CS225 Lab11: Reductions with OpenMP

Chase Geigle

CS 225, UIUC

April 10, 2012

Motivating example: Finding a vector's sum

- Consider the following code for finding the sum of the integers in a vector.

```
int findSum( const vector<int> & list ) {
    int sum = 0;
    for( int i = 0; i < list.size(); i++ )
        sum += list[i];
    return sum;
}
```

Motivating example: Finding a vector's sum

- Consider the following code for finding the sum of the integers in a vector.

```
int findSum( const vector<int> & list ) {
    int sum = 0;
    for( int i = 0; i < list.size(); i++ )
        sum += list[i];
    return sum;
}
```

- Can this be parallelized?

Motivating example: Finding a vector's sum

- Consider the following code for finding the sum of the integers in a vector.

```
int findSum( const vector<int> & list ) {
    int sum = 0;
    for( int i = 0; i < list.size(); i++ )
        sum += list[i];
    return sum;
}
```

- Can this be parallelized?
- Not trivially. Why? What happens with the `sum` variable?

Down to the machine level

- `sum += list[i]` translates to **multiple instructions** in machine code.

Down to the machine level

- `sum += list[i]` translates to **multiple instructions** in machine code.
- Each CPU core (which runs a thread) can only perform arithmetic on things that are currently loaded in.

Down to the machine level

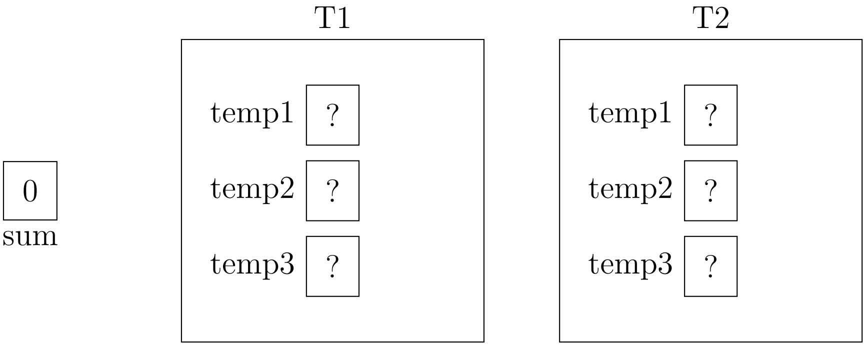
- `sum += list[i]` translates to **multiple instructions** in machine code.
- Each CPU core (which runs a thread) can only perform arithmetic on things that are currently loaded in.
- Need to load in `sum` and `list[i]` before they can be added.

