



Programming Shared-Memory Parallel Systems - OpenMP

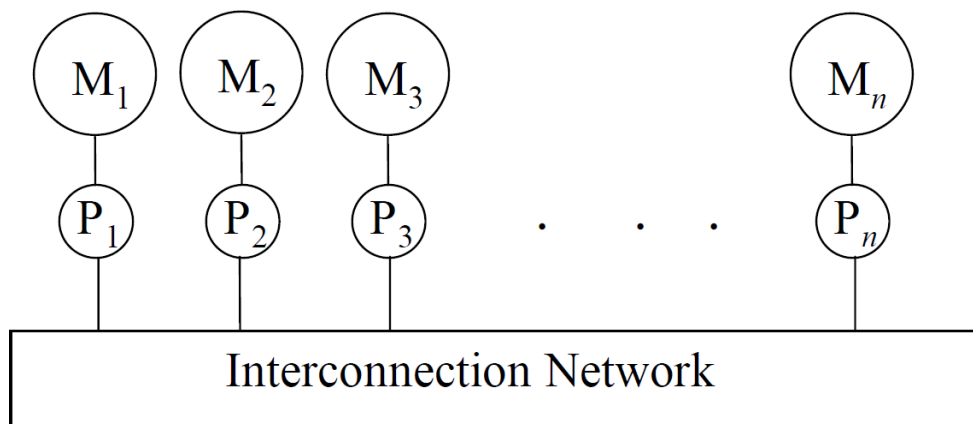
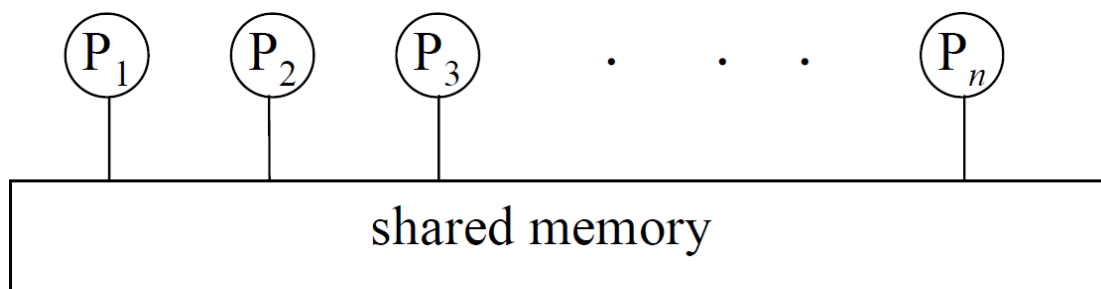
Usama Bin Amir

Department of Cyber Security,
Air University, Islamabad



Parallel and Distributed Computing

- The **difference** includes whether the **processes** **communicate** using **shared** or a **distributed memory**





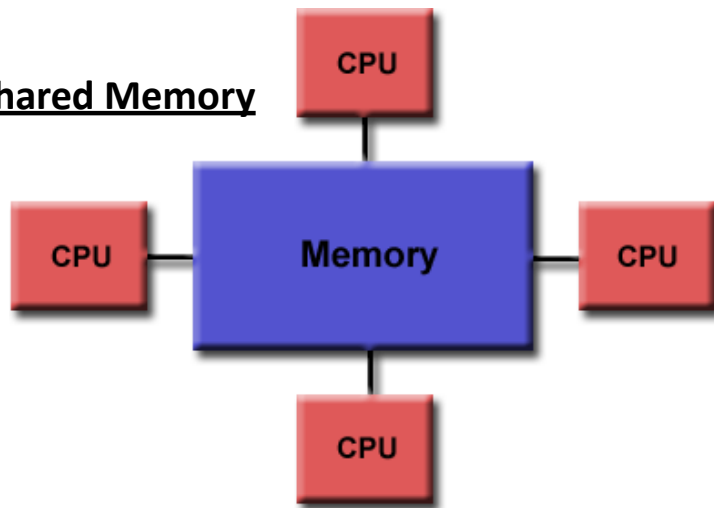
OpenMP

- **Programming** of **shared memory systems**
- An **API** for **Fortran** and **C/C++**
 - **Directives**
 - **Runtime routines**
 - **Environment variables**

