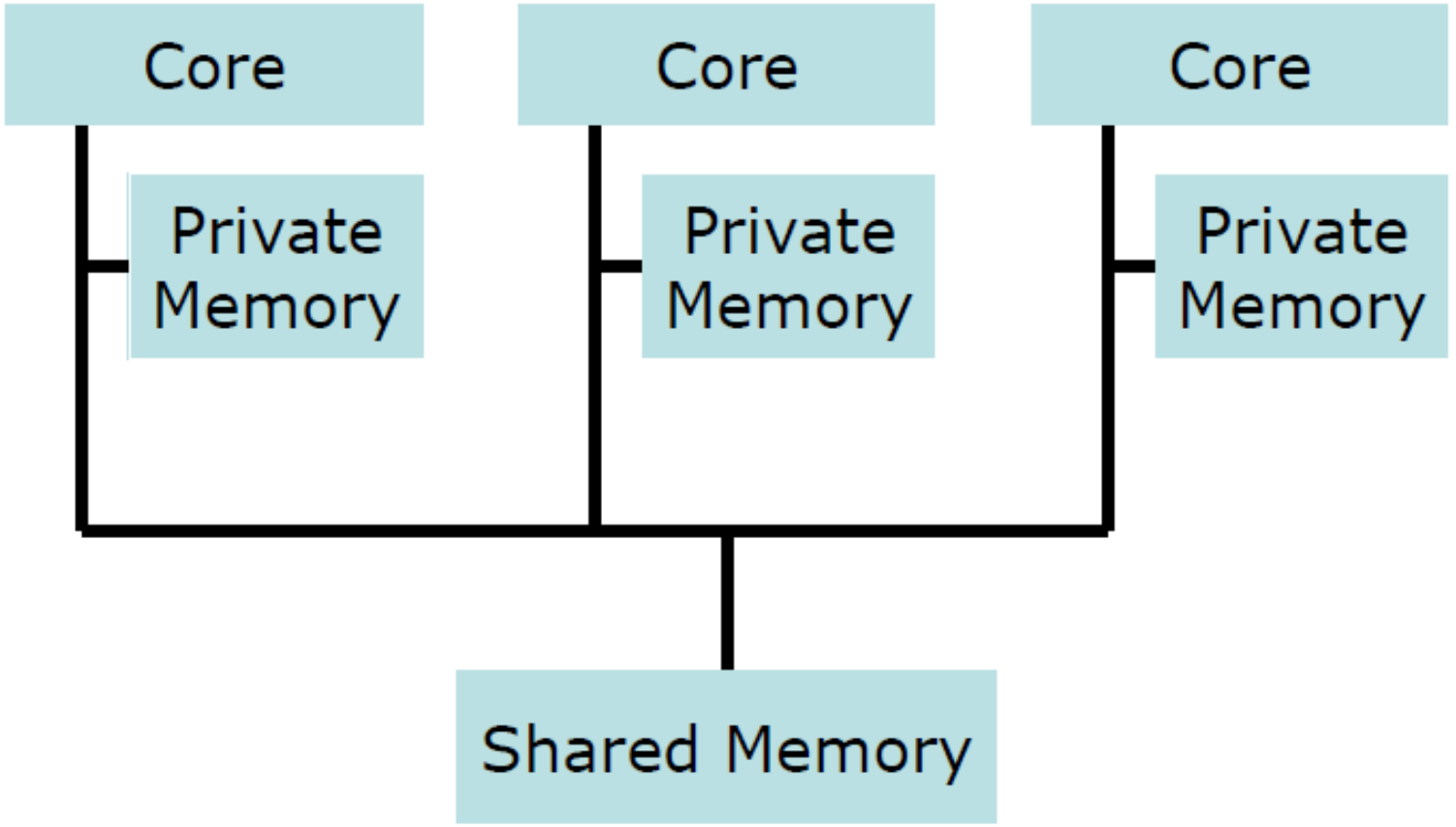
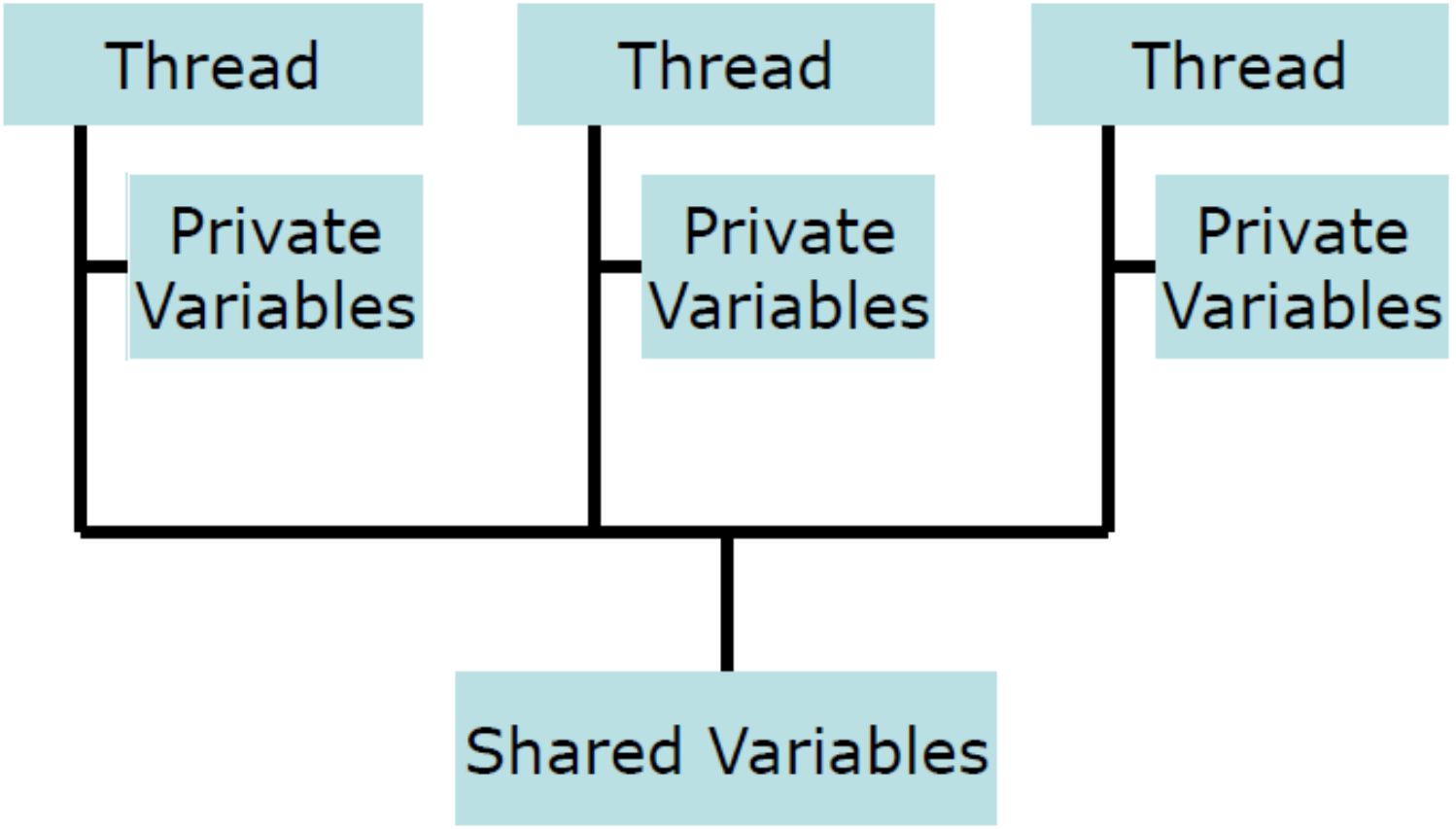
Shared Memory and OpenMP

Selected Slides from Intel's
INTEL® DEVELOPER ZONE:
Courseware - Parallel Programming Basics
http://software.intel.com/en-us/courseware/249633

The Shared-Memory Model



The Threads Model



What Is a Thread?

"A unit of control within a process" — Carver and Tai

Main thread executes program's "main" function

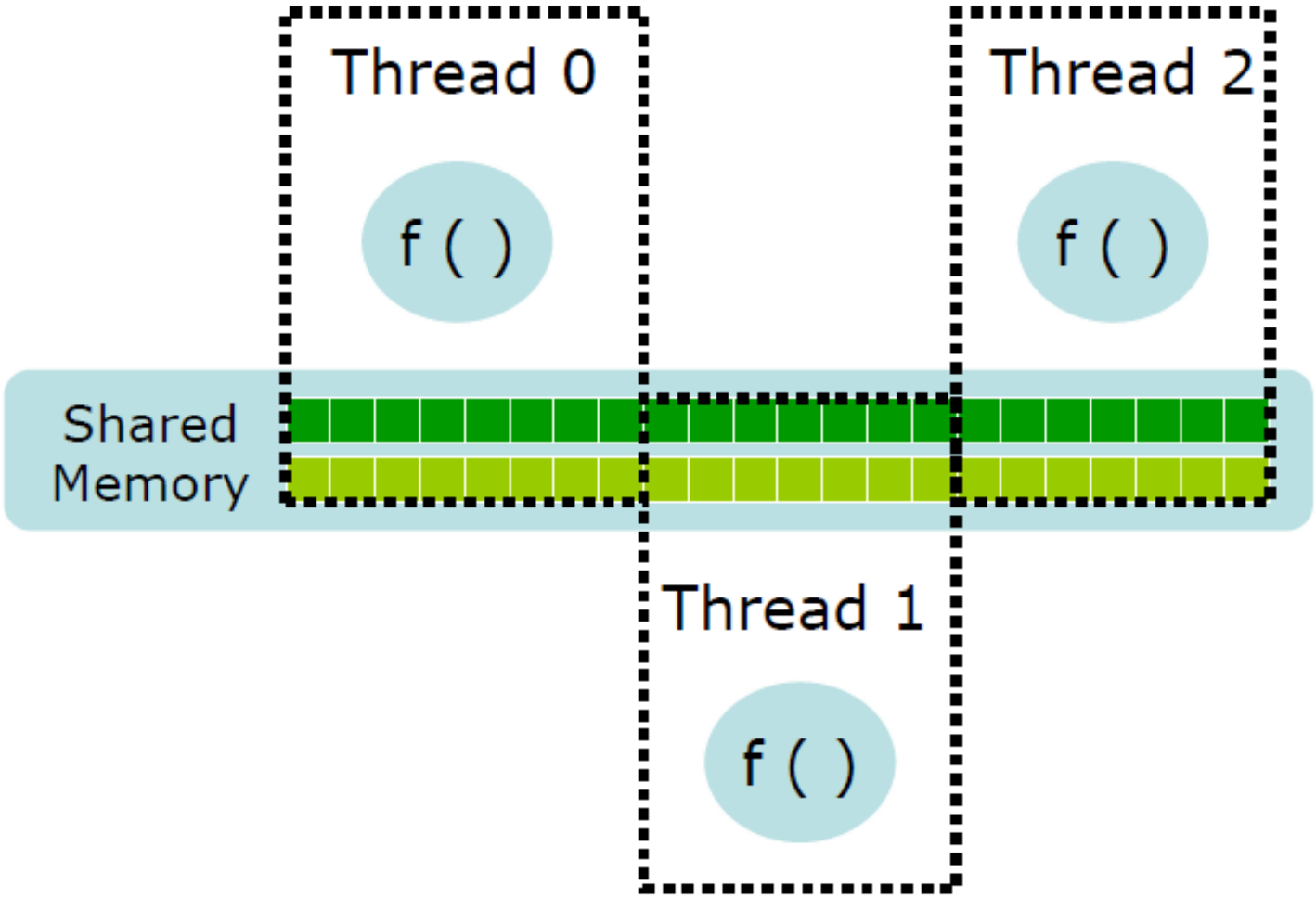
Main thread may create other threads to execute other functions

Threads have own program counter, copy of core registers, and stack of activation records

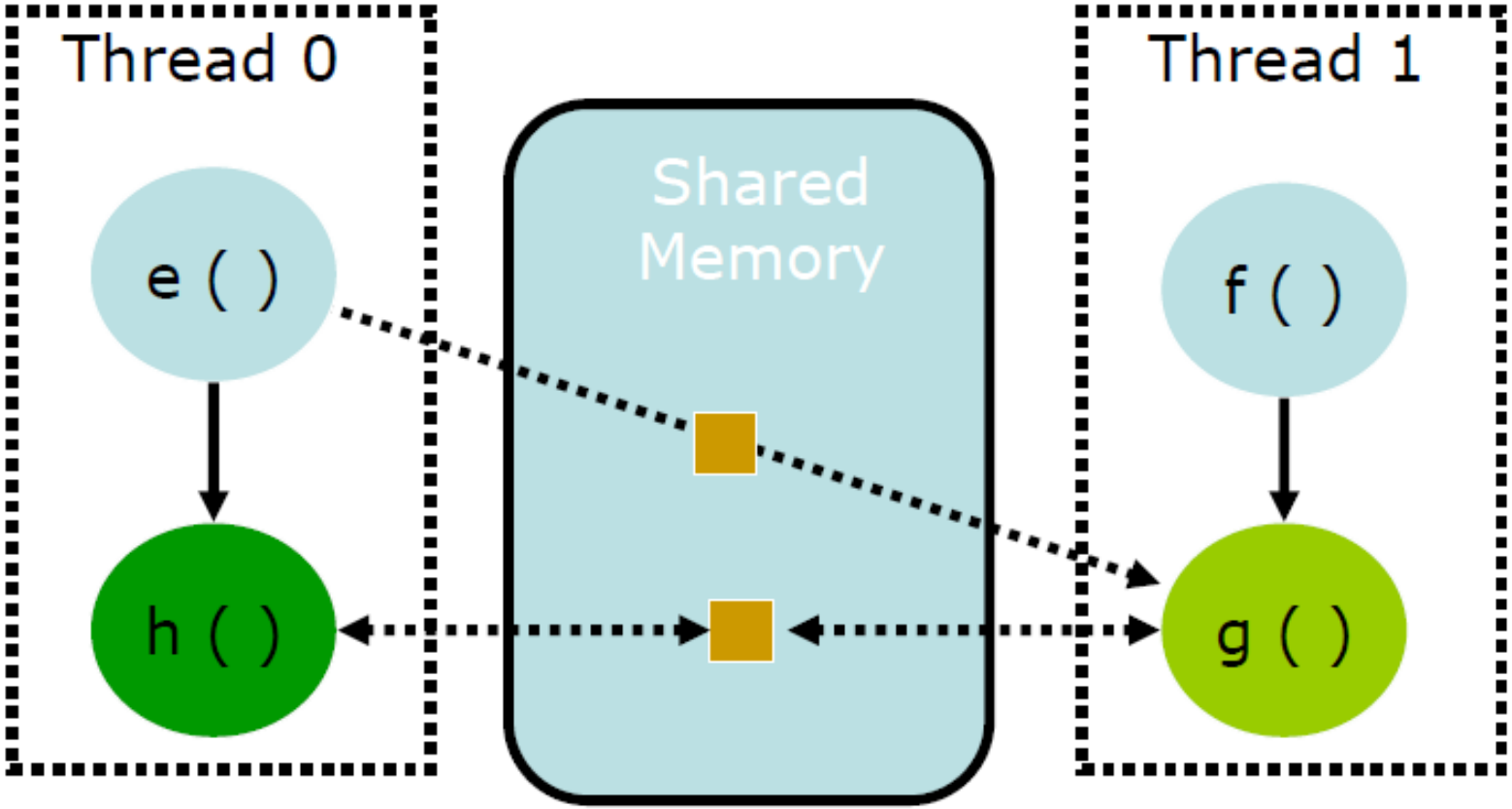
Threads share process's data, code, address space, and other resources

Threads have lower overhead than processes

Domain Decomposition Using Threads



Task Decomposition Using Threads



What are tasks?

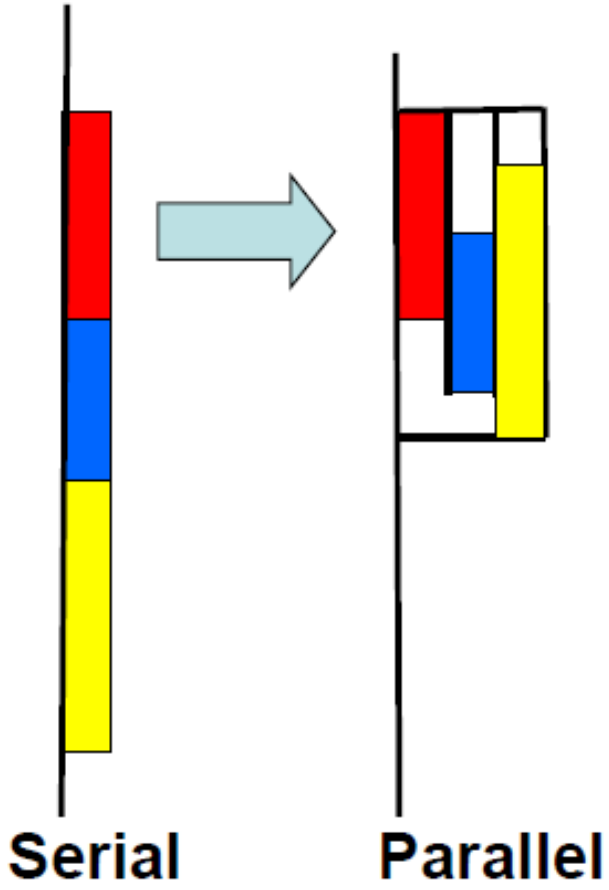
Tasks are independent units of work

- Threads are assigned to perform the work of each task
- Tasks may be deferred
- Tasks may be executed immediately

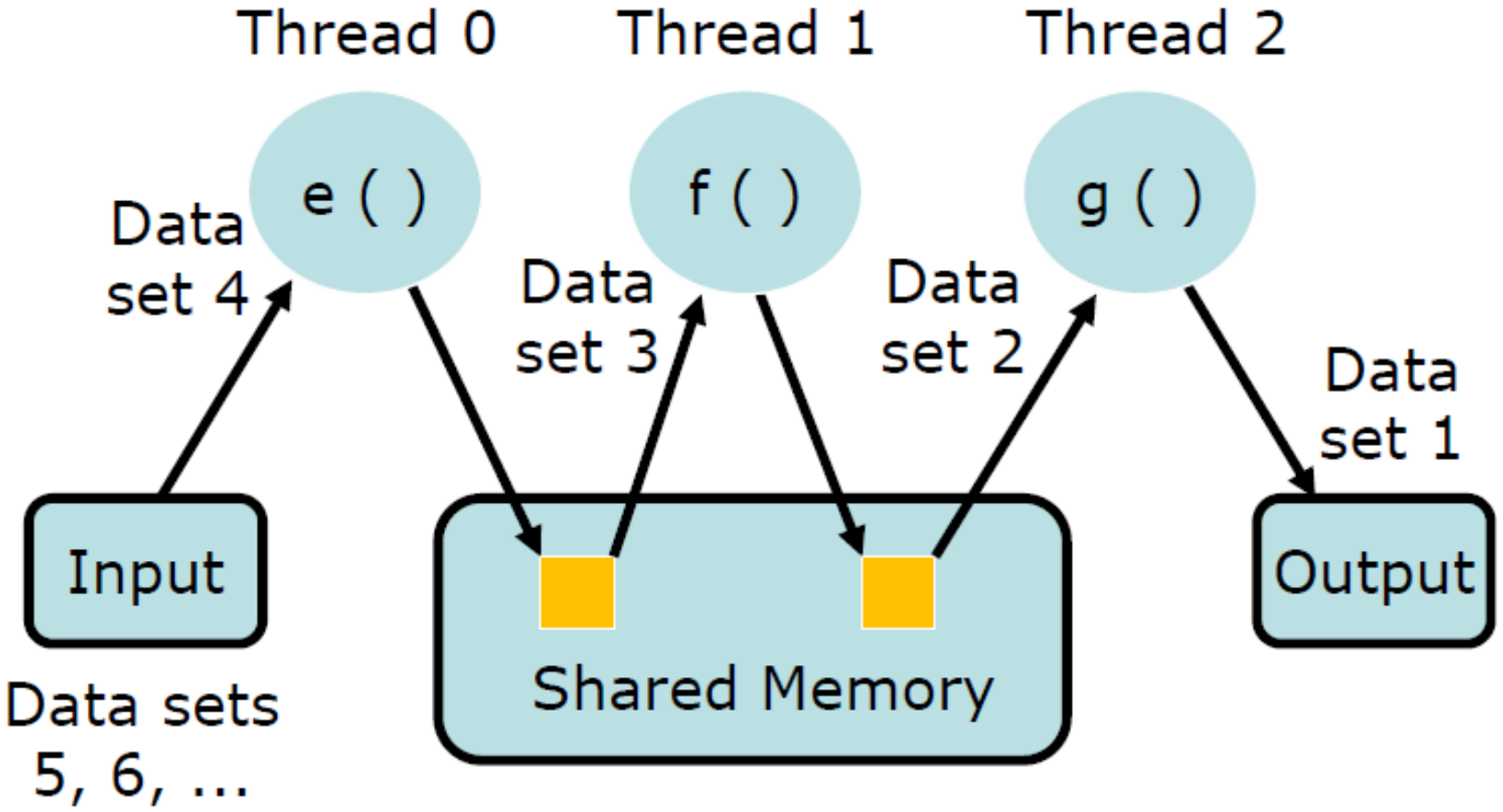
The runtime system decides which of the above