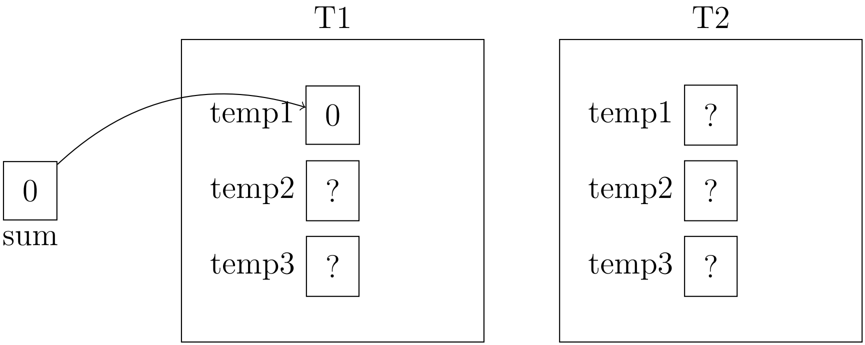
Down to the machine level

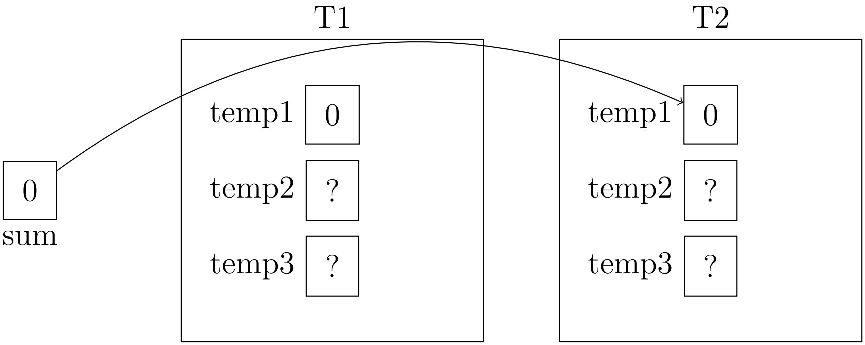
- `sum += list[i]` translates to **multiple instructions** in machine code.
- Each CPU core (which runs a thread) can only perform arithmetic on things that are currently loaded in.
- Need to load in `sum` and `list[i]` before they can be added.
- Then the result must be stored back into memory.

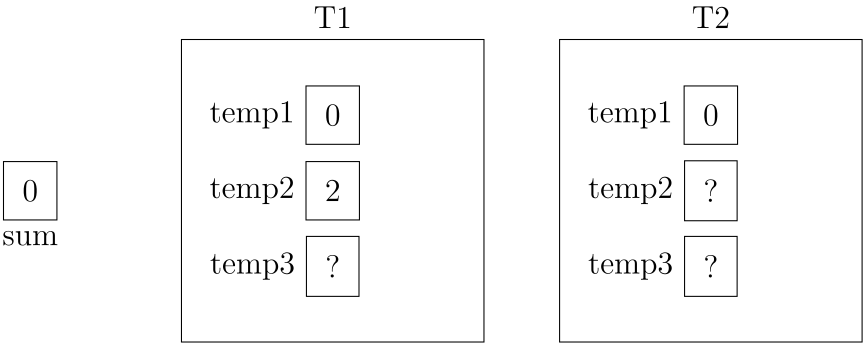
Down to the machine level

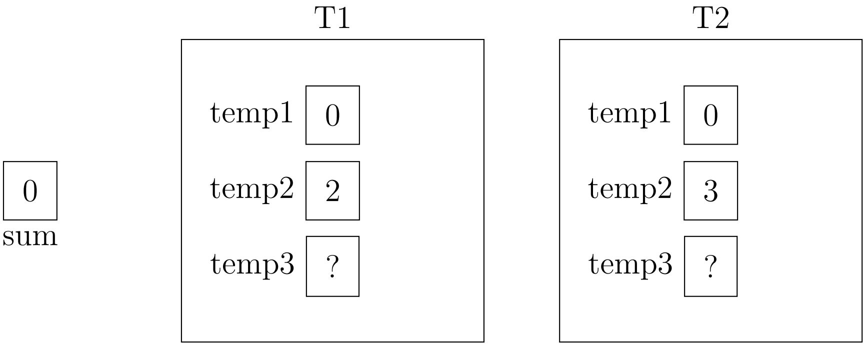
- `sum += list[i]` translates to **multiple instructions** in machine code.
- Each CPU core (which runs a thread) can only perform arithmetic on things that are currently loaded in.
- Need to load in `sum` and `list[i]` before they can be added.
- Then the result must be stored back into memory.
- But why is this a problem? What happens if we interleave these multiple machine instructions?

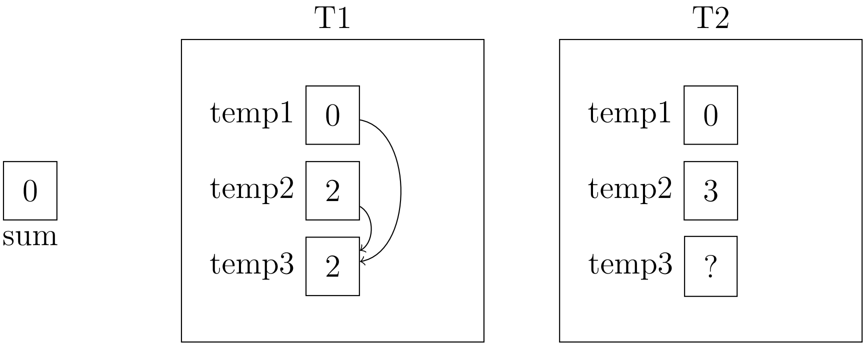

0
sum
T1
temp1 ?
temp2 ?
temp3 ?
T2
temp1 ?
temp2 ?
temp3 ?

