Programming Shared Address Space Platforms with OpenMP

High Performance Computing

Chapter 7.10

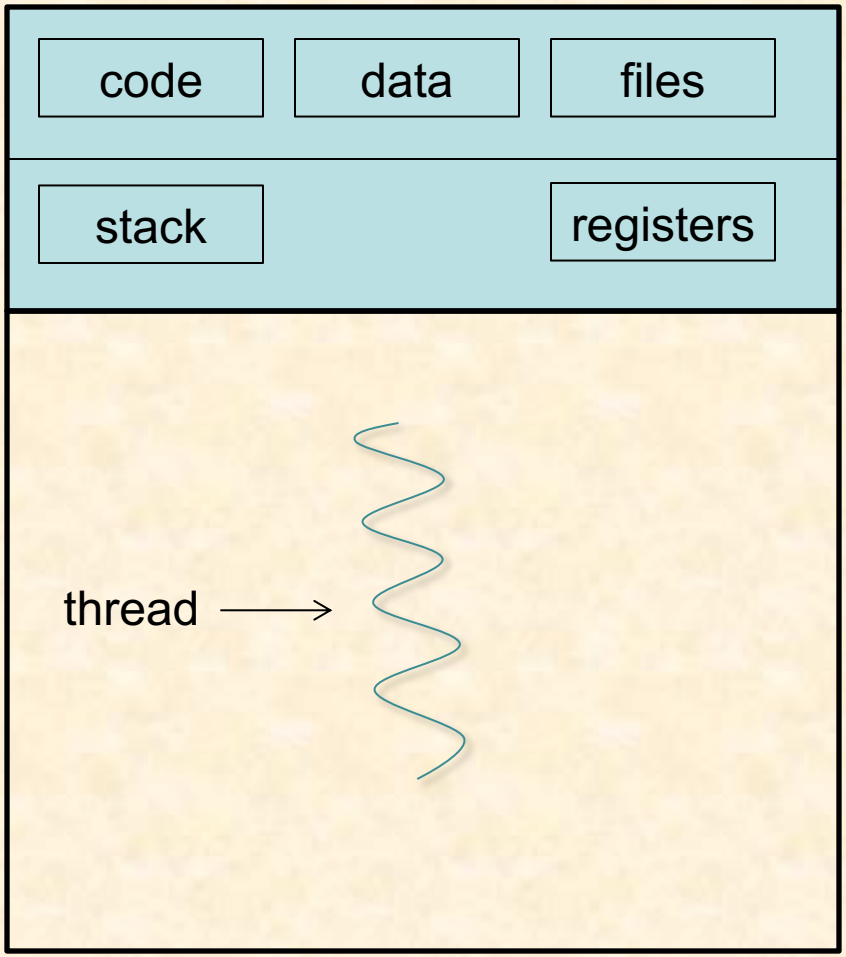
Target HPC Platform

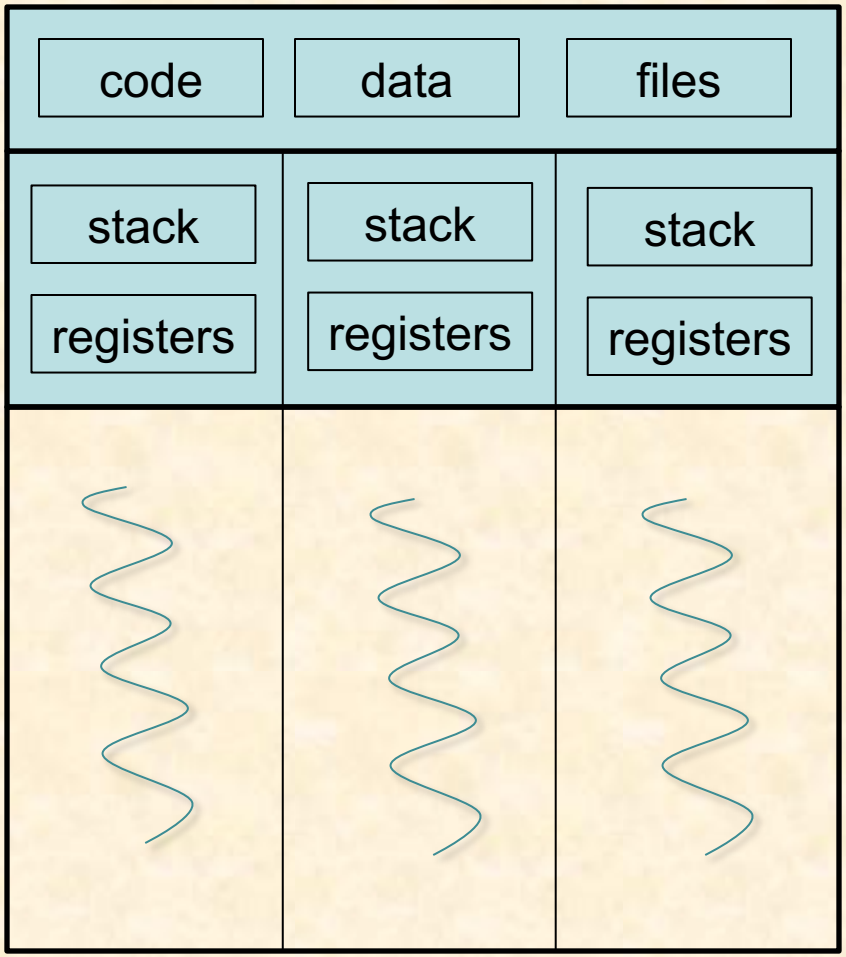
- Explicit parallelism
  - Requires specification of concurrent tasks
  - Interaction between concurrent tasks
    - Communication
    - Synchronization
- Shared address space
  - All processors share a common data space that is accessible to all processors.
  - Classified into
    - Uniform Memory Access (UMA)
      - Uni-processor (with multiple cores) fall in this category
    - Non-Uniform Memory Access (NUMA)
      - Multi-processor (with multiple cores) machines

Core concept underlying OpenMP: Threading

- Enables Concurrent execution within ***one*** process
  - Sub-process
  - Light weight process
  - Multiple threads per process
- Threads share process resources
  - Threads share process's virtual memory space
    - Consequently threads can access the same heap space
  - Threads share the same code!
    - Typically simply a method or part of a program runs as a thread
  - Share resources
    - File handles
    - Socket handles etc.
- Threads have some *limited* non-shared resources
  - Stack (limited size)
  - Thread local storage (TLS)
  - Different set of General Purpose Registers (GPR)
    - They are saved and restored by OS for each thread

Single vs. Multithreaded processes





Differences between processes & threads

Processes	***Threads***
Resources are not shared between processes. Consequently, can access more resources	Resources of a process are shared by multiple threads. Consequently, resources per thread is limited.
Can be terminated independent of other processes. Consequently, easy to monitor, control, and administer.	Can't kill just a thread. Have to kill a whole process. Harder to monitor and misbehaving threads are nightmares.

Differences between processes & threads (Contd.)

Processes	***Threads***
Heavy weight context switches	Light weight context switching
Have special streams for standard I/O.	No special streams for standard I/O. Simply share parent process's streams.
Shell can redirect or pipe I/O to other processes.	No redirection or piping concepts. Data has to be carefully shared between cooperating threads.

When to use Multiple threads?: Factor #1

1. *Application characteristics*
   - Application involves Overlapping CPU and I/O operations
     - Example: Real time Audio/Video processing
       - Data constantly flows in
       - One thread does audio processing (using separate hardware)
       - Another thread does video processing CPU
     - Example: Data processing
       - One thread does indexing of data (I/O intensive operation)
       - One thread does data compression for storage (CPU intensive)
   - Application consists of two or more concurrent tasks
       - Different tasks use CPU at different times
     - Example: Interaction with multiple users
       - Typically not all users type commands at the same time

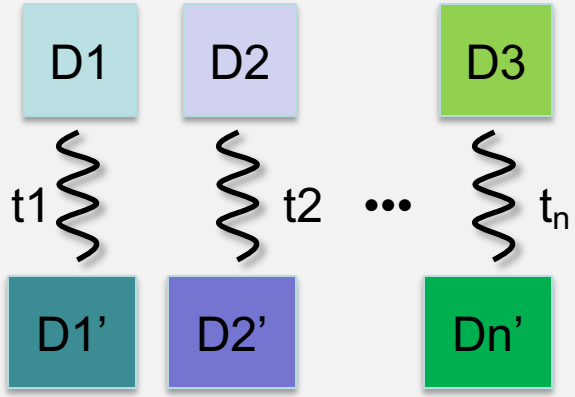
When to use Multiple threads?: Factor #2

2. *Hardware Configuration*
   - Most critical factor from performance perspective
   - Shared Memory Platform with multiple cores
     - Typical target platform for using multiple threads
   - Special hardware
     - Example: Intel Xeon Phi
   - One CPU but special support for multiple threads
     - **Hyperthreading**
       - CPU has modified pipeline for super scalar operations
       - Run 2 CPU intensive threads
   - Single CPU
     - If CPU intensive computation, better to use single thread!

