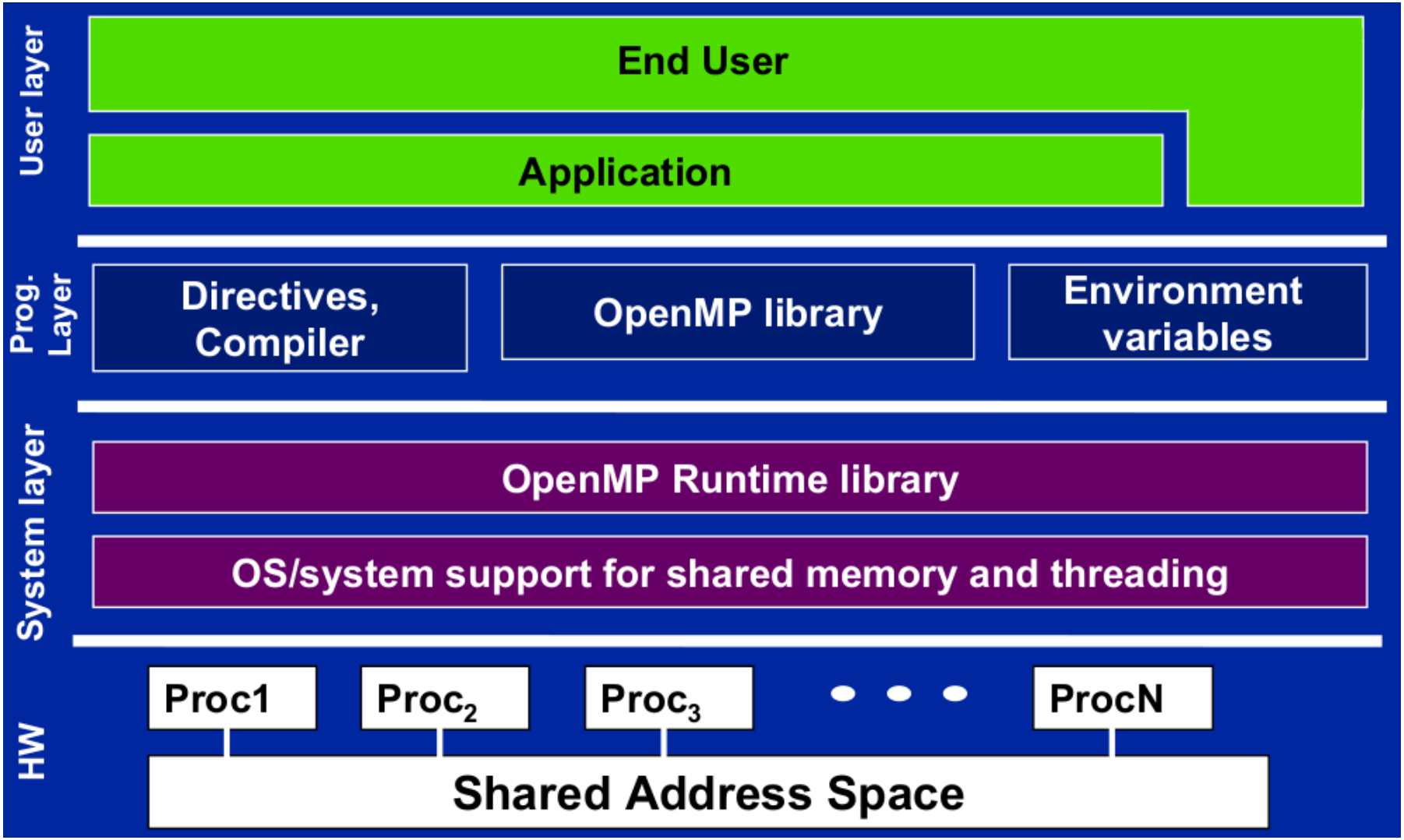
- Example 1: #include <stdio.h>
- Example 2: #define PI 3.1415926



Generating intermediate code



OpenMP software stack



Alternative: Interprocess-communication via messages (MPI)

