Parallel Programming
OpenMP

Pham Quang Dung

Hanoi, 2012

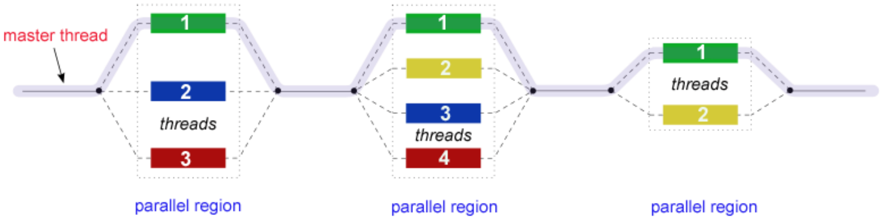
Outline

Introduction

- OpenMP is an API that may be used to explicitly direct multi-threaded shared memory parallel applications
- OpenMP has three major components :
  - Compiler directives
  - Runtime library routines
  - Environment variables
- website : openmp.org

Introduction

- Fork-Join Model
- An OpenMP begins as a single process (master thread). The master thread runs in a sequential mode until the first parallel region construct is encountered
- FORK : the master thread creates a group of parallel threads
- JOIN : When the group of threads finishes the statement in the paralle region construct, they synchronize and terminate. Only the master thread continues.



source : computing.llnl.gov/tutorials/openMP/#Introduction

Outline

OpenMP API overview

- Compiler directives appear as comments in the source code
  - When compiling without flag -fopenmp, then the directives are ignored
- OpenMP compiler directives are used for
  - Spawning a parallel region
  - Dividing blocks of code among threads
  - Distribute loop iterations between threads
  - Synchronization of work among threads

OpenMP API overview

- Run-time Library routines
  - Setting and querying the number of threads
  - Querying a thread's unique identifier (thread ID), a thread's ancestor's identifier, the thread team size
  - Setting, initializing and terminating locks
  - Querying wall clock time
  - ...

Outline

1. Introduction
2. OpenMP API overview
3. Compiling OpenMP programs
4. OpenMP directives

Compiling OpenMP programs

- Ubuntu : g++ with flag -fopenmp
- Example : g++ -o openmp-helloworld openmp-helloworld.cpp -fopenmp

Outline

OpenMP directives

- Directives format : #**pragma omp directive-name [clause,...] newline**
- Example : #**pragma omp parallel private(i)**
- General rules :
    - Case sensitive
    - Only one directive-name per directive
    - Each directive applies to at most one succeeding statement (structured block)

Parallel constructs

- Creates a team of threads and becomes the master thread of the team
- The code of the parallel region is duplicated and all threads will execute that code
- An implied barrier at the end of the parallel region
- Only the master thread continues the execution after this point
- If any thread terminates within the parallel region, then all threads of the team terminate

Parallel constructs

- syntax : #pragma omp parallel [clause ...] newline
- clause can be :
  - if (scalar_expression)
  - private (list)
  - shared (list)
  - default (shared || none)
  - firstprivate (list)
  - reduction (operator : list)
  - copyin (list)
  - num_threads (integer-expression)

Parallel constructs

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

main(int argc, char** argv){
  int nbthreads, tid;
  nbthreads = atoi(argv[1]);

  /*fork a team of threads with each thread having a private tid
      variable*/
  #pragma omp parallel if(nbthreads >= 4) num_threads(nbthreads)
  {
  tid = omp_get_thread_num();
  printf("Hello world from thread %d\n",tid);
  }
}
```

Parallel constructs

- Compile : g++ -o openMP-parallel-region-construct openMP-parallel-region-construct.cpp -fopenmp

Example

- Run : ./openMP-parallel-region-construct 3
- Output : Hello world from thread 0

Example

- Run : ./openMP-parallel-region-construct 6
- Output :

  Hello world from thread 3
  Hello world from thread 2
  Hello world from thread 0
  Hello world from thread 1
  Hello world from thread 4
  Hello world from thread 5

Parallel constructs

```c
#include "omp.h"
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

main(int argc, char** argv){
  int nbthreads,tid;
  nbthreads = atoi(argv[1]);
  int x = 123;
  int a = 456;