Classification of parallel programs

- Data parallelism
  - Each thread/process performs same computation
  - The data for each thread/process is different

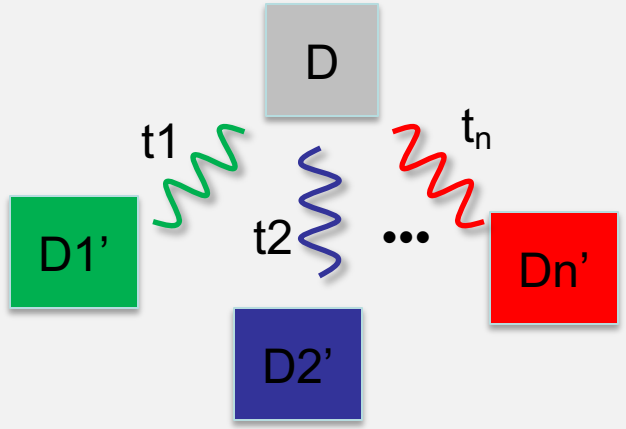
Input data is logically partitioned into independent subsets.



Each thread process a subset of data to generate logically independent subset of outputs

- Task parallelism
  - Each thread/process performs different computations
  - The data for each thread/process is the same

Same data but different processing



Each thread processes same data but differently generating different outputs

Examples of data & task parallelism

- Data parallelism
  - Use multiple threads/processes to change separate pixels in a image
  - Process multiple files concurrently using many threads or processes
- Task Parallelism
  - Run various data mining algorithms on a given piece of data using multiple threads
  - Update different indexes and relations in a database when a new record is added.
  - Convert a video to many different formats and resolutions using multiple threads or processes.
  - Searching for a given search-term on multiple indexes using several threads or processes to provide instant search results.

Explicit Threading C++ Example

```
#include <iostream>
#include <thread>

void someMethod() {
  std::cout << "Hello from Thread: "
            <<  std::this_thread::get_id()
            << std::endl;
}

int main() {
  std::thread t1(someMethod);
  std::cout << "Waiting for thread "
            << t1.get_id() << " to finish.\n";
  t1.join();
  return 0;
}
```

This method creates a thread that starts running `someMethod` as if it is a standard method call.

The `join` method waits for the thread to end *i.e.*, for the called method to end.

Advantages of Explicit Threaded programming Model

- Maximize efficiency
  - Programmer has full control of threads & resources
  - Enables fine tuning of applications for a given platform
- Better control on synchronization
  - Programs can be optimized to minimize Synchronization or "locking"
  - Depending on application lock-free solutions may be possible
- Scheduling & Load Balancing
  - Improved control on balancing load (i.e., number of instructions executed) between multiple threads
  - Balancing load on
- Enable use of special hardware
  - Dedicate some threads to run on different hardware (such as: GPUs)

Disadvantages of Explicit Threading

- Requires rewriting programs to effectively use multiple threads
  - Programmer is responsible for creating threads
  - Assigning tasks to threads
    - Can be very tricky in several situations
- Programmer must handle race conditions
  - Responsible for suitably synchronizing threads using critical sections
    - Using semaphores/mutex
  - Monitors must be used for optimizing data interchange
    - Typically used in producer-consumer type programs
- Programmer is responsible for using threads appropriate to hardware
  - Requires some understanding of underlying hardware
- Programming language constructs for threads vary
  - Programmer must be aware of idiosyncrasies
- These days threads are considered low-level primitives
  - Typically reserved for system programmers as opposed to application programmers.

Introduction to OpenMP

- An Application Program Interface (API) that may be used to explicit, shared memory parallelism
  - Uses threads to accomplish parallelism
- Comprised of three primary API components:
  - Compiler Directives
    - Directive have optional clauses that are used to customize operations of the directives
  - Runtime Library Routines
  - Environment Variables
- OpenMP is an abbreviation for Open Multi-Processing
  - Developed via collaborative work between the hardware and software industry, government and academia.

Objectives of OpenMP

- Standardization:
  - Provide a standard among a variety of shared memory architectures/platforms
  - Jointly defined and endorsed by a group of major computer hardware and software vendors
- Lean and Mean:
  - Establish a simple and limited set of directives for programming shared memory machines.
  - Significant parallelism can be implemented by using just 3 or 4 directives.
- Ease of Use:
  - Provide capability to incrementally parallelize a serial program, unlike message-passing libraries which typically require an all or nothing approach
  - Provide the capability to implement both coarse-grain and fine-grain parallelism
- Portability:
  - The API is specified for C/C++ and Fortran
- Public forum for API and membership
  - Most major platforms have been implemented including Unix/Linux platforms and Windows

Limitations of OpenMP

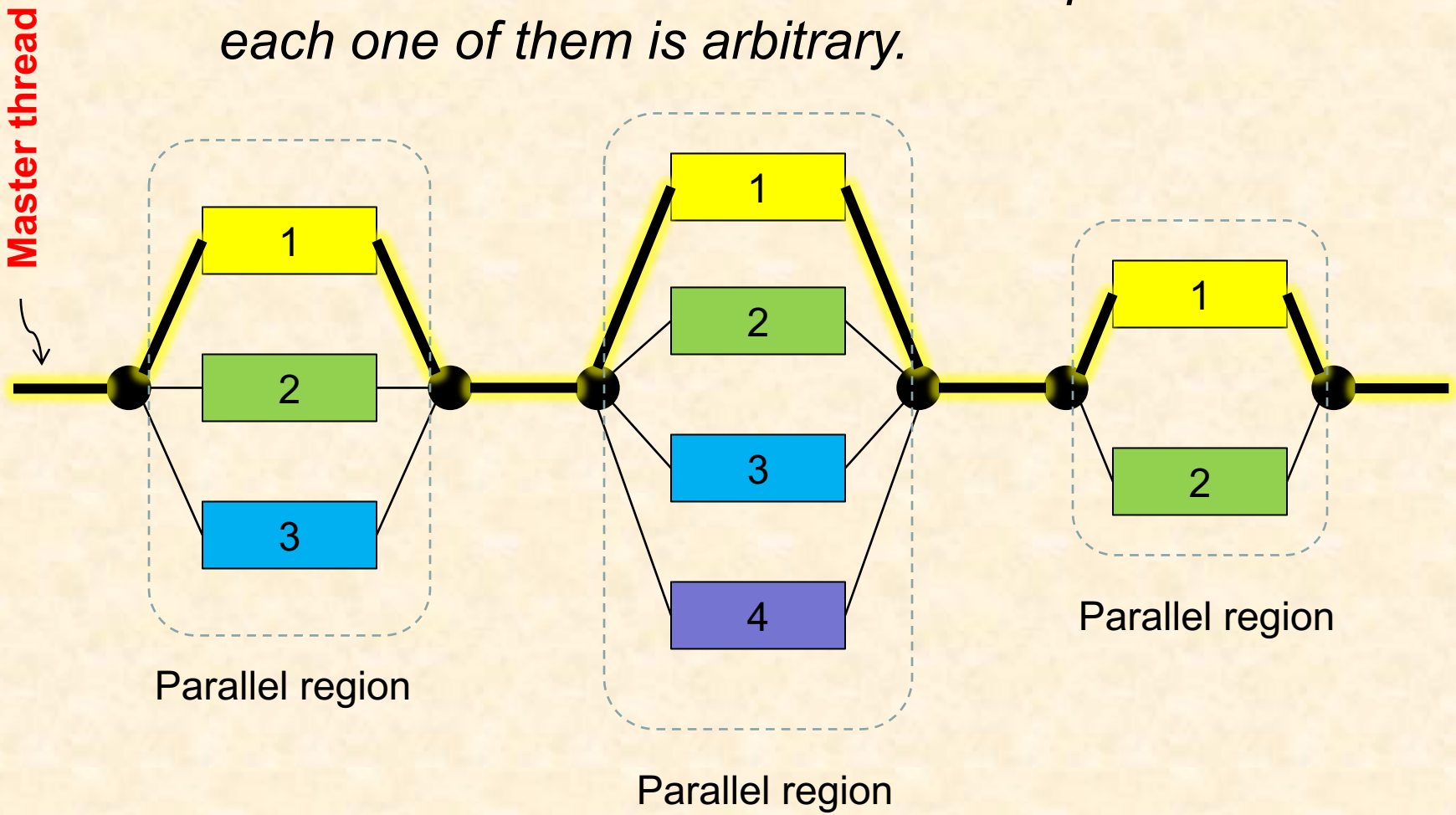
- Not meant for distributed memory parallel systems (by itself)
- Not guaranteed to make the most efficient use of shared memory
  - OpenMP performs best effort to optimize system usage
- Requires application program to handle certain tasks
  - Suitably handle data dependencies, data conflicts, or deadlocks
  - Requires programmer to check for code sequences that cause a program to be classified as non-conforming
  - The programmer is responsible for synchronizing input and output.
- Geared for C programs
  - Can be easily mixed with C++ programs