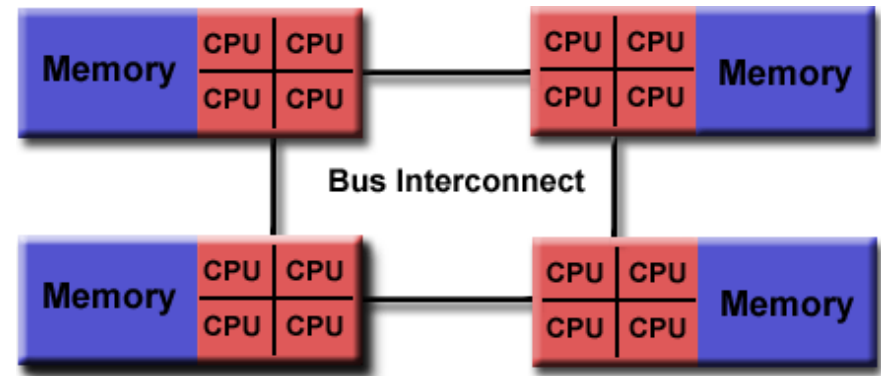


Memory Models

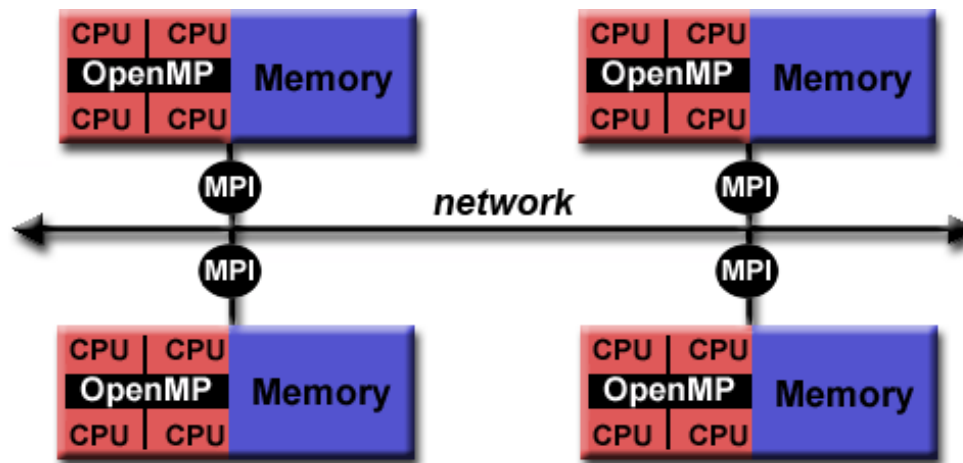
Shared Memory



Distributed Memory



Hybrid Memory





Goals

- **Standardization**
 - Provide a standard among a variety of shared memory architectures (platforms)
- **High-level interfaces** to thread programming
- **Multi-vendor support**
- **Multi-OS support** (Unix, Windows, Mac, etc.)
- The **MP** in **OpenMP** is for **M**ulti-**P**rocessing
- *Don't confuse **OpenMP** with **Open MPI**! :)*



Release History

Date	Version
Oct 1997	Fortran 1.0
Oct 1998	C/C++ 1.0
Nov 1999	Fortran 1.1
Nov 2000	Fortran 2.0
Mar 2002	C/C++ 2.0
May 2005	OpenMP 2.5
May 2008	OpenMP 3.0
Jul 2011	OpenMP 3.1
Jul 2013	OpenMP 4.0
Nov 2015	OpenMP 4.5
Nov 2018	OpenMP 5.0



Programming Shared Memory Systems

- **Explicit Parallelism**
 - For example, **pthread**s
- **Programmer Directed**
 - For example, **OpenMP**



Hello World – pthreads based version

```
#include <pthread.h>
#include <stdio.h>

void* thrfunc(void* arg) {
    printf("hello from thread %d\n", *(int*)arg);
}

int main(void) {
    pthread_t thread[4];
    pthread_attr_t attr;
    int arg[4] = {0,1,2,3};
    int i;
    // setup joinable threads
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
    // create N threads
    for(i=0; i<4; i++)
        pthread_create(&thread[i], &attr, thrfunc, (void*)&arg[i]);

    // wait for the N threads to finish
    for(i=0; i<4; i++)
        pthread_join(thread[i], NULL);
}
```




... and the OpenMP version

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

int main(int argc, char* argv[])
{
    #pragma omp parallel {
        printf("Hello World... from thread = %d\n", omp_get_thread_num());
    }
}
```

Compilation: \$ gcc -fopenmp hello.c -o hello

Execution: \$ export OMP_NUM_THREADS=4

\$./hello

Demo: hello.c



Compiling

Intel (**icc**, **ifort**, **icpc**)

-openmp

PGI (**pgcc**, **pgf90**, ...)

-mp

GNU (**gcc**, **gfortran**, **g++**)

-fopenmp



OpenMP - User Interface Model

- **Shared Memory** with **thread based parallelism**
- Not a new language
- **Compiler directives, library calls, and environment variables extend the base language**
 - f77, f90, f95, C, C++
- **Not automatic parallelization**
 - User **explicitly specifies parallelism**
 - **NOTE: Compiler does not ignore user directives even if wrong**



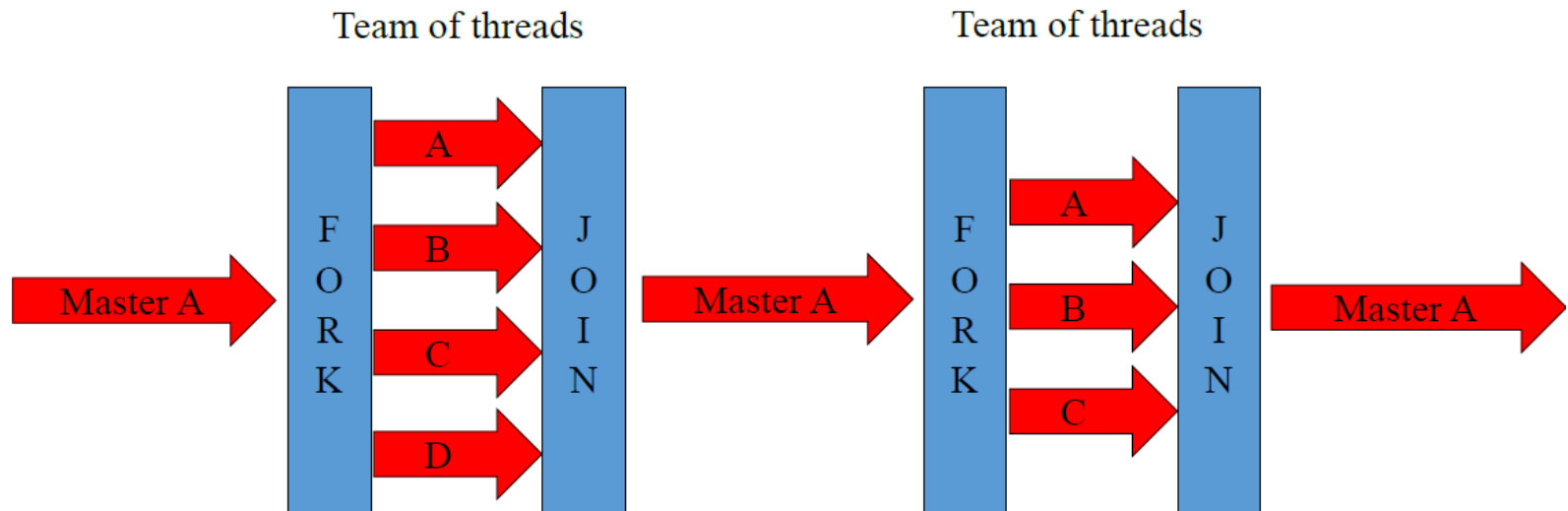
OpenMP - Syntax

- **Parallelism** is **highlighted** using **compiler directives** or **pragmas**
- For C and C++, the pragmas take the form:
#pragma omp *construct* [*clause* [*clause*]...]
- Any compiler (even if it does not have OpenMP support) can compile the program (with no parallelism though)



Fork*/Join Execution Model

- An **OpenMP** program starts as a **single thread** (*master thread*).
- **Additional threads** (*Team*) are **created** when the master hits a parallel region.
- When all threads finished the parallel region, the **new threads** are **given back** to the **runtime** or operating system



**Not to be confused with fork() system call*



Using OpenMP

- **OpenMP** is usually used to parallelize loops:
 - Find most time consuming loops
 - Split them among threads

Split-up this loop between multiple threads