  /*fork a team of threads with each thread having a private tid
     variable*/
  #pragma omp parallel num_threads(nbthreads) private(tid,a)
  {
    tid = omp_get_thread_num();
    printf("Hello world from thread %d x = %d a = %d\n",tid,x,a);

    x = tid * 10;
    sleep(3);
    printf("thread %d has x = %d\n",tid,x);
  }
}
```

Parallel constructs

```
Hello world from thread 0 x = 123 a = 0
Hello world from thread 2 x = 123 a = 0
Hello world from thread 1 x = 123 a = 32634
thread 0 has x = 10
thread 2 has x = 10
thread 1 has x = 10
```

Parallel constructs

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

main(int argc, char** argv){
  int nbthreads,tid;
  nbthreads = atoi(argv[1]);
  int x = 123;
  int a = 456;

  /*fork a team of threads with each thread having a private tid
      variable*/
 #pragma omp parallel num_threads(nbthreads) private(tid,a,x)
  {
    tid = omp_get_thread_num();
    printf("Hello world from thread %d x = %d a = %d\n",tid,x,a);

    x = tid * 10;
    sleep(3);
    printf("thread %d has x = %d\n",tid,x);
  }
}
```

Parallel constructs

```
Hello world from thread 2 x = -495646976 a = 32693
Hello world from thread 0 x = 0 a = 0
Hello world from thread 1 x = 0 a = 0
thread 2 has x = 20
thread 0 has x = 0
thread 1 has x = 10
```

Parallel constructs

- firstprivate(var-list) : variables are private and are initialized with the value of the shared copy before the parallel region
- lastprivate(var-list) : variable are private and the value of the thread executing the last iteration of a parallel loop in sequential order to the variable outside of the parallel region

Parallel constructs

```c
#include "omp.h"
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

main(int argc, char** argv){
  int nbthreads,tid;
  nbthreads = atoi(argv[1]);
  int x = 123;
  int a = 456;

  /*fork a team of threads with each thread having a private tid
     variable*/
  #pragma omp parallel num_threads(nbthreads) firstprivate(tid,a,x)
  {
    tid = omp_get_thread_num();
    printf("Hello world from thread %d x = %d a = %d\n",tid,x,a);

    x = tid * 10;
    sleep(3);
    printf("thread %d has x = %d\n",tid,x);
  }
}
```

Parallel constructs

```
Hello world from thread 0 x = 123 a = 456
Hello world from thread 2 x = 123 a = 456
Hello world from thread 1 x = 123 a = 456
thread 0 has x = 0
thread 2 has x = 20
thread 1 has x = 10
```

Work-Sharing constructs

- Divide the execution of the enclosed code region among the members of the team that encounter it
- Do not launch new threads
- No implied barrier upon the entry to a work-sharing construct
- An implied barrier at the end of the work-sharing construct
- Type
  - DO/FOR (Data parallelism)
  - SECTION (Functional parallelism)
  - SINGLE

Work-Sharing constructs : DO-FOR

- Specifies that the iterations of the loop immediately following it must be executed in parallel by the team
- Assumes that a parallel region has already been initiated

```
#pragma omp for [clause ...]  newline
                schedule (type [,chunk])
                ordered
                private (list)
                firstprivate (list)
                lastprivate (list)
                shared (list)
                reduction (operator: list)
                collapse (n)
                nowait

   for_loop
```

Work-Sharing constructs : DO-FOR

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

main(int argc, char** argv){
  int nbthreads,tid,i;
  nbthreads = 2;
  int N = 3*nbthreads;

  /*fork a team of threads with each thread having a private tid
      variable*/
  #pragma omp parallel num_threads(nbthreads) private(tid)
  {
    #pragma omp for
    for(i = 0; i < N; i++){
      tid = omp_get_thread_num();
      printf("Hello world from thread %d i = %d\n",tid,i);
    }
  }
}
```

Work-Sharing constructs : DO-FOR

