Introduction to OpenMP

Mohammad Mahdi Javanmard

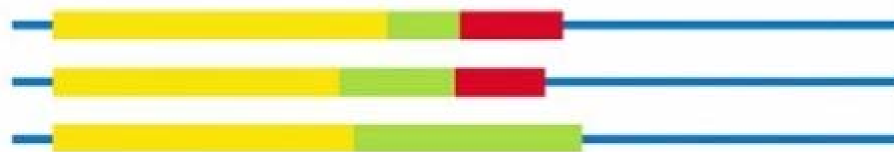
Acknowledgement: Contents Courtesy Tim Mattson @ intel

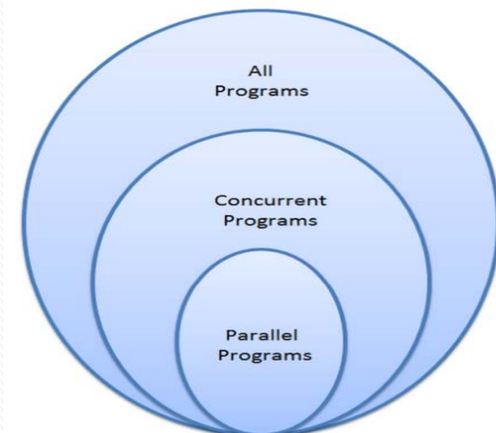
What is OpenMP?

- *OpenMP: An API for Writing Multithreaded Applications on Shared Memory model*
    - A set of compiler directives, library routines and environment variables for parallel application programmers
    - Makes it easy to create multi-threaded (MT) programs in Fortran, C and C++
    - Standardizes last 15 years of SPMD (Single Program Multiple Date) practice
- But, beforehand …
    - Difference between Concurrent and Parallel Programs
        - Concurrent: a condition of a system in which multiple tasks are *logically* active at one time (they *can* be active at one time)
        - Parallel: a condition of a system in which multiple tasks are *acutally* active at one time (they *are* active at one time).

What is OpenMP?

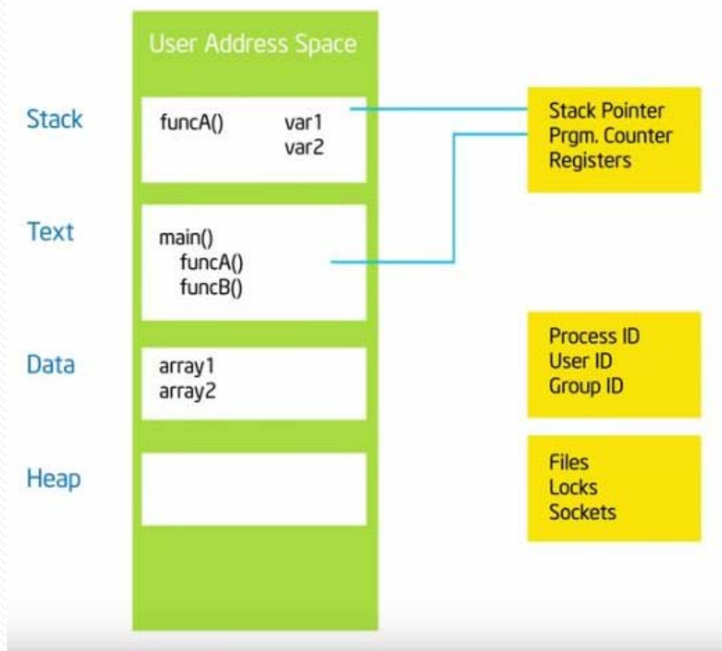
- But, beforehand …
  - Difference between Concurrent and Parallel Programs:



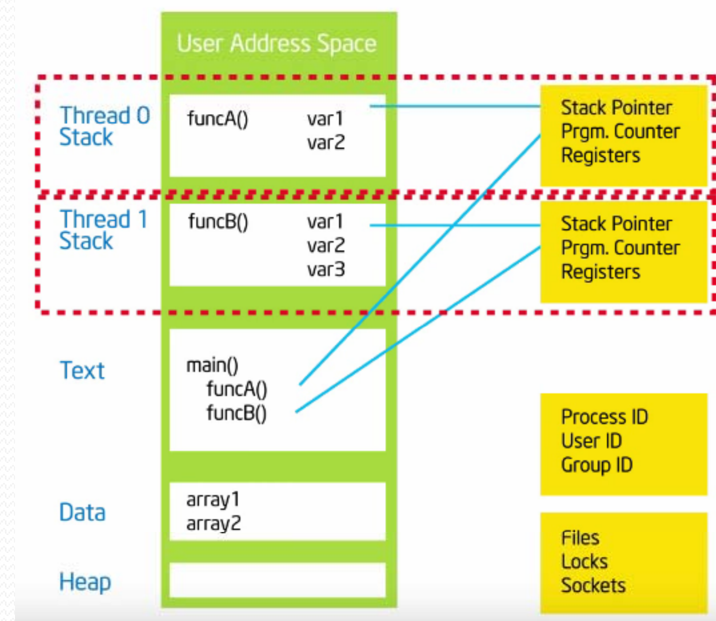


What is OpenMP?

- But, beforehand …
  - Process? An instance of a program execution.
  - Thread? A path of execution within a process (i.e., light weight processes).