```
void main( )  
{  
    double Res[1000];  
  
    for(int i=0;i<1000;i++) {  
        do_huge_comp(Res[i]);  
    }  
}
```

Sequential program

```
void main( )  
{  
    double Res[1000];  
  
    #pragma omp parallel for  
    for(int i=0;i<1000;i++) {  
        do_huge_comp(Res[i]);  
    }  
}
```

Parallel program



OpenMP Directives



OpenMP - Directives

- **OpenMP** compiler directives are used for various purposes:
 - Spawning a parallel region
 - Dividing blocks of code among threads
 - Distributing loop iterations between threads
 - ...

sentinel directive-name [clause, ...]

↓ ↓ ↓
#pragma omp parallel private(var)



Supported Clauses for the Parallel Construct

Valid Clauses:

if (logical expression)

num_threads (integer)

private (list of variables)

firstprivate (list of variables)

shared (list of variables)

default (none | shared | private **fortran only**)

copyin (list of variables)

reduction (operator: list)

...



OpenMP Constructs

- **OpenMP constructs** can be divided into **5 categories**:
 1. Parallel Regions
 2. Work-sharing
 3. Data Environment
 4. Synchronization
 5. Runtime functions/environment variables



OpenMP: Parallel Regions

- You create **threads** in **OpenMP** with “**omp parallel**” pragma
- For example: a **4-thread based Parallel region**:

```
int A[10];  
  
omp_set_num_threads(4);  
  
#pragma omp parallel  
{  
    int ID =omp_get_thread_num();  
    fun1(ID,A);  
}
```

Demo: helloFun.c

- **Implicit barrier** at the **end of parallel block**
- **Each thread** calls **fun1(ID,A)** *for ID = 0 to 3*
- **Each thread** executes the same code within the block



The *parallel* directive

- A **parallel region** is a **block** of **code** that will be executed by multiple threads
- When (in serial program) a **PARALLEL** directive is **found**, a **team of threads** is **created** and **main-thread** (serial execution thread) **becomes** the **master of the team**
- **Master thread** has **id** or **number 0** (*within that team*)
- The code is duplicated and all threads will execute that code
- There is an **implicit barrier** at the end of a parallel region
- **Master thread continues execution** after this point



The *parallel* directive

- Some **common clauses** include:
 - **if** (*expression*)
 - **private** (*list*)
 - **shared** (*list*)
 - **num_threads** (*integer-expression*)



How Many Threads?

- The **number of threads** in a **parallel region** is **determined** by the **following factors**, *in order of precedence*:
 1. **Evaluation** of the **if** clause
 2. **Setting** of the **NUM_THREADS** clause
 3. Use of the **omp_set_num_threads()** library function
 4. Setting of the **OMP_NUM_THREADS** **environment variable**
 5. **Implementation default**: Usually the ***number of CPUs on a node***
- **Threads are numbered** from **0** (master thread) to **N-1**



IF clause

```
#pragma omp parallel if (scalar_expression)
```

- Execute in parallel if expression is true
 - Otherwise serial execution



NUM_THREADS clause

```
#pragma omp parallel if(np>1) num_threads(np)
{
    ...
}
```

- Execute in parallel if expression is true
- Executes using *np* number of threads



omp_set_num_threads() function

```
#define TOTAL_THREADS 8
int main( )
{
    omp_set_num_threads(TOTAL_THREADS);
    #pragma omp parallel
    {
        . . .
    }
    . . .
}
```

- Execute in parallel using 8 threads



OMP_NUM_THREADS – Environment Variable

```
$ export OMP_NUM_THREADS=4  
$ echo $OMP_NUM_THREADS
```

- Sets and displays the value of the environment variable **OMP_NUM_THREADS**



Execution Status in Parallel Region

```
int omp_in_parallel()
```

- Returns **non-zero**: if execution is in parallel region
- Returns **zero**: if execution in non-parallel region

Demo: PRegion.c



Shared and Private Data

- **Shared data** are **accessible** by **all threads**
- A reference **a[5]** to a **shared array** accesses the same address in all threads
- **Private data** are accessible only by a thread
 - **Each thread** has its own copy
- The **default** is **shared**



Shared and Private Data

```
int main(int argc, char* argv[])
{
    int threadData = 10;

    // Beginning of parallel region
    #pragma omp parallel private(threadData)
    {
        threadData = 200;
    }

    // Ending of parallel region
    printf("Value: %d\n", threadData);
}
```

Demo: SPData.c



Shared and Private Data

```
#pragma omp parallel shared(list)
```

- Default behavior
- List will be shared
- Each thread access the same memory location
- Initial value (for the first thread) will be same as before the region
- Final value will be updated by the last thread leaving the region
- Problems: Data Race



Shared and Private Data

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

- Data local to thread
- You should not rely on any initial and terminal value (after execution of the parallel region)
- Separate “**Stack Memory**” for each thread’s private data
- No storage associated with original object (even with same name for data-items)
- Use *firstprivate* and/or *lastprivate clause* to override



Shared and Private Data

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

- Variables in **list** are **private**
- **Initialized** with the **value** the variable had *before* entering the **construct**