Tasks are composed of:

- **code** to execute
- **data** environment
- **internal control variables** (ICV)



Pipelining Using Threads



Domain Decomposition

Sequential Code:

```
int a[1000], i;
for (i = 0; i < 1000; i++) a[i] = foo(i);
```

Thread 0:

```
for (i = 0; i < 500; i++) a[i] = foo(i);
```

Thread 1:

```
for (i = 500; i < 1000; i++) a[i] = foo(i);
```

Domain Decomposition

Sequential Code:

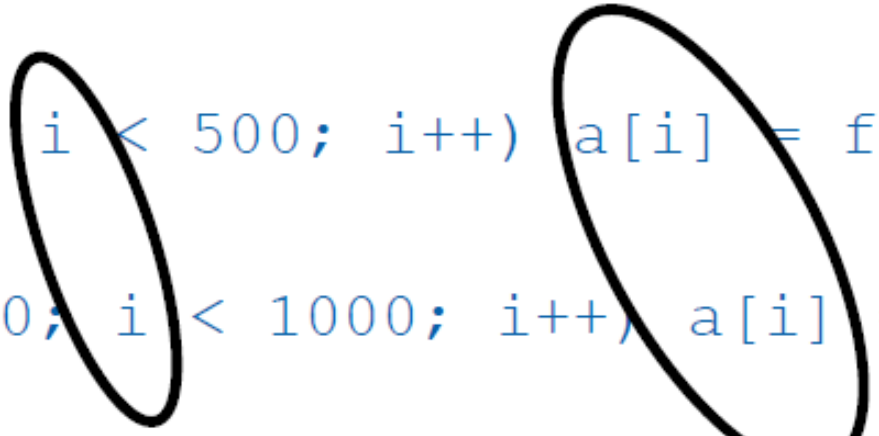
```
int a[1000], i;
for (i = 0; i < 1000; i++) a[i] = foo(i);
```

Thread 0:

```
for (i = 0; i < 500; i++) a[i] = foo(i);
```

Thread 1:

```
for (i = 500; i < 1000; i++) a[i] = foo(i);
```



Task Decomposition

```
int e;

main () {
   int x[10], j, k, m;   j = f(x, k); m = g(x, k); ...
}

int f(int *x, int k)
{
   int a;   a = e * x[k] * x[k];   return a;
}

int g(int *x, int k)
{
   int a;   k = k-1;  a = e / x[k];   return a;
}
```

Task Decomposition

```
int e;

main () {
    int x[10], j, k, m;    j = f(x, k);   m = g(x, k);
}
```

Thread 0

```
int f(int *x, int k)
{
    int a;    a = e * x[k] * x[k];    return a;
}
```

Thread 1

```
int g(int *x, int k)
{
    int a;    k = k-1;   a = e / x[k];    return a;
}
```

Task Decomposition

```
Volatile int e;
```
Static variable: Shared

```
main () {
   int x[10], j, k, m;   j = f(x, k);   m = g(x, k);
}
```

Thread 0

```
int f(int *x, int k)
{
   int a;   a = e * x[k] * x[k];   return a;
}
```

Thread 1

```
int g(int *x, int k)
{
   int a;   k = k-1;   a = e / x[k];   return a;
}
```

Task Decomposition

```
Volatile int e;

main () {
    int x[10], j, k, m;    j = f(x, k);   m = g(x, k);
}
```

Heap variable: Shared

Thread 0

```
int f(int *x, int k)
{
    int a;    a = e * x[k] * x[k];    return a;
}
```

Thread 1

```
int g(int *x, int k)
{
    int a;    k = k-1;   a = e / x[k];    return a;
}
```

Task Decomposition

```
Volatile int e;

main () {
    int x[10], j, k, m;    j = f(x, k);   m = g(x, k);
}
```

Function's local variables: Private

Thread 0

```
int f(int *x, int k)
{
   int a;    a = e * x[k] * x[k];    return a;
}
```

Thread 1

```
int g(int *x, int k)
{
   int a;    k = k-1;   a = e / x[k];    return a;
}
```

OpenMP

Fork/Join Programming Model

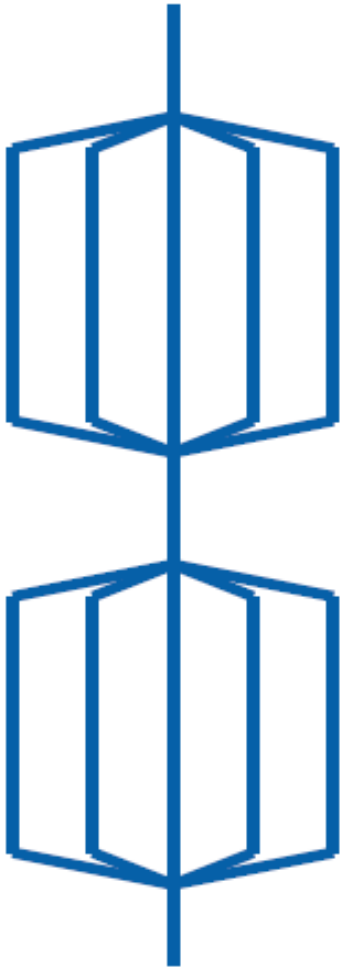
When program begins execution, only master thread active

Master thread executes sequential portions of program

For parallel portions of program, master thread forks (creates or awakens) additional threads

At join (end of parallel section of code), extra threads are suspended or die

Relating Fork/Join to Code



Sequential code

Parallel code

Sequential code

Parallel code

Sequential code

Pragma: parallel for

The compiler directive

```
#pragma omp parallel for
```

tells the compiler that the `for` loop which immediately follows can be executed in parallel

The number of loop iterations must be computable at run time before loop executes

Loop must not contain a `break`, `return`, or `exit`

Loop must not contain a `goto` to a label outside loop

Example

```
int first, *marked, prime, size;
...
#pragma omp parallel for
for (i = first; i < size; i += prime)
  marked[i] = 1;
```

- Threads are assigned an independent set of iterations
- Threads must wait at the end of construct

Pragma: parallel

Sometimes the code that should be executed in parallel goes beyond a single `for` loop

The `parallel` pragma is used when a block of code should be executed in parallel

```
#pragma omp parallel
{
  DoSomeWork(res, M);
  DoSomeOtherWork(res, M);
}
```

Pragma: for

- The `for` pragma can be used inside a block of code already marked with the `parallel` pragma
- Loop iterations should be divided among the active threads
- There is a *barrier synchronization* at the end of the `for` loop

```
#pragma omp parallel
{
  DoSomeWork(res, M);
 #pragma omp for
  for (i = 0; i < M; i++){
     res[i] = huge();
   }
  DoSomeMoreWork(res, M);
}
```

Which Loop to Make Parallel?

```
main () {
int i, j, k;
float **a, **b;
...
for (k = 0; k < N; k++)
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      a[i][j] = MIN(a[i][j], a[i][k] + a[k][j]);
```

Loop-carried dependences

Can execute in parallel

Can execute in parallel

Minimizing Threading Overhead

There is a fork/join for every instance of

```
#pragma omp parallel for
for ( ) {
    ...
}
```

Since fork/join is a source of overhead, we want to maximize the amount of work done for each fork/join

Hence we choose to make the middle loop parallel

Almost Right, but Not Quite

```
main () {
int i, j, k;
float **a, **b;
...
for (k = 0; k < N; k++)
  #pragma omp parallel for
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      a[i][j] = MIN(a[i][j], a[i][k] + a[k][j]);
```

Problem: j is a shared variable

Problem Solved with private Clause

```
main () {
int i, j, k;
float **a, **b;
...
for (k = 0; k < N; k++)
  #pragma omp parallel for private (j)
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      a[i][j] = MIN(a[i][j], a[i][k] + a[k][j]);
```

Tells compiler to make listed variables private

OpenMP `reduction` Clause

Reductions are so common that OpenMP provides a `reduction` clause for the `parallel for` pragma

```
reduction (op : list)
```

A PRIVATE copy of each list variable is created and initialized depending on the "op"

- The identity value "op" (e.g., 0 for addition)

These copies are updated locally by threads

At end of construct, local copies are combined through "op" into a single value and combined with the value in the original SHARED variable

Reduction Example

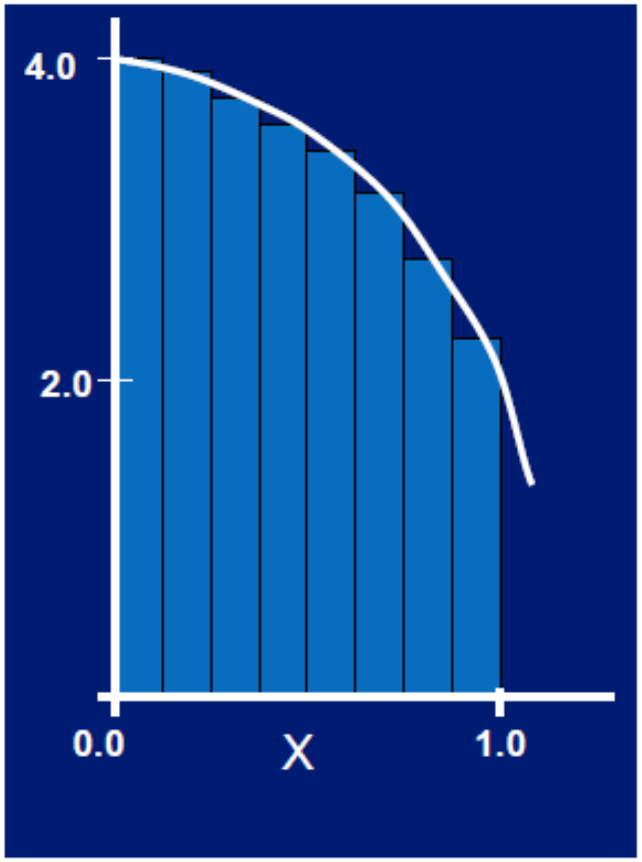
```
#pragma omp parallel for reduction(+:sum)
    for(i = 0; i < N; i++) {
      sum += a[i] * b[i];
    }
```

Local copy of `sum` for each thread

All local copies of sum added together and stored in shared copy

Numerical Integration Example

$$\int_0^1 \frac{4.0}{(1+x^2)} dx = \pi$$




```
static long num_rects=100000;
double width, pi;

void main()
{   int i;
    double x, sum = 0.0;

    width = 1.0/(double) num_rects;
    for (i = 0; i < num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0 + x*x);
    }
    pi = width * sum;
    printf("Pi = %f\n",pi);
}
```

Numerical Integration: What's Shared?

```
static long num_rects=100000;
double width, pi;

void main()
{   int i;
    double x, sum = 0.0;

    width = 1.0/(double) num_rects;
    for (i = 0; i < num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0 + x*x);
    }
    pi = step * sum;
    printf("Pi = %f\n",pi);
}
```

What variables can be shared?

width, num_rects

Numerical Integration: What's Private?

```
static long num_rects=100000;
double width, pi;

void main()
{   int i;
    double x, sum = 0.0;

    width = 1.0/(double) num_rects;
    for (i = 0; i < num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0 + x*x);
    }
    pi = step * sum;
    printf("Pi = %f\n",pi);
}
```

What variables need to be private?

x, i

Numerical Integration: Any Reductions?

```
static long num_rects=100000;
double width, pi;

void main()
{   int i;
    double x, sum = 0.0;

    width = 1.0/(double) num_rects;
    for (i = 0; i < num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0 + x*x);
    }
    pi = step * sum;
    printf("Pi = %f\n",pi);
}
```

What variables should be set up for reduction?

sum

Solution to Computing Pi

```
static long num_rects=100000;
double width, pi;

void main()
{   int i;
    double x, sum = 0.0;
#pragma omp parallel for private(x) reduction(+:sum)
    width = 1.0/(double) num_rects;
    for (i = 0; i < num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0 + x*x);
    }
    pi = step * sum;
    printf("Pi = %f\n",pi);
}
```

Locks Are Dangerous

Suppose a lock is used to guarantee mutually exclusive access to a shared variable

Imagine two threads, each with its own critical region

Thread 1

```
a += 5;
b += 7;
a += b;
a += 11;
```

Thread 2

```
b += 5;
a += 7;
a += b;
b += 11;
```

Faulty Implementation

Thread 1

```
lock (lock_a);
a += 5;
lock (lock_b);
b += 7;
a += b;
unlock (lock_b);
a += 11;
unlock (lock_a);
```

Thread 2

```
lock (lock_b);
b += 5;
lock (lock_a);
a += 7;
a += b;
unlock (lock_a);
b += 11;
unlock (lock_b);
```

Faulty Implementation

What happens if threads are at this point at the same time?

Thread 1

```
lock (lock_a);
a += 5;
lock (lock_b);
b += 7;
a += b;
unlock (lock_b);
a += 11;
unlock (lock_a);
```

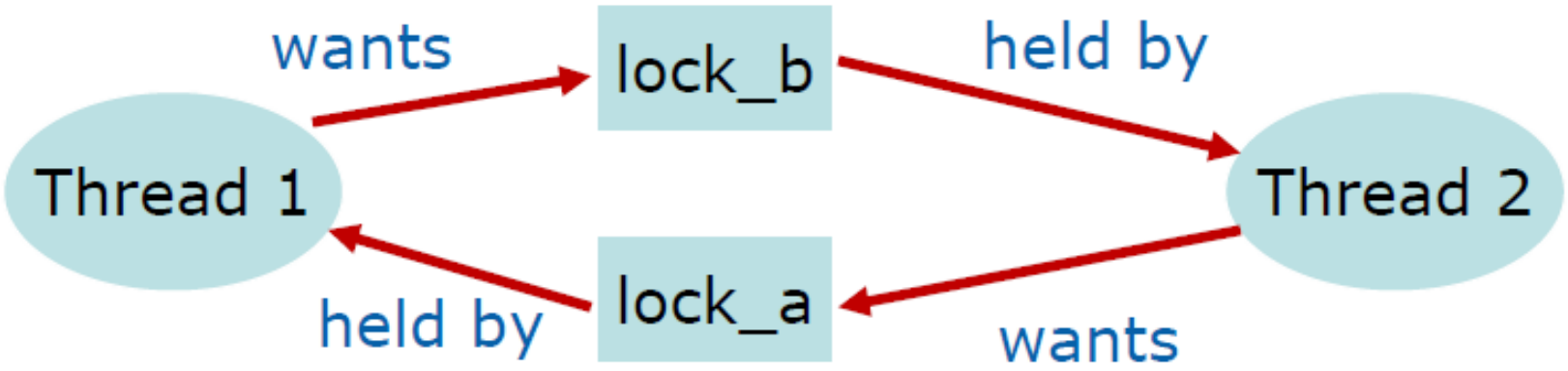
Thread 2

```
lock (lock_b);
b += 5;
lock (lock_a);
a += 7;
a += b;
unlock (lock_a);
b += 11;
unlock (lock_b);
```

Deadlock

A situation involving two or more threads (processes) in which no thread may proceed because each is waiting for a resource held by another

Can be represented by a resource allocation graph



A graph of deadlock contains a cycle

Correct Implementation

Thread 1

```
lock (lock_a);
a += 5;
lock (lock_b);
b += 7;
a += b;
unlock (lock_b);
a += 11;
unlock (lock_a);
```

Thread 2

```
lock (lock_a);
lock (lock_b);
b += 5;
a += 7;
a += b;
unlock (lock_a);
b += 11;
unlock (lock_b);
```

Threads must lock
lock_a before lock_b

Pragma: single

Denotes block of code to be executed by only one thread

- First thread to arrive is chosen

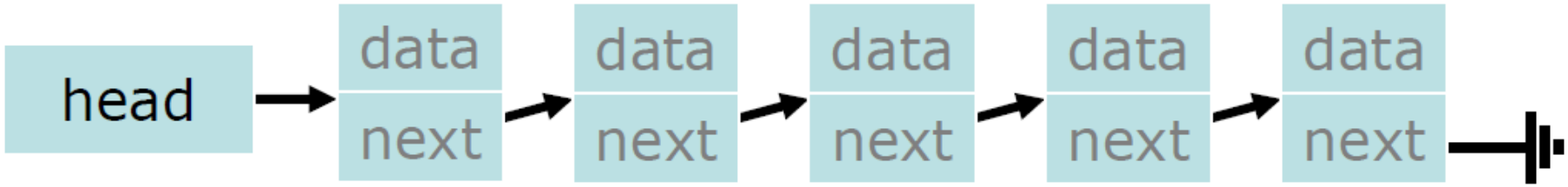
Implicit barrier at end

```
#pragma omp parallel
{
   DoManyThings();
#pragma omp single
   {
     printf("Many Things done\n");
   }  // threads wait here for single
   DoManyMoreThings();
}
```

A Linked List Example

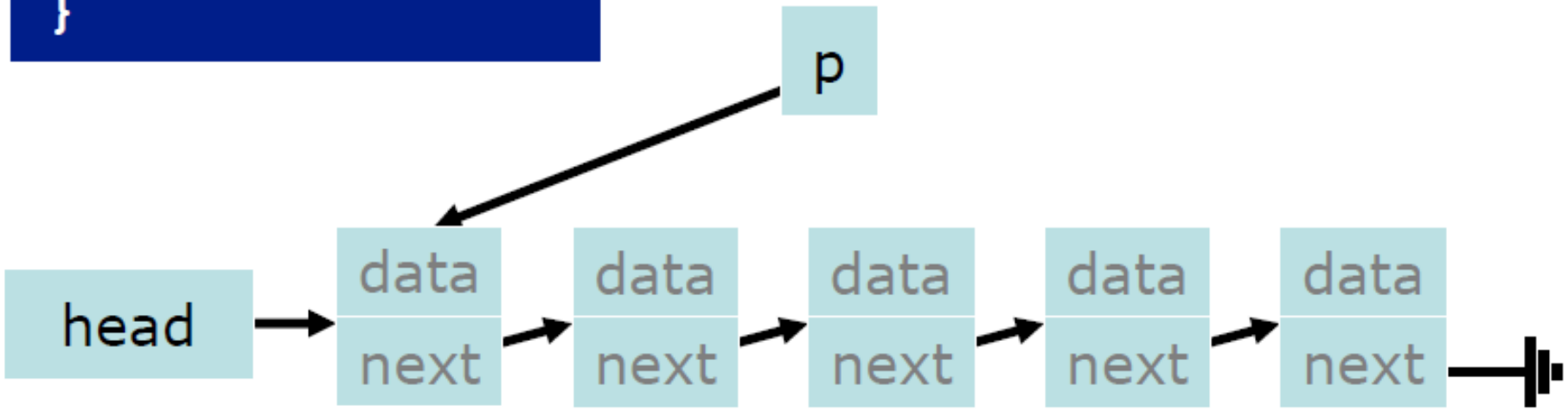
```
node *p = head;
while (p) {
  process(p);
  p = p->next;
}
```

Serial version



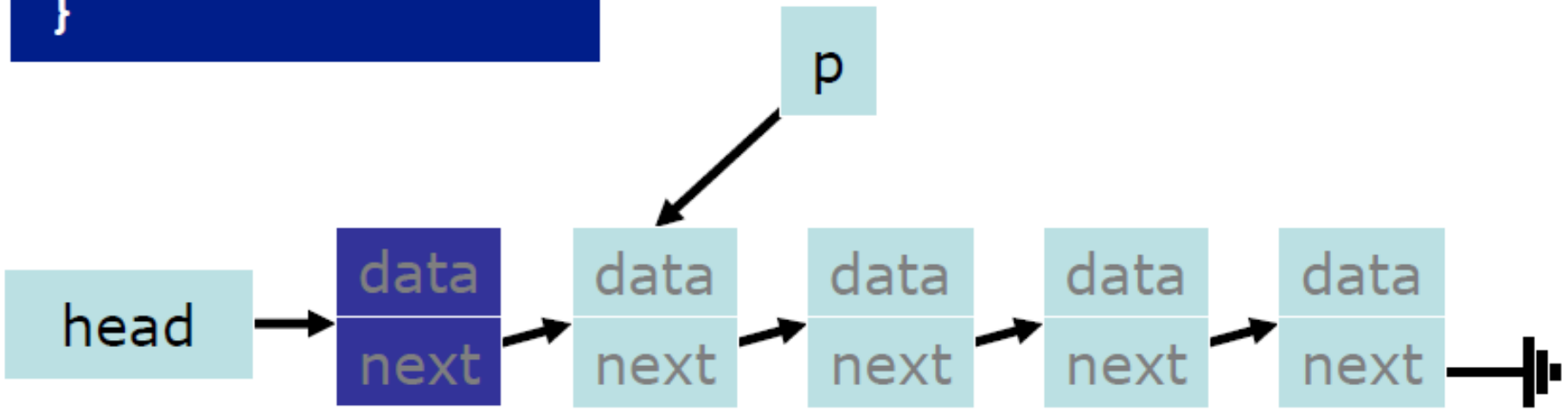
A Linked List Example

```
node *p = head;
while (p) {
   process(p);
   p = p->next;
}
```