```
Hello world from thread 0 i = 0
Hello world from thread 0 i = 1
Hello world from thread 0 i = 2
Hello world from thread 1 i = 3
Hello world from thread 1 i = 4
Hello world from thread 1 i = 5
```

Work-Sharing constructs : DO-FOR (scheduling strategies)

- Schedule clause : schedule(type[,size])
- Scheduling types
  - static : chunks of the specified size are assigned in a round robin fashion to the threads
  - dynamic : the iterations are broken into chunks of speficied size. When a thread finishes the execution of a chunk, the next chunk is assigned to that thread
  - guided : similar to dynamic, the size of the chunks is exponentially decreasing. The size parameter specifies the smallest chunk. The initial chunk is implementation depedent
  - runtime : the scheduling and the size of chunks are determined via environment variables

Work-Sharing constructs : DO-FOR (scheduling strategies)

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

main(int argc, char** argv){
  int nbthreads,tid,i;
  nbthreads = 3;//atoi(argv[1]);
  int N = 3*nbthreads;
  #pragma omp parallel num_threads(nbthreads) private(tid)
  {
    #pragma omp for schedule(dynamic, 4)
    for(i = 0; i < N; i++){
      tid = omp_get_thread_num();
      printf("Hello world schedule(dynamic,4) from thread %d i = %d
          \n",tid,i);
    }

    #pragma omp for schedule(guided)
    for(i = 0; i < N; i++){
      tid = omp_get_thread_num();
      printf("Hello world schedule(guide) from thread %d i = %d\n",
          tid,i);
    }
  }
```

Work-Sharing constructs : DO-FOR (scheduling strategies)

```
Hello world schedule(dynamic,4) from thread 2 i = 0
Hello world schedule(dynamic,4) from thread 2 i = 1
Hello world schedule(dynamic,4) from thread 2 i = 2
Hello world schedule(dynamic,4) from thread 2 i = 3
Hello world schedule(dynamic,4) from thread 1 i = 8
Hello world schedule(dynamic,4) from thread 0 i = 4
Hello world schedule(dynamic,4) from thread 0 i = 5
Hello world schedule(dynamic,4) from thread 0 i = 6
Hello world schedule(dynamic,4) from thread 0 i = 7
Hello world schedule(guide) from thread 0 i = 0
Hello world schedule(guide) from thread 0 i = 1
Hello world schedule(guide) from thread 0 i = 2
Hello world schedule(guide) from thread 0 i = 7
Hello world schedule(guide) from thread 0 i = 8
Hello world schedule(guide) from thread 2 i = 3
Hello world schedule(guide) from thread 2 i = 4
Hello world schedule(guide) from thread 1 i = 5
Hello world schedule(guide) from thread 1 i = 6
```

Work-Sharing constructs : DO-FOR (firstprivate vs. lastprivate)

```c
#include "omp.h"
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

main(int argc, char** argv){
  int nbthreads,tid,i;
  nbthreads = atoi(argv[1]);
  int x = 123;
  int a = 456;
  #pragma omp parallel num_threads(nbthreads){
    #pragma omp for lastprivate(tid,x,i) firstprivate(a)
    for(i = 0; i < nbthreads; i++){
      tid = omp_get_thread_num();
      printf("Hello world from thread %d x = %d a = %d\n",tid,x,a);
      x = tid * 10; a = tid * 10;
      sleep(1);
      printf("thread %d has x = %d a = %d\n",tid,x,a);
    }
  }
  printf("tid = %d x = %d a = %d\n",tid,x,a);
}
```

Work-Sharing constructs : DO-FOR

```
Hello world from thread 4 x = 0 a = 456
Hello world from thread 5 x = 0 a = 456
Hello world from thread 3 x = 0 a = 456
Hello world from thread 1 x = 32606 a = 456
Hello world from thread 0 x = 32767 a = 456
Hello world from thread 2 x = 0 a = 456
thread 4 has x = 40 a = 40
thread 1 has x = 10 a = 10
thread 0 has x = 0 a = 0
thread 2 has x = 20 a = 20
thread 3 has x = 30 a = 30
thread 5 has x = 50 a = 50
tid = 5 x = 50 a = 456
```

Work-Sharing constructs : SECTION

- Non-iterative work-sharing construct
- Specify that the enclosed sections of code are divided among threads in the team
- Each SECTION is executed once by a thread in the team
- Different sections may be executed by different threads
- It is possible that one thread executes more than one section
- There is an implied barrier at the end of a SECTION directive unless a nowait clause is specified
- Restriction : It is illegal to branch (goto) into or out of sections blocks