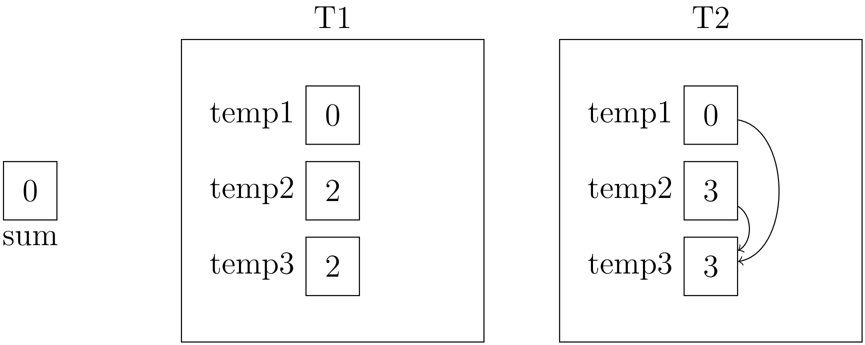

T1
T2
temp1
0
temp2
?
temp3
?
temp1
?
temp2
?
temp3
?
0
sum

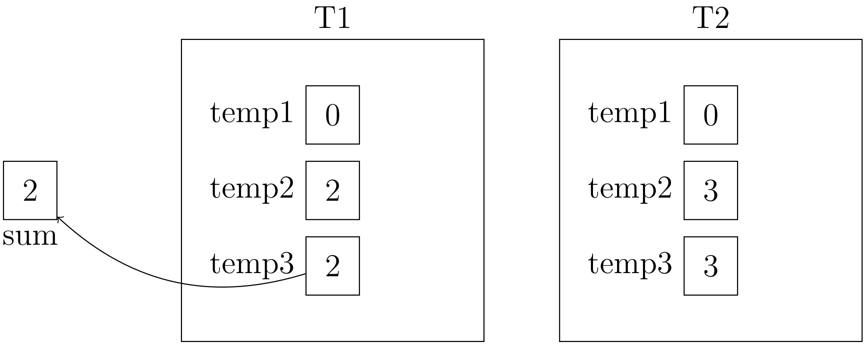

T1
T2
temp1
0
temp2
?
temp3
?
temp1
0
temp2
?
temp3
?
0
sum

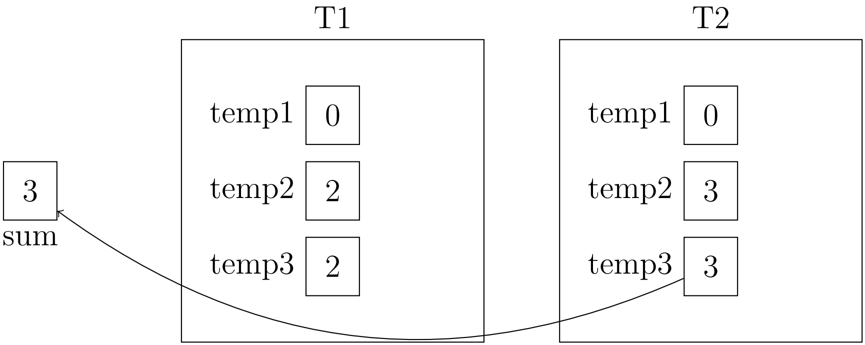

T1
T2
temp1
0
temp2
2
temp3
?
temp1
0
temp2
?
temp3
?
0
sum


T1
T2
temp1
0
temp2
2
temp3
?
temp1
0
temp2
3
temp3
?
0
sum


T1
T2
0
sum
temp1
0
temp2
2
temp3
2
temp1
0
temp2
3
temp3
?


T1
T2
temp1
0
temp2
2
temp3
2
temp1
0
temp2
3
temp3
3
0
sum


T1
T2
temp1
0
temp2
2
temp3
2
temp1
0
temp2
3
temp3
3
2
sum


T1
T2
temp1
0
temp2
2
temp3
2
temp1
0
temp2
3
temp3
3
3
sum

Parallelizing the vector sum problem

- Suppose you have t threads, and you know the answer to your t parallel subproblems (each thread's result for its section of the vector) is the set T.