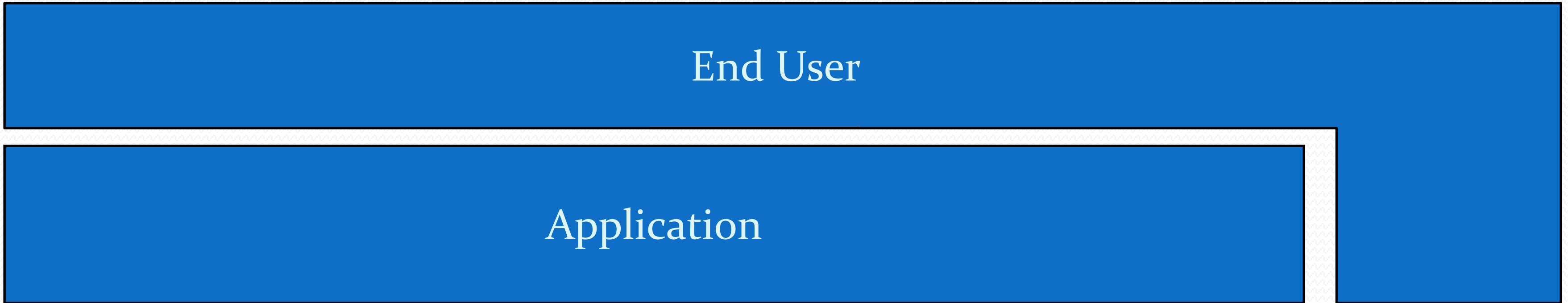


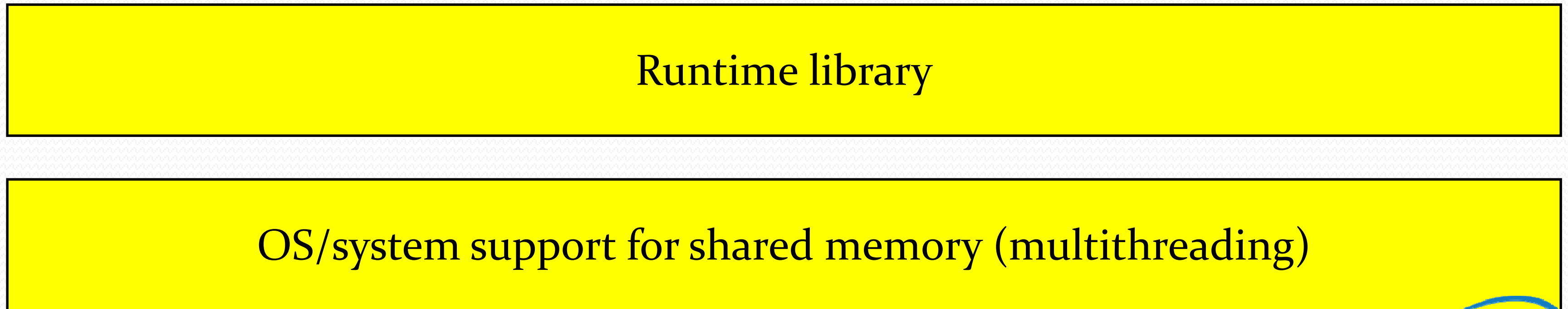
Process



Multi-threaded Process

OpenMP Parallel Computing Solution Stack





OpenMP Basic Concepts

- OpenMP Constructs
  - General Syntax:
    - #pragma omp construct [clause [clause] …]
      - Example:
        - #pragma omp parallel num_threads(4)
      - All functions, constructs, data types defined in the header file omp.h
        - #include "omp.h"
- Structured Blocks:
  - A block of one or more statements with one point of entry at the top and one point of exit at the bottom.
  - However, it is OK to have exit() within the structured block.

Running OpenMP Programs

- As it is widely used, it is already built inside gcc:
  - gcc –**fopenmp** FILE_NAME.c
  - Assuming you are linux/mac user:
- The most important Construct:
  - **#pragma omp parallel**
    - Creates bunch of threads and run the code in its corresponding block (parallel region) in parallel
    - Example:

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

void main() {
#pragma omp parallel
{
    int id = omp_get_thread_num();
    printf("hello(%d)", id);
    printf("world(%d)\n", id);
}
}
```

Sample Execution with 8 TH

```
hello(2)hello(4)hello(7)hello(3)world(4)
hello(0)world(0)
hello(5)world(5)
world(7)
world(3)
world(2)
hello(6)world(6)
hello(1)world(1)
```

Running OpenMP Programs

- The most important Construct:
  - **#pragma omp parallel**
    - Creates bunch of threads and run the code in its corresponding block (parallel region) in parallel.
    - Without this construct, you can’t get multithreaded execution.
    - Example:

```
double A[1000]; // resides in the heap (public, shared among threads)
// omp_set_num_threads(4); // REQUESTING for 4 threads -- default: number of cores
#pragma omp parallel num_threads(4) // the only way to create multi-thraded program
{
    int id; // resides in stack of each thread (private, local)
    id = omp_get_thread_num();
    pooh(id, A);
}
printf("done!\n");
```

Running OpenMP Programs

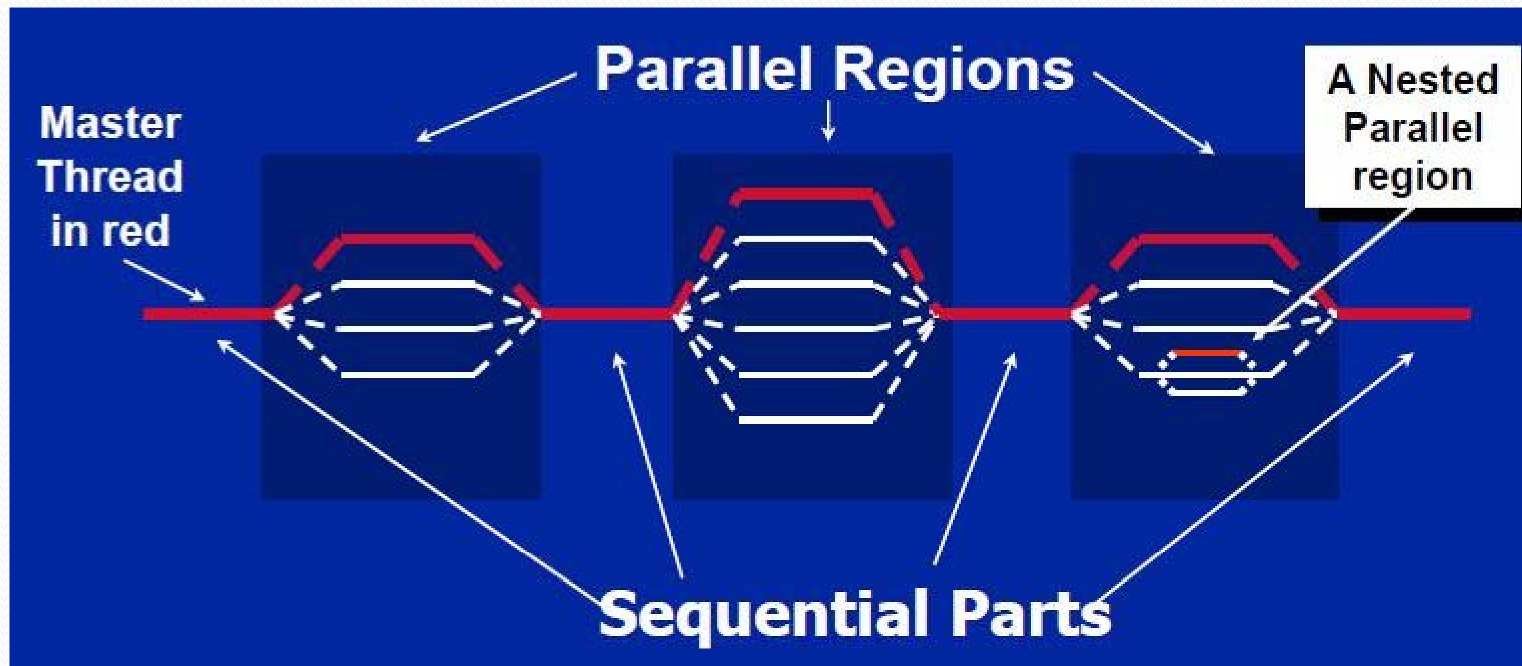
- The parallel program written in OpenMP runs slower on serial machine than the serial program (without OpenMP). Why?

Running OpenMP Programs

- The parallel program written in OpenMP runs slower on serial machine than the serial program (without OpenMP). Why?
  - Having Parallel runtime overhead
    - Setting a runtime environment(s) such as number of threads etc
    - Creating threads

OpenMP Parallel Model

- The same as intel cilk, it is based on fork join model:
  - Master thread (id = 0) spawns a team of threads as needed.



OpenMP Program, some important notes

- OpenMP program gets translated to pthread program ultimately:

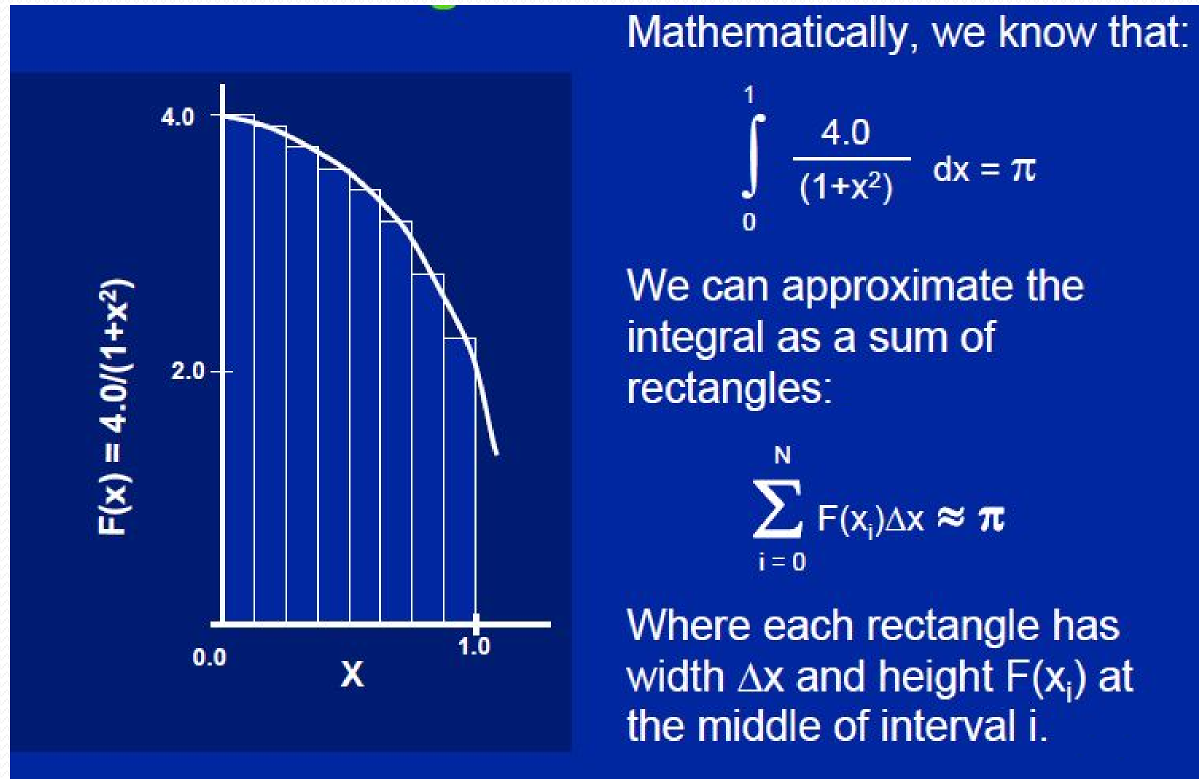
```
#pragma omp num_threads(4)
{
    foo();
}
```