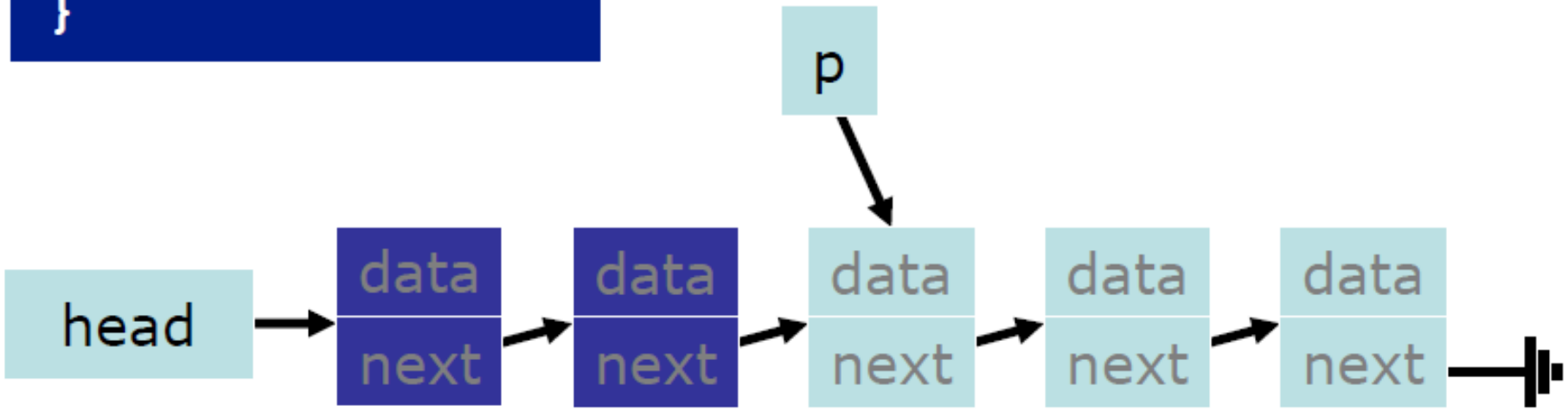
A Linked List Example

```
node *p = head;
while (p) {
   process(p);
   p = p->next;
}
```



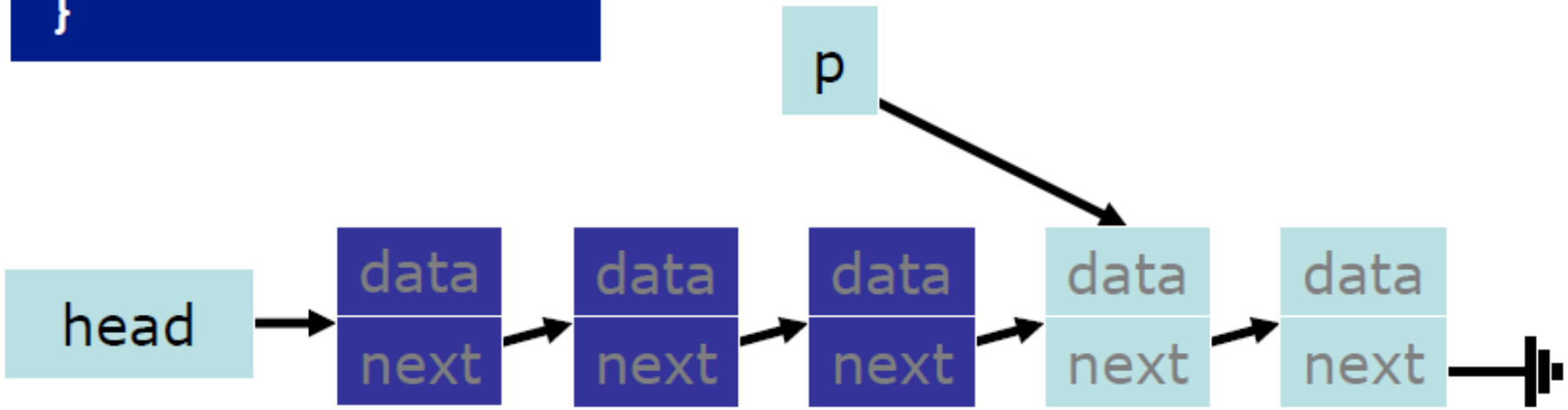
A Linked List Example

```
node *p = head;
while (p) {
   process(p);
   p = p->next;
}
```



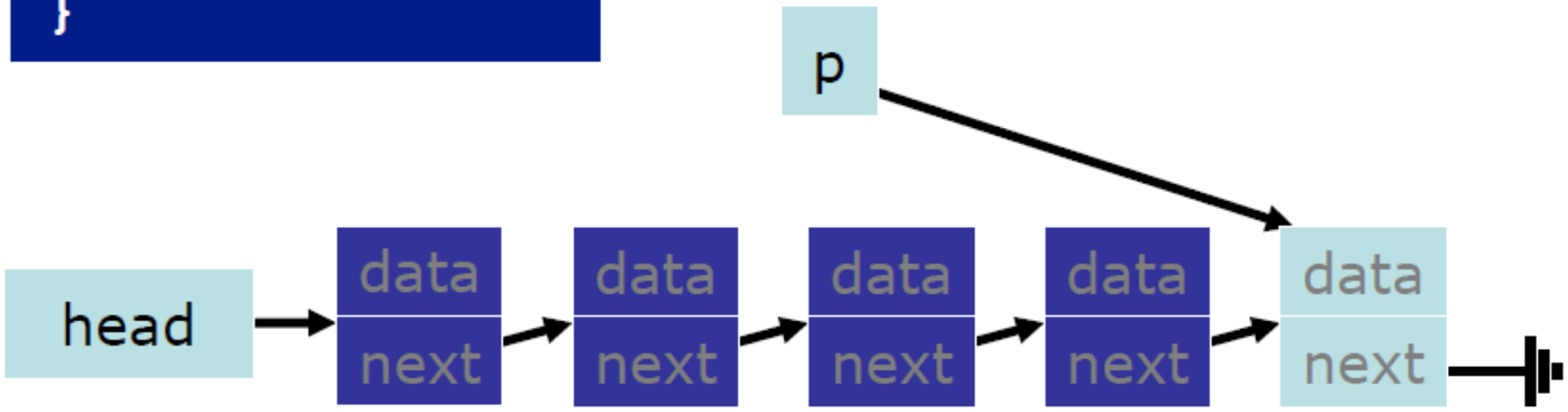
A Linked List Example

```
node *p = head;
while (p) {
   process(p);
   p = p->next;
}
```



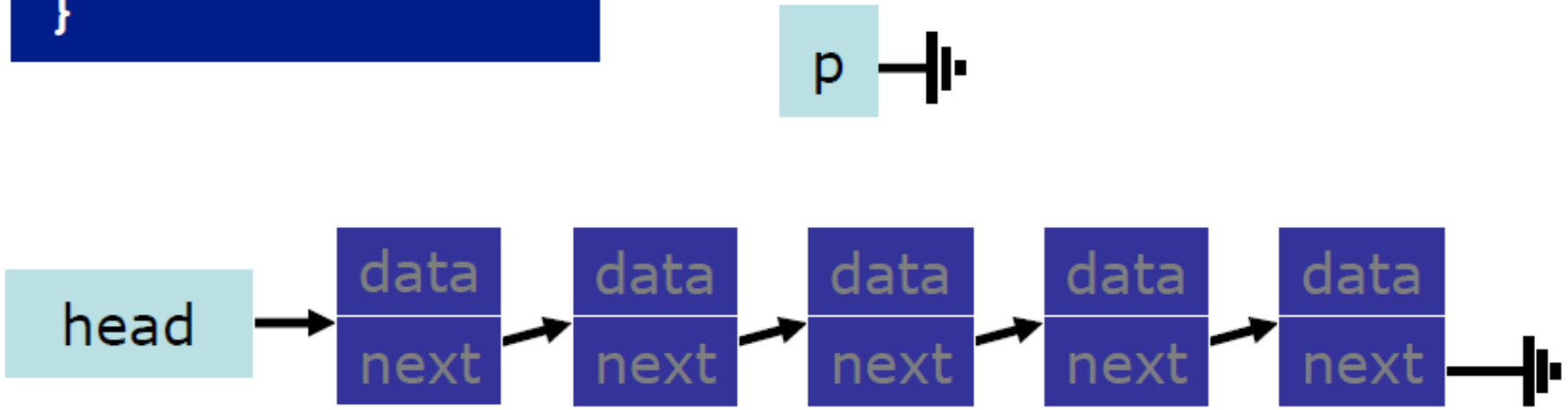
A Linked List Example

```
node *p = head;
while (p) {
   process(p);
   p = p->next;
}
```



A Linked List Example

```
node *p = head;
while (p) {
   process(p);
   p = p->next;
}
```



Task Construct – Explicit Task View

```
node *p = head;
#pragma omp parallel
{
#pragma omp single
while (p) {
  #pragma omp task
    process(p);
    p = p->next;
 }
}
```

A team of threads is forked at the `omp parallel` construct

A single thread, T0, executes the `while` loop

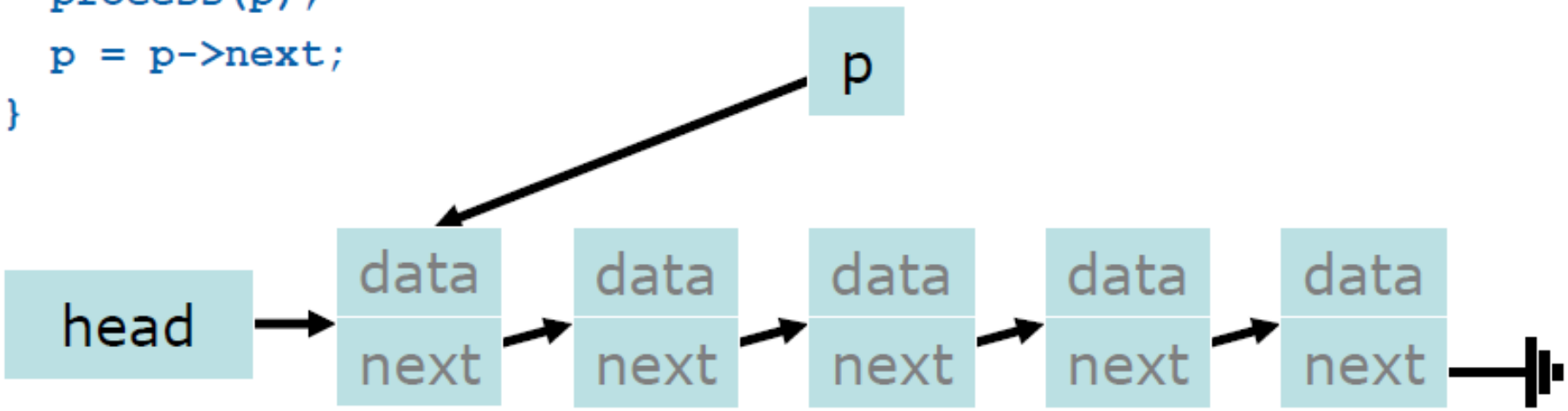
Each time T0 crosses the `omp task` construct it generates a new task

Each task runs in a thread

All tasks complete at the barrier at the end of the parallel region's single construct

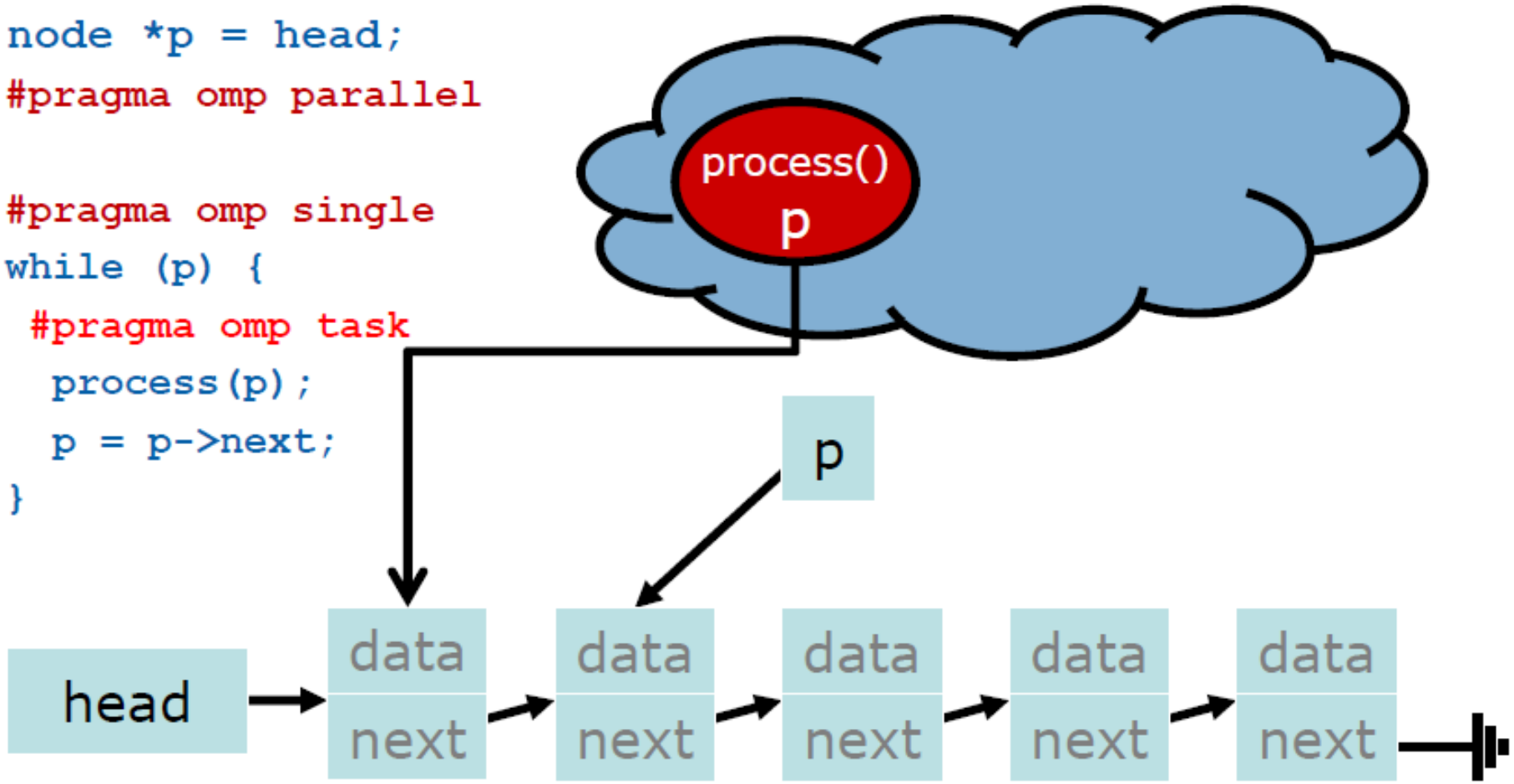
A Linked List Example

```
node *p = head;
#pragma omp parallel
{
#pragma omp single
while (p) {
 #pragma omp task
  process(p);
  p = p->next;
}
}
```



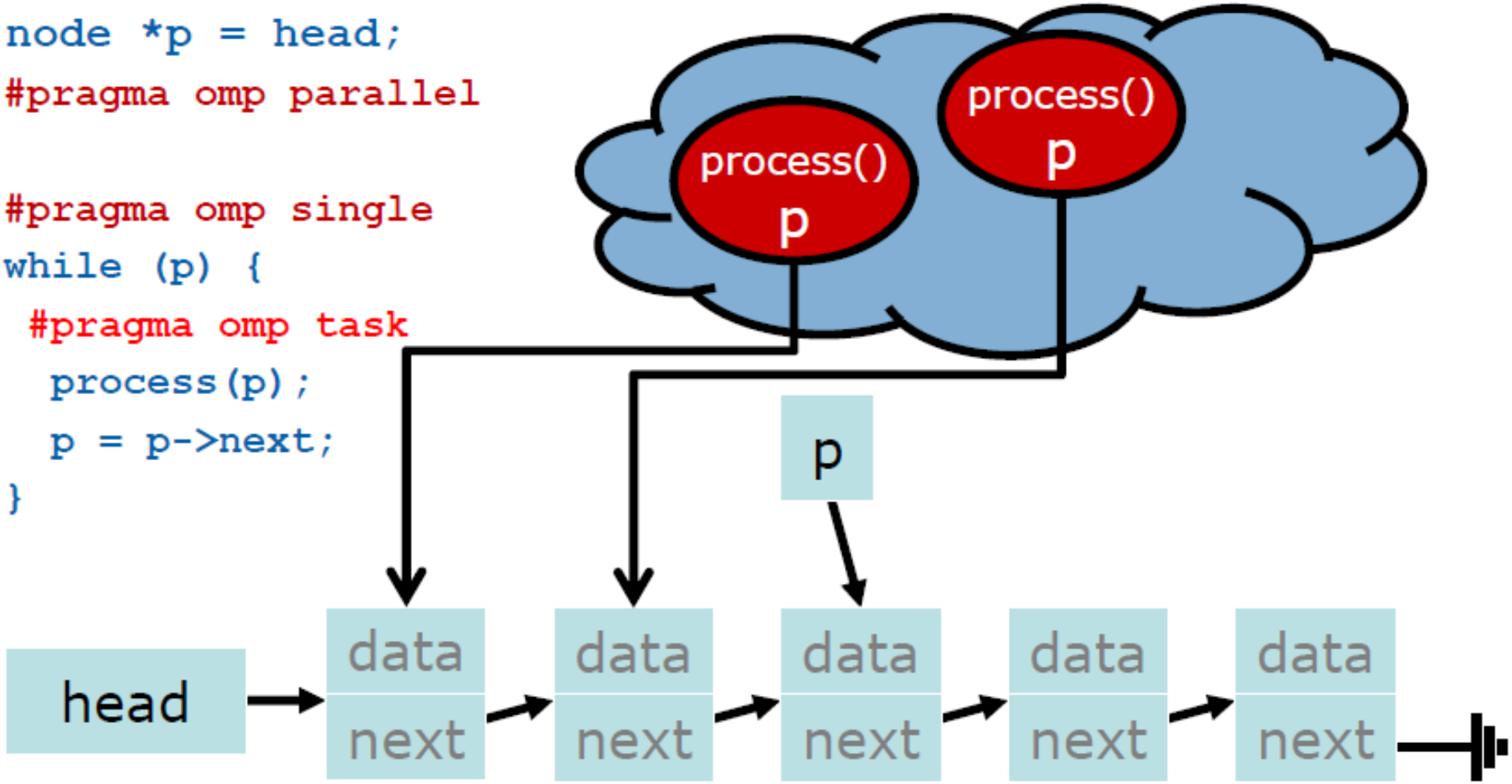
A Linked List Example

```
node *p = head;
#pragma omp parallel
{
#pragma omp single
while (p) {
 #pragma omp task
  process(p);
  p = p->next;
}
}
```