Fork-Join OpenMP Model

- OpenMP uses the fork-join model of parallel execution
  - An OpenMP program consists of a collection of threads
    - There is always one main thread called the master thread
  - All programs being as a single master thread
    - Master thread runs sequential parts (non-openMP parts) of the program
  - Whenever parallel regions are encountered
    - FORK: Master thread creates (or reuses) a team of parallel threads
    - JOIN: Once all the threads have completed, they synchronize and logically end the block of parallel execution
    - Master thread continues

Fork-Join: Conceptual view

The number of parallel regions in a program and the number of threads that comprise each one of them is arbitrary.



Important facts on OpenMP

- OpenMP API is primarily compiler-directives
  - OpenMP parallelism directives are embedded in C/C++ program
- OpenMP supports nested parallelism
  - Parallel regions can be placed inside other parallel regions
  - Some implementation may not support it
  - Nest parallelism can be controlled using environment variables
- Dynamic threads
  - OpenMP can dynamically alter number of threads used to execute parallel regions
  - Can be controlled using compiler-directives
- OpenMP does not deal with I/O
  - It is entirely delegated to programmer
  - Best to perform I/O from sequential sections of the code
- Conceptual synchronization of memory
  - OpenMP provides a “relaxed-consistency” implying threads may cache data
  - Programmer needs to override this behavior if needed

OpenMP Programming model

- OpenMP programming is primarily performed using compiler directives
  - It is possible to directly use the library but such approaches are extremely rare in practice
- OpenMP model is primary for C language
  - Can be easily mixed with C++
  - Newer versions of OpenMP are fully streamlined with C++ as well
- All OpenMP directives begin with the keywords **`#pragma omp`**
  - The `#pragma` is similar to `#include`
    - However `#pragma` is used by the compiler rather than the preprocessor
- General syntax of OpenMP directive:

```
#pragma omp directive [clause list]
{
  // structured block
}
```

  - There are different directives for different types of concurrent code
  - The optional clause list provides parameters and details for a concurrent block.

The `parallel` OpenMP directive

- The `parallel` directive is the primary clause used in OpenMP programs

```
#pragma omp parallel [clauses]
    { // start parallel
    } // end parallel
```

  - The master thread forks a team of threads at the open brace
  - The threads join at the close brace
- The following optional clauses can be used with the `parallel` directive
  - Conditional parallelization
    - A simple scalar expression to determine if block must be run in parallel
  - Degree of concurrency
    - The number of threads to use if block is to be run in parallel
  - Data handling
    - How variables and their values are used before and after the parallel block

Hello OpenMP

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

int main() {
#pragma omp parallel
    { // start parallel
        const int numThreads = omp_get_num_threads();
        const int threadID   = omp_get_thread_num();
        std::cout << "hello OpenMP from "
                  << threadID   << " of "
                  << numThreads << " threads.\n";
    } // end parallel
    return 0;
}
```

Header provides declaration for some OpenMP functions (that are C language based)

```
$ g++ -std=c++11 -g -Wall -fopenmp omp.cpp -o omp
$ export OMP_NUM_THREADS=4
$ ./omp
hello OpenMP from 0 of 4 threads.
hello OpenMP from 1 of 4 threads.
hello OpenMP from 2 of 4 threads.
hello OpenMP from 3 of 4 threads.
```

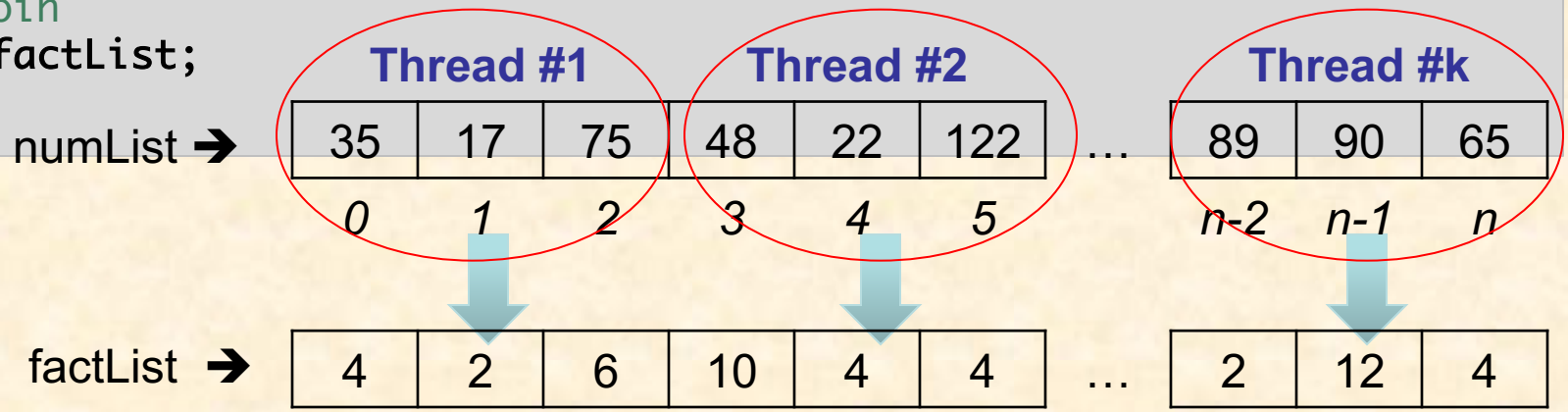
Data parallelism: Parallelizing a method

- Recollect key data parallel concepts:
  - Each thread must operate on an independent unit of data
  - The output is typically stored in an independent location
    - For now assume two threads cannot modify the same region of memory

```
// Returns the number of factors for a given number.
int getFactorCount(int num);

// Compute factors for a list of numbers in a
// data parallel manner
IntVec getFactors(const IntVec& numList) {
    IntVec factList(numList.size());
    for(size_t idx = 0; (idx < numList.size()); idx++) {
        factList[idx] = getFactorCount(numList[idx]);
    }
    return factList;
}
```

Data parallel implementation

```
// Compute factors for a list of numbers in a
// data parallel manner
IntVec getFactors(const IntVec& numList) {
  IntVec factList(numList.size());
#pragma omp parallel
  {  // fork

     const int iterPerThr = numList.size() / omp_get_num_threads();
     const int startIndex = omp_get_thread_num() * iterPerThr;
     const int endIndex   = startIndex + iterPerThr;
     for(int idx = startIndex; (idx < endIndex); idx++) {
        factList[idx] = getFactorCount(numList[idx]);
     }

  }  // join
  return factList;
}
```



Data parallel implementation (fix edge case)