```
void thunk() {
    foo();
}
pthread_t tid[4];
for (int i = 1; i < 4; ++i)
        pthread_create(&tid[i], 0, thunk, 0); // fork part

thunk(); // for the master thread
for (int i = 1; i < 4; ++i)
        pthread_join(tid[i]); // synchronization - join part
```

- In OpenMP, you always request bunch of threads. However, environment might give you less !!!
  - So, you can't always assume that you have requested number of threads !!

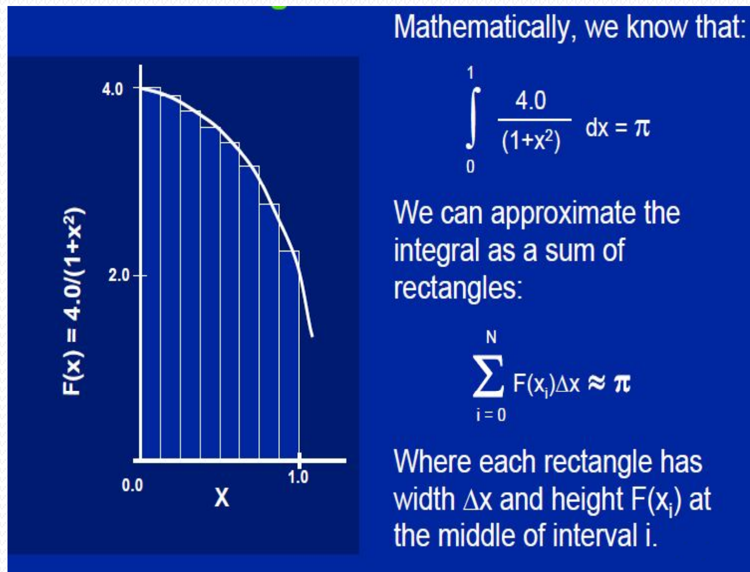
OpenMP Program, some important notes

- In OpenMP, you always request bunch of threads. However, environment might give you less !!!
  - Example:



OpenMP Program, some important notes

- Solution in OpenMP [SPMD pattern]:



```
#include "omp.h"
static long num_steps = 100000;
double step;
#define NUM_THREADS 4
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_set_num_threads();
        if (id == 0) nthreads = nthrds;
        for (i = id, sum[id] = 0.0; i < num_steps; i+= nthrds)
        {
            x = (i + 0.5) * step;
            sum[id] += 4.0/(1.0 + x * x);
        }
    }

    /* NUM_THREADS is dangerous to use rather than nthreads
    because, you are not sure, if you are given NUM_THREADS threads !!
    */
    for (i = 0, pi = 0.0; i < nthreads; ++i)
        pi += sum[i] * step;
}
```

OpenMP Synchronization Constructs

- Different synchronization constructs:
  - **#pragma omp barrier**:
    - All the threads wait until everyone reaches:

```
#pragma omp parallel
{
    int id = omp_get_thread_num();
    A[id] = big_calculation1(id);

    #pragma omp barrier
    B[id] = big_calculation2(id);
}
```

  - **#pragma omp critical**:
    - Only one thread at a time can execute the code in the block of critical.

OpenMP Synchronization Constructs

- Different synchronization constructs:
  - **#pragma omp critical**:
    - Only one thread at a time can execute the code in the block of critical.
    - If someone else is in the critical section, I will wait.
    - Used to resolve race condition issue.
    - Example:

```
float res = 0.0;
#pragma omp parallel
{
    float B = 0.0;
    int i, id, nthrds;
    nthrds = omp_get_num_threads();
    for (i = id; i < nthrds; i += nthrds)
        B += big_calculation(i);
    #pragma omp critical
        res += consume(B);
}
```

OpenMP Synchronization Constructs

- Different synchronization constructs:
  - **#pragma omp critical**:
    - Use it very carefully. Excessive use of this will serialize your program and hence, kill the performance of your program:
    - Example of bad practice:

```
float res = 0.0;
#pragma omp parallel
{
    float B;
    int i, id, nthrds;
    nthrds = omp_get_num_threads();
    for (i = id; i < nthrds; i += nthrds) {
        #pragma omp critical
            res += big_calculation(i);
    }
}
```

  - **#pragma omp atomic**:
    - Very similar to the critical.
    - Search more about it ☺

OpenMP Parallel for loop

- One of the work-sharing constructs.
  - Always, there is an implied/default barrier at the end of work-sharing constructs (to be discussed further later)
- Syntax:
  - **#pragma omp for**
    for (…) {
    }
- Semantic:
  - Take the immediately following for loop, split up its iterations among the threads, give them to execute them.
  - Example:

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

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

OpenMP Parallel for – Loop Scheduling

- How should OpenMP splits up the iterations of the for loop among the threads?
  - You, as a programmer, should decide.
  - How? By using different possible *schedule* clauses:
    - schedule (static[,chunk])
    - schedule (dynamic [,chunk])
    - schedule (guided [,chunk])
    - schedule (runtime)
    - schedule (auto)

OpenMP Parallel for – Loop Scheduling

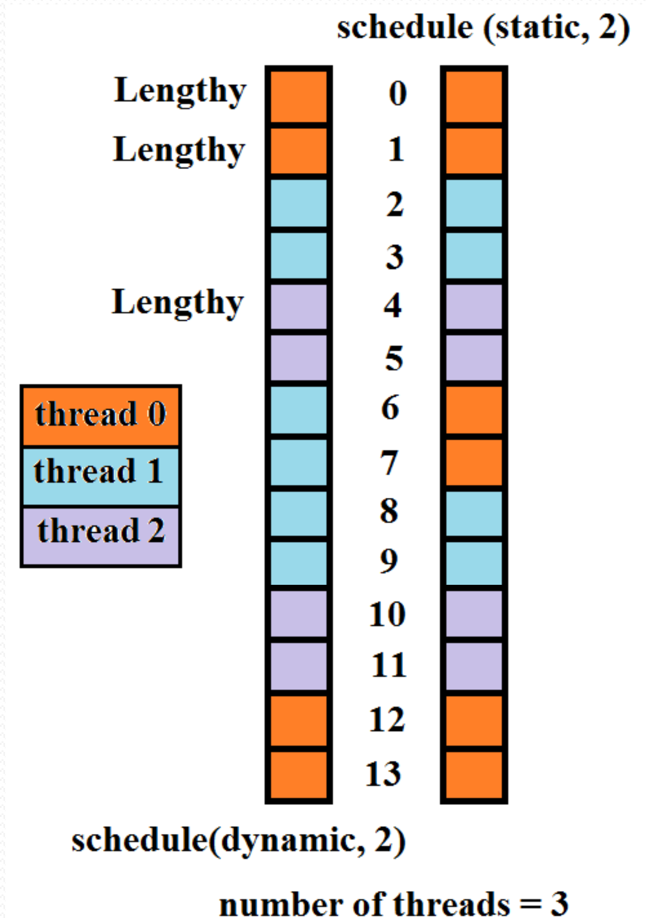
- **schedule (static[,chunk])**
  - At **compile time**, you determine the number of iterations to be given to each of the threads !
  - Used when having predictable and similar work per iterations
  - Default chunk size = (# of iterations/# of threads)
  - Example:
    - #pragma omp parallel for schedule (static, 4)
    - Number of threads = 3

static

OpenMP Parallel for – Loop Scheduling

- **schedule (dynamic[,chunk])**
  - At **run time**, you take the loop iterations & put them in the logical task queue.
  - Threads grab the iterations to execute them at **run time**.
  - Used when having unpredictable, highly variable work per iteration.
  - Default chunk size =

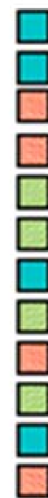
(# of iterations/# of threads)



OpenMP Parallel for – Loop Scheduling

- **schedule (guided[,chunk])**
  - Similar to dynamic. However, the size of the blocks of iterations (to be given to the threads) shrink as the calculation proceeds.
  - Search more about it ☺
- **schedule (runtime)**
  - You, as the programmer or user, will pass the schedule information at runtime (either through runtime library routines or environment variables)

guided(1)



OpenMP Parallel for – Loop Scheduling

- **schedule (auto)**
  - You rely on the compiler and you give it the permission to be flexible on how it does the schedule.

OpenMP Parallel loop –Important Point

- **Be very careful of loop-carried dependencies.**
  - Example:

```
int i, A[MAX];
int j = 5;
#pragma omp parallel for
for (i = 1; i < MAX; ++i) {
    j += 2; // loop-carried dependency
    A[i] = big_calculation(j);
}
```

  - How to resolve?

```
int i, A[MAX];
int j = 5;
#pragma omp parallel for
for (i = 1; i < MAX; ++i) {
    j = 5 + 2 * (i + 1); // NO loop-carried dependency
    A[i] = big_calculation(j);
}
```

OpenMP Parallel for loop – Using Reductions

- One way to resolve race condition is to use reduction:
  - The Problem:
    - Variable *ave* is shared among threads !!

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

  - The solution:

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

    - By having the clause reduction (+:ave), the compiler creates a local copy of variable *ave* (for each thread), initialize it with 0 and does the required computation. After the thread execution is done, it combines the local copies to the shared global one !

OpenMP Parallel for loop – Using Reductions

- One way to resolve race condition is to use reduction:
  - Using Reductions:
    - List of variables in the reduction clause get initialized with the identity value:
      - + → 0
      - * → 1
      - Min → largest positive number
      - Max → least negative number

OpenMP Parallel for loop – nowait clause

- As mentioned before, always, there is an implied/default barrier at the end of work-sharing constructs.
  - However, by using the clause *nowait*, you can eliminate the implied/default barrier.
  - Example:

```
#pragma omp parallel shared(A, B, C) private(id)
{
    id = omp_get_thread_num();
    A[id] = big_calculation(id);

    #pragma omp barrier
    #pragma omp for
        for (i = 0; i < N; ++i) {C[i] = big_calculation2(i, A);}
    // IMPLIED BARRIER

    #pragma omp for nowait
        for (i = 0; i < N; ++i) {B[i] = big_calculation3(C, i);}
    // NO IMPLIED BARRIER

    A[id] = big_calculation4(id);
}
```

OpenMP Master Construct

- By using master construct, only the master thread will do the corresponding block and others will just skip it.
  - There is NO barrier for other threads to wait for the master thread to finish the master block !!!!

```
#pragma omp parallel
{
    do_many_things();

    #pragma omp master
    {
        exchange_boundaries();
    }
    // NO DEFAULT BARRIER HERE !!
    #pragma omp barrier

    do_many_other_things();
}
```

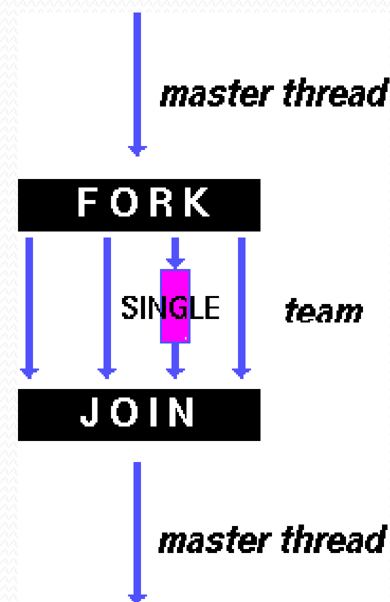
OpenMP work-sharing Constructs

- Emphasizing again: there is an implied/default barrier at the end of the work-sharing constructs.
- If you want to skip the implied barrier, use *nowait* clause.
- You already know one of them:
  - #pragma omp parallel for
  - Single Construct:
    - With #pragma omp single
  - Sections/Section construct:
    - With:
      - #pragma omp sections
      - #pragma omp section

OpenMP Single Construct

- By using this construct in parallel region, only one thread (not necessarily the master thread) will execute the corresponding code:
- Example:

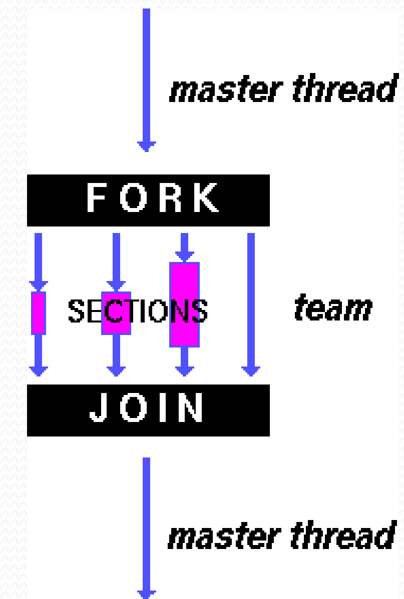
```
#pragma omp parallel
{
    do_many_things();