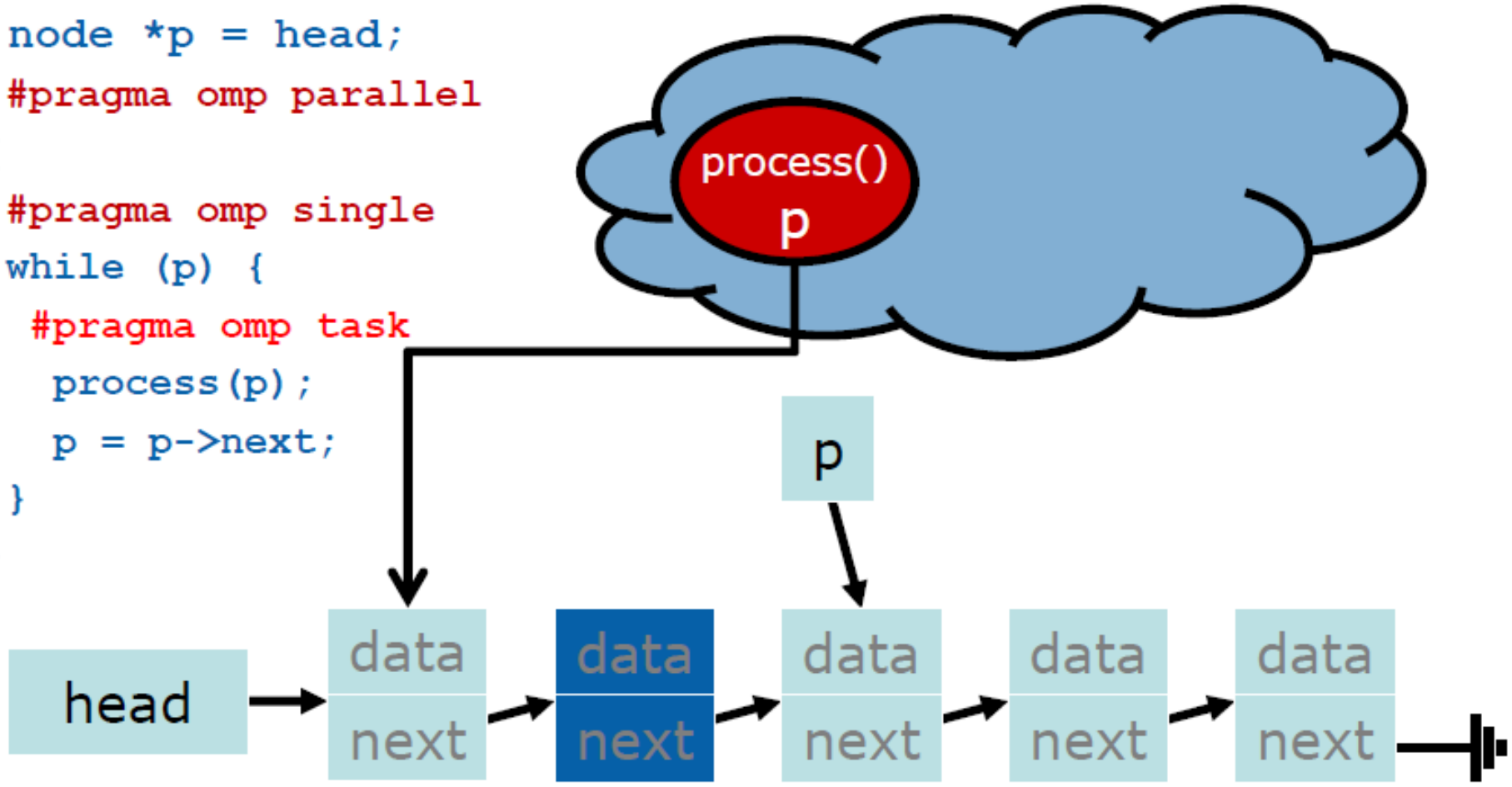
A Linked List Example

```
node *p = head;
#pragma omp parallel
{
#pragma omp single
while (p) {
 #pragma omp task
  process(p);
  p = p->next;
}
}
```



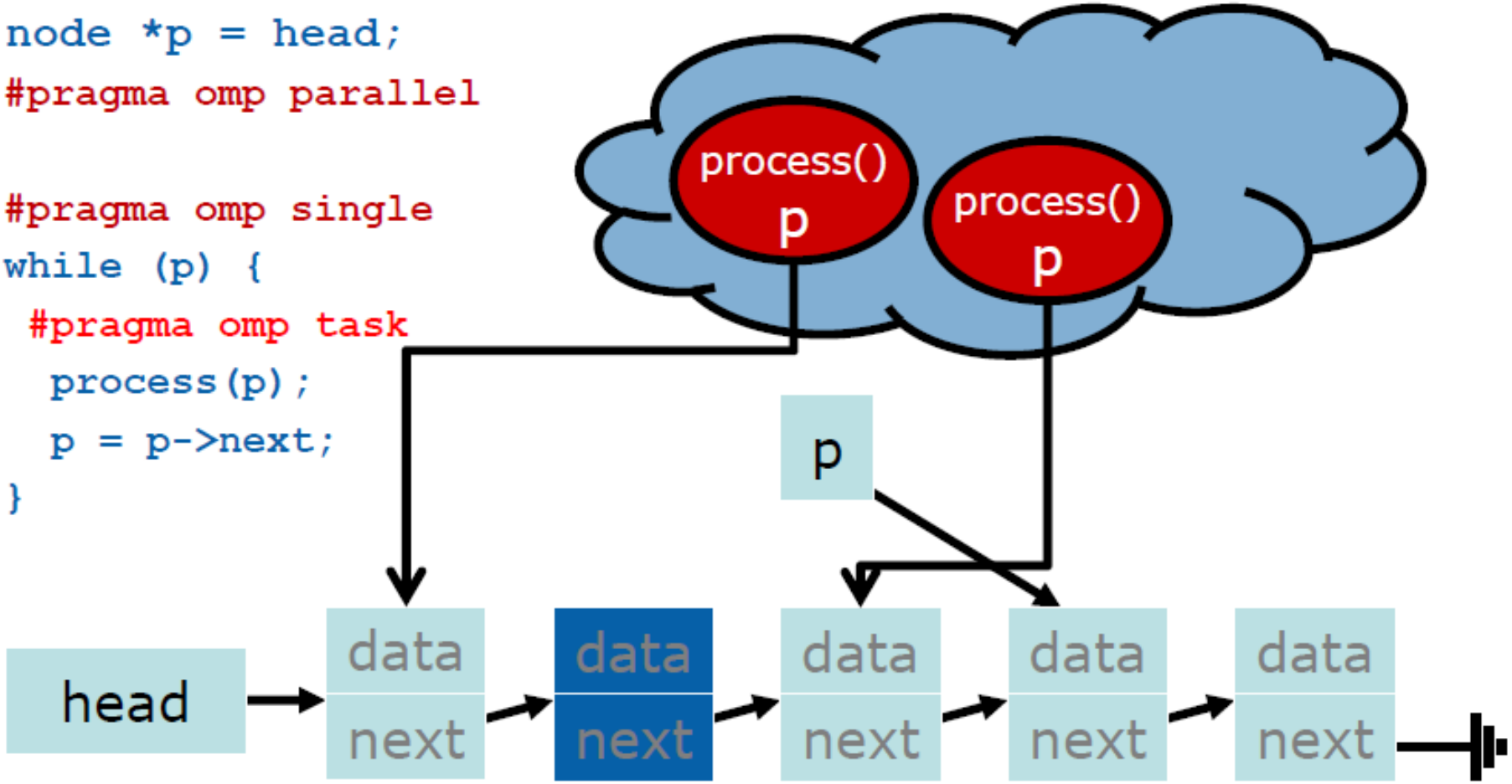
A Linked List Example

```
node *p = head;
#pragma omp parallel
{
#pragma omp single
while (p) {
 #pragma omp task
  process(p);
  p = p->next;
}
}
```



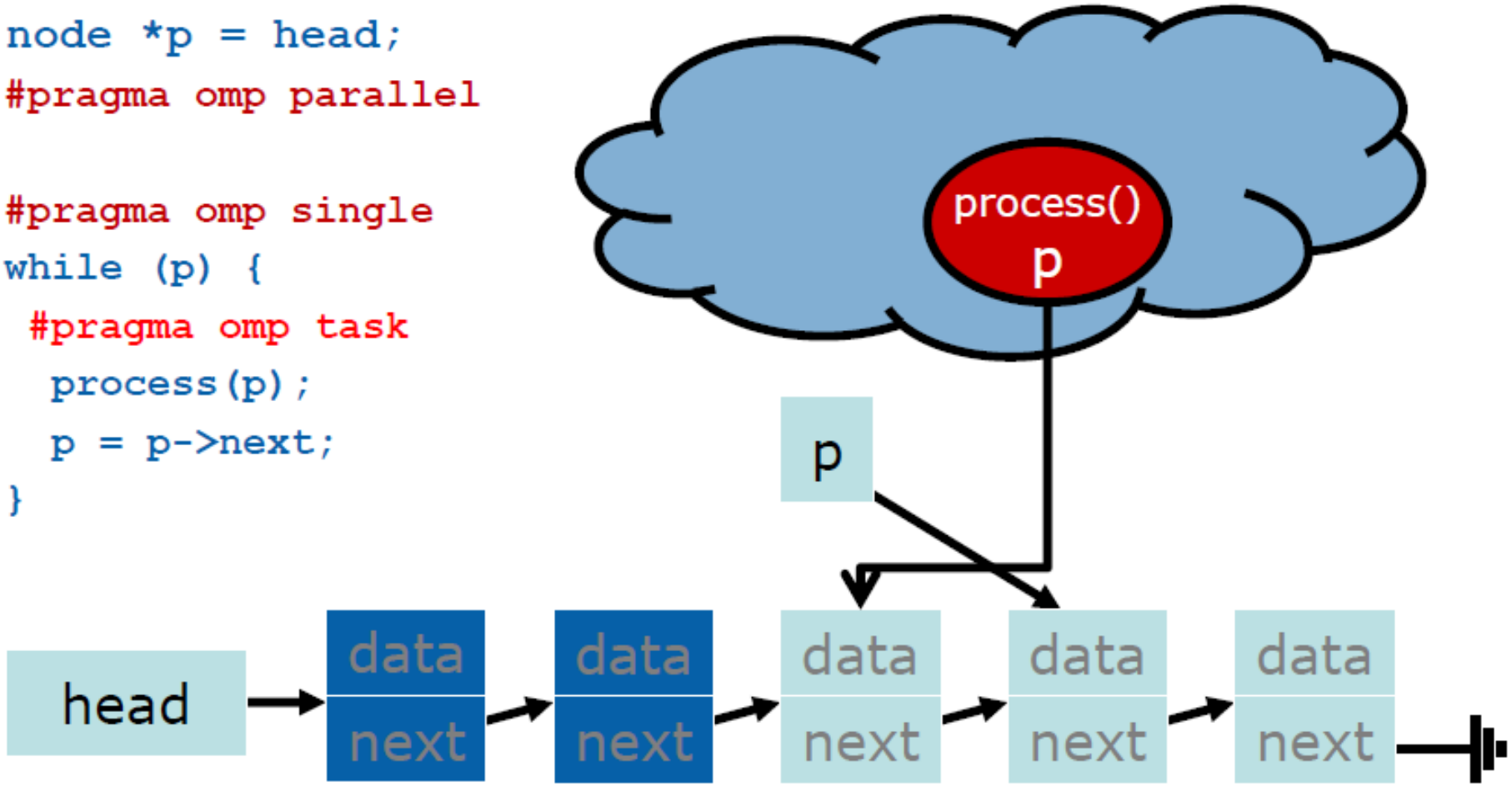
A Linked List Example

```
node *p = head;
#pragma omp parallel
{
#pragma omp single
while (p) {
 #pragma omp task
  process(p);
  p = p->next;
}
}
```



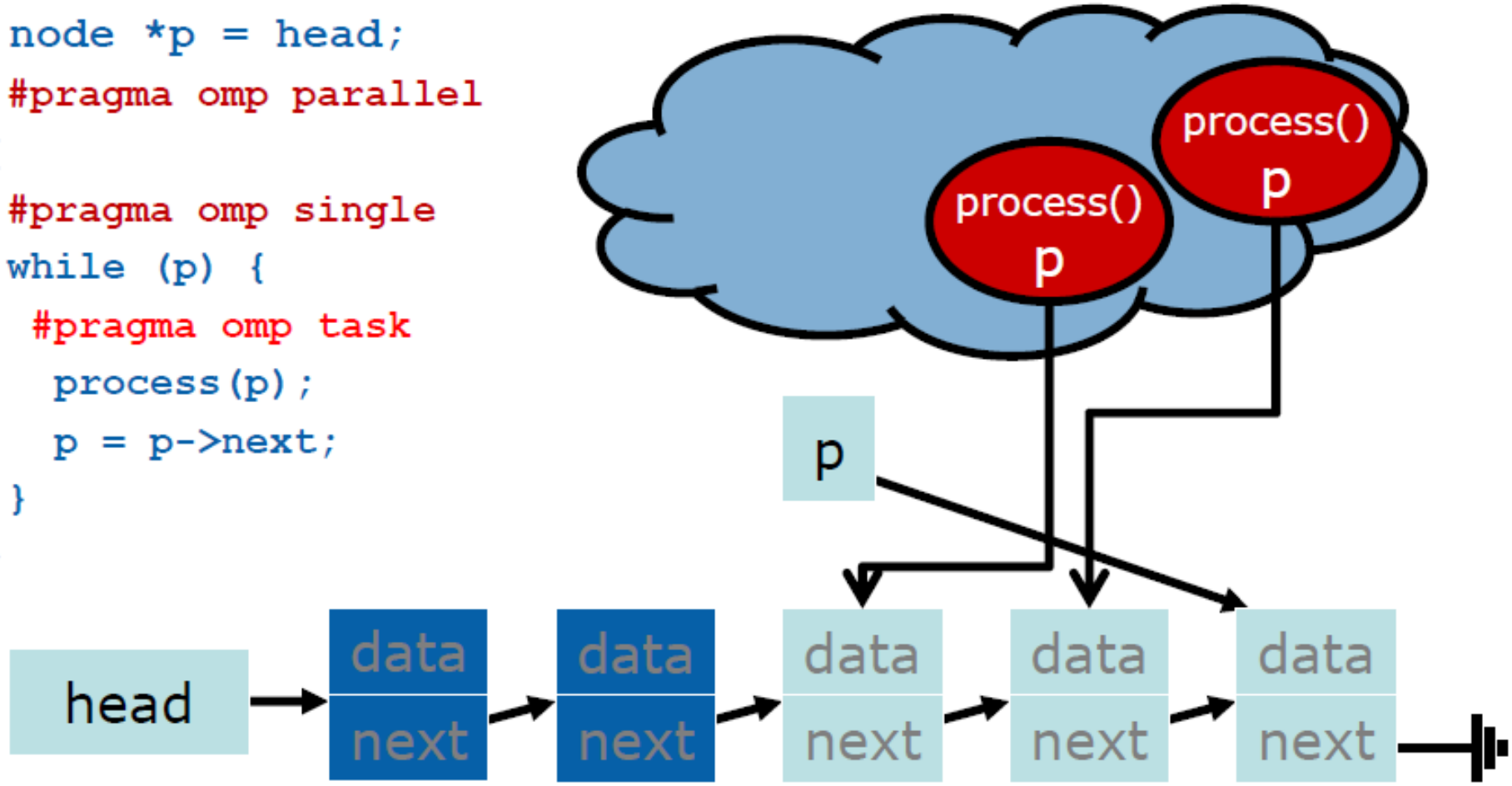
A Linked List Example

```
node *p = head;
#pragma omp parallel
{
#pragma omp single
while (p) {
 #pragma omp task
  process(p);
  p = p->next;
}
}
```



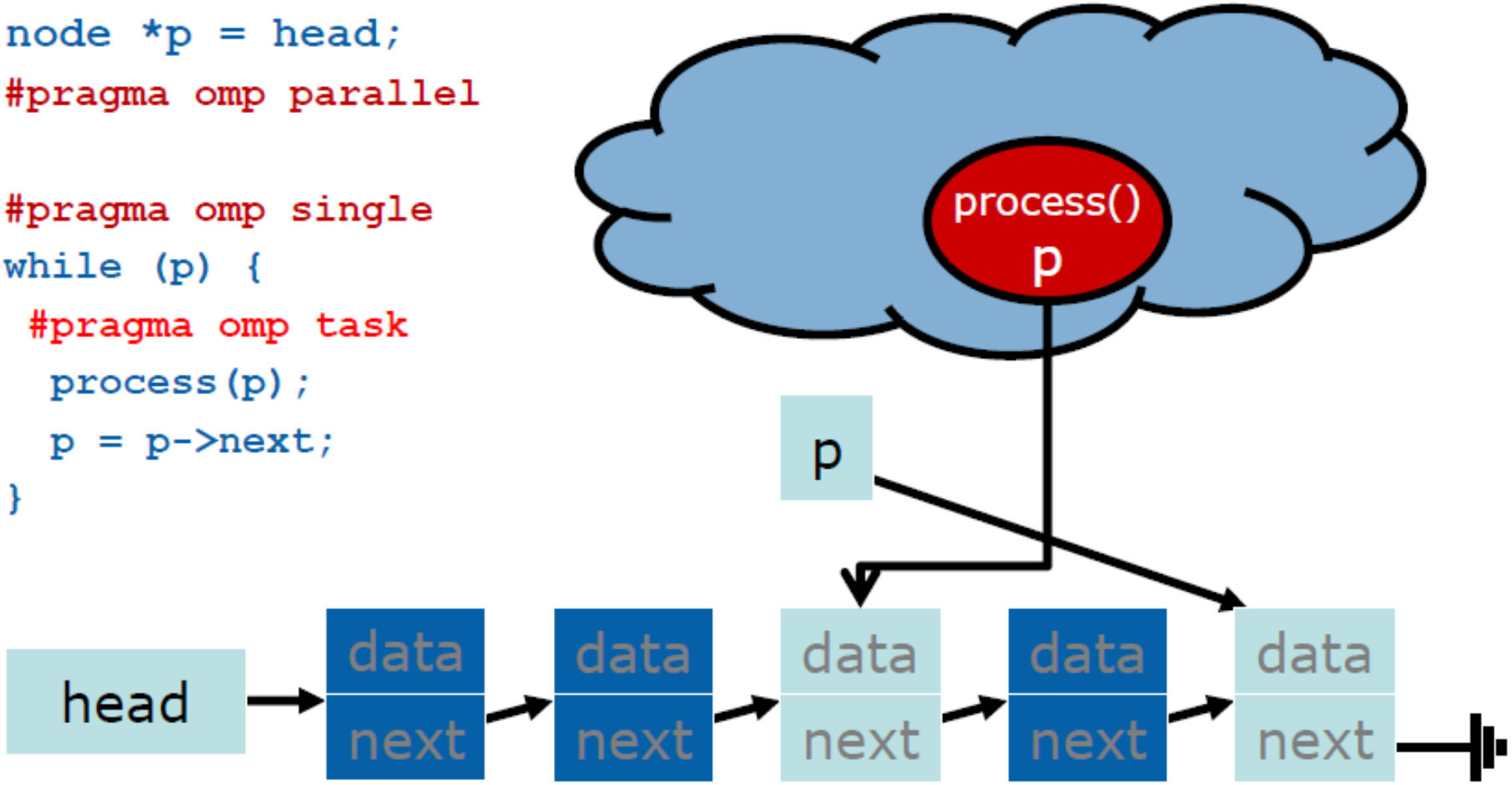
A Linked List Example

```
node *p = head;
#pragma omp parallel
{
#pragma omp single
while (p) {
 #pragma omp task
  process(p);
  p = p->next;
}
}
```



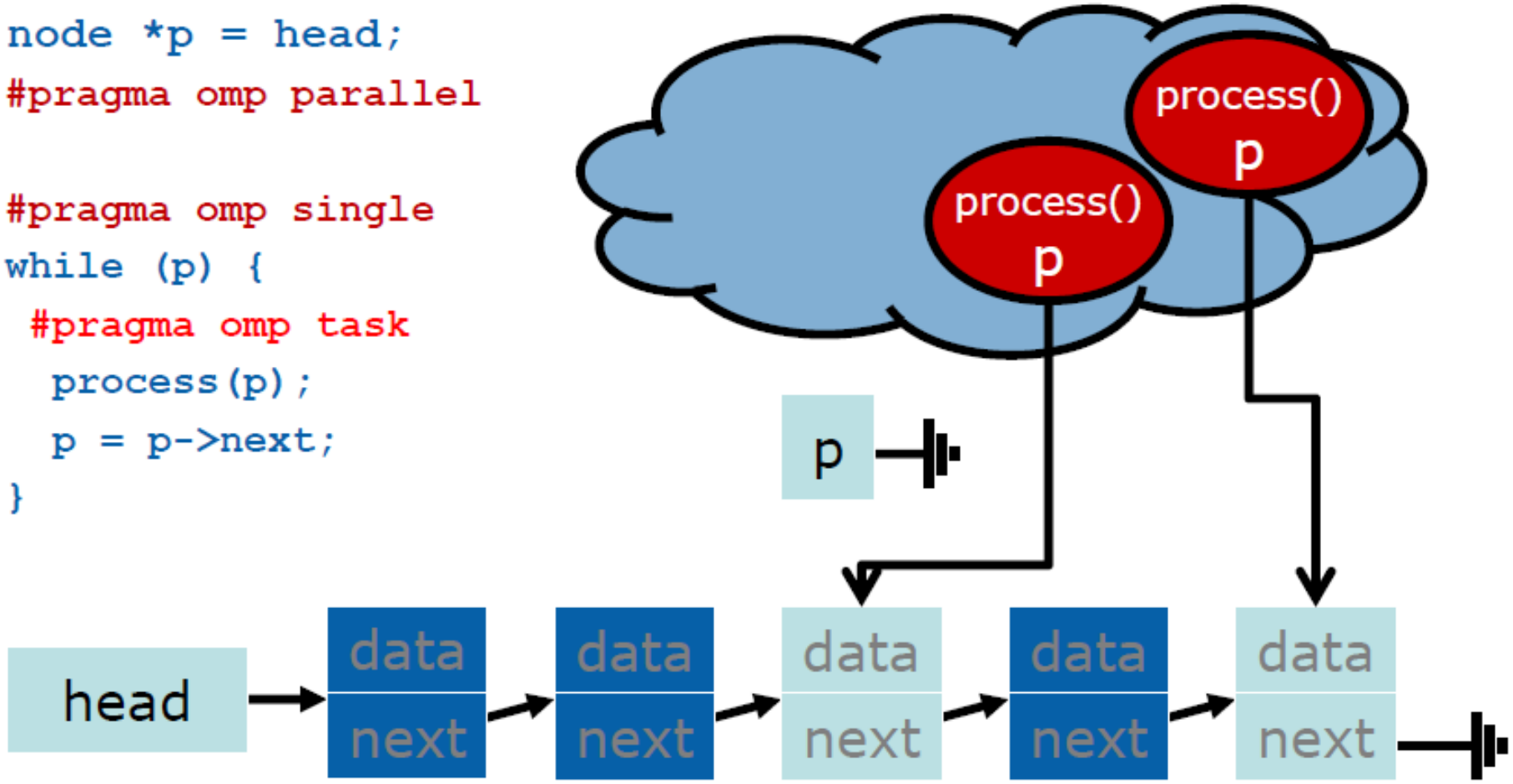
A Linked List Example

```
node *p = head;
#pragma omp parallel
{
#pragma omp single
while (p) {
 #pragma omp task
  process(p);
  p = p->next;
}
}
```