```
IntVec getFactors(const IntVec& numList) {
  IntVec factList(numList.size());
#pragma omp parallel
  {  // fork
    const bool isLastThr = omp_get_num_threads() ==
                                  omp_get_thread_num() + 1;
    const int iterPerThr = numList.size() / omp_get_num_threads();
    const int startIndex = omp_get_thread_num() * iterPerThr;
    const int endIndex   = (isLastThr ? numList.size() :
                                startIndex + iterPerThr);
    for(int idx = startIndex; (idx < endIndex); idx++) {
       factList[idx] = getFactorCount(numList[idx]);
    }
  }  // join
  return factList;
}
```

This math fixes logic from previous slide to handle the case where the list's size is not exactly divisible by number of threads

General rules for OpenMP directives in C++ programs

- All directives are case sensitive
- Directives follow conventions of the C/C++ standards for compiler directives
- Only one directive-name may be specified per directive
- Each directive applies to at most one succeeding statement
  - Which can be a block and therefore applies to the whole structured block.
- Long directive lines can be "continued" on succeeding lines by escaping the newline character with a backslash ("\") at the end of a directive line.

Running Programs on a cluster

- The following steps are involved in running an OpenMP program on the cluster
  - Compile the program using `g++` command
    - Ensure you compile `-fopenmp` feature flag
  - Create a SLURM script file to run a job on the cluster
    - This is a standard bash script
    - The script file specifies information about the resources needed for the parallel program, such as: number of CPUs, memory, runtime etc.
    - The script should include calls to your program program
  - Submit the script file via `sbatch` command to create a new job on the cluster
    - The job can be run in interactive or batch mode

Example SLURM script (file name: hello_slurm.sh)

```
#!/bin/bash
# Lines beginning with # are comments
# Lines beginning #SBATCH are processed by slurm
#SBATCH --account=PMIU0184
#SBATCH --time=1:00:00
#SBATCH --mem=2GB
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=3

export OMP_NUM_THREADS=$SLURM_NTAS

/usr/bin/time -v ./hello one two

#end of script
```

The line specifies that this job is requesting 1 hour to wall clock time to run on the cluster.

This line specifies that the job is requesting a total of 2 gigabytes of memory. This value is the sum of the anticipated peak virtual memory used by all the parallel processes in the program.

Example SLURM script (file name: hello_slurm.sh)

```
#!/bin/bash
# Lines beginning with # are comments
# Lines beginning #SB
#SBATCH --account=PMI
#SBATCH --time=1:00:00
#SBATCH --mem=2GB
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=3

export OMP_NUM_THREADS=$SLURM_NTASKS

/usr/bin/time -v ./hello one two

#end of script
```

This line specifies that the job is requesting 1 compute nodes with 3 cores on each compute node dedicated to the SLURM job.

Set number of threads that OpenMP should use based on SLURM job configuration.

Scoping or Extent Rules for OpenMP Directives

- Static (Lexical) Extent:
  - The code textually enclosed between the beginning and the end of a structured block following a directive.
    - This is the most common case
  - The static extent of a directives does not span multiple routines or code files
- Orphaned Directive:
  - An OpenMP directive that appears independently from another enclosing directive
    - Such as a `parallel` directive
  - It exists outside of another directive's static (lexical) extent.
  - Can span routines and possibly code files
- Dynamic Extent:
  - The dynamic extent of a directive includes both its static (lexical) extent and the extents of its orphaned directives.
  - Used for defining critical sections

Clauses for parallel section

- The clauses that can be used with a `parallel` section are:

  `#pragma omp parallel` *[clause ...]*

  Where *clause* can be:

  - `if` (*scalar_expression*)
  - `num_threads` (integer-expression)
  - `private` (list)
  - `shared` (list)
  - `default` (`shared` | `none`)
  - `firstprivate` (list)
  - `reduction` (operator: list)
  - `copyin` (list)

The `if` clause for a `parallel` section

- The `if` clause permits conditional parallelization of a section of code
  - If the specified expression evaluates to `true` then the section is run in parallel
    - If the conditional expression evaluates to `false` then the section is executed serially by the master thread

```
// Figure out if parallelization is desired
bool parallelize = shouldRunInParallel();

#pragma omp parallel if (parallelize)
    { // start parallel


    } // end parallel
```

The `num_threads` clause

- The `num_threads` clause can be used to fix the number of threads to be used to run a parallel section
  - The number of threads can be either a literal constant
  - Or a value computed in the program

```
// The number of threads depends on getNumThread
// method's return value
#pragma omp parallel num_threads(getNumThreads())
    { // start parallel


    } // end parallel
```

Threads for a parallel block

- The number of threads in a parallel region is determined by the following factors, in order of precedence:
  - Evaluation of the **`if`** clause
  - Setting of the **`num_threads`** clause
  - Use of the **`omp_set_num_threads()`** library function in the C++ program
  - Setting of the **`OMP_NUM_THREADS`** environment variable
  - Implementation default
    - Usually 1
    - Or the number of CPUs on a node, though it could be dynamic
- Threads are numbered from 0 to N-1
  - The master thread is always thread with number 0.

Data Handling clauses

- The `private` clause specifies list of private variables
  - Each thread gets a different copy of the private variable (defined earlier)
    - The initial value is undefined in the thread
  - Changes to private variables are not visible outside the thread
- The `firstprivate` clause specifies list of private variables
  - However, each copy in every thread is initialized to the value just before the parallel block
- The `shared` clause is used to share the same variable between multiple threads
  - Care must be taken when using shared variables to avoid race conditions
  - Frequently used for large data structures in data-parallel applications

Example of data handling clauses

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

int main() {
    int a = 0, b = 20, c = 100;
#pragma omp parallel private(a) firstprivate(b) \
shared(c)
    {
        a = 10;
        if (omp_get_thread_num() == 0) {
            c = c + a + b;  // Only one thread changes 'c'
        }
    }
    std::cout << " a = " << a << " b = " << b
              << " c = " << c << std::endl;
    return 0;
}
```

```
$ g++ -std=c++14 -g -Wall -fopenmp omp.cpp -o omp
$ export OMP_NUM_THREADS=4
$ ./omp
a = 0 b = 20 c = 130
```

Default data handling

- The `default` clause defines the default handling for variables
  - Variables defined outside a parallel directive and used within a parallel section
  - The `default(shared)` implies a variable is shared by default
  - The `default(none)` implies that the state of each variable must be specified
    - This is the preferred setting to guard against errors

```
int a = 0, b = 20, c = 100;
#pragma omp parallel private(a, b) default(shared) \
shared(c) // Variable c is shared by default
    {
        a = 10;
        c = c + a + b;
    }
```

Race Conditions

- Multiple threads updating a shared variable causes race conditions
  - Be cautious when developing data parallel applications
  - Here is an example of a code that has potential race-condition:

```
int racer() {
  int sum = 0;
#pragma omp parallel
  {    // fork
       for (int i = 0; (i < 10000); i++) {
          sum++;
       }
  }    // join
       return sum;
}
```