OpenMP syntax

- Most of the constructs in OpenMP are compiler directives:
  - **#pragma omp construct [clause [clause]…]**
  - Example: **#pragma omp parallel num_threads(4)**
- Function prototypes and types in the file:

  #include <omp.h>
- Most OpenMP constructs apply to a “structured block”
  - 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

Installation (check video tutorial)

- Install Code::Blocks (http://www.codeblocks.org/)
  - Mac users would prefer another IDE…
- Install the GCC compiler
  - Nothing to do for Linux/Mac users (check you have ‘gcc’ and ‘g++’)
  - Windows users: Install MinGW (http://www.mingw.org/)
- Tell your IDE to use OpenMP
  - Install library (libomp, libomp-dev)
  - “-fopenmp” flag (compiler)
  - Link to library (“gomp” or MinGW folder)
- You might use http://www.omp4j.org plus e.g. Eclipse

Warming up A: Hello world

- First verify that your environment works by writing a hello world program:

```
void main()
{


    int ID = 0;

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

}
```

Warming up B: Hello world

- Then verify that your OpenMP environment works by writing a hello world program:

```
#include “omp.h”
void main()
{
 #pragma omp parallel
 {

    int ID = 0;

    printf(“ hello(%d) ”, ID);
    printf(“ world(%d) \n”, ID);
 }
}
```

- Compile it using: gcc -fopenmp -o exec_name source_name.c
- Each thread can print its own ID using **omp_get_thread_num();** inside the parallel construct

How do threads interact?

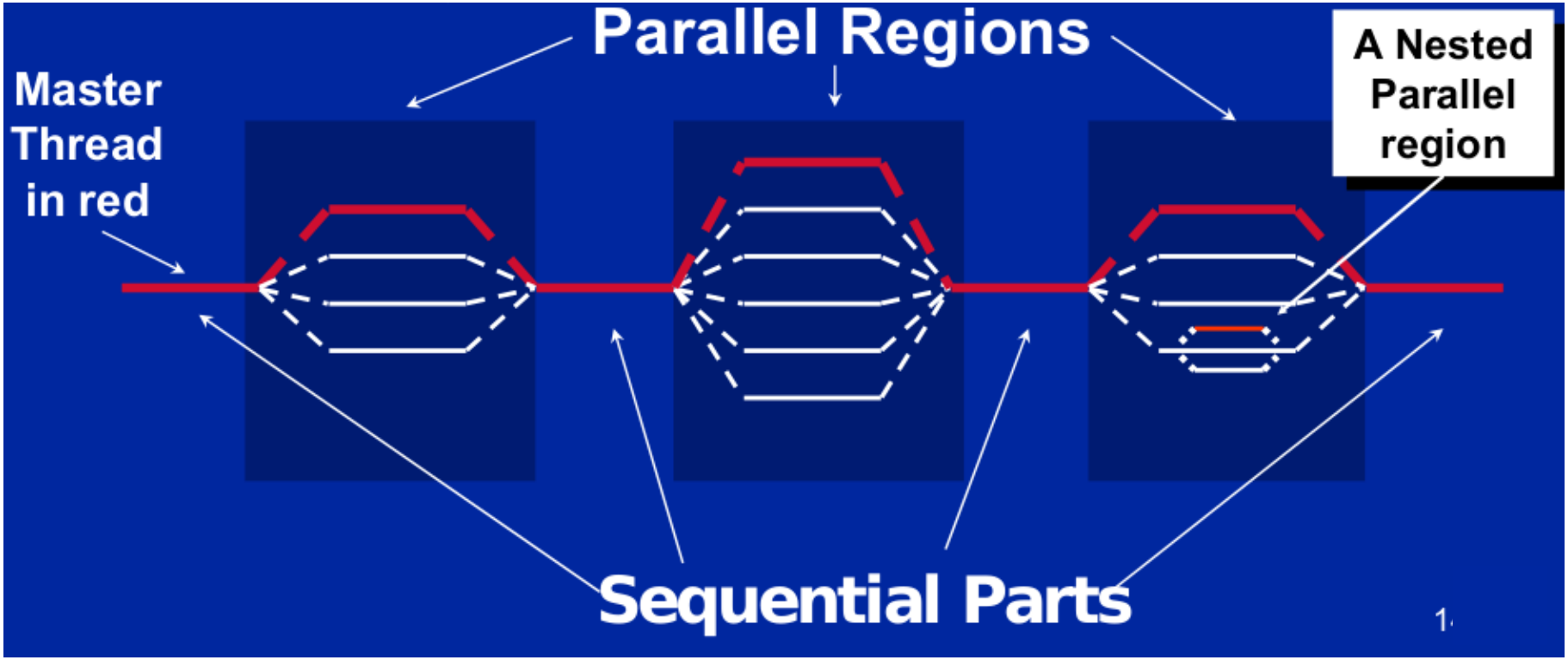
- OpenMP uses a multi-threading, shared address model
  - Threads are created implicitly
  - Threads communicate by sharing variables
- Unintended sharing of data causes race conditions:
  - *race condition*: when the program's outcome changes as the threads are scheduled differently
- To control race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