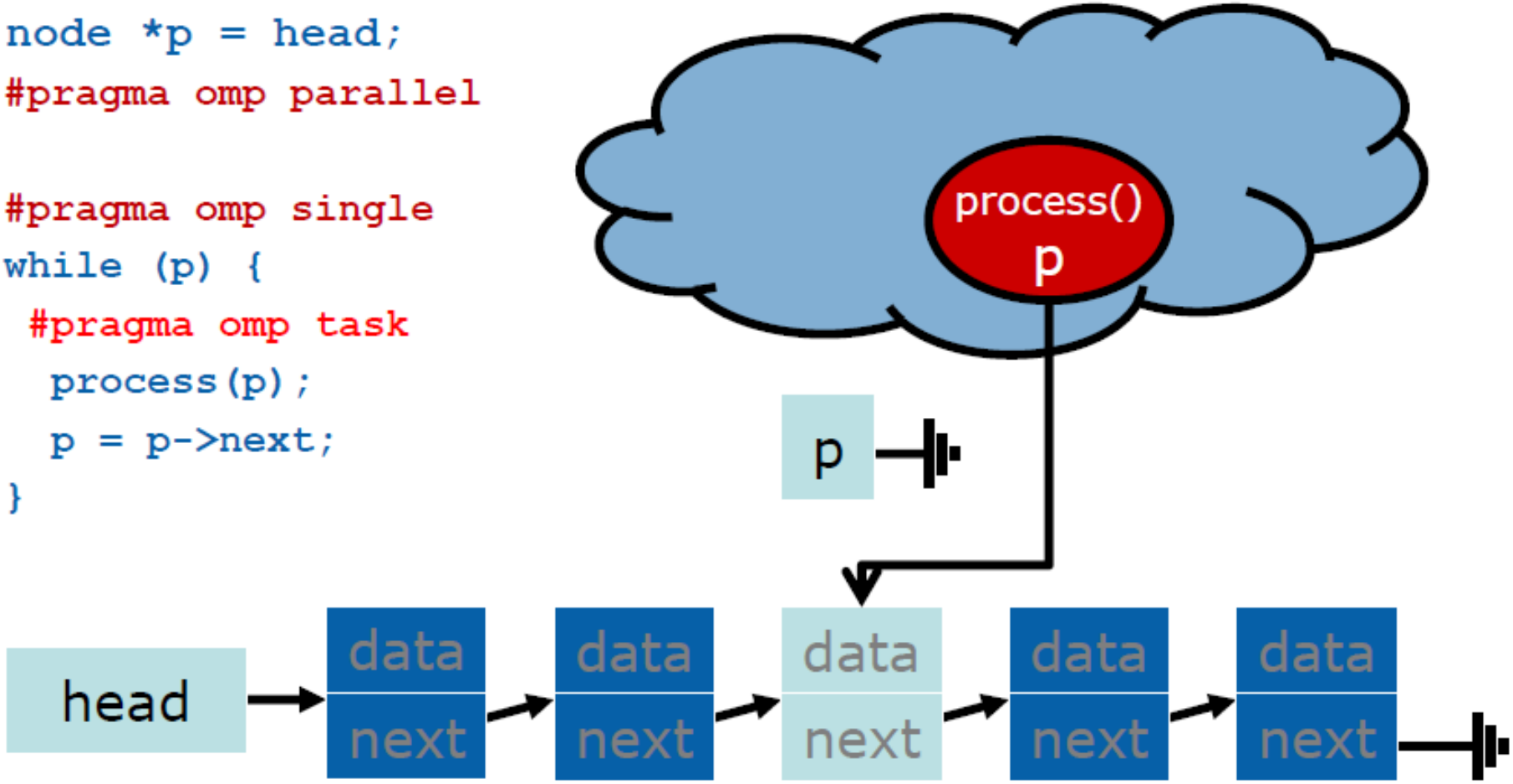
A Linked List Example

```
node *p = head;
#pragma omp parallel
{
#pragma omp single
while (p) {
 #pragma omp task
  process(p);
  p = p->next;
}
}
```



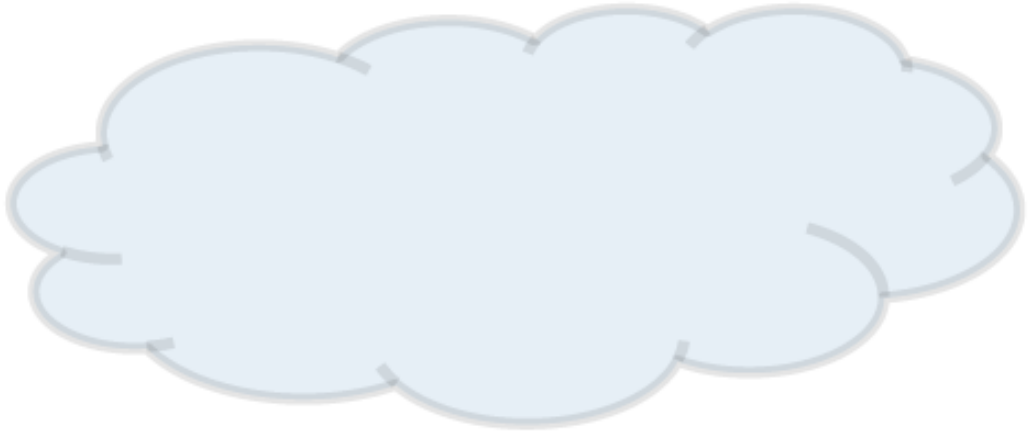
A Linked List Example

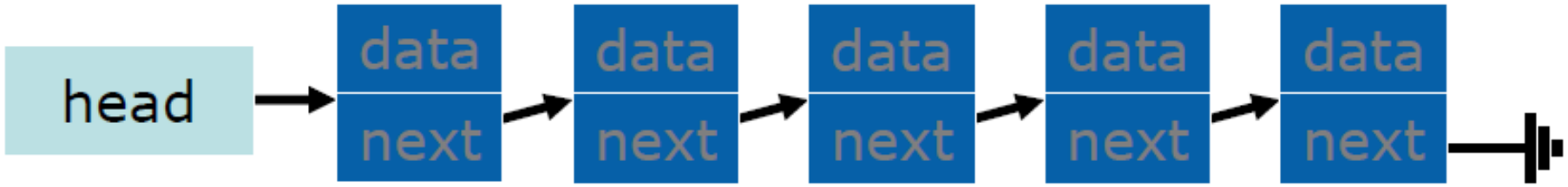
```
node *p = head;
#pragma omp parallel
{
#pragma omp single
while (p) {
 #pragma omp task
  process(p);
  p = p->next;
}
}
```



A Linked List Example

```
node *p = head;
#pragma omp parallel
{
#pragma omp single
while (p) {
 #pragma omp task
  process(p);
  p = p->next;
}
}
```





When are tasks gauranteed to be complete?

Tasks are gauranteed to be complete:

- At thread or task barriers
- At the directive: `#pragma omp barrier`
- At the directive: `#pragma omp taskwait`

Example: Naive Fibonacci Calculation

Recursion typically used to calculate Fibonacci number

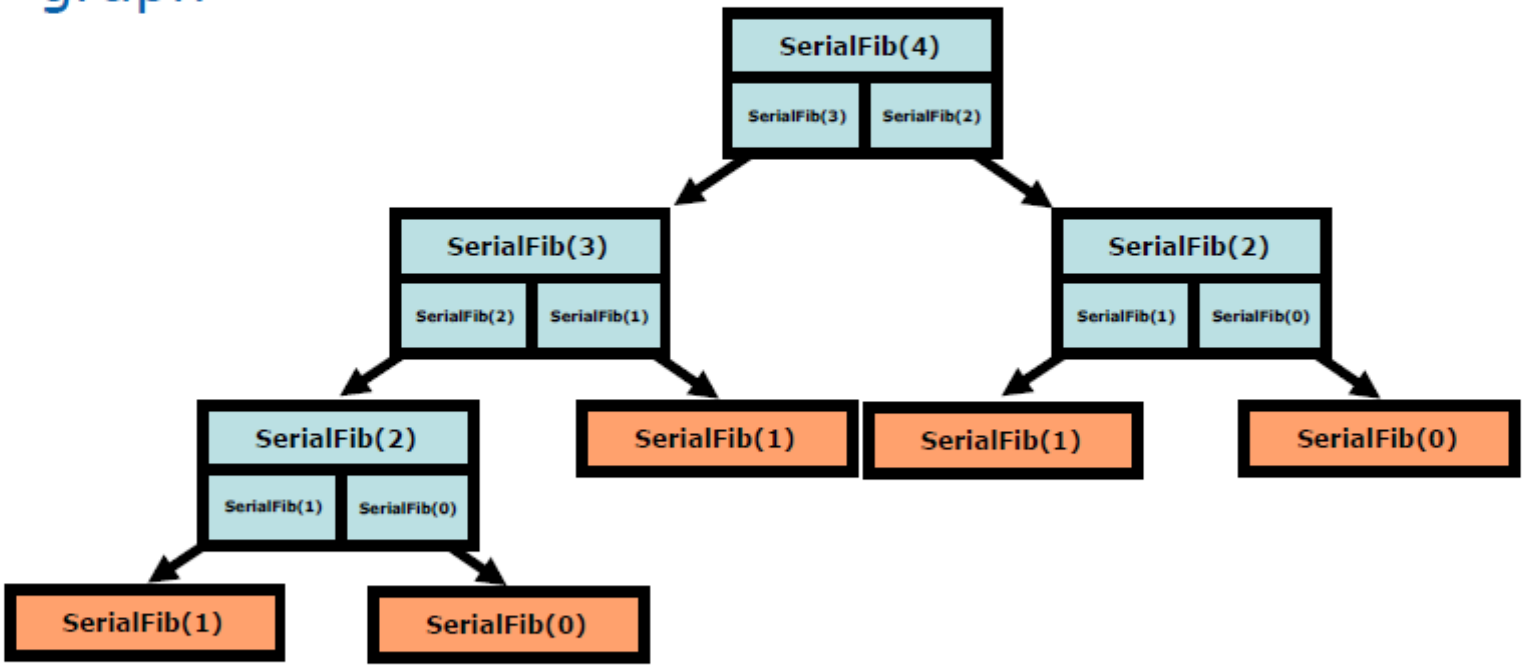
Widely used as toy benchmark

- Easy to code
- Has unbalanced task graph

```
long SerialFib( long n ) {
    if( n < 2 )
        return n;
    else
        return SerialFib(n-1) + SerialFib(n-2);
}
```

Example: Naive Fibonacci Calculation

We can envision Fibonacci computation as a task graph



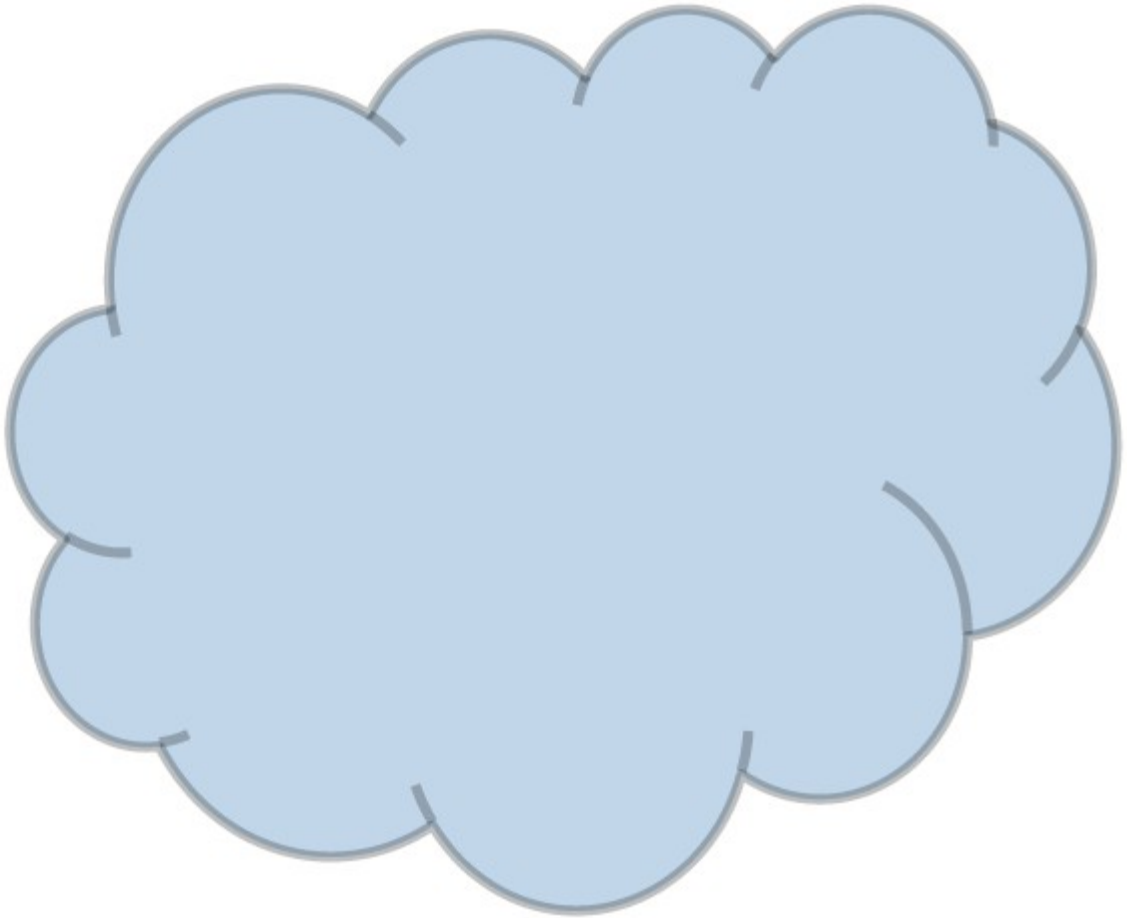
Fibonacci - Task Spawning Solution

```
long ParallelFib(long n)
{ long sum;
 #pragma omp parallel
 {
  #pragma omp single
   FibTask(n,&sum);
 }
  return sum;
}
```

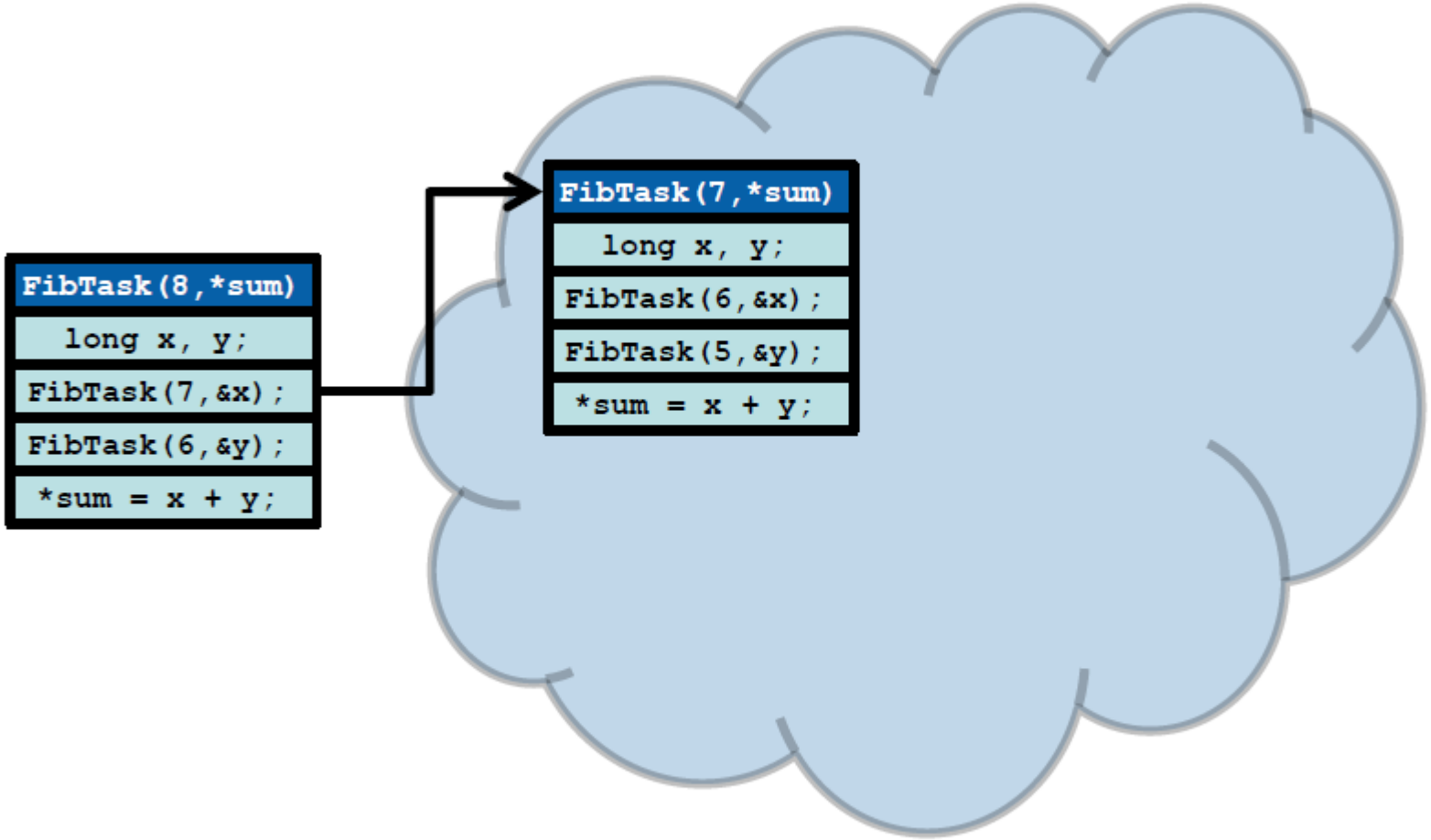
- Write a helper function to set up parallel region
- Call `FibTask()` to do computation
- Use `sum` return parameter in `FibTask()`