```
#pragma omp parallel for lastprivate (list)
```

- **Used in “for” loops**
- Variables in **list** are **private**
- The **thread** that **executes** the ***final iteration of the loop*** **updates the value** (of the variables in the list)



Shared and Private Data

- Alter the **default behavior**
- To implement **customized access behavior**



Shared and Private Data – Example (1/4)

```
#include <omp.h>
#include <stdio.h>
int main() {
    int i;
    const int N = 1000;
    int a = 50;
    int b = 0;

    #pragma omp parallel for default(shared)
    for (i=0; i<N; i++) {
        b = a + i;
    }

    printf("a=%d b=%d (expected a=50 b=1049)\n", a, b);
}
```

Demo: SPDE1.c



Shared and Private Data – Example (2/4)

```
#include <omp.h>
#include <stdio.h>
int main() {
    int i;
    const int N = 1000;
    int a = 50;
    int b = 0;

    #pragma omp parallel for default(none) private(i) private(a) private(b)
    for (i=0; i<N; i++) {
        b = a + i;
    }

    printf("a=%d b=%d (expected a=50 b=1049)\n", a, b);
}
```

Demo: SPDE1.c



Shared and Private Data – Example (3/4)

```
#include <omp.h>
#include <stdio.h>
int main() {
    int i;
    const int N = 1000;
    int a = 50;
    int b = 0;

    #pragma omp parallel for default(none) private(i) private(a) lastprivate(b)
    for (i=0; i<N; i++) {
        b = a + i;
    }

    printf("a=%d b=%d (expected a=50 b=1049)\n", a, b);
}
```

Demo: SPDE1.c



Shared and Private Data – Example (4/4)

```
#include <omp.h>
#include <stdio.h>
int main() {
    int i;
    const int N = 1000;
    int a = 50;
    int b = 0;

    #pragma omp parallel for default(none) private(i) firstprivate(a) lastprivate(b)
    for (i=0; i<N; i++) {
        b = a + i;
    }

    printf("a=%d b=%d (expected a=50 b=1049)\n", a, b);
}
```

Demo: SPDE1.c



Getting ID of Current Thread

```
int main(int argc, char* argv[])
{
    int iam, nthreads;
    #pragma omp parallel private(iam,nthreads) num_threads(2)
    {
        iam = omp_get_thread_num();
        nthreads = omp_get_num_threads();
        printf("ThradID %d, out of %d threads\n", iam, nthreads);
        if (iam == 0)
            printf("Here is the Master Thread.\n");
        else
            printf("Here is another thread.\n");
    }
}
```

Demo: CTID.c



Work-Sharing Constructs

- If **all** the **threads** are **doing** the **same thing**, *what is the advantage then?*
- Within each “Team” threads are assigned IDs, with master thread assigned ID 0
 - `omp_get_thread_num()` //to get thread number

Can we use this to distribute tasks amongst the “team” members?

- **Work-sharing** constructs **distribute** the **specified work** to **all threads** within the **current team**
- **Work-sharing constructs** **do not launch new threads**

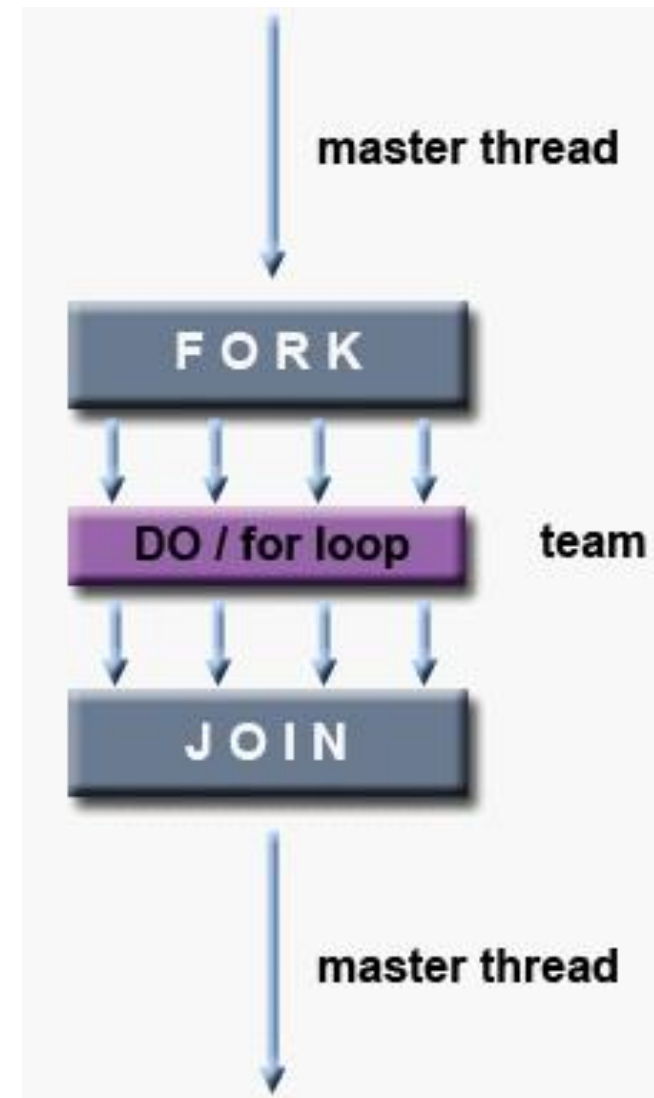


For Work-Sharing Construct

- **for** - shares iterations of a loop across the team

#pragma omp for [clause ...] newline

There is an **implicit synchronization** after
#pragma omp for





For Work-Sharing Construct

- **SCHEDULE** clause describes how iterations of the loop are divided among the threads in the team

STATIC



Chunks of specified size assigned round-robin

DYNAMIC



Chunks of specified size are assigned when thread finishes previous chunk (*work-Stealing mechanism*)



Do/For Work-Sharing Construct