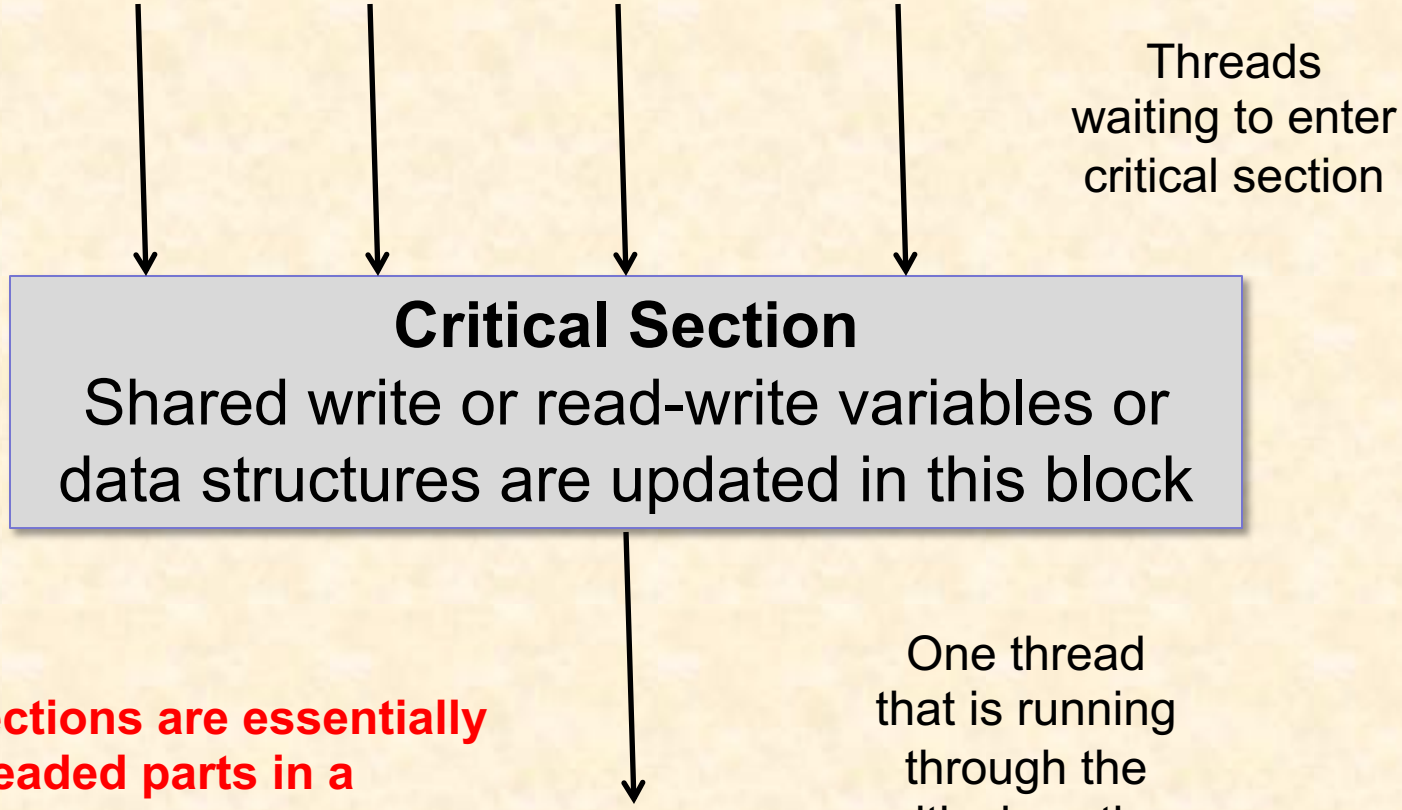
Multiple threads updating the value of `sum` causes a race condition. The value of `sum` is indeterminable.

Avoiding race conditions: Critical Sections

- Critical sections are used to eliminate race conditions
  - Only one thread executes in the critical section
    - Order in which threads run through a critical section is unspecified
  - Other threads block if they need access to the same critical section
  - Must be used sparingly
    - Degrades performance if it is overused
- General form for critical section:

```
#pragma omp critical [(name)]
{
  // Critical Section
}
```

Conceptual view of critical section



Critical sections are essentially single threaded parts in a multithreaded code to ensure consistent read/write of shared variables or data structures

Example of Critical Section

```
// Assume this method is called from multiple threads!!
std::pair<int, int>
oddEven(const std::vector<int>& values, int start, int stop) {
    int oddSum = 0, evenSum = 0;
    for (int i = start; (i < stop); i++) {
        if (values[i] % 2 == 0) {
#pragma omp critical (evenCS)
            {
                evenSum += values[i];
            }
        } else {
#pragma omp critical (oddCS)
            {
                oddSum += values[i];
            }
        }
    }
    return std::make_pair(oddSum, evenSum);
}
```

Example of split critical sections

```
using IntVec = std::vector<int>;
// Shared read-write lists that are operated on by multiple threads!!
IntVec list1, list2;
```

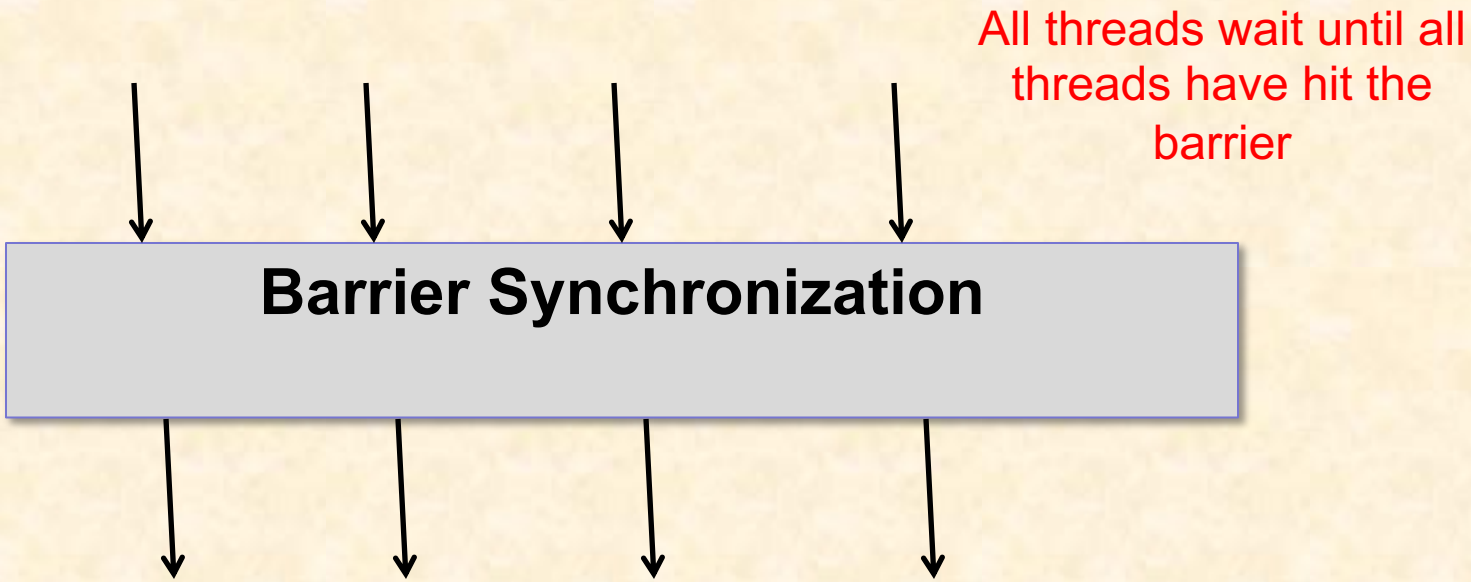
```
int get1(size_t index) {
#pragma omp critical (vec1)
  return list1.at(index);
}

void add1(int val) {
#pragma omp critical (vec1)
  list1.push_back(val);
}
```

```
int get2(size_t index) {
#pragma omp critical (vec2)
  return list2.at(index);
}

void add2(int val) {
#pragma omp critical (vec2)
  list2.push_back(val);
}
```

```
void swap(size_t index1, size_t index2) {
#pragma omp critical (vec1)
#pragma omp critical (vec2)
  std::swap(list1.at(index1), list2.at(index2));
}
```

Barrier synchronization

- Barriers are used to synchronize threads
  - They are meant to be used in a parallel code path
- Barrier is a conceptual point in code where threads wait until all threads reach that barrier
  - They are useful to ensure all threads complete an operation