    #pragma omp single
    {
        exchange_boundaries();
    }
    // THERE IS A DEFAULT BARRIER HERE !!

    do_many_other_things();
}
```



OpenMP Sections/Section Construct

- The Sections work-sharing construct gives a different structured block to each thread.
- Example:

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
            X_calculations();

        #pragma omp section
            Y_calculations();

        #pragma omp section
            Z_calculations();
    }
    // DEFAULT/IMPLIED BARRIER HERE
    other_stuff();
}
```



Low level Synchronization in OpenMP - Locks

- Constructs such as barrier, critical and atomic provide synchronization facility in OpenMP.
- OpenMP also provides **lock mechanism** which is a *lower level synchronization mechanism* than the previous constructs, hence, it is more powerful. *Programmer has greater control over his/her program*.
- How?
  1. First, defining variable(s) of type *lock_t*
  2. Initializing them by the function *omp_initi_lock(&lock variable)*
  3. Trying to grab the lock by calling the function *omp_set_lock(&lock variable)* ➔ If other thread has the lock, you sit and wait until it becomes available !
  4. Releasing the lock by calling the function *omp_unset_lock(&lock variable)*
  5. Freeing the lock resource (when you are done with the locks) by calling the function *omp_destroy_lock(&lock variable)*

Low level Synchronization in OpenMP - Locks

- Lock Example:
  - Histogram example:
  - Assumption: chances of having contention on the lock is low if the histogram is large enough !
  - Think about how to implement with critical construct !!

```
lock_t hist_locks[NBUCKETS];
int hist[NBUCKETS];
#pragma omp parallel for
    for(i = 0; i < NBUCKETS; ++i) {
        omp_init_lock(&hist_locks[i]);
        hist[i] = 0;
    }
#pragma omp parallel for
    for (i = 0; i < NVALS; ++i) {
        iVal = (int) sample(arr[i]);
        omp_set_lock(&hist_locks[iVal]);
        hist[iVal]++;
        omp_unset_lock(&hist_locks[iVal]);
    }
for(i = 0; i < NBUCKETS; ++i)
    omp_destroy_lock(&hist_locks[i]);
```

OpenMP Runtime Library

- Some useful OpenMP Runtime Library Routines:
  - *omp_set_num_threads(…)*
    - // you could use clause num_threads(…) instead
  - *omp_get_num_threads():*
    - **Be very careful where using this routine !!** If it is called outside the parallel region, the value returned is obviously 1 !!
  - *omp_get_thread_num()*
  - *omp_get_max_thread()*
  - *omp_in_parallel()*
    - Returns true if the function is called inside a parallel region
    - Used mainly inside the functions which might be called from the parallel region

OpenMP Runtime Library

- Some useful OpenMP Runtime Library Routines:
  - *omp_num_procs()*: returns number of processors
  - *omp_set_dynamic(…)*
    - When set, the compiler might give you different number of threads from one parallel region to another parallel region in your program (runtime has the power to choose the number of threads for you)
  - *omp_get_dynamic()*: boolean
  - Example:

```
void main() {
    omp_set_dynamic(0);

    // ...

    omp_set_num_threads(omp_num_proc());
    ...
}
```

OpenMP Environment Variables

- Environment variables give the facility to change the execution environment without a requirement to recompile the program:
- Examples:
  - OMP_NUM_THREADS
  - OMP_POLICY_WAIT
    - Either ACTIVE or PASSIVE
    - Used in case of having barrier and critical sections and locks:
      - If thread needs to wait:
        - If ACTIVE:
          - Though the thread is awake and burns up the CPU cycles, it does nothing !!
          - Burning the CPU cycles is **cheap**.
        - If PASSIVE:
          - The thread is suspended, put to sleep, hence not burning up the CPU cycles.
          - Suspending and then, waking up the thread is **expensive**.
    - A trade-off in using the policy:
      - Which policy do you choose if you have huge functions in critical sections?
  - OMP_STACKSIZE
  - OMP_PROC_BIND
    - Search more about them ☺
- How to set?
  - In bash shell: export OMP_NUM_THREADS=8

OpenMP Data Storage Attribute

- Data can reside either in the heap or stack.
  - If it is in the heap, it is public and shared among the threads. Hence, race condition is possible if using in the parallel region.
  - If it in the stack, it is private and local to the thread. In other words, the thread has its own local copy.
  - Example:
    - *A*, *index* and *count* are public, sitting on the heap ! They are alive even after the parallel region.
    - *Temp* and *p* are local/private to each thread, each copy of *temp* and *p* are sitting in the stack of each thread. They vanish after the thread is done with its execution !!

```
// file1.c
double A[10];
int main() {
    int index[10];
    #pragma omp parallel
    {
        int p = work(index);
    }
    printf("%d\n", index[0]);
}

// another file
extern double A[10];
int work(int * index) {
    double temp[10];
    static int count;
    // ...
}
```

OpenMP Data Storage Attribute

- However, OpenMP provides a way to change the storage attribute:
  - SHARED
  - PRIVATE
  - FIRSTPRIVATE
  - LASTPRIVATE

  - DEFAULT(SHARED|NONE)

OpenMP Data Storage Attribute

- However, OpenMP provides a way to change the storage attribute, through the following clauses:
  - SHARED
    - As its name indicate, the variable is considered to be shared among the threads (on the process heap)
  - PRIVATE
    - As its name indicates, the variable is considered to be private for the thread (on the thread stack)
    - BE CAREFUL: private variables are not initialized !!!!!
    - Example:

```
void wrong() {
    int temp = 0;
    #pragma omp parallel for private(temp)
        for (int j = 0; j < 1000; ++j)
            temp += j;
    printf("%d\n", temp); // it will print the global one which is 0
}
```

OpenMP Data Storage Attribute

- However, OpenMP provides a way to change the storage attribute, through the following clauses:
  - PRIVATE → FIRSTPRIVATE
    - How to fix the problem with un-initialized private variable?
      - Either initialize it in the parallel region
      - Or, use the FIRSTPRIVATE clause rather than PRIVATE clause.
      - Example:

```
int incr = 0;
#pragma omp parallel for firstprivate(incr)
    for(i = 0; i <= MAX; ++i) {
        if ((i % 2) == 0) incr++;
        A[i] = incr;
    }
```

      - The private variable *incr* is allocated in the *thread stack* and it is *initialized to zero*.