Fibonacci Task Solution Example

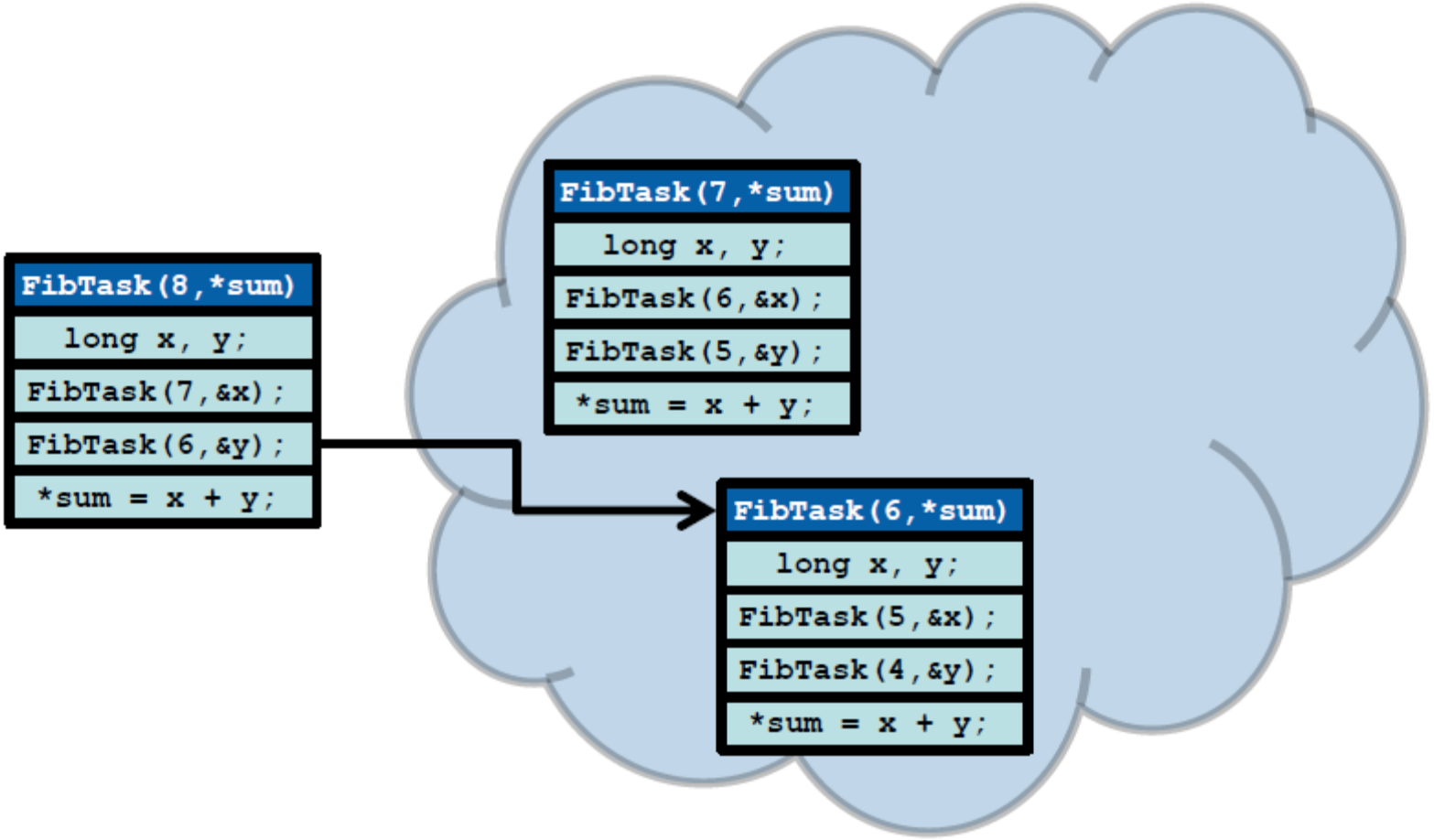
```
FibTask(8,*sum)
  long x, y;
FibTask(7,&x);
FibTask(6,&y);
*sum = x + y;
```