Outline

- Introduction to OpenMP
- **Creating Threads (you are here!)**
- Synchronization
- Parallel Loops
- Synchronize single masters and stuff
- Schedule your for and sections

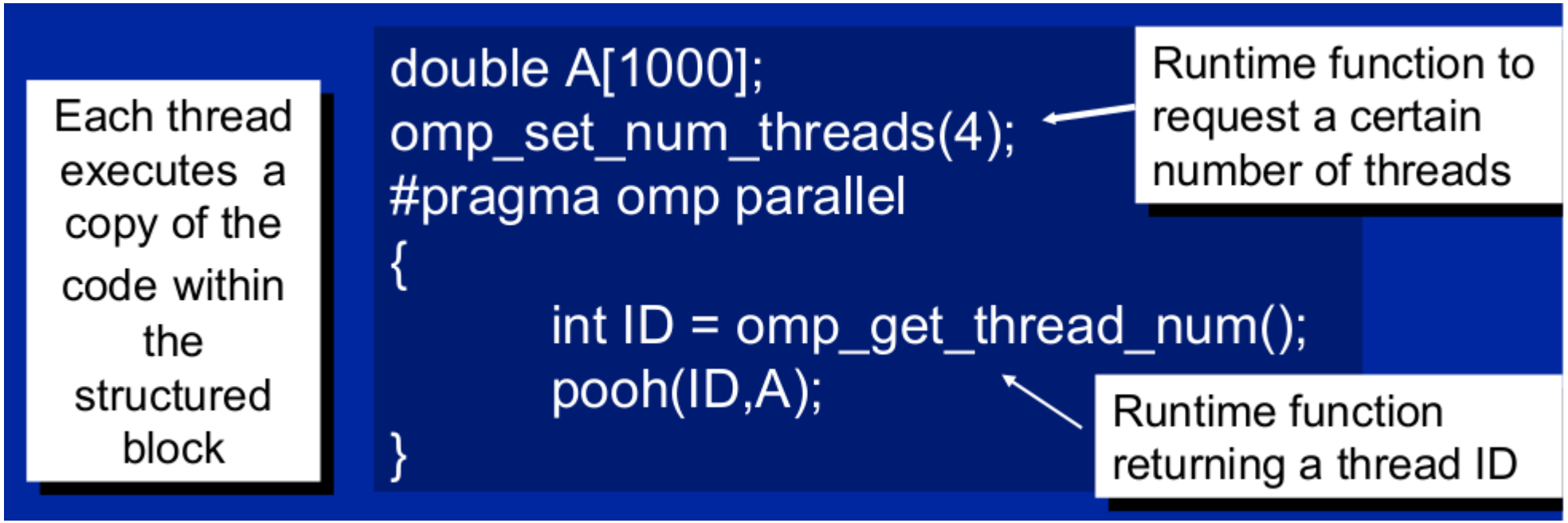
OpenMP Programming Model: Fork-Join Parallelism