Initializing shared data structures

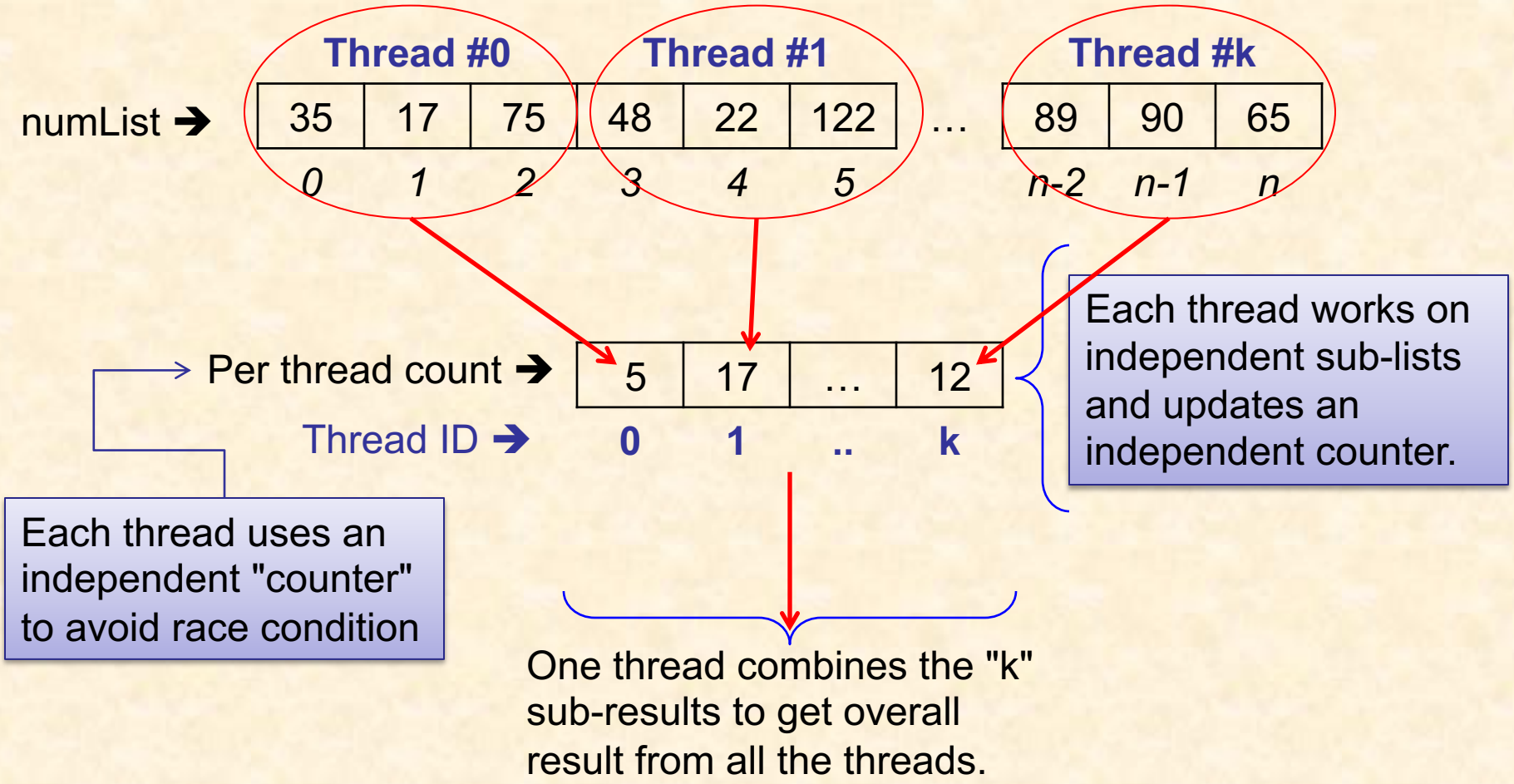
```
int parallelMethod() {
    IntVec vec;
#pragma omp parallel shared(vec)
    {   // fork
        const int thrCnt = omp_get_num_threads();
        const int thrID  = omp_get_thread_num();
#pragma omp critical
        if (thrID == 0) {
            vec.resize(thrCnt);
        }
#pragma omp barrier
        // Now each thread can operate on vec[thrID]
        // safely.
        // Do some data parallel operations here...

    }  // join
    return 0;
}
```

Parallelization Example

```
int countEvensSerial(const IntVec& vec) {
    // Count even numbers
    int count = 0;
    for (int val : vec) {
        if (val % 2 == 0) {
            count++;
        }
    }
    return count;
}
```

How would you parallelize this `countEvensSerial` method?

Overview of data parallel approach



Parallelization

```
int countEvens(const IntVec& vec) {
    const int MaxThr = 8;   // Assumption just for this example!
    IntVec counts(MaxThr);  // Per-thread counts
#pragma omp parallel shared(vec, counts)
    {
        // Compute information about this thread.
        int  threadId  = omp_get_thread_num();
        bool isLastThr = (threadId == omp_get_num_threads() - 1);
        // Compute the range of numbers this thread must check.
        int iterPerThr = vec.size() / omp_get_num_threads();
        int startIdx   = threadId * iterPerThr;
        int endIdx     = (isLastThr ? vec.size() : startIdx + iterPerThr);
        // Count even numbers for this thread.
        for (int i = startIdx; (i < endIdx); i++) {
            if (vec[i] % 2 == 0) {
                counts[threadId]++;
            }
        }
    }
    // Find total across all threads
    int total = std::accumulate(counts.begin(), counts.end(), 0);
    return total;
}
```

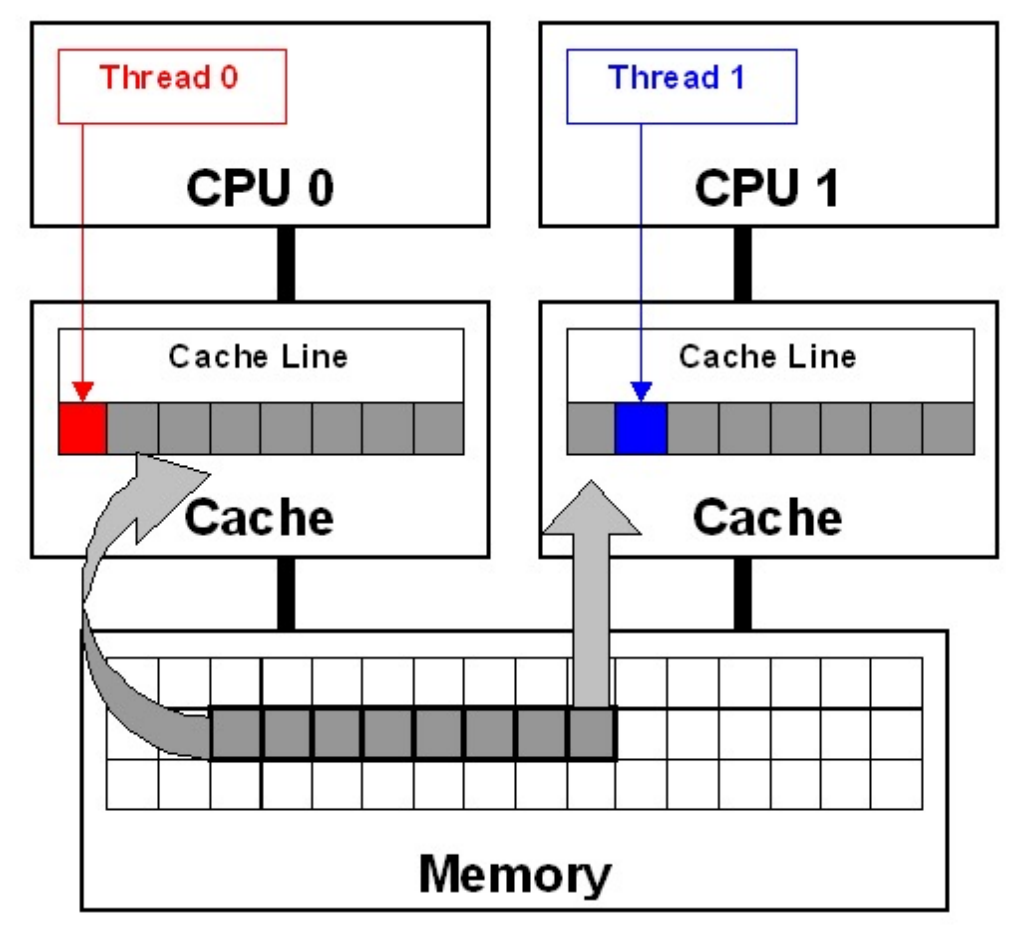
Problem with Parallelization

```
int countEvens(const IntVec& vec) {
    const int MaxThr = 8;   // Assumption just for this example!
    IntVec counts(MaxThr);  // Per-thread counts
#pragma omp parallel shared(vec, counts)
    {
        // Compute information about this thread.
        int  threadId  = omp_get_thread_num();
        bool isLastThr = (threadId == omp_get_num_threads() - 1);
        // Compute the range of numbers this thread must check.
        int iterPerThr = vec.size() / omp_get_num_threads();
        int startIdx   = threadId * iterPerThr;
        int endIdx     = (isLastThr ? vec.size() : startIdx + iterPerThr);
        // Count even numbers for this thread.
        for (int i = startIdx; (i < endIdx); i++) {
            if (vec[i] % 2 == 0) {
                counts[threadId]++;
            }
        }
    }
    // Find total across all threads
    int total = std::accumulate(counts.begin(), counts.end(), 0);
    return total;
}
```

This 1 line of code that is constantly changing value in an array can negatively impact caching!