Work-Sharing constructs : SECTION

```
#pragma omp sections [clause ...]  newline
                     private (list)
                     firstprivate (list)
                     lastprivate (list)
                     reduction (operator: list)
                     nowait
  {

  #pragma omp section   newline

     structured_block

  #pragma omp section   newline

     structured_block

  }
```

Work-Sharing constructs : SECTION

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

main(int argc, char** argv){
  int tid,i,S,P;
  #pragma omp parallel num_threads(2) private(tid,i,S,P)
  {
    tid = omp_get_thread_num();
    #pragma omp sections nowait
    {
      #pragma omp section
      {
        S = 0;
        for(i = 1; i <= 5; i++) S = S + i;
        printf("Thread %d compute sum S = %d\n",tid,S);
      }
      #pragma omp section
      {
        P = 1;
        for(i = 1; i <= 5; i++) P = P * i;
        printf("Thread %d compute sum P = %d\n",tid,P);
      }
    }
  }
```

Work-Sharing constructs : SECTION

```
Thread 0 compute sum P = 120
Thread 1 compute sum S = 15
```

Work-Sharing constructs : SINGLE

- The SINGLE directive specifies that the enclosed code is executed by only one thread in the team
- Useful when dealing with sections of code that are not thread safe (e.g., I/O)
- Threads in the team that do not execute the SINGLE directive wait at the end of the enclosed code block, unless a nowait clause is specified
- Restriction : It is illegal to branch into or out of a SINGLE block

Work-Sharing constructs : SINGLE

- syntax : #pragma omp single [clause ...] newline structured_block
- clause can be :
  - private (list)
  - firstprivate (list)
  - nowait

Work-Sharing constructs : SINGLE

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

main(int argc, char** argv){
  int tid;
  #pragma omp parallel num_threads(3) private(tid)
  {
    tid = omp_get_thread_num();
    printf("Thread %d created\n",tid);
    #pragma omp single nowait
    {
      printf("Thread %d execute single section\n",tid);
    }
  }
}
```

Work-Sharing constructs : SINGLE

```
Thread 1 created
Thread 1 executes single section
Thread 2 created
Thread 0 created
```

Combined Parallel Work-sharing constructs

```
#include "omp.h"
#include <stdio.h>
#define N 10
#define CHUNK 4

main(int argc, char** argv){
  int tid, i, chunk;
  chunk = CHUNK;
  float a[N], b[N], c[N];
  for(i = 0; i < N; i++)
    a[i] = b[i] = i*1.0;

  #pragma omp parallel for \
    shared(a,b,c,chunk) private(i,tid) \
      schedule(static, chunk)
  for(i = 0; i < N; i++){
    c[i] = a[i] + b[i];
    tid = omp_get_thread_num();
    printf("thread %d computes c[%d] = %f\n",tid,i,c[i]);
  }
}
```

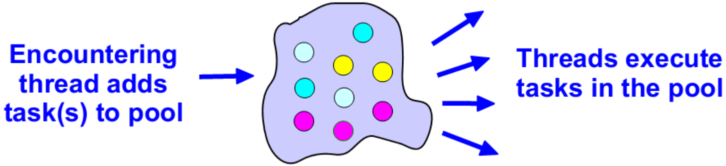
Combined Parallel Work-sharing constructs

```
thread 2 computes c[8] = 16.000000
thread 2 computes c[9] = 18.000000
thread 1 computes c[4] = 8.000000
thread 1 computes c[5] = 10.000000
thread 1 computes c[6] = 12.000000
thread 1 computes c[7] = 14.000000
thread 0 computes c[0] = 0.000000
thread 0 computes c[1] = 2.000000
thread 0 computes c[2] = 4.000000
thread 0 computes c[3] = 6.000000
```

TASK construct

- TASK construct defines an explicit task
  - may be executed by the encoutering thread, OR
  - deferred for execution by another thread in the team
- A Task
  - A specific instance of executable code and its data environment, generated when a thread encounters a taks construct or a parallel construct
  - When a thread executes a task, it produces a task region
- Task region
  - A region consists of all code encountered during the execution of a task
  - A parallel region consists of one or more implicit task regions
- Explicit task : A task generated when a task construct is encountered during the execution