- *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 one



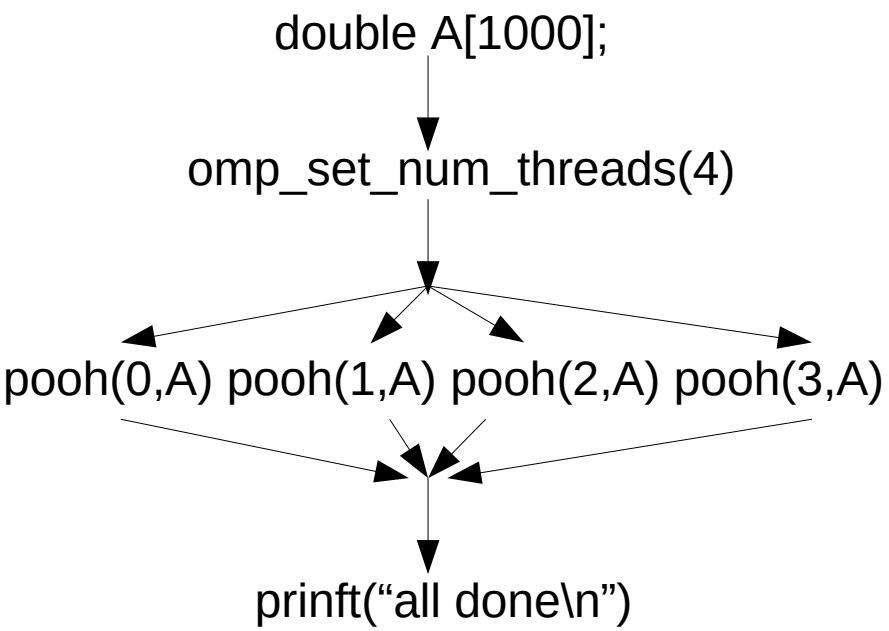
Thread Creation: Parallel Regions

- You “create” threads in OpenMP with the **parallel** construct
- Example: Creating a 4-thread parallel region:



Thread Creation: Parallel Regions example

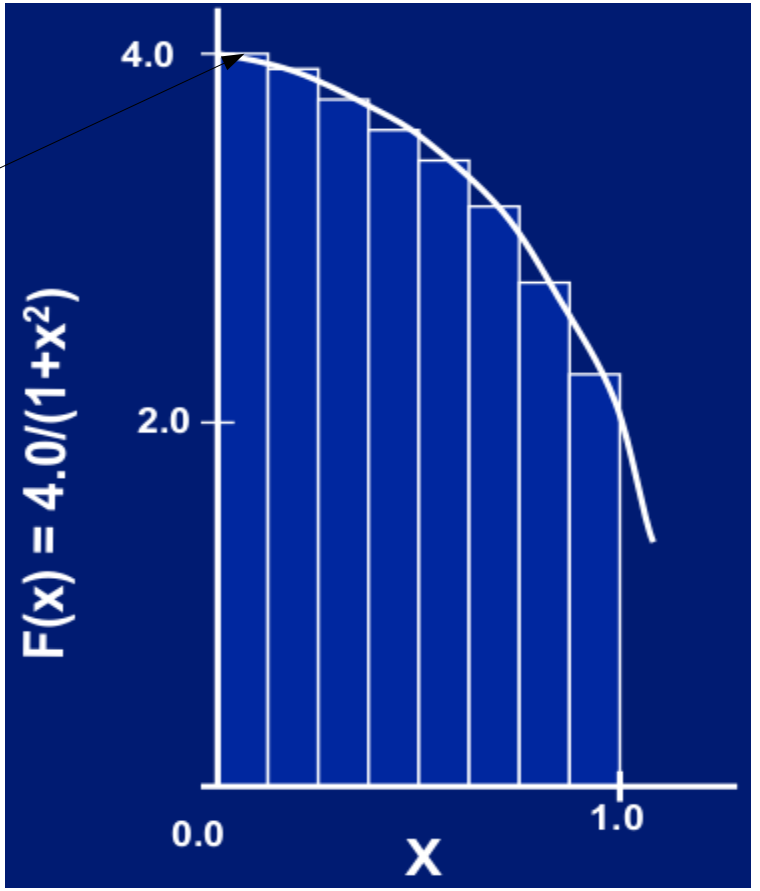
- A single copy of “A” is shared
- A *barrier* is established at the end of the region