```
int main(int argc, char* argv[])
{
    int i, a[10];
    #pragma omp parallel num_threads(2)
    {
        #pragma omp for schedule(static, 2)
        for ( i=0; i<10;i++)
            a[i] = omp_get_thread_num();
    }

    for ( i=0; i<10;i++)
        printf("%d",a[i]);
}
```

Demo: ForConst.c



Do/For Work-Sharing Construct

```
int main(int argc, char* argv[])
{
    int sum, counter, inputList[6] = {11,45,3,5,12,-3};
    #pragma omp parallel num_threads(2)
    {
        #pragma omp for schedule(static, 3)
        for (counter=0; counter<6; counter++) {
            printf("%d adding %d to the\n",omp_get_thread_num(), inputList[counter]);

            sum+=inputList[counter];
        } //end of for
    } //end of parallel section

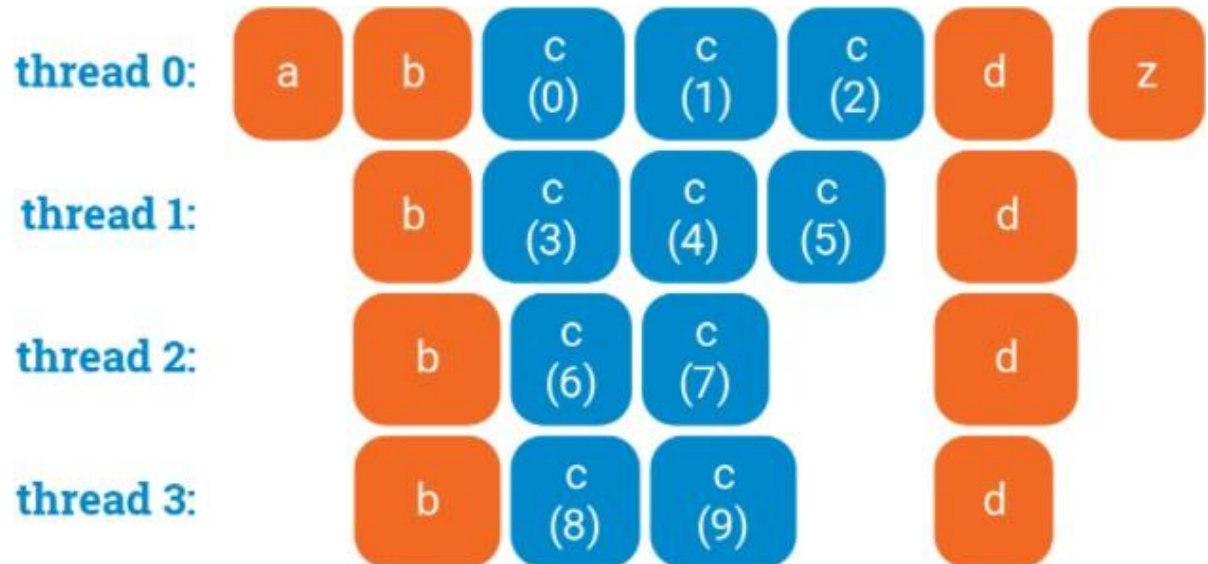
    printf("The summed up Value: %d", sum);