Parallelizing the vector sum problem

- Suppose you have t threads, and you know the answer to your t parallel subproblems (each thread's result for its section of the vector) is the set T.
- How can the final answer be computed?

Parallelizing the vector sum problem

- Suppose you have t threads, and you know the answer to your t parallel subproblems (each thread's result for its section of the vector) is the set T.
- How can the final answer be computed?

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
2	5	2	1	3	2	1	1	3	4	3	5	2	5	1	4	2	4	3	1

13 + 11 + 16 + 14

Parallelizing the vector sum problem

- Suppose you have t threads, and you know the answer to your t parallel subproblems (each thread's result for its section of the vector) is the set T.
- How can the final answer be computed?

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
2	5	2	1	3	2	1	1	3	4	3	5	2	5	1	4	2	4	3	1

$$\underbrace{\hphantom{xxxxx}}_{13} + \underbrace{\hphantom{xxxxx}}_{11} + \underbrace{\hphantom{xxxxx}}_{16} + \underbrace{\hphantom{xxxxx}}_{14}$$

- $\sum_{\forall a \in T}(a)$ (that is, sum up the values in T)

Parallelizing the vector sum problem

- Suppose you have t threads, and you know the answer to your t parallel subproblems (each thread's result for its section of the vector) is the set T.
- How can the final answer be computed?

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
2	5	2	1	3	2	1	1	3	4	3	5	2	5	1	4	2	4	3	1

$$\underbrace{\qquad\qquad}_{13} + \underbrace{\qquad\qquad}_{11} + \underbrace{\qquad\qquad}_{16} + \underbrace{\qquad\qquad}_{14}$$

- $\sum_{\forall a \in T}(a)$ (that is, sum up the values in T)
- This step is an example of a *reduction*.

Reductions

Definition

- A *reduction* is an operation applied to an array to produce a result of a lesser rank.

Reductions

Definition

- A *reduction* is an operation applied to an array to produce a result of a lesser rank.
- In our example, we have an array of results, T from each thread's subproblem.

Reductions

Definition

- A *reduction* is an operation applied to an array to produce a result of a lesser rank.
- In our example, we have an array of results, T from each thread's subproblem.
- If x is the answer computed in a parallel subproblem, we run a "reduction on x" to get our final answer.

Reductions in OpenMP

- How do we write reductions in OpenMP?

Reductions in OpenMP

- How do we write reductions in OpenMP?
- In general, we have two options:

Reductions in OpenMP

- How do we write reductions in OpenMP?
- In general, we have two options:
  1. Use an OpenMP feature called the "reduction clause".

Reductions in OpenMP

- How do we write reductions in OpenMP?
- In general, we have two options:
  1. Use an OpenMP feature called the "reduction clause".
     - This is a useful shorthand, but it isn't always an option for more complicated reduction statements.

Reductions in OpenMP

- How do we write reductions in OpenMP?
- In general, we have two options:
  1. Use an OpenMP feature called the “reduction clause”.
     - This is a useful shorthand, but it isn’t always an option for more complicated reduction statements.
  2. Manually fix the race condition on the reduction variable.

Reductions in OpenMP

- How do we write reductions in OpenMP?
- In general, we have two options:
  1. Use an OpenMP feature called the "reduction clause".
     - This is a useful shorthand, but it isn't always an option for more complicated reduction statements.
  2. Manually fix the race condition on the reduction variable.
     - This is always an option, regardless of the complexity of the reduction statement.

Critical Regions and Atomicity

Definition

An **atomic** operation is one that happens entirely in one step.

Critical Regions and Atomicity

Definition

An **atomic** operation is one that happens entirely in one step.

Definition

A **critical region** is a section of code whose operation must be **atomic** for correctness. Critical regions that are not safeguarded are often the root causes of race conditions.

Critical Regions and Atomicity

Definition

An **atomic** operation is one that happens entirely in one step.