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

Assignment: Numerical integration

- Mathematically, integrating 4.0/(1+X*X) in [0,1] equals pi
- We can approximate the integral as a sum of rectangles:
  - ∑ F(Xi) ΔX ≈ pi, i={0..N}
  - Each rectangle has width ΔX and height F(Xi) at the middle of interval i



Assignment: Numerical integration

```
static long num_steps = 100000;
double step;
void 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;
}
```

Assignment: Numerical integration

- Create a parallel version of the pi program using a parallel construct
- Print running time per thread
- Pay close attention to shared versus private variables
- In addition to a parallel construct, you will need the runtime library routines:
  - *int omp_get_num_threads();*
  - *int omp_get_thread_num();*
  - *double omp_get_wtime();*

Outline

- Introduction to OpenMP
- Creating Threads
- **Synchronization (you are here!)**
- Parallel Loops
- Synchronize single masters and stuff
- Schedule your for and sections

Synchronization

- Used to impose order constraints and to protect access to shared data
- High level synchronization
  - Critical
  - Atomic
  - Barrier
  - Ordered

Synchronization: critical

- Mutual exclusion as in a binary semaphore
- #pragma omp critical(someName) → named section
- Beware! All the unnamed critical sections are mutually exclusive

```
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
           consume (B, res);
     }
}
```

Synchronization: atomic

- Mutual exclusion but only to protect updates of variables
- Much faster compared to critical; it might take advantage of low-level instructions for e.g. atomic variable + 1

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

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- **Parallel Loops (you are here!)**
- Synchronize single masters and stuff
- Schedule your for and sections

SPMD versus worksharing

- A parallel construct by itself creates an SPMD (Single Program Multiple Data) program from a sequential one
- Each thread redundantly executes the same code
- How do you split up pathways through the code between threads within a team? → **worksharing**
  - Loop construct
  - Sections/section constructs (discussed later)
  - Single construct (discussed later)

A note on variables

- shared(list) → limits what is shared
- private(var) → *var* copy initialized randomly in the region
- firstprivate(var) → *var* is initialized with the value it has before the parallel region
- lastprivate(var) → the value of the last thread after the region is stored

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

int main (void)
{
    int i = 10;

    #pragma omp parallel private(i)
    {
        printf("thread %d: i = %d\n", omp_get_thread_num(), i);
        i = 1000 + omp_get_thread_num();
    }

    printf("i = %d\n", i);

    return 0;
}
```

The loop construct

- Splits up loop iterations among the threads in a team
- In the example below, the variable “I” is made “private” to each thread by default (you could do this explicitly with a “private(I)” clause in other constructs as discussed)

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

The loop construct: Motivating example

- Sequential for

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

- Parallel construct

```
#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];}
}
```

- Loop construct

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

Combining loop and parallel constructs

```
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 codes are functionally equivalent
- But, barriers are implicitly place at the end of each construct

Exploiting the for construct

- Find compute intensive loops
- Make the loop iterations independent so they can safely execute in any order without loop-carried dependencies
- Place the appropriate directive, run and test

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

Remove dependency

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

Reduction

- See this example:

```
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; this true dependence can’t be trivially removed
- This is a common situation and is called **reduction**
- Both arithmetic and logical reductions are supported

Reduction

- To specify a reduction, we use **reduction (op : list)**
  - A local copy of each list variable is made and initialized depending on the "op" (e.g. 0 for "+")
  - Compiler finds standard reduction expressions containing "op" and uses them to update the local copy
  - Local copies are reduced into a single value and combined with the original global value
  - Variables in "list" must be shared in the region

```
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;
```

Assignment: loop construct

- I will ask you to go back to the serial pi program and parallelize it with a loop construct and reductions
- I will ask you to perform as few changes to the serial program as possible

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- **Synchronize single masters stuff (you are here!)**
- Schedule your for and sections

Synchronization: Barrier

- Each thread that encounters this pragma must wait until all threads in the team have arrived
- The code below has both **explicit** and **implicit** barriers