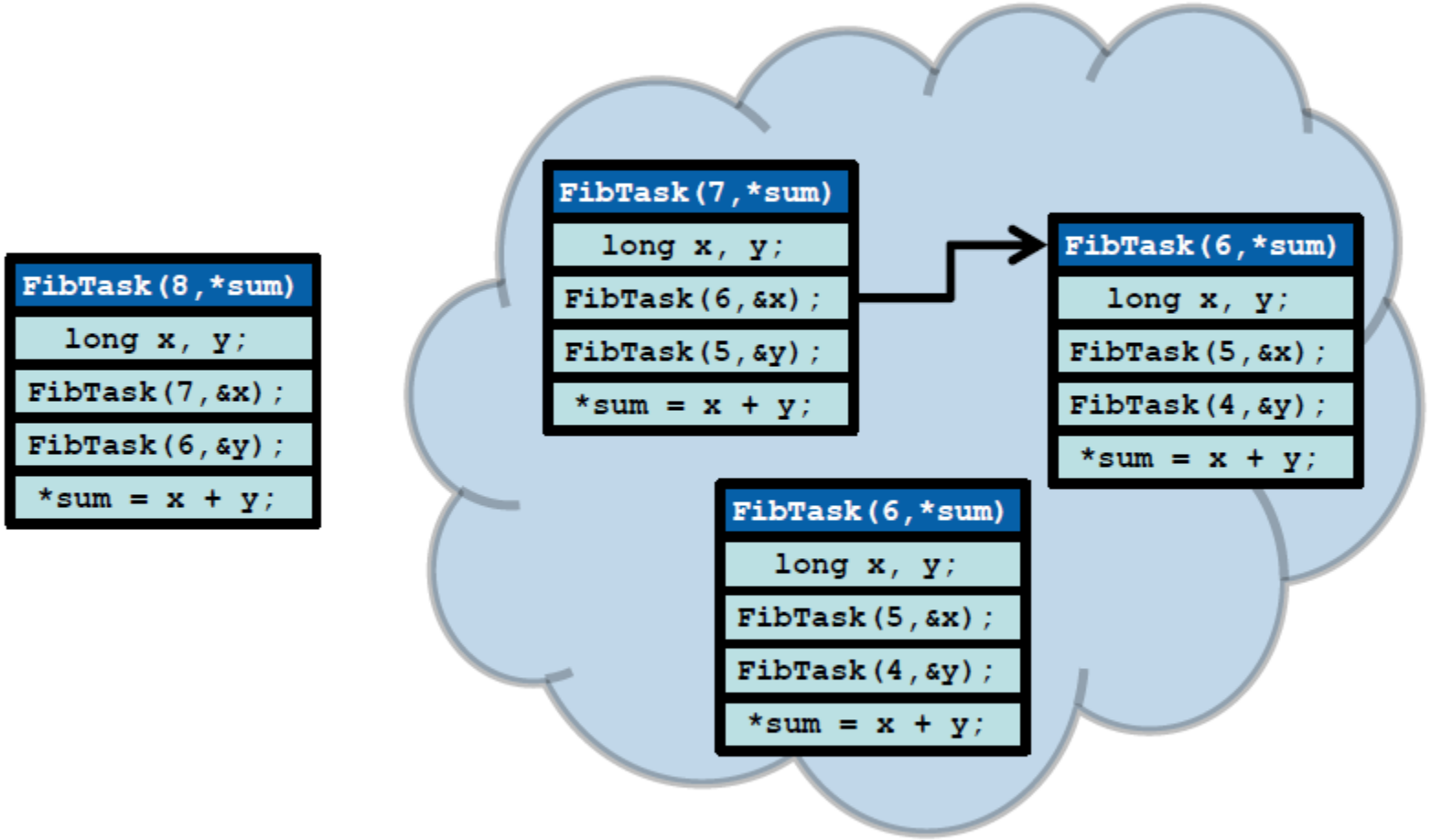
Fibonacci Task Solution Example



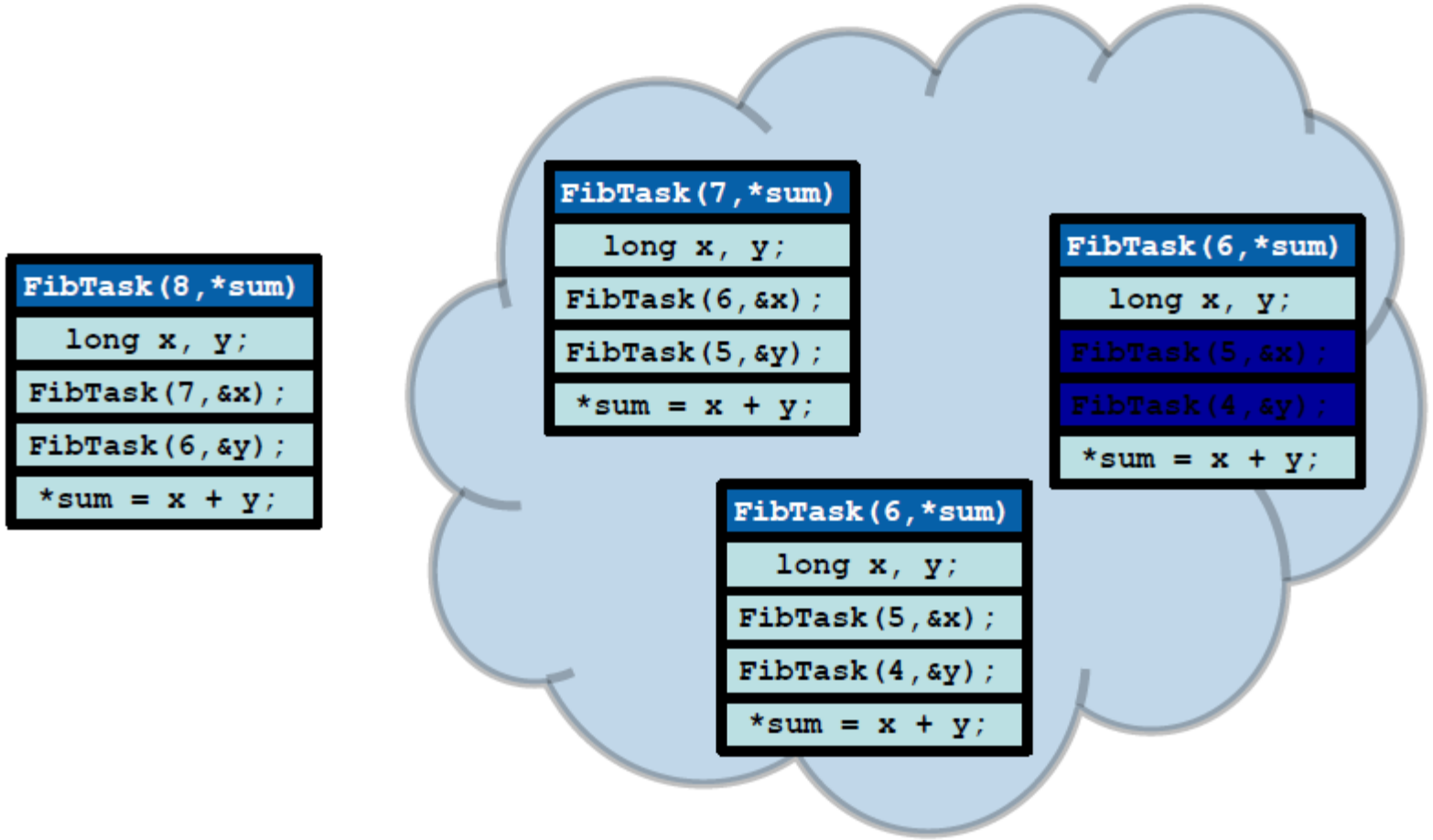
Fibonacci Task Solution Example



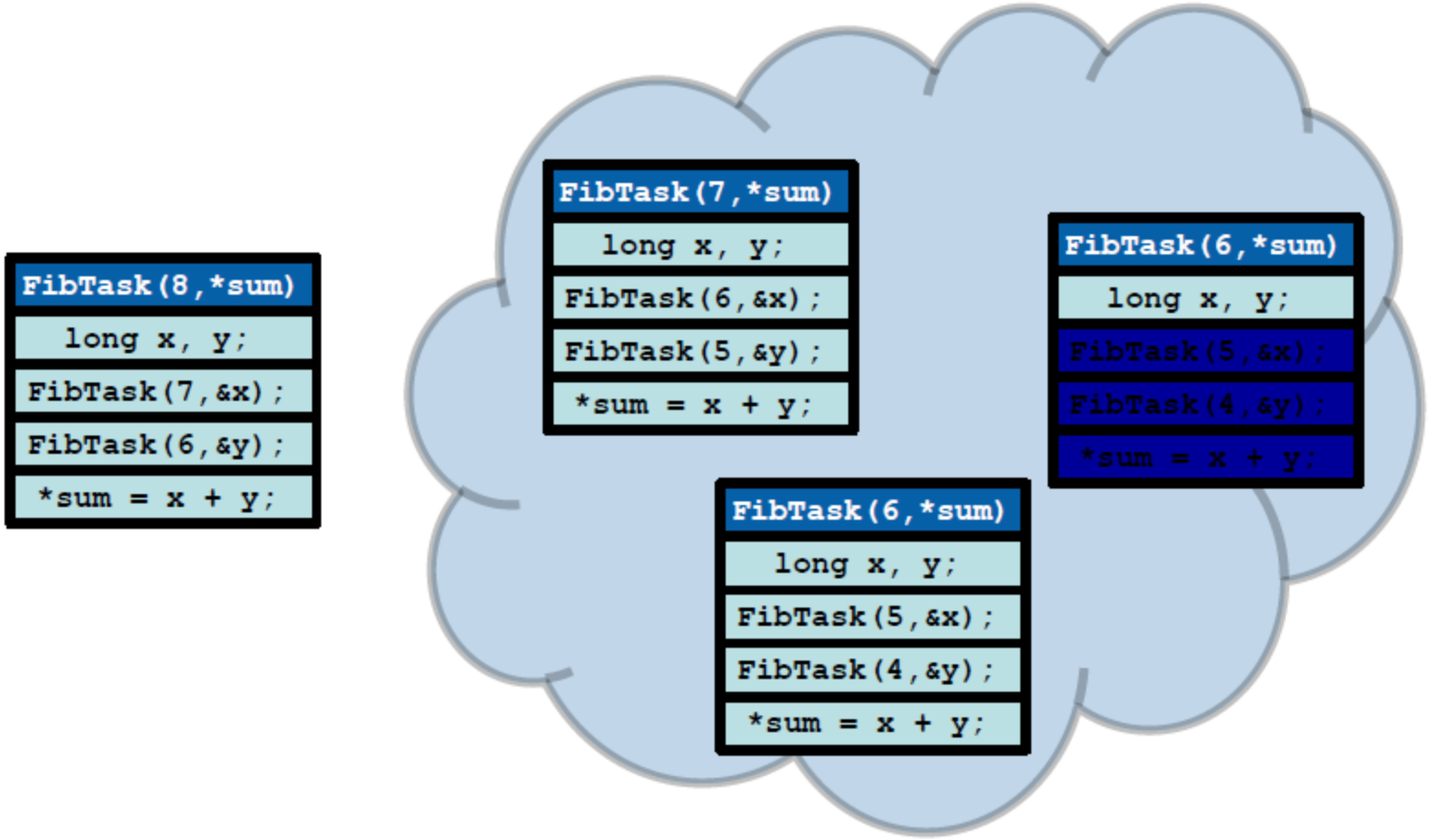
Fibonacci Task Solution Example



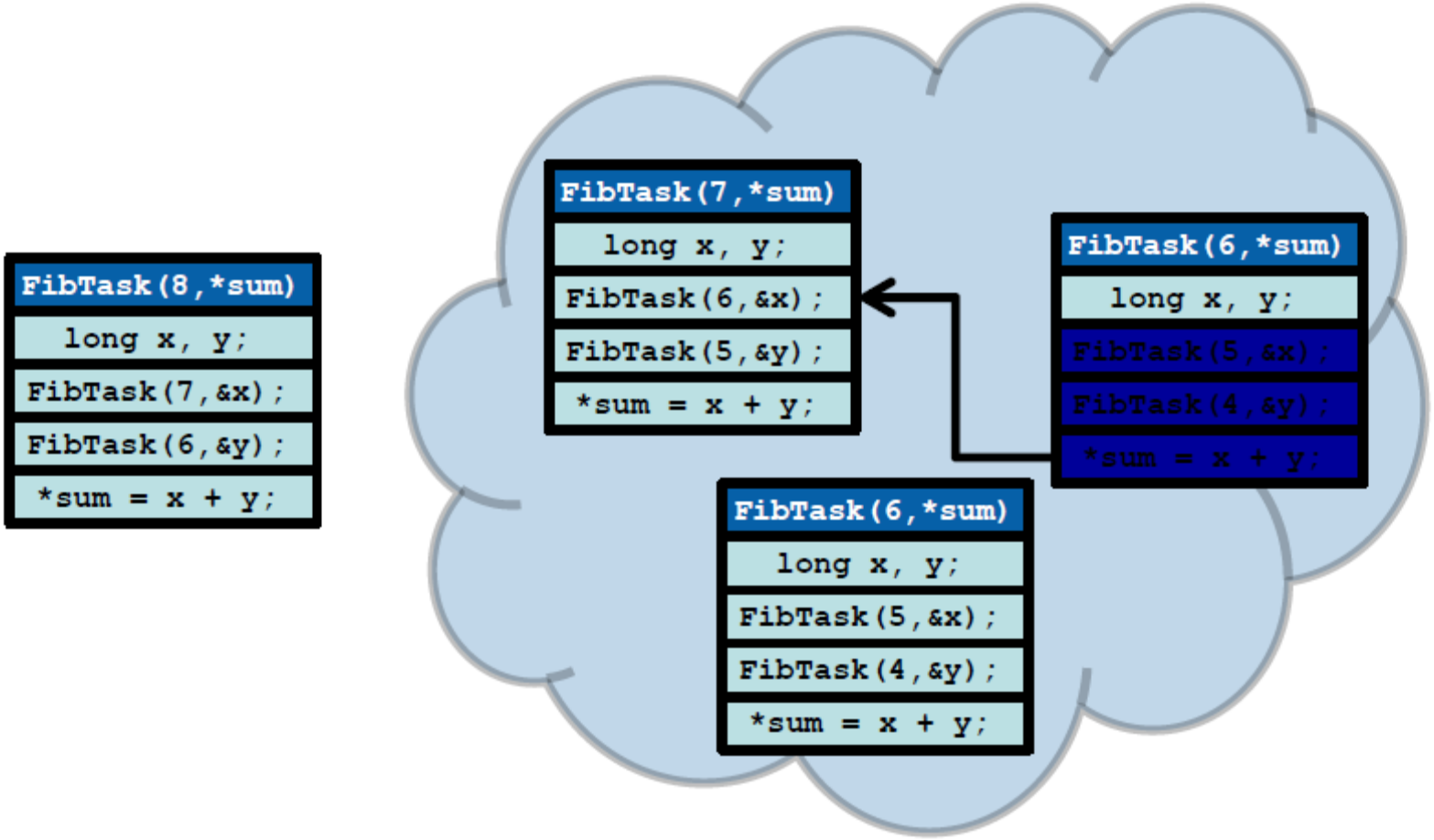
Fibonacci Task Solution Example



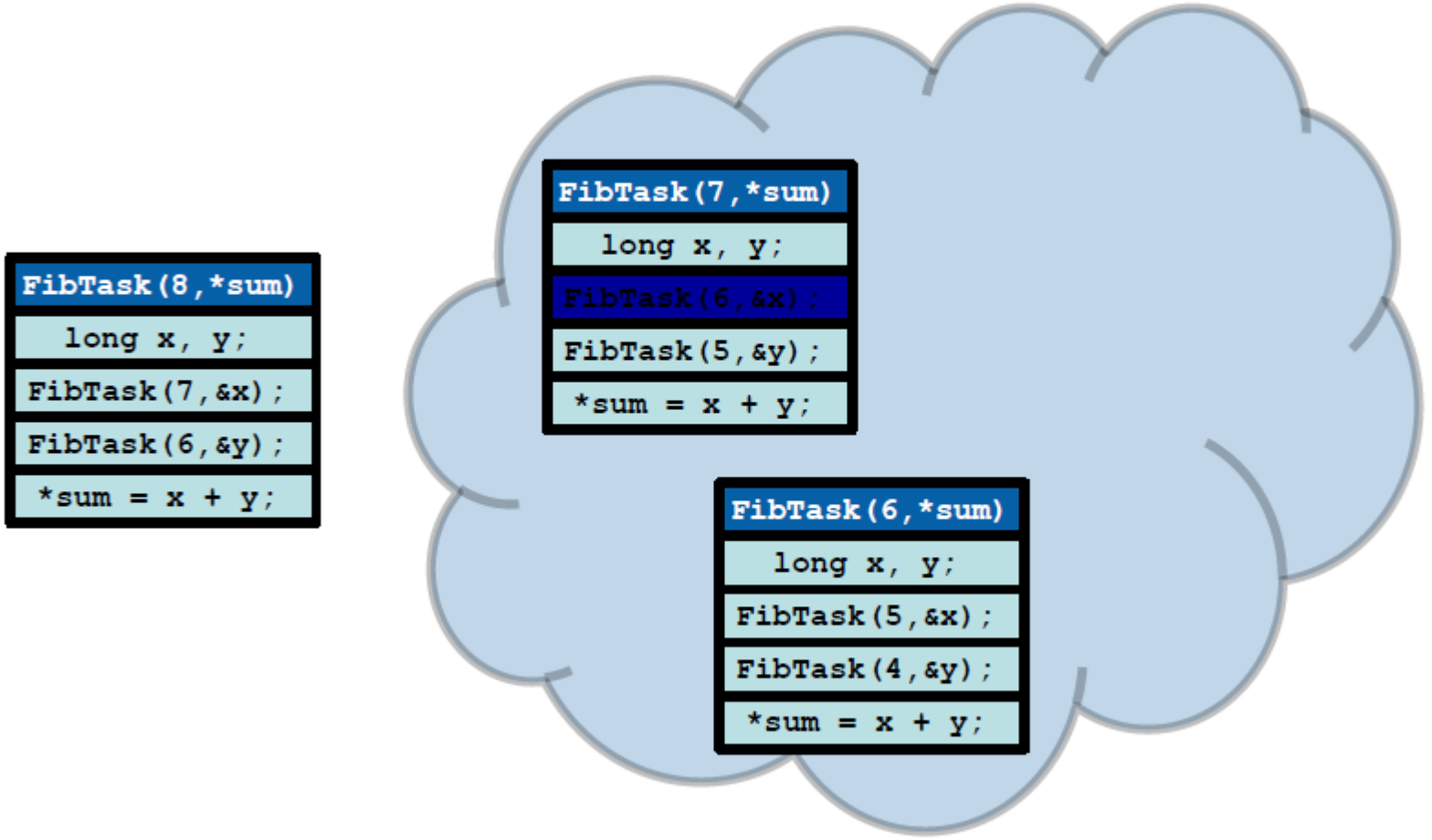
Fibonacci Task Solution Example



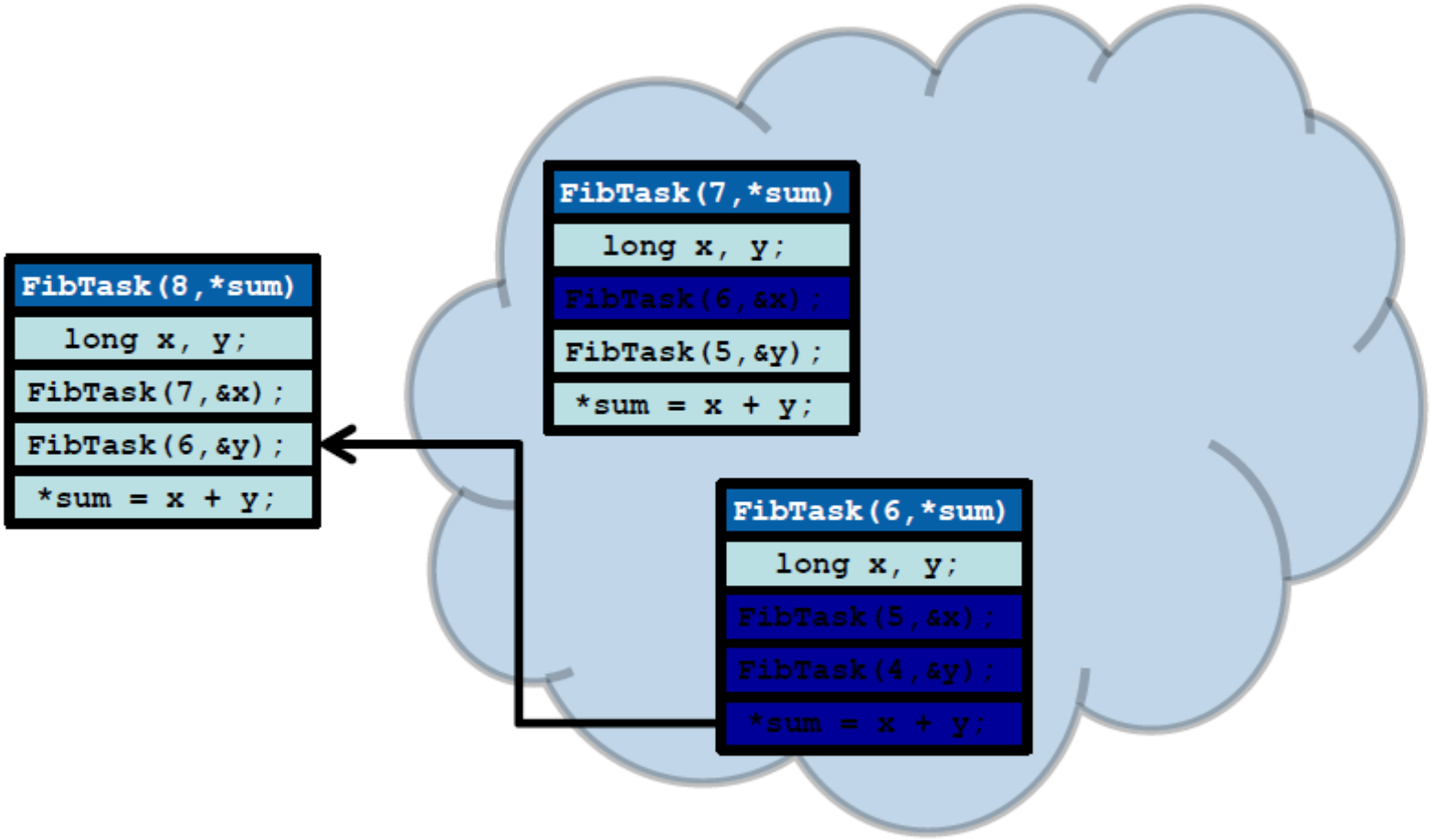
Fibonacci Task Solution Example



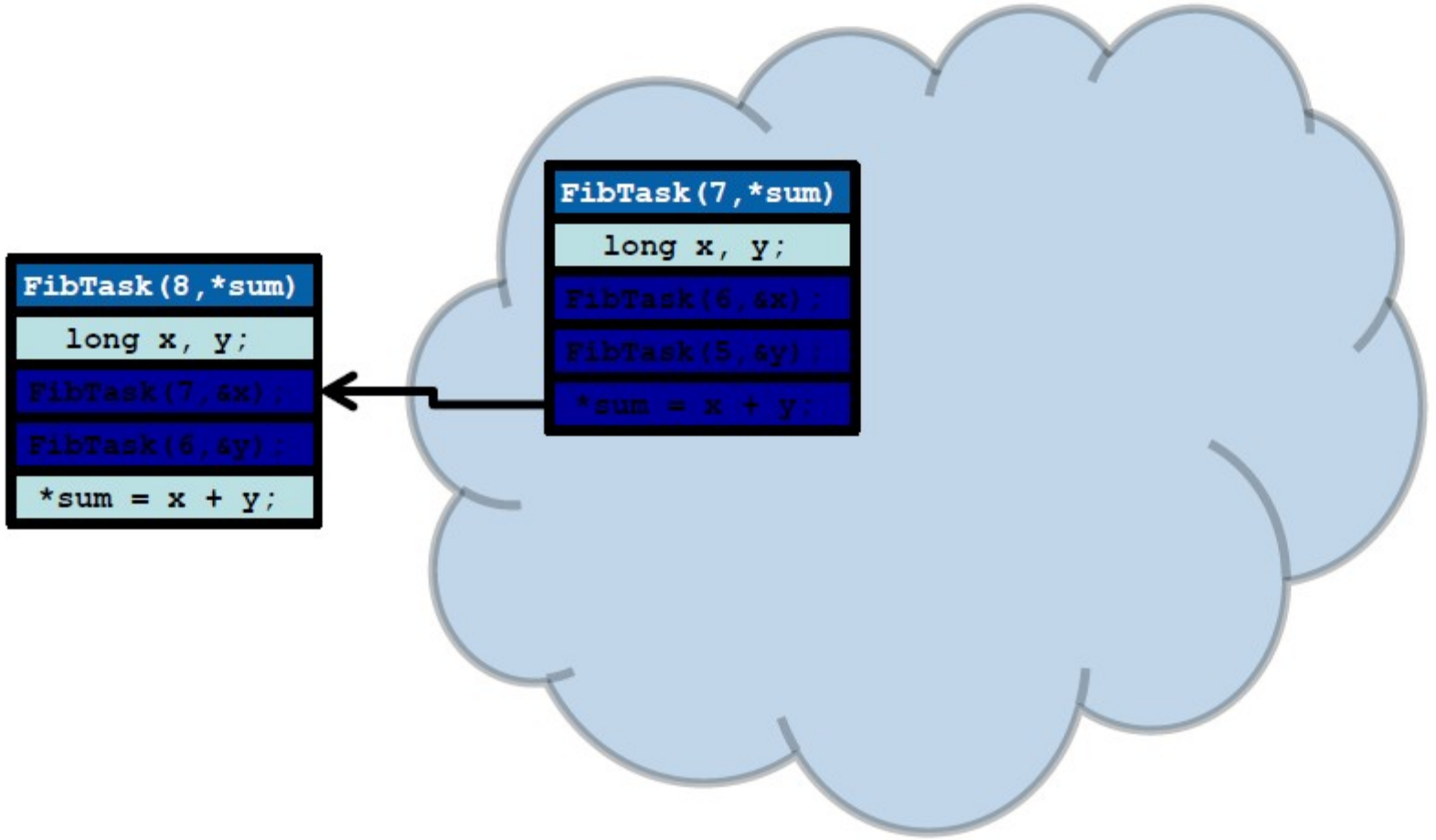
Fibonacci Task Solution Example



Fibonacci Task Solution Example

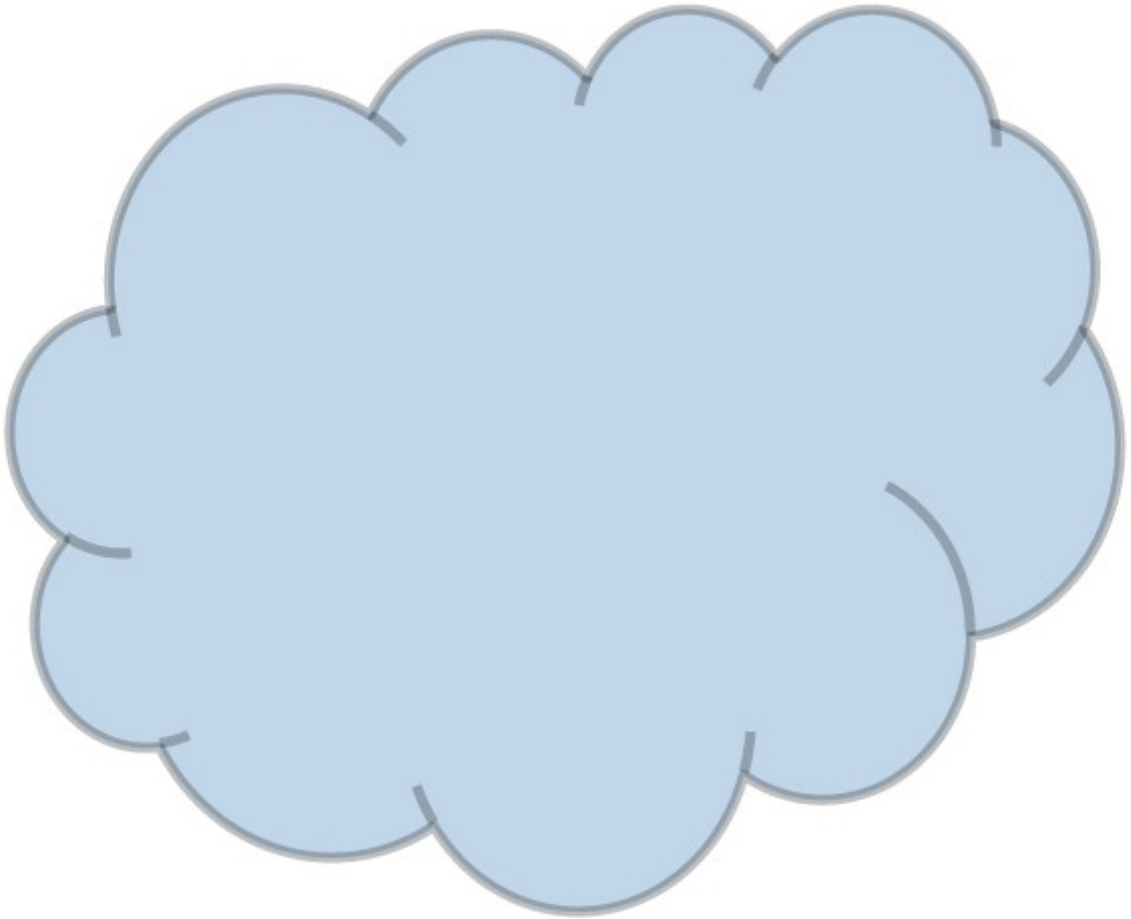


Fibonacci Task Solution Example



Fibonacci Task Solution Example

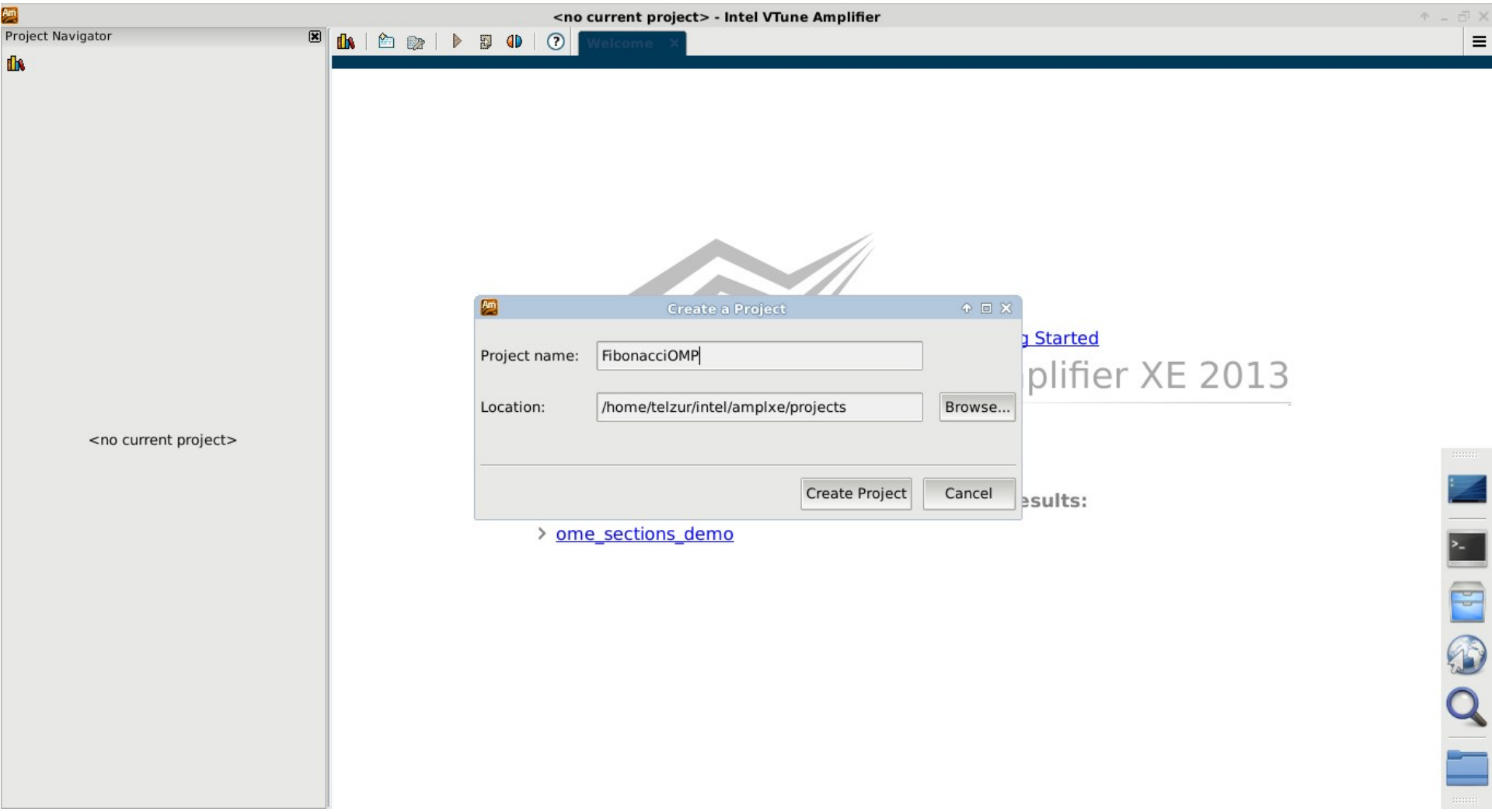
```
FibTask(8,*sum)
  long x, y;
  FibTask(7,&x);
  FibTask(6,&y);
  *sum = x + y;
```