Definition

A **critical region** is a section of code whose operation must be **atomic** for correctness. Critical regions that are not safeguarded are often the root causes of race conditions.

- We guarantee critical regions are **atomic** by only allowing one thread to execute in the critical region at a time.

Critical Regions and Atomicity

Definition

An **atomic** operation is one that happens entirely in one step.

Definition

A **critical region** is a section of code whose operation must be **atomic** for correctness. Critical regions that are not safeguarded are often the root causes of race conditions.

- We guarantee critical regions are **atomic** by only allowing one thread to execute in the critical region at a time.
- How can we use the idea of a “critical region” to help manually fix the race condition on the `sum` variable?

Example: Manual Reduction With a Critical Region

```
int findSum(const vector<int> & list) {
  int sum = 0;
  #pragma omp parallel
  {
    int local_sum = 0;
    #pragma omp for
    for( int i = 0; i < list.size(); i++ )
      local_sum += list[i];
    #pragma omp critical
    sum += local_sum;
  }
  return sum;
}
```

Un-fuzzying the example

What are the pragmas here doing?

```
int findSum(const vector<int> & list) {
  int sum = 0;
  #pragma omp parallel
  {
    int local_sum = 0;
    #pragma omp for
    for( int i = 0; i < list.size(); i++ )
      local_sum += list[i];
    #pragma omp critical
    sum += local_sum;
  }
  return sum;
}
```

Un-fuzzying the example

- omp parallel

What are the pragmas here doing?

```
int findSum(const vector<int>& list) {
  int sum = 0;
  #pragma omp parallel
  {
    int local_sum = 0;
    #pragma omp for
    for( int i = 0; i < list.size(); i++ )
      local_sum += list[i];
    #pragma omp critical
    sum += local_sum;
  }
  return sum;
}
```

Un-fuzzying the example

What are the pragmas here doing?

```
int findSum(const vector<int>& list) {
  int sum = 0;
  #pragma omp parallel
  {
    int local_sum = 0;
    #pragma omp for
    for( int i = 0; i < list.size(); i++ )
      local_sum += list[i];
    #pragma omp critical
    sum += local_sum;
  }
  return sum;
}
```

- `omp parallel`
  - Creates a team of threads to operate on the given block (a block is between {}).

Un-fuzzying the example

What are the pragmas here doing?

```
int findSum(const vector<int>& list) {
  int sum = 0;
  #pragma omp parallel
  {
    int local_sum = 0;
    #pragma omp for
    for( int i = 0; i < list.size(); i++ )
      local_sum += list[i];
    #pragma omp critical
    sum += local_sum;
  }
  return sum;
}
```

- `omp parallel`
  - Creates a team of threads to operate on the given block (a block is between {}).
- `omp for`

Un-fuzzying the example

What are the pragmas here doing?

```
int findSum(const vector<int>& list) {
  int sum = 0;
  #pragma omp parallel
  {
    int local_sum = 0;
    #pragma omp for
    for( int i = 0; i < list.size(); i++ )
      local_sum += list[i];
    #pragma omp critical
    sum += local_sum;
  }
  return sum;
}
```

- `omp parallel`
  - Creates a team of threads to operate on the given block (a block is between {}).
- `omp for`
  - Parallelizes a for loop, using the team of threads that exist at this point.

Un-fuzzying the example

What are the pragmas here doing?

```
int findSum(const vector<int> & list) {
  int sum = 0;
  #pragma omp parallel
  {
    int local_sum = 0;
    #pragma omp for
    for( int i = 0; i < list.size(); i++ )
      local_sum += list[i];
    #pragma omp critical
    sum += local_sum;
  }
  return sum;
}
```

- `omp parallel`
  - Creates a team of threads to operate on the given block (a block is between {}).
- `omp for`
  - Parallelizes a for loop, using the team of threads that exist at this point.
  - Note that we omit the `parallel` directive here because we already have spawned a team of threads.