False Sharing

- False sharing is a scenario where independent values used by independent threads are in 1 cache line needing the whole cache line to be updated in multiple caches
  - False sharing can be avoided by using thread-private variables



Avoiding false sharing

```
int countEvensNoShare(const IntVec& vec) {
    const int MaxThr = 8;   // Assumption!
    IntVec counts(MaxThr);  // Per-thread counts
#pragma omp parallel shared(vec, counts)
    {
        // Compute information about this thread.
        int  threadId  = omp_get_thread_num();
        bool isLastThr = (threadId == omp_get_num_threads() - 1);
        // Compute the range of numbers this thread must check.
        int iterPerThr = vec.size() / omp_get_num_threads();
        int startIdx   = threadId * iterPerThr;
        int endIdx     = (isLastThr ? vec.size() : startIdx + iterPerThr);
        // Count even numbers for this thread.
          int counter = 0;
        for (int i = startIdx; (i < endIdx); i++) {
            if (vec[i] % 2 == 0) {
                counter++;
            }
        }
        // Set counts per thread
        counts[threadId] = counter;
    }
    // Find total across all threads
    int total = std::accumulate(counts.begin(), counts.end(), 0);
    return total;
}
```

Shared variable is updated just once, thereby reducing (but not eliminating) impact of false sharing

The `reduction` clause

- The reduction clause defines how values of private variables are combined when threads join
  - The syntax of the reduction clause is

    `reduction`(operator`:` variable list)
  - Where the operator can be one of: `+`, `-`, `*`, `&`, `|`, `^`, `&&` or `||`
  - Variables are treated as `firstprivate` variables

Parallel Accumulate (std::accumulate)

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <cstdlib>
#include <omp.h>


int main() {
    int sum = 0;
    std::vector<int> values(1000000);
    std::generate(values.begin(), values.end(), rand);
#pragma omp parallel default(none) reduction(+: sum) shared(values)
    {
        const int iterPerThr = values.size() / omp_get_num_threads();
        const int startIndex = omp_get_thread_num() * iterPerThr;
        const int endIndex   = startIndex + iterPerThr;
        for(int idx = startIndex; (idx < endIndex); idx++) {
            sum += values[idx];
        }
    }
    std::cout << "sum = " << sum << std::endl;
    return 0;
}
```

The entries in `values` vector is split between multiple threads to compute local-sum. The `reduction` clause is used to add local-sum from each thread to determine overall sum of all numbers in `values` vector.

Data Handling in OpenMP

- Data handling is a critical factor that influences performance
  - If a thread initializes and uses a variable (such as loop indices) and no other thread access the data, then such variables should be `private`.
    - If a thread needs to obtain pre-initialized values for such variables use `firstprivate`.
  - If multiple threads manipulate a single piece of data:
    - First explore ways of breaking these manipulations into independent local operations
      - Using `reduction` clause can help in certain cases
    - Try to organize larger data structures as independent smaller data structures
  - Large data structures that are read-only
    - They may be shared
  - Shared read-write variables should be used as a last resort

Checking for race conditions

- Race conditions can be hard to troubleshoot
  - Particularly if program is large / complex
  - Involves multiple 3rd party libraries
- GCC provides thread sanitizers to help locating race conditions
  - Need to compile programs with following additional flags:

    ```
    -fsanitize=thread -fPIC -pie
    ```

  - Need to provide comprehensive set of test cases to utilize various code paths
  - Programs do run slower with sanitizer enabled

Output from thread sanitizer

```
[raodm@mualhpcp01 c++]$ g++ -fopenmp -std=c++11 -g -Wall -fpic -pie -fsanitize=thread racer.cpp -o racer
[raodm@mualhpcp01 c++]$ ./racer
==================
WARNING: ThreadSanitizer: data race (pid=11906)
  Write of size 4 at 0x7fff1cb0aec0 by thread T3:
    #0 racer() [clone ._omp_fn.0] /home/raodm/hackArea/c++/racer.cpp:9 (racer+0x000000000f81)
    #1 gomp_thread_start ../../../gcc-4.9.2/libgomp/team.c:117 (libgomp.so.1+0x00000000d7d5)

  Previous read of size 4 at 0x7fff1cb0aec0 by thread T2:
    #0 racer() [clone ._omp_fn.0] /home/raodm/hackArea/c++/racer.cpp:9 (racer+0x000000000f6c)
    #1 gomp_thread_start ../../../gcc-4.9.2/libgomp/team.c:117 (libgomp.so.1+0x00000000d7d5)

SUMMARY: ThreadSanitizer: data race /home/raodm/hackArea/c++/racer.cpp:9 racer() [clone ._omp_fn.0]
==================
sum = 22230
ThreadSanitizer: reported 1 warnings
```

OpenMP `for` directive

- Most programs involve iteration over a finite space or finite number of elements
  - Typically the `for`-loop is used for such iterations
- OpenMP `for` directive
  - Used within an OpenMP parallel section
  - Used to split parallel iteration spaces across threads
  - Provides various clauses to control properties of the parallel `for`-loop
    - Data handling clauses
    - Clause to handle update of loop iteration variable
    - Scheduling clauses to manage iterations with varying times

Restrictions for using `for` directive

- Restrictions on using for directive
  - The directive must be placed immediately before the concurrent `for`-loop.
  - The loop iteration variable must be an integer
    - Initialization must be an integer expression
    - The iteration variable maybe incremented or decremented
  - The loop control parameters must be the same for all threads
  - Program correctness must not depend on which thread executes a particular iteration
  - It is illegal to branch out of the loop (using a `goto` or `break` statement)

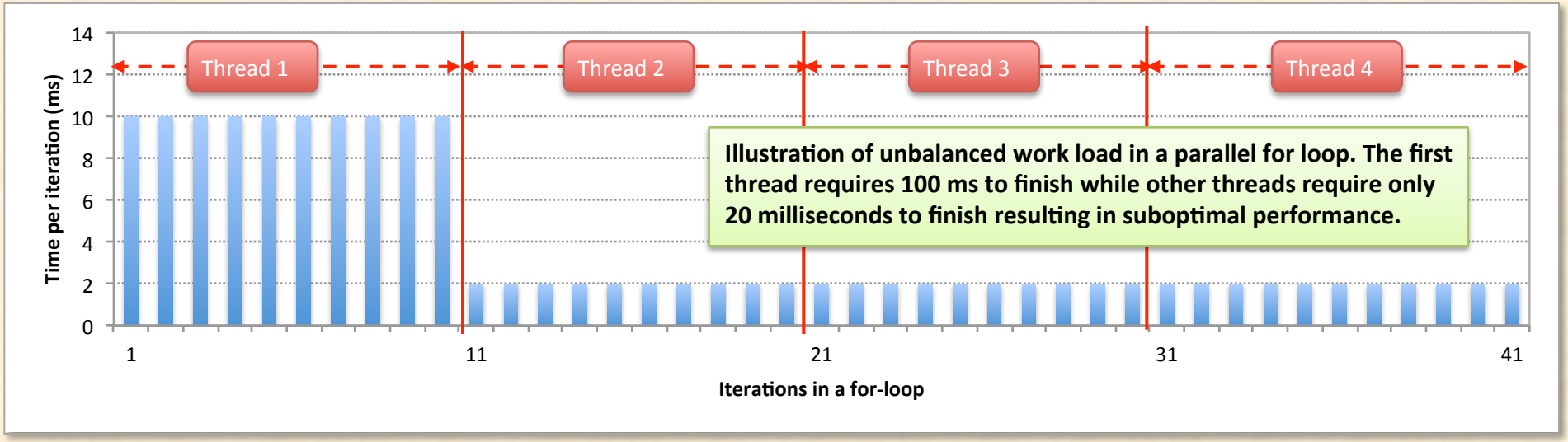
Parallel Accumulate (std::accumulate)