```
#pragma omp parallel shared (A, B, C) private(id)
{
        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);
}
```

Synchronization: Master

- The master construct denotes a structured block that is only executed by the master thread (every region has one!)
- The other threads just skip it (no synchronization is implied)
- The master construct can be used inside a loop construct

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

Synchronization: Single

- The single construct denotes a block of code that is executed by any but only one thread
- A barrier is implied at the end of the single block
- Can remove the barrier with a *nowait* clause

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

Synchronization: Ordered

- The ordered region executes in the sequential order
- Affects next statement only
- E.g. print results for each “I”

```
#pragma omp parallel private (tmp)
#pragma omp for ordered reduction(+:res)
        for (I=0;I<N;I++){
                tmp = NEAT_STUFF(I);
#pragma ordered
                res += consum(tmp);
        }
```

Runtime library routines

- Modifying/checking the number of threads
  - *omp_set_num_threads()*, *omp_get_num_threads()*, *omp_get_thread_num()*, *omp_get_max_threads()*
- Are we executing code in an active parallel region?
  - *omp_in_parallel()*
- Do you want the system to dynamically vary the number of threads from one parallel construct to another?
  - *omp_set_dynamic()*, *omp_get_dynamic()*
- How many processors in the system?
  - *omp_num_procs()*
- *… plus a few less commonly used routines*

Runtime library routines

- To use a known, fixed number of threads in a program, (1) tell OpenMP that you don't want dynamic adjustment of threads, (2) set the number of threads, then (3) save the number you got (you might actually get less than expected)
- Note we are protecting the "num_threads" variable

```
#include <omp.h>
void main()
{   int num_threads;
     omp_set_dynamic( 0 );
     omp_set_num_threads( omp_num_procs() );
#pragma omp parallel
   {    int id=omp_get_thread_num();
#pragma omp single
            num_threads = omp_get_num_threads();
          do_lots_of_stuff(id);
    }
}
```

Environment variables

- *OMP_NUM_THREADS* stores the global default number of threads to use in parallel constructs
- Program output:
  - *OMP_NUM_THREADS (e.g. “4”)*
  - *8*
  - *2*
  - *8*
- *OMP_SCHEDULE “schedule[, chunk_size]”*
  - *schedule → e.g. dynamic, static*
  - *chunk_size → default “1”*

```
int main() {
    #pragma omp parallel
    {
        #pragma omp single
        {
            printf("%d\n", omp_get_num_threads());
        }
    }
    omp_set_num_threads(8);
    #pragma omp parallel
    {
        #pragma omp single
        {
            printf("%d\n", omp_get_num_threads());
        }
    }
    #pragma omp parallel num_threads(2)
    {
        #pragma omp single
        {
            printf("%d\n", omp_get_num_threads());
        }
    }
    #pragma omp parallel
    {
        #pragma omp single
        {
            printf("%d\n", omp_get_num_threads());
        }
    }
}
```

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Synchronize single masters and stuff
- **Schedule your for and sections here!)**

Schedule your for and sections

- sections construct gives a different block to each thread in the team
- Some threads in the team might have nothing to do
- By default, there is a barrier at the end of the “omp sections” → use the “nowait” clause to turn off the barrier

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

The schedule clause

- The schedule clause affects how loop iterations are mapped onto threads
  - *schedule(static [,chunk])* → assign blocks of iterations of size “chunk” to each thread
  - *schedule(dynamic [,chunk])* → each thread grabs “chunk” iterations off a queue until all iterations have been handled
  - *schedule(guided [,chunk])* → threads dynamically grab blocks of iterations. The size of the block starts large and shrinks down to size “chunk” as the calculation proceeds
  - *schedule(runtime)* → schedule and chunk size taken from the OMP_SCHEDULE environment variable
  - Omp4j supports static and dynamic scheduling

Static vs dynamic scheduling

- Example: Assume that each thread gets assigned two (blocks of) iterations and these blocks take gradually less and less time, with 8 block in total
  - Dynamic scheduling might provide better **load balance**