Un-fuzzying the example

What are the pragmas here doing?

```
int findSum(const vector<int>& list) {
  int sum = 0;
  #pragma omp parallel
  {
    int local_sum = 0;
    #pragma omp for
    for( int i = 0; i < list.size(); i++ )
      local_sum += list[i];
    #pragma omp critical
    sum += local_sum;
  }
  return sum;
}
```

- `omp parallel`
  - Creates a team of threads to operate on the given block (a block is between {}).
- `omp for`
  - Parallelizes a for loop, using the team of threads that exist at this point.
  - Note that we omit the `parallel` directive here because we already have spawned a team of threads.
- `omp critical`

Un-fuzzying the example

What are the pragmas here doing?

```
int findSum(const vector<int>& list) {
  int sum = 0;
  #pragma omp parallel
  {
    int local_sum = 0;
    #pragma omp for
    for( int i = 0; i < list.size(); i++ )
      local_sum += list[i];
    #pragma omp critical
    sum += local_sum;
  }
  return sum;
}
```

- `omp parallel`
  - Creates a team of threads to operate on the given block (a block is between {}).
- `omp for`
  - Parallelizes a for loop, using the team of threads that exist at this point.
  - Note that we omit the `parallel` directive here because we already have spawned a team of threads.
- `omp critical`
  - Designates the given block of code as a “critical region”.

Un-fuzzying the example

What are the pragmas here doing?

```
int findSum(const vector<int>& list) {
  int sum = 0;
  #pragma omp parallel
  {
    int local_sum = 0;
    #pragma omp for
    for( int i = 0; i < list.size(); i++ )
      local_sum += list[i];
    #pragma omp critical
    sum += local_sum;
  }
  return sum;
}
```

- `omp parallel`
  - Creates a team of threads to operate on the given block (a block is between {}).
- `omp for`
  - Parallelizes a for loop, using the team of threads that exist at this point.
  - Note that we omit the `parallel` directive here because we already have spawned a team of threads.
- `omp critical`
  - Designates the given block of code as a "critical region".
  - Only one thread in the team is allowed to be executing in the critical region at a time.

Un-fuzzying the example

What are the pragmas here doing?

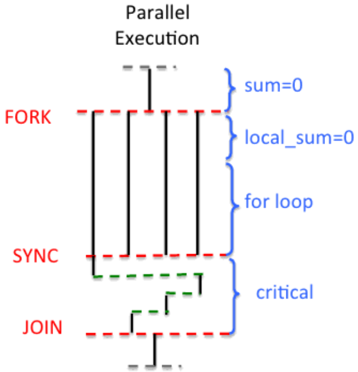
```
int findSum(const vector<int>& list) {
  int sum = 0;
  #pragma omp parallel
  {
    int local_sum = 0;
    #pragma omp for
    for( int i = 0; i < list.size(); i++ )
      local_sum += list[i];
    #pragma omp critical
    sum += local_sum;
  }
  return sum;
}
```

- `omp parallel`
  - Creates a team of threads to operate on the given block (a block is between {}).
- `omp for`
  - Parallelizes a for loop, using the team of threads that exist at this point.
  - Note that we omit the `parallel` directive here because we already have spawned a team of threads.
- `omp critical`
  - Designates the given block of code as a "critical region".
  - Only one thread in the team is allowed to be executing in the critical region at a time.
  - This fixes our race condition!

Execution Diagram

What does this look like when running?

```
int findSum(const vector<int>& list) {
  int sum = 0;
  #pragma omp parallel
  {
    int local_sum = 0;
    #pragma omp for
    for( int i = 0; i < list.size(); i++ )
      local_sum += list[i];
    #pragma omp critical
    sum += local_sum;
  }
  return sum;
}
```



The Reduction Clause

In this example, we can apply the “reduction clause” to simplify our code.

The Reduction Clause

In this example, we can apply the “reduction clause” to simplify our code.

- The “reduction clause” comes after a for construct and takes the form `reduction(operator: variables)`.

The Reduction Clause

In this example, we can apply the "reduction clause" to simplify our code.

- The "reduction clause" comes after a for construct and takes the form `reduction(operator: variables)`.
- The `operator` can only be of the following: $+$, $*$, $-$, $/$, &, ^, $|$, &&, $||$.

The Reduction Clause

In this example, we can apply the "reduction clause" to simplify our code.