TASK construct



source : http ://openmp.org/wp/resources/

- Developers specify tasks in applications and run-time system executes taks

TASK construct

```
int main(int argc, char** argv){
  int nbT = atoi(argv[1]);
  #pragma omp parallel num_threads(nbT)
  {
    #pragma omp single
    {
      #pragma omp task
      printf("hello ");

      #pragma omp task
      printf("world ");

      #pragma omp taskwait
      printf("\nthank you!");
    }
  }
  printf("\n");
}
```

TASK construct

Example

```
world hello
thank you!
```

Example

```
hello world
thank you!
```

TASK construct

```
void process(int tid, int i){
  printf("Process(%d,%d)\n",tid,i);
  sleep(4-3*tid);
}
int main(int argc, char** argv){
  int nbT = atoi(argv[1]);
  int i = 1;
  int N = 20;
  #pragma omp parallel num_threads(nbT)
  {
    #pragma omp single nowait
    {
      while(i < N){
       #pragma omp task firstprivate(i)
       {
         int tid = omp_get_thread_num();
         process(tid,i);
       }
       i = i+1;
      }
    }
  }
}
```

TASK construct

Run with 2 threads

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

Sychronization constructs

Directives

- Critical : identifies that a section of code must be executed by a single thread at a time
- Barrier : specifies a synchronization point at which threads in a parallel region will wait until all other threads in that section reach the same point
  - Statement execution past the omp barrier point then continues in parallel
- Taskwait : specifies a wait on the completion of child tasks generated by the current task

Sychronization constructs - Critical section

```
1  int main(int argc, char** argv){
     int N = atoi(argv[1]);
3    int nbT = atoi(argv[2]);
     int T = 0;
5    #pragma omp parallel
     {
7      #pragma omp for
       for(int i = 1; i <= 20; i++){
9        #pragma omp critical
         {
11         T = T + i;
         }
13     }
     }
15   printf("T = %d\n",T);
   }
```

What happen if we remove line 9 ?

Sychronization constructs - barrier

Code without barrier

```
int main(int argc, char** argv){
  int nbT = atoi(argv[1]);
  int tid;
  #pragma omp parallel num_threads(nbT) private(tid)
  {
    tid = omp_get_thread_num();
    double st = omp_get_wtime();
    for(int i = 0; i < 5*(tid+1); i++){
       printf("T %d, i = %d\n",tid,i);
       sleep(1);
    }
    printf("Thread %d finished at %lf\n",tid,omp_get_wtime()-st);
  }
}
```

Sychronization constructs - barrier

```
T 0, i = 0
T 1, i = 0
T 0, i = 1
T 1, i = 1
T 0, i = 2
T 1, i = 2
T 0, i = 3
T 1, i = 3
T 0, i = 4
T 1, i = 4
 Thread 0 finished at 5.000866
T 1, i = 5
T 1, i = 6
T 1, i = 7
T 1, i = 8
T 1, i = 9
 Thread 1 finished at 10.001761
```

Sychronization constructs - barrier

Code with barrier

```
int main(int argc, char** argv){
  int nbT = atoi(argv[1]);
  int tid;
  #pragma omp parallel num_threads(nbT) private(tid)
  {
    tid = omp_get_thread_num();
    double st = omp_get_wtime();
    for(int i = 0; i < 5*(tid+1); i++){
      printf("T %d, i = %d\n",tid,i);
      sleep(1);
    }
    #pragma omp barrier
    printf("Thread %d finished at %lf\n",tid,omp_get_wtime()-st);
  }
}
```

Sychronization constructs - barrier

```
T 0, i = 0
T 1, i = 0
T 0, i = 1
T 1, i = 1
T 0, i = 2
T 1, i = 2
T 0, i = 3
T 1, i = 3
T 0, i = 4
T 1, i = 4
T 1, i = 5
T 1, i = 6
T 1, i = 7
T 1, i = 8
T 1, i = 9
 Thread 1 finished at 10.001736
 Thread 0 finished at 10.001799
```

THREADPRIVATE directive

- #**pragma omp threadprivate(***var_list***)**
- Make global file scope variables (C,C++) local and persistent to a thread throught the execution of multiple parallel regions
- The directive must appear after the declaration of listed variables
- Each thread get its own local location for variables listed
- On the entry of a parallel region, data in THREADPRIVATE variables are undefined, unless a COPYIN clause is specified in the PARALLEL directive
- Data in THREADPRIVATE objects is guaranteed to persist only if the dynamic threads mechanism is "turned off" and the number of threads in different parallel regions remains constant