OpenMP Data Storage Attribute

- However, OpenMP provides a way to change the storage attribute, through the following clauses:
  - LASTPRIVATE:
    - Example:

```
void sq2(int n, double * lastTerm)
{
    double x;
    int i;
    #pragma omp parallel for lastprivate(x)
    {
        for (i = 0; i < n; ++i) {
            x = a[i] * a[i] + b[i] * b[i];
            b[i] = sqrt(x);
        }
    }

    *lastTerm = x;
}
```

    - Among the threads, whoever does the iteration (n-1), i.e., the last iteration, the value of x that it has is copied to the global scope !!!

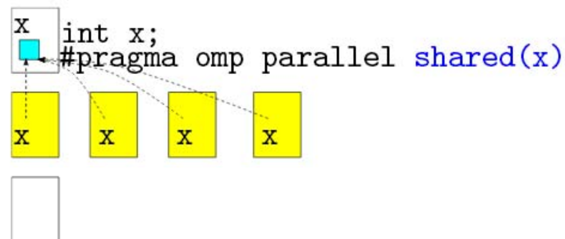
OpenMP Data Storage Attribute

- However, OpenMP provides a way to change the storage attribute, through the following clauses:
  - For the variables defined outside the parallel region, the default storage attribute is SHARED.
  - However, you can also change the default by using the clause DEFAULT(SHARED | NONE).
    - If DEFAULT (NONE) is used, the programmer should determine the storage attribute of each and every variable explicitly.
      - It is handy for debugging purposes !

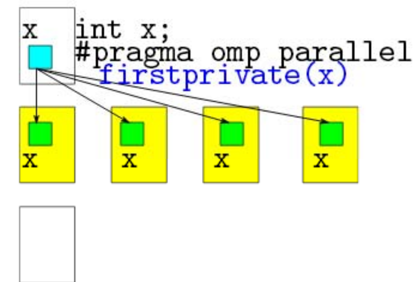
OpenMP Data Storage Attribute

- Summary of Data Storage Attributes:

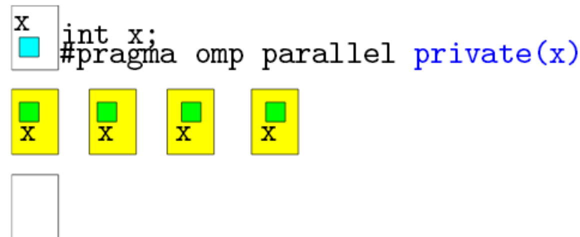
shared



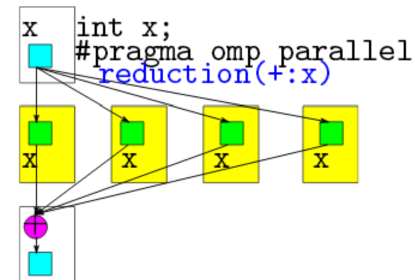
firstprivate



private



reduction

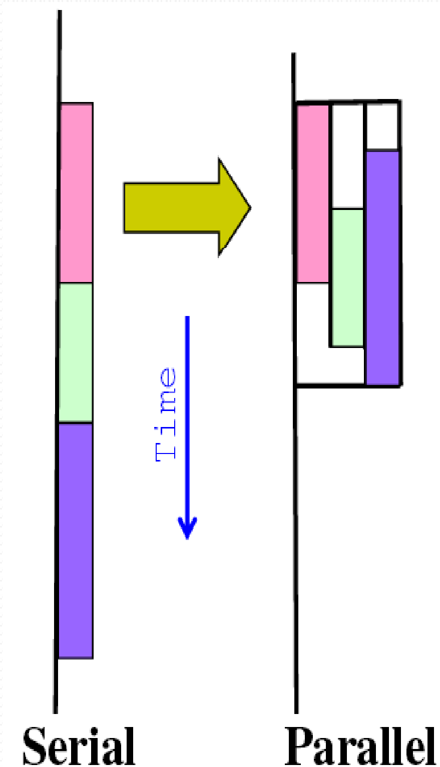


Source of Image:
http://www.eidos.ic.i.u-tokyo.ac.jp/~tau/lecture/parallel_distributed/2014/slides/openmp.pdf

Parallelizing Linked List based operations – OpenMP task parallelism

- Based on the current knowledge of OpenMP, how can parallelize the following program?

```
p = head;
while (p) {
    big_ugly_calculation(p);
    p = p -> next;
}
```



Parallelizing Linked List based operations – OpenMP task parallelism

- Based on the current knowledge of OpenMP, how can parallelize the following program?
- Solution:

```
p = head;
while (p) {
    big_ugly_calculation(p);
    p = p -> next;
}
```

```
int count = 0;
while (p) {
    p = p -> next;
    count++;
}
p = head;
int i;
// allocate an array of size count, called parr
int *parr=(int*)malloc(count*sizeof(int));
for (i = 0; i < count; ++i) {
    parr[i] = p;
    p = p -> next;
}

#pragma omp parallel for
for (i = 0; i < count; ++i)
    big_ugly_calculation(parr[i]);
```

Parallelizing Linked List based operations – OpenMP task parallelism

- Solution:

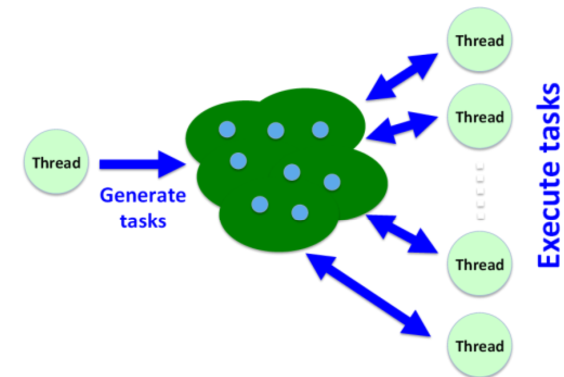
```
p = head;
while (p) {
    big_ugly_calculation(p);
    p = p -> next;
}
```

```
int count = 0;
while (p) {
    p = p -> next;
    count++;
}
p = head;
int i;
// allocate an array of size count, called parr
int *parr=(int*)malloc(count*sizeof(int));
for (i = 0; i < count; ++i) {
    parr[i] = p;
    p = p -> next;
}

#pragma omp parallel for
for (i = 0; i < count; ++i)
    big_ugly_calculation(parr[i]);
```

- But …
- Do you really want to traverse the list 3 times???
  - It doesn't seem to be efficient. Right?
- Can be easily done in cilk by using cilk_spawn …

Parallelizing Linked List based operations – OpenMP task parallelism

- Can be easily done in cilk by using cilk_spawn
- OpenMP provids a construct called task which does almost the same as cilk_spawn:
  - Task is an independent unit of work.
  - It is composed of:
    - Code to execute.
    - Data environment
    - Internal Control Variables (ICVs):
      - Certain features of the environment that constructs control. E.g., number of threads, etc.
  - Created by using the construct *#pragma omp task*



Parallelizing Linked List based operations – OpenMP task parallelism

- Task is an independent unit of work.
  - Tasks are added to the task queue (by the thread which encounters the #pragma omp task construct) and
  - *The runtime system decides when tasks are executed.*
  - Example:
    - In this example, the programmer asks for bunch of threads, then, each thread is creating a task, called foo and add it to the task queue (to be executed).
      In #…single, only one thread is making the task called bar.

```
#pragma omp prallel
{
    #pragma omp task
    foo();

    #pragma omp barrier
    // ALL foo TASKS ARE
    // GUARANTEED TO BE COMPLETED
    // HERE
    #pragma omp single
    {
        #pragma omp task
        bar();
    } // IMPLIED BARRIER HERE
} // IMPLIED BARRIER HERE
```

Parallelizing Linked List based operations – OpenMP task parallelism

- Task is an independent unit of work.
  - Example2 – divide and conquer pattern:
    - This method will be called inside a parallel region (#pragma omp parallel).
    - Why are x and y defined to be shared for calculation of *fib(n-1)* and *fib(n-2)* ?

```
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);
}
```

Parallelizing Linked List based operations – OpenMP task parallelism

- Task is an independent unit of work.
  - Example2 – divide and conquer pattern:
    - This method will be called inside a parallel region (*#pragma omp parallel*).
    - Why are x and y defined to be shared for calculation of *fib(n-1)* and *fib(n-2)* ? Because, otherwise, they are considered to be private inside the tasks and never written back to the global x and y defined at the beginning of the function !! In other words, If we did not declare the variables shared, each task would set its own local variable, then throw away the results.

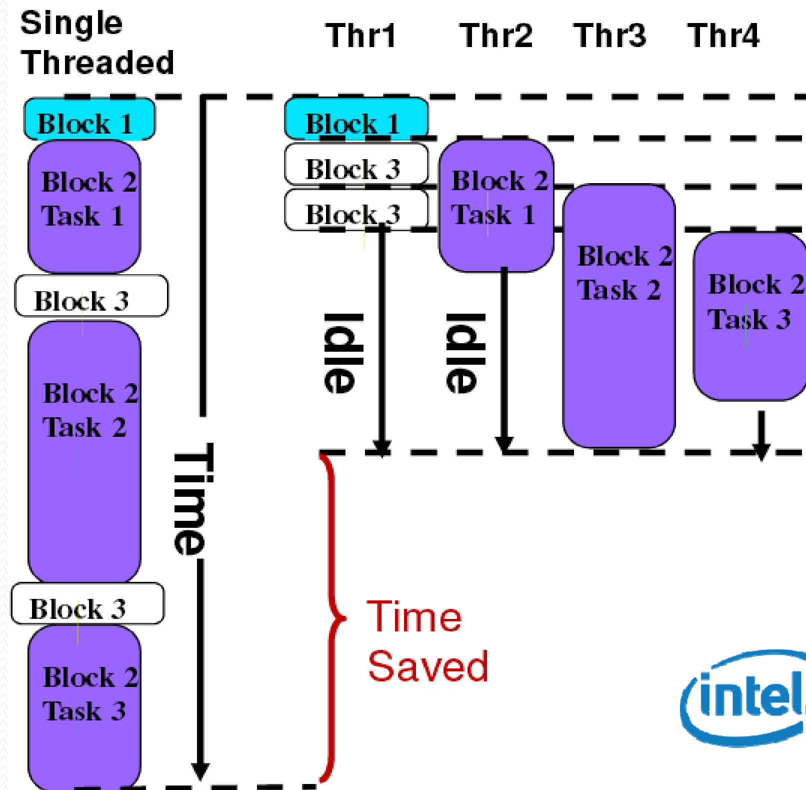
```
int fib(int n) {
    int x, y;
    if (n < 2) return n;

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

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

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

  - Just 2 emphasize once again:
    - Automatic variables defined inside the subroutines (within a parallel region) are considered as private to each task !!

Parallelizing Linked List based operations – OpenMP task parallelism

- Solution, using openMP task construct:

```
#pragma omp parallel
#pragma omp single
{   // block 1
    p = head;
    while(p) {
        #pragma omp task firstprivate(p)
            big_ugly_calculation(p); //block 2
        p = p -> next; // block 3
    }
} // DEFAULT BARRIER
```



Advanced topics about openMP tasks:
https://www.archer.ac.uk/training/course-material/2014/05/AdvancedOpenMP_Oxford/Slides/L05-Tasks.pdf