- The "reduction clause" comes after a for construct and takes the form `reduction(operator: variables)`.
- The `operator` can only be of the following: $+$, $*$, $-$, $/$, &, ^, |, &&, ||.
- It *cannot be overloaded*.

The Reduction Clause

In this example, we can apply the "reduction clause" to simplify our code.

- The "reduction clause" comes after a for construct and takes the form `reduction(operator: variables)`.
- The `operator` can only be of the following: $+$, $*$, $-$, $/$, &, ^, $|$, &&, $||$.
- It *cannot be overloaded*.
- The elements provided as `variables` must be scalar (primitive) types.

Example: Using the Reduction Clause

```
int findSum( const vector<int> & list ) {
    int sum = 0;
    #pragma omp parallel for reduction(+: sum)
    for( int i = 0; i < list.size(); i++ )
        sum += list[i];
    return sum;
}
```

Example: Using the Reduction Clause

```
int findSum( const vector<int> & list ) {
    int sum = 0;
    #pragma omp parallel for reduction(+: sum)
    for( int i = 0; i < list.size(); i++ )
        sum += list[i];
    return sum;
}
```

This is shorthand for resolving the race condition manually.

Another Example: Largest Integer Problem

- Consider the code below for finding the maximum integer in an array.

```
int findMax( const vector<int> & list ) {
    int max = INT_MIN;
    for( int i = 0; i < list.size(); i++ )
        if( max < list[i] )
            max = list[i];
    return max;
}
```

Another Example: Largest Integer Problem

- Consider the code below for finding the maximum integer in an array.

```
int findMax( const vector<int> & list ) {
    int max = INT_MIN;
    for( int i = 0; i < list.size(); i++ )
        if( max < list[i] )
            max = list[i];
    return max;
}
```

- Can this be parallelized?

Another Example: Largest Integer Problem

- Consider the code below for finding the maximum integer in an array.

```
int findMax( const vector<int> & list ) {
    int max = INT_MIN;
    for( int i = 0; i < list.size(); i++ )
        if( max < list[i] )
            max = list[i];
    return max;
}
```

- Can this be parallelized?
- What reduction method will you use?

Another Example: Largest Integer Problem

- Consider the code below for finding the maximum integer in an array.

```
int findMax( const vector<int> & list ) {
    int max = INT_MIN;
    for( int i = 0; i < list.size(); i++ )
        if( max < list[i] )
            max = list[i];
    return max;
}
```

- Can this be parallelized?
- What reduction method will you use?
- Why aren't we able to use a reduction clause here?

Another Example: Largest Integer Problem

- Consider the code below for finding the maximum integer in an array.

```
int findMax( const vector<int> & list ) {
    int max = INT_MIN;
    for( int i = 0; i < list.size(); i++ )
        if( max < list[i] )
            max = list[i];
    return max;
}
```

- Can this be parallelized?
- What reduction method will you use?
- Why aren't we able to use a reduction clause here?
- Full disclosure: This will eventually be possible with a reduction clause in OpenMP 3.1 (supported in GCC 4.7).

Parallelizing the Maximum Integer Problem

Filling in the code:

```
int findMax( const vector<int> & list ) {
    int max = INT_MIN;

    {
        int local_max = max;

        for( int i = 0; i < list.size(); i++ )
            if( local_max < list[i] )
                local_max = list[i];

        {


        }
    }
    return max;
}
```

Parallelizing the Maximum Integer Problem

Filling in the code:

```
int findMax( const vector<int> & list ) {
    int max = INT_MIN;
    #pragma omp parallel
    {
        int local_max = max;

        for( int i = 0; i < list.size(); i++ )
            if( local_max < list[i] )
                local_max = list[i];

        {


        }
    }
    return max;
}
```

Parallelizing the Maximum Integer Problem

Filling in the code:

```
int findMax( const vector<int> & list ) {
    int max = INT_MIN;
    #pragma omp parallel
    {
        int local_max = max;
        #pragma omp for
        for( int i = 0; i < list.size(); i++ )
            if( local_max < list[i] )
                local_max = list[i];

        {


        }
    }
    return max;
}
```