}
```

Demo: ForConst2.c



For Work-Sharing –Synchronized

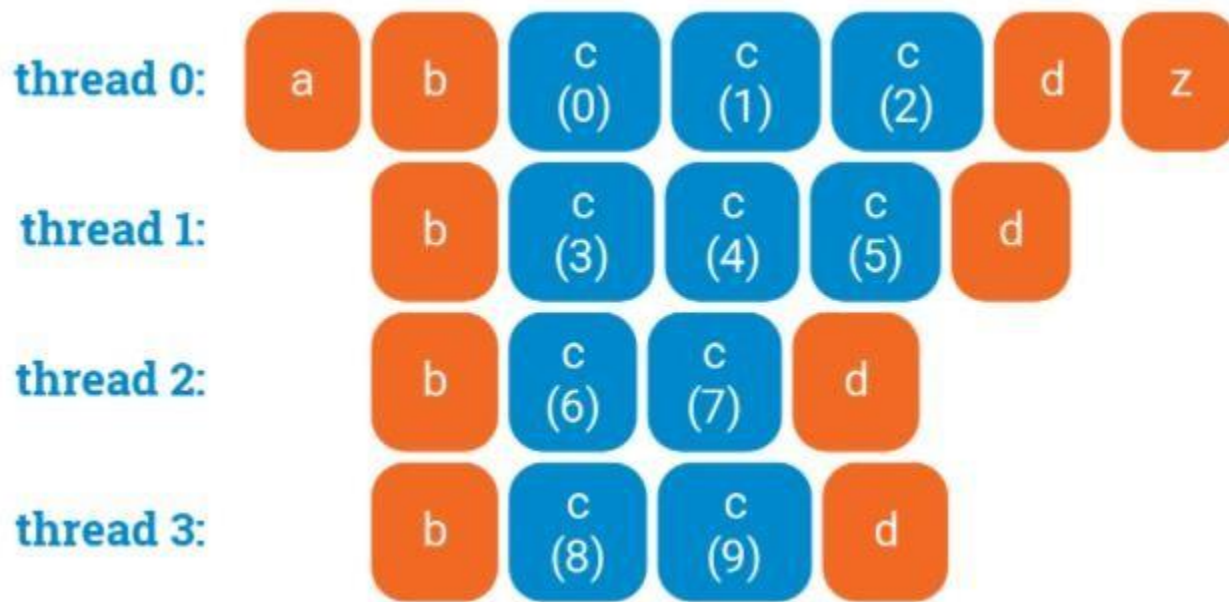
```
a();  
#pragma omp parallel  
{  
    b();  
    #pragma omp for  
    for (int i = 0; i < 10; ++i) {  
        c(i);  
    }  
    d();  
}  
z();
```





For Work-Sharing – Non Synchronized

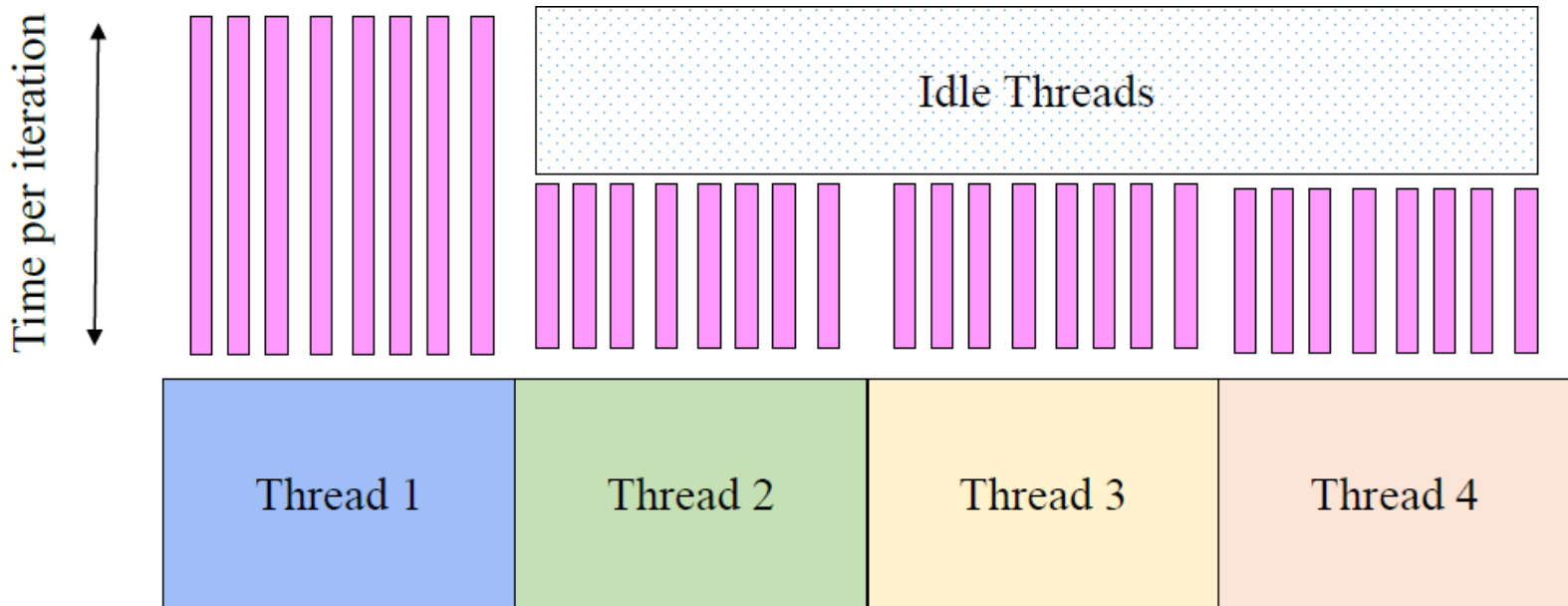
```
a();  
#pragma omp parallel  
{  
    b();  
    #pragma omp for nowait  
    for (int i = 0; i < 10; ++i) {  
        c(i);  
    }  
    d();  
}  
z();
```





Problems with Static Scheduling

- What happens if loop iterations do not take the same amount of time?
 - Load imbalance**

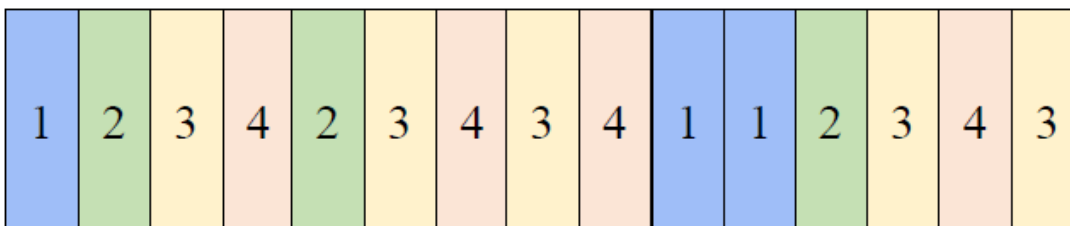




Dynamic Scheduling

- **Fixed size chunks assigned on the fly**
 - Work-stealing mechanism
- **Disadvantage:** more overhead as compared to Static