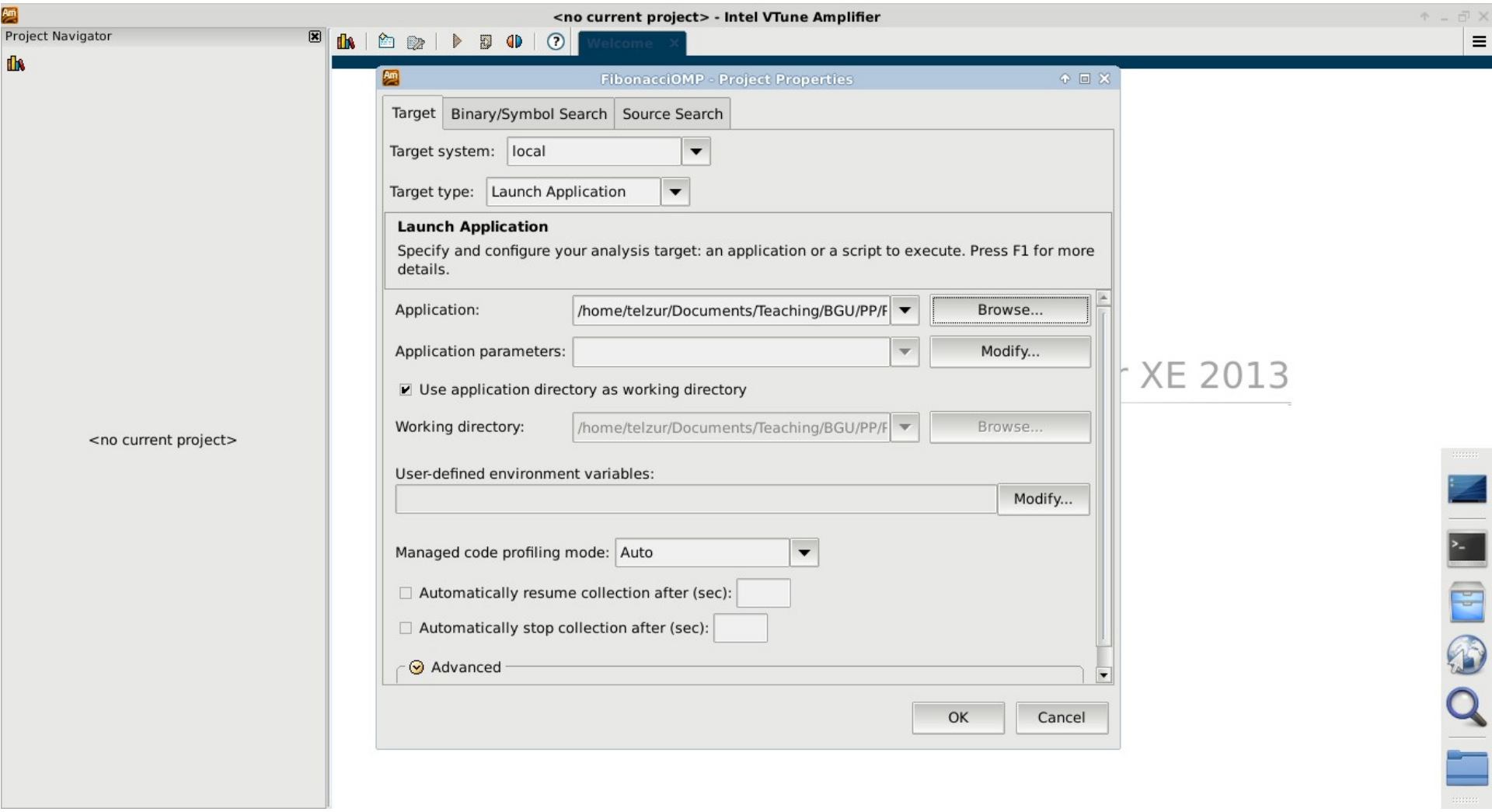


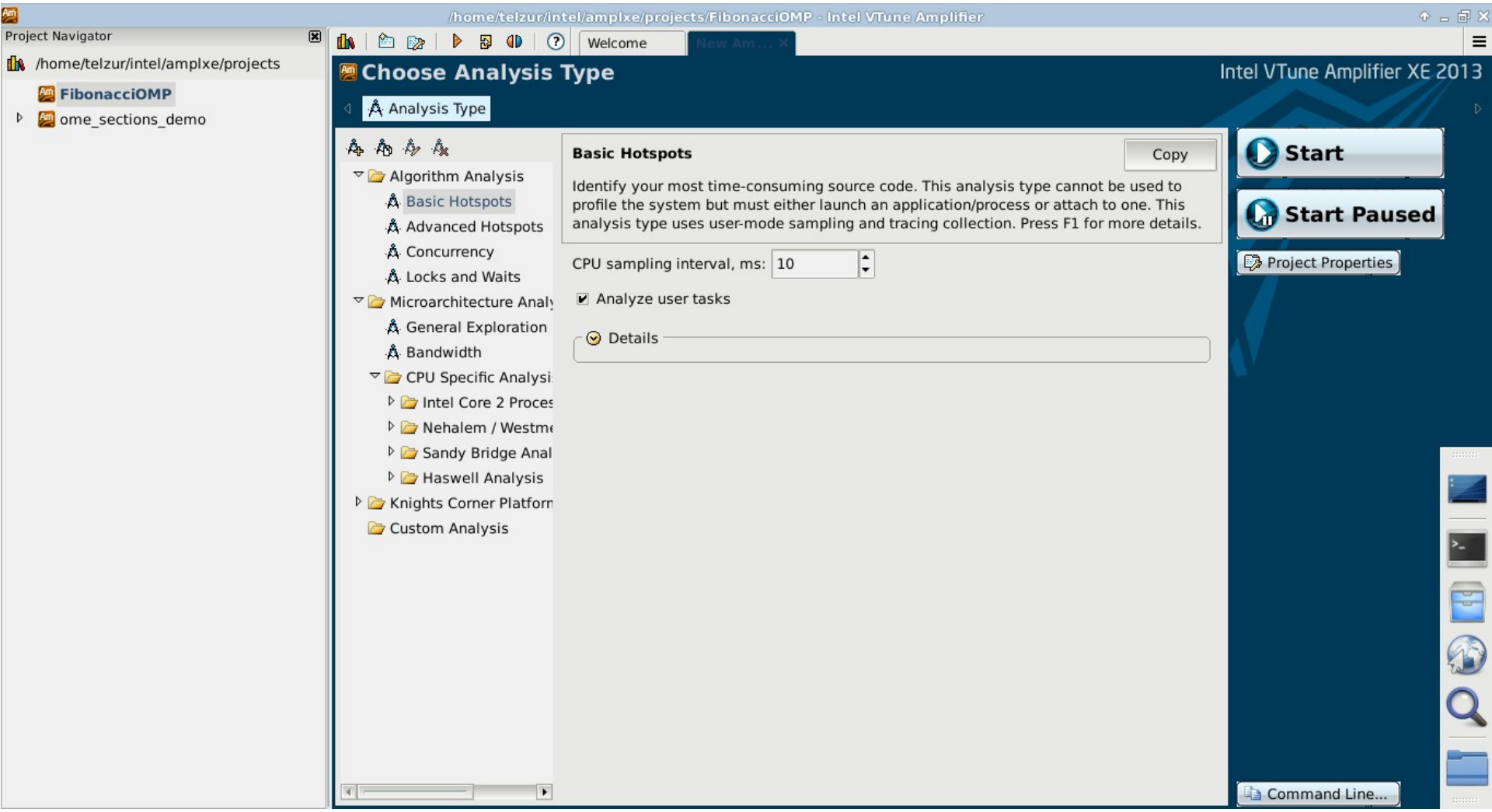
Back to lecture09 presentation

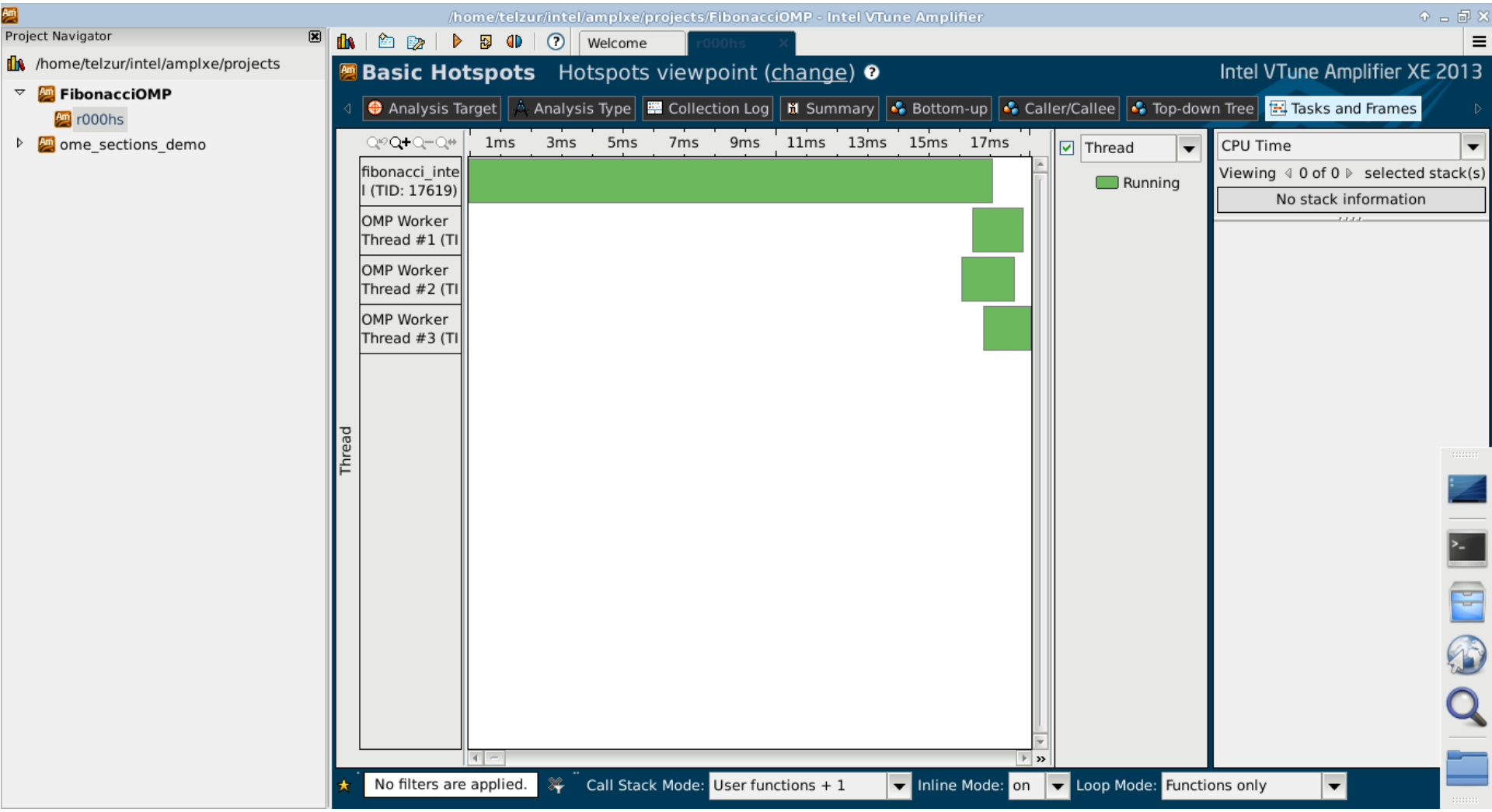
Next slides added by Guy

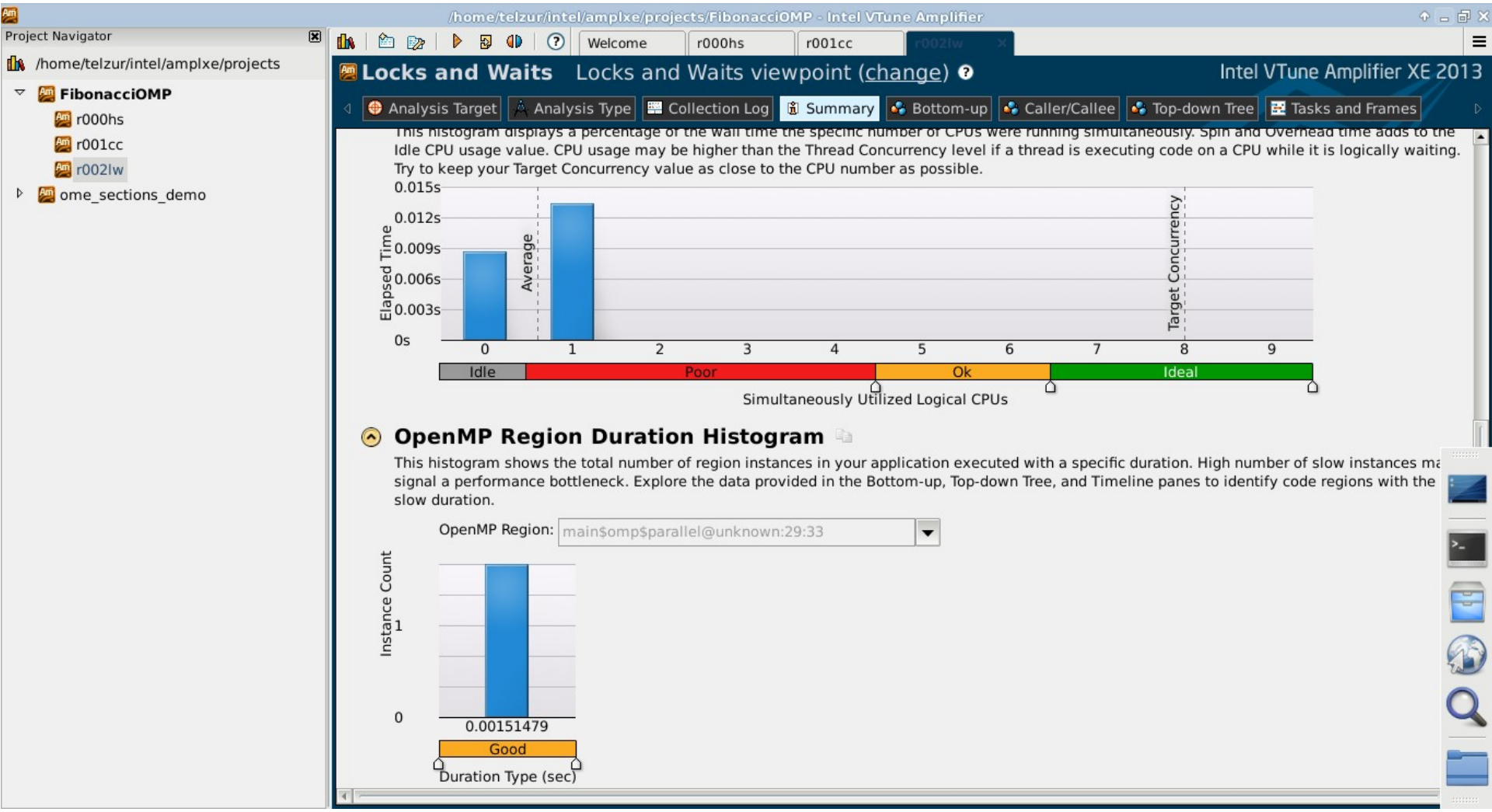
```
$ cp ./fibonacci.c  fibonacci_intel.c


$ /opt/intel/bin/icc -g -openmp  -o ./fibonacci_intel
./fibonacci_intel.c


$ /opt/intel/vtune_amplifier_xe_2013/bin64/amplxe-gui &
```

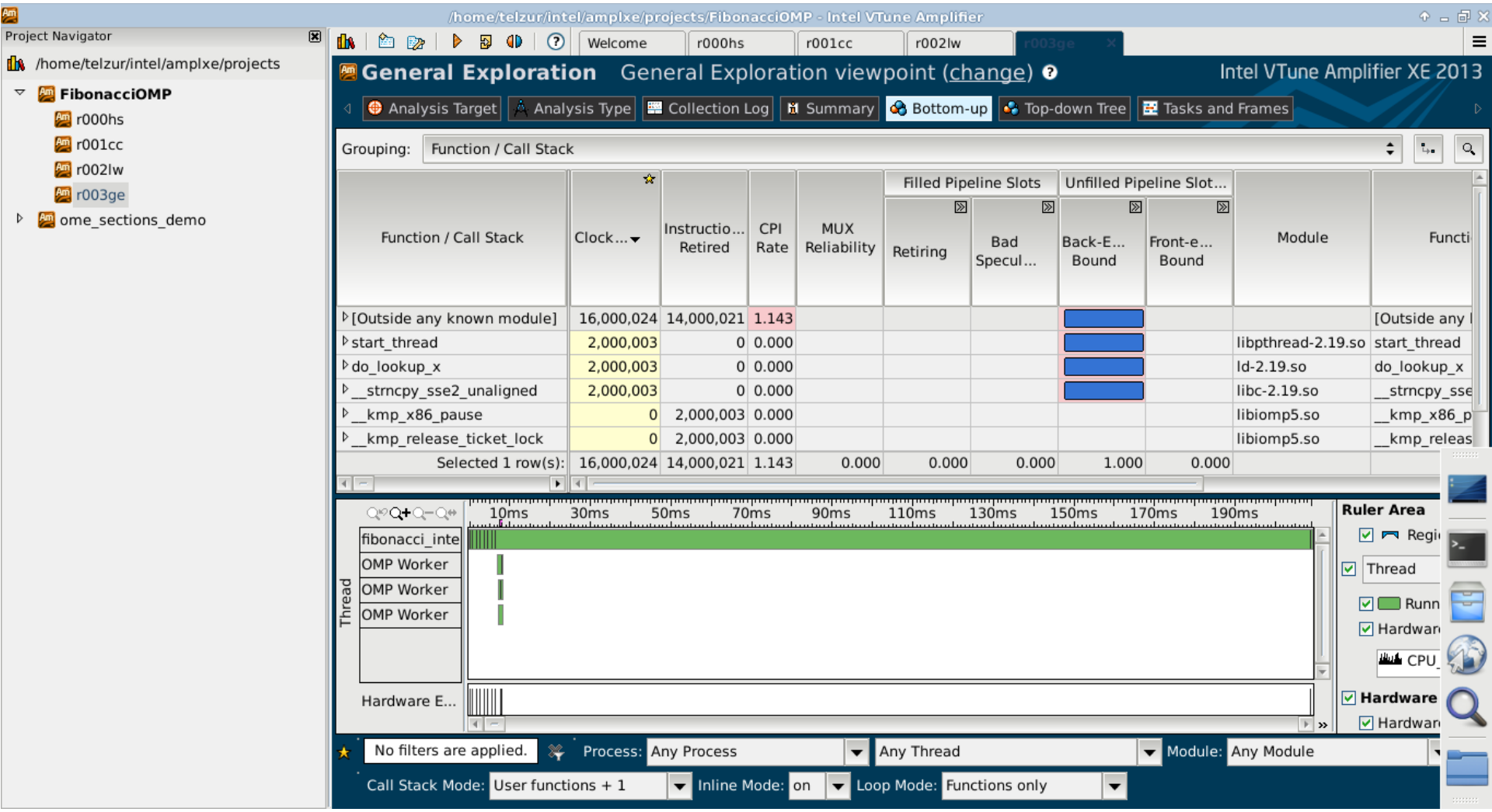

<no current project> - Intel VTune Amplifier
Project Navigator
<no current project>
Welcome
Create a Project
Project name:
FibonacciOMP
Location:
/home/telzur/intel/amplxe/projects
Browse...
Create Project
Cancel
g Started
plifier XE 2013
esults:
ome_sections_demo


<no current project> - Intel VTune Amplifier
Project Navigator
<no current project>
Welcome
FibonacciOMP - Project Properties
Target
Binary/Symbol Search
Source Search
Target system: local
Target type: Launch Application
Launch Application
Specify and configure your analysis target: an application or a script to execute. Press F1 for more details.
Application: /home/telzur/Documents/Teaching/BGU/PP/F
Browse...
Application parameters:
Modify...
Use application directory as working directory
Working directory: /home/telzur/Documents/Teaching/BGU/PP/F
Browse...
User-defined environment variables:
Modify...
Managed code profiling mode: Auto
Automatically resume collection after (sec):
Automatically stop collection after (sec):
Advanced
OK
Cancel
XE 2013


/home/telzur/intel/amplxe/projects/FibonacciOMP - Intel VTune Amplifier
Project Navigator
/home/telzur/intel/amplxe/projects
FibonacciOMP
ome_sections_demo
Welcome
Choose Analysis Type
Intel VTune Amplifier XE 2013
Analysis Type
Algorithm Analysis
Basic Hotspots
Advanced Hotspots
Concurrency
Locks and Waits
Microarchitecture Analy
General Exploration
Bandwidth
CPU Specific Analysi
Intel Core 2 Proces
Nehalem / Westme
Sandy Bridge Anal
Haswell Analysis
Knights Corner Platforn
Custom Analysis
Basic Hotspots
Copy
Identify your most time-consuming source code. This analysis type cannot be used to profile the system but must either launch an application/process or attach to one. This analysis type uses user-mode sampling and tracing collection. Press F1 for more details.
CPU sampling interval, ms: 10
Analyze user tasks
Details
Start
Start Paused
Project Properties
Command Line...


/home/telzur/intel/amplxe/projects/FibonacciOMP - Intel VTune Amplifier
Project Navigator
/home/telzur/intel/amplxe/projects
FibonacciOMP
r000hs
ome_sections_demo
Welcome
r000hs
Basic Hotspots
Hotspots viewpoint (change)
Intel VTune Amplifier XE 2013
Analysis Target
Analysis Type
Collection Log
Summary
Bottom-up
Caller/Callee
Top-down Tree
Tasks and Frames
1ms
3ms
5ms
7ms
9ms
11ms
13ms
15ms
17ms
Thread
fibonacci_inte
l (TID: 17619)
OMP Worker
Thread #1 (TI
OMP Worker
Thread #2 (TI
OMP Worker
Thread #3 (TI
Thread
Running
CPU Time
Viewing 0 of 0 selected stack(s)
No stack information
No filters are applied.
Call Stack Mode:
User functions + 1
Inline Mode:
on
Loop Mode:
Functions only


/home/telzur/intel/amplxe/projects/FibonacciOMP - Intel VTune Amplifier
Project Navigator
/home/telzur/intel/amplxe/projects
FibonacciOMP
r000hs
r001cc
r002lw
ome_sections_demo
Welcome
r000hs
r001cc
r002lw
Locks and Waits
Locks and Waits viewpoint (change)
Intel VTune Amplifier XE 2013
Analysis Target
Analysis Type
Collection Log
Summary
Bottom-up
Caller/Callee
Top-down Tree
Tasks and Frames
Idle CPU usage value. CPU usage may be higher than the Thread Concurrency level if a thread is executing code on a CPU while it is logically waiting.
Try to keep your Target Concurrency value as close to the CPU number as possible.
0.015s
0.012s
0.009s
0.006s
0.003s
0s
Elapsed Time
Average
Target Concurrency
0
1
2
3
4
5
6
7
8
9
Idle
Poor
Ok
Ideal
Simultaneously Utilized Logical CPUs
OpenMP Region Duration Histogram
This histogram shows the total number of region instances in your application executed with a specific duration. High number of slow instances ma
signal a performance bottleneck. Explore the data provided in the Bottom-up, Top-down Tree, and Timeline panes to identify code regions with the
slow duration.
OpenMP Region:
mainompparallel@unknown:29:33
Instance Count
1
0
0.00151479
Good
Duration Type (sec)


/home/telzur/intel/amplxe/projects/FibonacciOMP - Intel VTune Amplifier
Project Navigator
/home/telzur/intel/amplxe/projects
FibonacciOMP
r000hs
r001cc
r002lw
r003ge
ome_sections_demo
Welcome
r000hs
r001cc
r002lw
r003ge
General Exploration
General Exploration viewpoint (change)
Intel VTune Amplifier XE 2013
Analysis Target
Analysis Type
Collection Log
Summary
Bottom-up
Top-down Tree
Tasks and Frames
Grouping:
Function / Call Stack
Function / Call Stack
Clock...
Instructio... Retired
CPI Rate
MUX Reliability
Filled Pipeline Slots
Retiring
Bad Specul...
Unfilled Pipeline Slot...
Back-E... Bound
Front-e... Bound
Module
Functi
[Outside any known module] 16,000,024 14,000,021 1.143
start_thread 2,000,003 0 0.000 libpthread-2.19.so start_thread
do_lookup_x 2,000,003 0 0.000 ld-2.19.so do_lookup_x
__strncpy_sse2_unaligned 2,000,003 0 0.000 libc-2.19.so __strncpy_sse
__kmp_x86_pause 0 2,000,003 0.000 libiomp5.so __kmp_x86_p
__kmp_release_ticket_lock 0 2,000,003 0.000 libiomp5.so __kmp_releas
Selected 1 row(s): 16,000,024 14,000,021 1.143 0.000 0.000 0.000 1.000 0.000
10ms 30ms 50ms 70ms 90ms 110ms 130ms 150ms 170ms 190ms
Thread
fibonacci_inte
OMP Worker
OMP Worker
OMP Worker
Hardware E...
Ruler Area
Thread
Hardware
No filters are applied.
Process: Any Process
Any Thread
Module: Any Module
Call Stack Mode: User functions + 1
Inline Mode: on
Loop Mode: Functions only