Parallelizing the Maximum Integer Problem

Filling in the code:

```
int findMax( const vector<int> & list ) {
    int max = INT_MIN;
    #pragma omp parallel
    {
        int local_max = max;
        #pragma omp for
        for( int i = 0; i < list.size(); i++ )
            if( local_max < list[i] )
                local_max = list[i];
        #pragma omp critical
        {


        }
    }
    return max;
}
```

Parallelizing the Maximum Integer Problem

Filling in the code:

```
int findMax( const vector<int> & list ) {
    int max = INT_MIN;
    #pragma omp parallel
    {
        int local_max = max;
        #pragma omp for
        for( int i = 0; i < list.size(); i++ )
            if( local_max < list[i] )
                local_max = list[i];
        #pragma omp critical
        {
            if( max < local_max )
                max = local_max;
        }
    }
    return max;
}
```

Aside: When to use the Reduction Clause

- In general, the reduction clause can only be used if our reduction is a simplistic one.

Aside: When to use the Reduction Clause

- In general, the reduction clause can only be used if our reduction is a simplistic one.
- For more complicated reductions (such as ones involving 2D arrays or more interesting data structures), we will have to write a manual reduction.

Map Reduce

- What we have seen so far is a generalization of the Map Reduce pattern.

Map Reduce

- What we have seen so far is a generalization of the Map Reduce pattern.
- **Map Reduce** was introduced by Google in 2004 as a way of processing large amounts of data in parallel.

Map Reduce

- What we have seen so far is a generalization of the Map Reduce pattern.
- **Map Reduce** was introduced by Google in 2004 as a way of processing large amounts of data in parallel.
- Uses a cluster of machines to do the processing.

Map Reduce

- What we have seen so far is a generalization of the Map Reduce pattern.
- **Map Reduce** was introduced by Google in 2004 as a way of processing large amounts of data in parallel.
- Uses a cluster of machines to do the processing.
- Broken into two steps:

Map Reduce

- What we have seen so far is a generalization of the Map Reduce pattern.
- **Map Reduce** was introduced by Google in 2004 as a way of processing large amounts of data in parallel.
- Uses a cluster of machines to do the processing.
- Broken into two steps:
  - Map: Partition the input into small pieces, and run a function (`map()`) on each piece. The result is a list of (key, value) pairs.

Map Reduce

- What we have seen so far is a generalization of the Map Reduce pattern.
- **Map Reduce** was introduced by Google in 2004 as a way of processing large amounts of data in parallel.
- Uses a cluster of machines to do the processing.
- Broken into two steps:
  - Map: Partition the input into small pieces, and run a function (`map()`) on each piece. The result is a list of (key, value) pairs.
  - Reduce: Combines many lists produced from each Map step into a smaller set of (key, value) pairs.

Map Reduce

- What we have seen so far is a generalization of the Map Reduce pattern.
- **Map Reduce** was introduced by Google in 2004 as a way of processing large amounts of data in parallel.
- Uses a cluster of machines to do the processing.
- Broken into two steps:
  - Map: Partition the input into small pieces, and run a function (`map()`) on each piece. The result is a list of (key, value) pairs.
  - Reduce: Combines many lists produced from each Map step into a smaller set of (key, value) pairs.
- In our examples, our `parallel for` directive is our mapping function, and our reduction setup is the reduce step.

Summary

So what have we seen?

- **Race conditions** are a big concern when writing parallel code.

Summary

So what have we seen?

- **Race conditions** are a big concern when writing parallel code.
- **Reductions** can help us assemble a main result from a list of results of parallel subproblems.

Summary

So what have we seen?

- **Race conditions** are a big concern when writing parallel code.
- **Reductions** can help us assemble a main result from a list of results of parallel subproblems.
- There is a subset of Race conditions that can be resolved by using **reductions**.

Summary

So what have we seen?

- **Race conditions** are a big concern when writing parallel code.
- **Reductions** can help us assemble a main result from a list of results of parallel subproblems.
- There is a subset of Race conditions that can be resolved by using **reductions**.
- The general paradigm in use here is the same as that used in **Mapreduce**!