THREADPRIVATE directive

```c
int  a, b, tid;
float x;
#pragma omp threadprivate(a, x)
int main(int argc, char** argv)  {
  omp_set_dynamic(0);
  a = 123; x = 456;
  printf("Serial a = %d, x = %f\n",a,x);
  printf("1st Parallel Region:\n");
  #pragma omp parallel private(b,tid) num_threads(3) copyin(a)
  {
    tid = omp_get_thread_num();
    printf("at the entry of parallel region of thread %d a= %d, x = %f\n",tid,a,x);
    a = tid; b = tid; x = 10*tid;
    printf("Thread %d:   a,b,x= %d %d %f\n",tid,a,b,x);
  }

  printf("2nd Parallel Region:\n");
  #pragma omp parallel private(tid) num_threads(4)
  {
    tid = omp_get_thread_num();
    printf("Thread %d:   a,b,x= %d %d %f\n",tid,a,b,x);
  }
}
```

THREADPRIVATE directive

```
Serial a = 123, x = 456.000000
1st Parallel Region:
at the entry of parallel region of thread 1 a= 123, x = 0.000000
Thread 1:   a,b,x = 1 1 10.000000
at the entry of parallel region of thread 0 a= 123, x = 456.000000
Thread 0:   a,b,x = 0 0 0.000000
at the entry of parallel region of thread 2 a= 123, x = 0.000000
Thread 2:   a,b,x = 2 2 20.000000
2nd Parallel Region:
Thread 2:   a,b,x = 2 0 20.000000
Thread 1:   a,b,x = 1 0 10.000000
Thread 0:   a,b,x = 0 0 0.000000
Thread 3:   a,b,x = 0 0 0.000000
```

REDUCTION clause

#pragma omp reduction(+ : *var_list***)**

- The REDUCTION clause performs a reduction on the variables appearing in the list
  - A private copy for each variables listed is created for each thread
  - At the end of the reduction, the reduction is applied to all private copies of the shared variables, and the final result is written to the global shared variable

REDUCTION clause

```c
int main(int argc, char** argv){
  int N = atoi(argv[1]);
  int nbT = atoi(argv[2]);

  int T = 0;
 #pragma omp parallel for reduction(+:T)
  for(int i = 1; i <= N; i++)
  {
    int tid = omp_get_thread_num();
    T = T + i;
    printf("Thread %d has i = %d compute local T = %d\n",tid,i,T);
  }
  printf("T = %d\n",T);
}
```

REDUCTION clause

```
Thread 0 has i = 1 compute local T = 1
Thread 0 has i = 2 compute local T = 3
Thread 0 has i = 3 compute local T = 6
Thread 1 has i = 4 compute local T = 4
Thread 1 has i = 5 compute local T = 9
Thread 1 has i = 6 compute local T = 15
Thread 2 has i = 7 compute local T = 7
Thread 2 has i = 8 compute local T = 15
Thread 2 has i = 9 compute local T = 24
Thread 3 has i = 10 compute local T = 10
T = 55
```

REDUCTION clause

```
int main(int argc, char** argv){
  int N = atoi(argv[1]);
  int nbT = atoi(argv[2]);

  int T = 1;
 #pragma omp parallel for reduction(*:T)
  for(int i = 1; i <= N; i++)
  {
    int tid = omp_get_thread_num();
    T = T + i;
    printf("Thread %d has i = %d compute local T = %d\n",tid,i,T);
  }
  printf("T = %d\n",T);
}
```

REDUCTION clause

```
Thread 0 has i = 1 compute local T = 2
Thread 0 has i = 2 compute local T = 4
Thread 0 has i = 3 compute local T = 7
Thread 2 has i = 7 compute local T = 8
Thread 2 has i = 8 compute local T = 16
Thread 2 has i = 9 compute local T = 25
Thread 3 has i = 10 compute local T = 11
Thread 1 has i = 4 compute local T = 5
Thread 1 has i = 5 compute local T = 10
Thread 1 has i = 6 compute local T = 16
T = 30800
```