```
#pragma omp parallel for default(shared) private(j) schedule(dynamic,10)  
    for (j=0; j<N; j++) {  
        ... // some work here  
    }
```



Demo: LoopSched.c



Threads share Global variables!

```
#include <pthread.h>
#include <iostream>
#include <unistd.h>
using namespace std;
#define NUM_THREADS 10

int sharedData = 0;
void* incrementData(void* arg) {
    sharedData++;
    pthread_exit(NULL);
}

int main()
{
    pthread_t threadID;
    for (int counter=0; counter<NUM_THREADS;counter++) {
        pthread_create(&threadID, NULL, incrementData, NULL);
    }
    cout << "ThreadCount:" << sharedData <<endl;
    pthread_exit(NULL);
}
```




The output for the pthread version?

```
>./globalData
```

```
ThreadCount:10
```

```
>./globalData
```

```
ThreadCount:8
```



ThreadCount: A better implementation

```
#include <pthread.h>
#include <iostream>
#include <unistd.h>
using namespace std;
#define NUM_THREADS 100
int sharedData = 0;
void* incrementData(void* arg)
{
    sharedData++;
    pthread_exit(NULL); }

int main()
{
    pthread_t threadID[NUM_THREADS];
    for (int counter=0; counter<NUM_THREADS;counter++) {
        pthread_create(&threadID[counter], NULL, incrementData, NULL);
    }
    //waiting for all threads
    int statusReturned;
    for (int counter=0; counter<NUM_THREADS;counter++) {
        pthread_join(threadID[counter], NULL);
    }
    cout << "ThreadCount:" << sharedData <<endl;
    pthread_exit(NULL);
}
```



Is the problem solved?

- Unfortunately, not yet :(
- The **output** from running it with 1000 threads is as below:

```
>./6join  
ThreadCount:990  
>./6join  
ThreadCount:978  
>./6join  
ThreadCount:1000  
>
```

- Reasons?
- What can be done?



ThreadCount: OpenMP Implementation

```
int main(int argc, char* argv[])
{
    int threadCount=0;
    #pragma omp parallel num_threads(100)
    {
        int myLocalCount = threadCount;
        sleep(1);
        myLocalCount++;
        threadCount = myLocalCount;
    }
    printf("Total Number of Threads: %d\n", threadCount);
}
```

Demo: TCount1.c

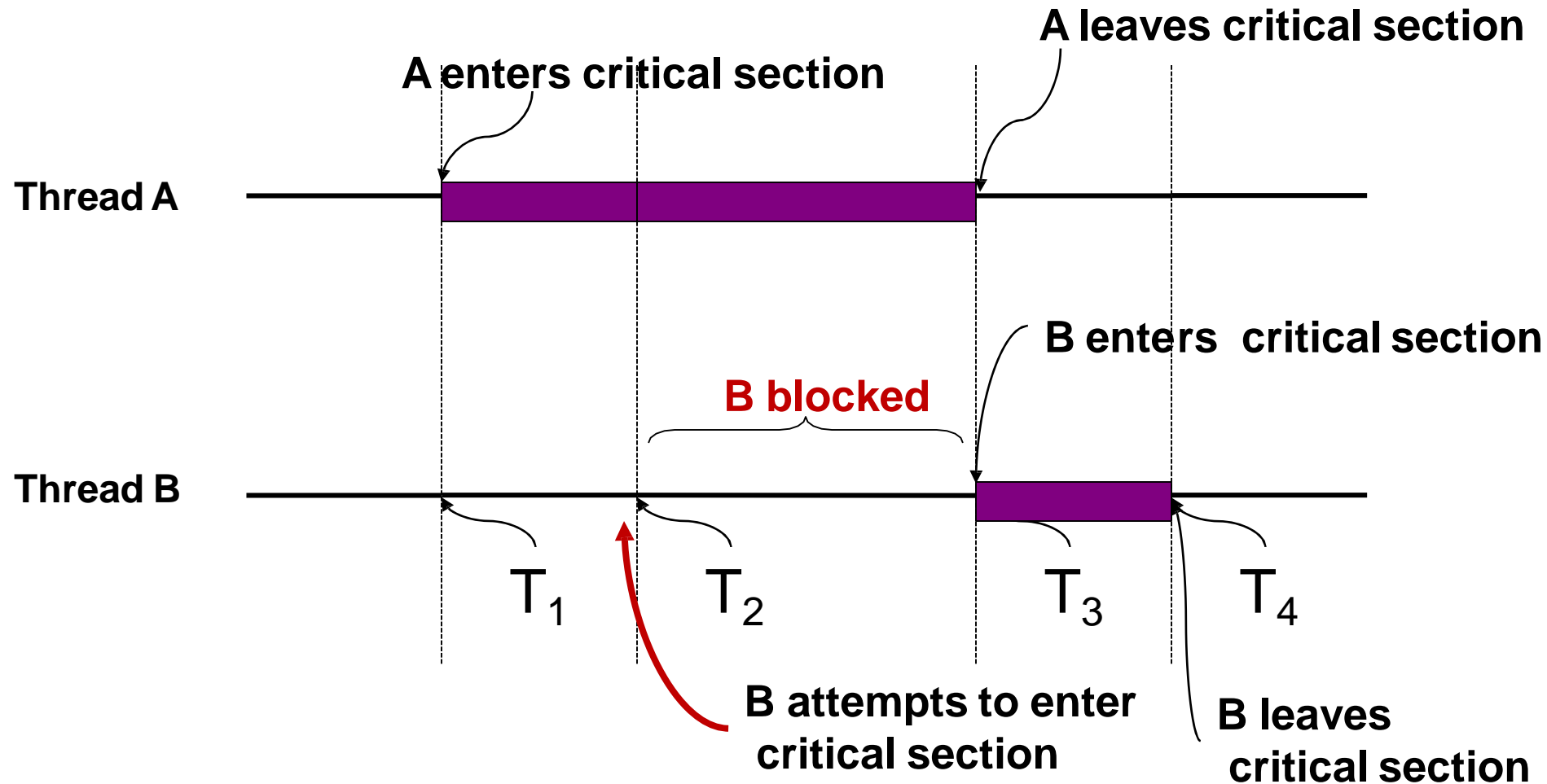


Critical-Section (CS) Problem

- n processes all competing to use some shared data
- Each process has a code segment, called **critical section**, in which the shared data is accessed
- **Problem** (ensures that):
 - Two process are not allowed to execute in their critical section at the same time
 - Access to the critical section must be an atomic action



Critical Section



Mutual Exclusion

At any given time, only one Thread is in the critical section



...back to threads counting

```
int sharedData = 0;
pthread_mutex_t mutexIncrement;

void* incrementData(void* arg)
{
    pthread_mutex_lock(&mutexIncrement);
    sharedData++;
    pthread_mutex_unlock(&mutexIncrement);
    pthread_exit(NULL);
}

int main()
{
    pthread_mutex_init(&mutexIncrement, NULL);

    pthread_t threadID[NUM_THREADS];
    for (int counter=0; counter<NUM_THREADS;counter++) {
        pthread_create(&threadID[counter], NULL, incrementData,
            NULL);
    }
    //waiting for all threads
    int statusReturned;
    for (int counter=0; counter<NUM_THREADS;counter++)
        { pthread_join(threadID[counter], NULL);
        }
    cout << "ThreadCount:" << sharedData
    <<endl; pthread_exit(NULL);
}
```



OpenMP - Synchronization Constructs

- The **CRITICAL** directive specifies a region of code that must be executed by only one thread at a time
- If a thread is currently executing inside a **CRITICAL** region and another thread attempts to execute it, it will block until the first thread exits that CRITICAL region.