```
int accumulate(const std::vector<int>&
  values) {
  int sum = 0;

#pragma omp parallel for default(none) \
        shared(values) reduction(+: sum)
  for(size_t i = 0; (i < values.size()); i++)
  {
      sum += values[i];
    }
  } // end parallel
  return sum;
}
```

The entries in `values` vector is split between multiple threads in the for-loop to compute a local-sum. The `reduction` clause is used to add local-sum from each thread to determine overall sum of all numbers in `values` vector.

Performance of OMP for

- Each loop iteration can take differing amount of time
  - Until all threads have joined, a parallel for cannot complete
    - The slowest thread controls when the loop finishes
- Example of 40 iterations divided between 4 threads:
  - Note that the load on threads is unbalanced



Load balancing

- Load balancing is a common theme on many distributed systems
  - Applicable to web-servers etc.
  - Similar issues arise in distributed databases
- Load on multiple threads must be balanced
  - For efficient use of multiple threads
- Load balancing can be tricky
  - Inputs may not be known
  - Number of threads may not be known
  - Time for running each iteration may vary

Load balancing / scheduling

- OpenMP `parallel for` facilitates load balancing via a `schedule` clause:
  - Schedule clause requires 1 scheduling method
    - The method must be one of : `static`, `dynamic`, `guided`, or `runtime`.
    - Some scheduling methods accept optional parameters
  - Programmer is responsible for choosing appropriate scheduling method based on application characteristics

```
#pragma omp parallel for default(none) \
        shared(values) schedule(static, 1)
for(size_t i = 0; (i < values.size()); i++) {
  process(values[i]);
} // end parallel for
```

The `for` directive clause: `schedule`

- There are four scheduling / load-balancing options:
  1. Static scheduling (default)
     - Accepts an optional chunk size
     - Chunk size is set/determined at beginning of the loop
       - No additional runtime overhead. Best approach if variance in time-per-iteration is small.
     - Chunks are assigned to threads in a round robin manner
  2. Dynamic scheduling
     - Chunk sizes are determined similar to static scheduling
     - But, chunks are assigned to threads when threads become idle
       - Incurs additional runtime overhead as it has to maintain an internal queue to enable scheduling and manage threads
       - Best approach when time per chunk can vary; but time per item in a chunk is sufficiently close (~15% to 20% difference).

The `for` directive clause: `schedule` (contd.)

- Scheduling / load-balancing options:
  3. Guided scheduling
     - Accepts initial maximum chunk size as parameter
     - Similar to dynamic scheduling, but chunk sizes are exponentially decreased as loop progresses
     - Chunks are dynamically assigned to threads
       - Effective when data is imbalanced
       - Incurs overheads for managing queue, chunk size, and scheduling/managing threads.
  4. Runtime scheduling
     - Environment variable (`OMP_SCHEDULE`) is used to chose between *static*, *dynamic*, or *guided* scheduling
       - Chunk size is also specified in the environment variable.
       - Example: `export OMP_SCHEDULE="guided, 100"`

The `for` directive: General format

- The general form of a for directive is:

  `#pragma omp for` *[clause ...]*

  Where *clause* can be:

  - `private` (list)
    - Similar to usage in `parallel` directive
  - `firstprivate` (list)
    - Similar to use in `parallel` directive
  - `reduction` (operator: list)
    - Similar to use in `parallel` directive
  - `lastprivate` (list)
  - `ordered`
  - `nowait`
  - `schedule` (list)

The `for` directive clause: `ordered`

- The `ordered` clause is used to preserve sequential order of iteration in a parallel `for`-loop
  - Required in certain cases due to data dependencies across iterations of a loop
  - Ordered for-loops typically run sequentially
    - Consequently performance impacted

```
void mystery(const std::vector<int>& list) {
#pragma omp parallel shared(list)
 {
    // Parallel for-loop
    // Compute cumulative sum of values in list.
    int std::vector<int> sum(list);
#pragma omp for ordered
    for(size_t i = 1; (i < list.size()); i++) {
      sum[i] = sum[i - 1] + list[i];
    }
    // Parallel operation using cumulative sum
  } // End parallel
}
```

The `for` directive clause: `nowait`

- The `for`-directive within a `parallel` construct executes an implicit barrier
  - At the end of each `for`-directive the system waits for all threads to finish the `for`-loop
- The `nowait` clause is used to avoid the implicit barrier
  - Execution proceeds to subsequent statements without waiting for all threads to finish
    - Saves on idling and synchronization overheads
  - Subsequent statements typically perform operations that are mutually exclusive of the previous ones
  - This is applicable only for directives and statements within the same `parallel` construct

Example of `nowait`

```
int count(const std::vector<int>& list1, std::vector<int>& list2,
   int value) {
    int count = 0;
#pragma omp parallel default(none) shared(list1, list2),
   firstprivate(value), reduction(+: count)
#pragma omp for nowait
    for(size_t i = 0; (i < list1.size()); i++) {
        if (list1[i] == value) {
            count++;
        }
    }
#pragma omp for
    for(size_t i = 0; (i < list2.size()); i++) {
        if (list2[i] == value) {
            count++;
        }
    }
    return count;
}
```

The `for` directive clause: `lastprivate`

- The `lastprivate` clause is used to ensure that the value copied back into the original variable is obtained from the thread that ran the last batch (sequentially speaking) of iterations
  - Handy for using loop-index values after a parallel `for` has finished.
  - A reasonably rarely used clause

Atomic directive

- The `atomic` directive provides a simple critical section that just update a scalar variable:
  - The update instructions can be one of the form:
    - `x` binary_operation = expr
  - The expr cannot have a reference to `x` itself.
    - The atomic directive only applies to the scalar variable `x` and not to expr!
  - The binary_operation is one of:
    - +, *, -, /, &, ||
    - <<, >>
      - This is shift-left and shift-right (not stream insertion or extraction operators)

Example of `atomic` directive

```
std::pair<int, int>
oddEven(const std::vector<int>& values) {
    int oddSum = 0, evenSum = 0;
#pragma omp parallel for
    for(size_t i = 0; (i < values.size()); i++) {
        if (values[i] % 2 == 0) {
#pragma omp atomic
            evenSum += values[i];
        } else {
#pragma omp atomic
            oddSum += values[i];
        }
    }
    return std::make_pair(oddSum, evenSum);
}
```

The `sections` directive

- The `sections` directive can be used to assign different tasks to different threads
  - Multiple tasks can be assigned to one thread depending on number of tasks vs. number of threads
  - Each task should be conceptually independent of the other
- The general form of the `section` directive is:

```
#pragma omp parallel
{
#pragma omp sections [clause list]
  {
#pragma omp section
    /* structured block for task */
#pragma omp section
    /* structured block for task */
  } // sections
} // parallel
```

Example of sections

```
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
    {
      taskA();
    }
    #pragma omp section
    {
      taskB();
    }
    #pragma omp section
    {
      taskC();
    }
  } // sections
} // parallel
```

- Valid clauses for sections includes:
  - private
  - firstprivate
  - lastprivate
  - reduction
  - nowait

Merging Directives

- The `parallel` directive can be merged with `for` and `sections` directives to streamline code
  - Examples (right-hand-side shows merged directives):

```
#pragma omp parallel shared (n)
  {
    #pragma omp for
    for (i = 0 < i < n; i++) {
      // body of parallel for loop
    }
  }
```

```
#pragma omp parallel for shared (n)
   {
    for (i = 0 < i < n; i++) {
      // body of parallel for loop
      }
   }
```

```
#pragma omp parallel
{
  #pragma omp sections
  {
    …
  } // sections
} // parallel
```

```
#pragma omp parallel sections
  {
    …
  } // parallel sections
```

Summary

- OpenMP provides a layer on top of native threads
  - Several applications can be easily multithreaded
    - Particularly when underlying problem has a static or regular structure.
  - OpenMP handles overheads of creating and managing threads
- There are a few drawbacks
  - Explicit threading can make data exchange more apparent
    - Alleviates data movement, false sharing, and contention
  - OpenMP is not supported in all programming languages
  - Explicit threading provides more composite synchronization features