```
pragma omp critical [ name ]
```

```
...
```




... back to threadCount

```
int main(int argc, char* argv[])
{
    int threadCount;
    #pragma omp parallel num_threads(5)
    {
        #pragma omp critical
        {
            int myLocalCount = threadCount;
            sleep(1);
            myLocalCount++;
            threadCount = myLocalCount;
        }
    }
    printf("Total Number of Threads: %d\n", threadCount);
}
```

Demo: TCount2.c



OpenMP - Synchronization Constructs

- The **MASTER directive** specifies a **region** that is to be executed only by the master thread of the team
- **All other threads** on the team **skip** this section of code

```
#pragma omp master
```

```
...
```

Demo: MasterOnly.c



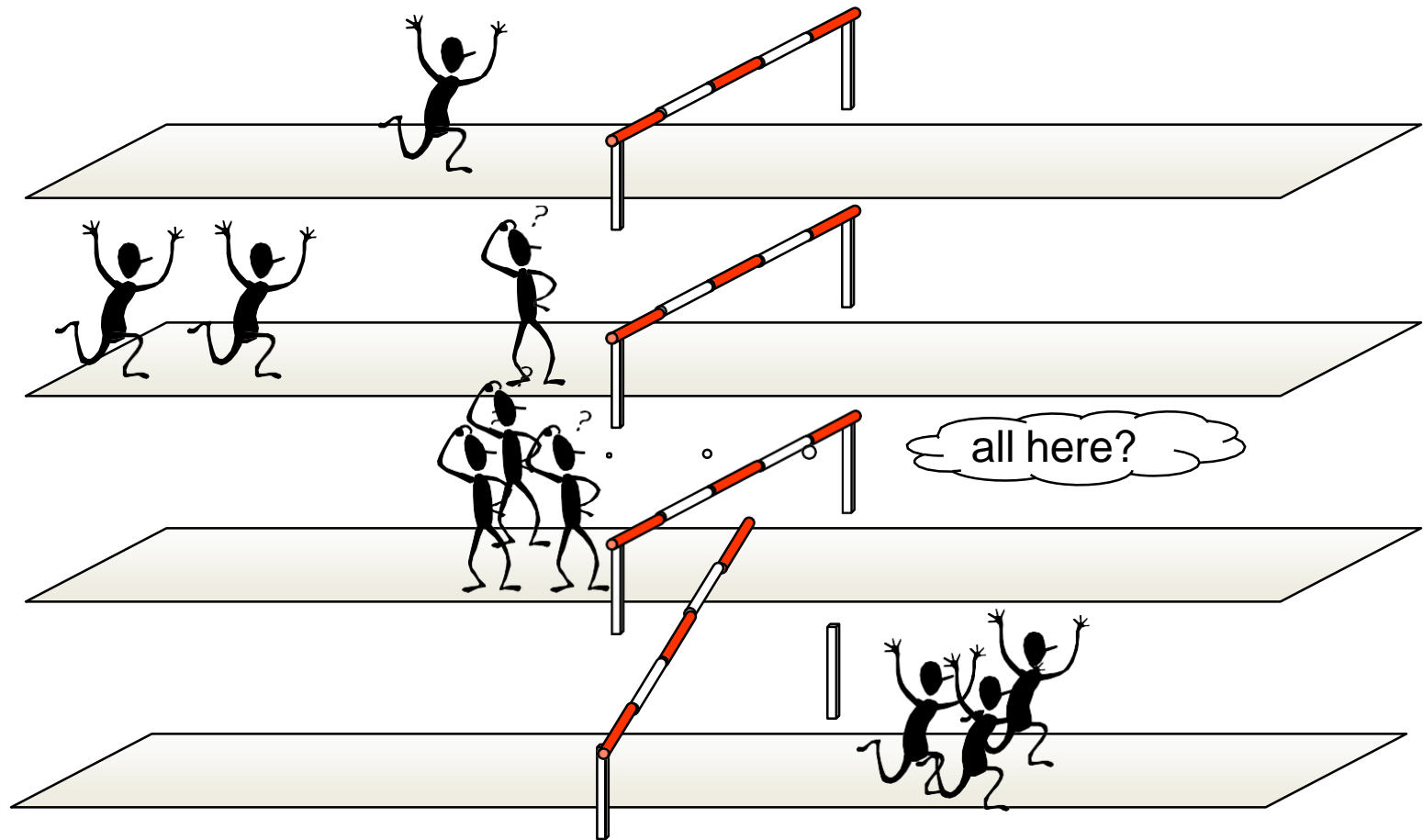
OpenMP - Synchronization Constructs

- When a **BARRIER** directive is reached, a thread will wait at that point until all other threads have reached that barrier
- *All threads then resume executing in parallel the code that follows the barrier.*

#pragma omp barrier

...

Barrier Synchronization



Demo: Barrier.c



Reduction (Data-sharing Attribute Clause)

- The **REDUCTION** clause performs a reduction operation on the variables that appear in the list
- A private copy for each list variable is created and initialized for each thread
- At the end of the reduction, the reduction variable (all private copies) is examined and the shared variable's final result is written.

#pragma omp operator: list

...

operator can be +, -, *, &&, ||, max, min ...



Reduction (Data-sharing Attribute Clause)

```
int main(int argc, char* argv[])
{
    srand(time(NULL));
    int winner;
    #pragma omp parallel reduction(max:winner) num_threads(10)
    {
        winner = (rand() % 1000) + omp_get_thread_num();
        printf("Thread: %d has Chosen: %d\n",
omp_get_thread_num(),winner);
    }
    printf("Winner: %d\n", winner);
}
```

Demo: Reduction.c



Any Questions?