HPCSE - II

«OpenMP Programming Model - Tasks»

Panos Hadjidoukas

Outline

- Recap of OpenMP
  - nested loop parallelism
  - functional parallelism
- OpenMP tasking model
  - how to use
  - how it works
  - examples

Nested Loop Parallelization - I

```
void work(int i, int j);

void nesting(int n)
{
  #pragma omp parallel
  {
    #pragma omp for
    for (int i=0; i<n; i++) {
      #pragma omp parallel
      {
        #pragma omp for
        for (int j=0; j<n; j++) {
          work(i, j);
        }
      }
    }
  }
}
```

several implicit barriers

Nested Loop Parallelization - II

```
void work(int i, int j);

void nesting(int n)
{
  #pragma omp parallel for
  for (int i=0; i<n; i++) {
    #pragma omp parallel for
    for (int j=0; j<n; j++) {
      work(i, j);
    }
  }
}
```

nested parallel regions

we avoided some implicit barriers

Nested Loop Parallelization - III

```
void work(int i, int j);

void nesting(int n)
{
  #pragma omp parallel for
  for (int k=0; k<n*n; k++) {
    int i = k / n;
    int j = k % n;
    work(i, j);
  }
}
```

loop fusion: we avoided nested parallelism

Nested Loop Parallelization - IV

```
void work(int i, int j);

void nesting(int n)
{
  #pragma omp parallel for collapse(2)
  for (int i=0; i<n; i++) {
    for (int j=0; j<n; j++) {
     work(i, j);
    }
  }
}
```

collapse clause: let the OpenMP compiler do it for us

Functional parallelism

- Parallelize the following sequential code
  - what is the total execution time if each function takes one second?

```
V = alpha();
W = beta();
X = gamma(V, W);
Y = delta();
printf("%f\n", epsilon(X,Y));
```

total time = 5s

Functional parallelism - Solution 1

```
#pragma omp parallel num_threads(3)
#pragma omg sections
{
     #pragma omp section
     V = alpha();

     #pragma omp section
     W = beta();

     #pragma omp section
     Y = delta();
}
X = gamma(V, W);
printf("%f\n", epsilon(X,Y));
```

no sense to use more threads

total time = 3s

Functional parallelism - Solution 2

```
#pragma omp parallel num_threads(2)
{
   #pragma omp sections
   {
       #pragma omp section
       V = alpha();

       #pragma omp section
       W = beta();
   }
   #pragma omp sections
   {
       #pragma omp section
       X = gamma(V, W);

       #pragma omp section
       Y = delta();
   }
}
printf("%f\n", epsilon(X,Y));
```

no sense to use more threads

implicit barrier

total time = 3s
but with fewer threads

Functional Parallelism I

- Implement an equivalent version of the following code without using parallel sections

```
void XAXIS();
void YAXIS();
void ZAXIS();

void a9()
{
   #pragma omp parallel
   {
       #pragma omp section
              XAXIS();
       #pragma omp section
              YAXIS();
       #pragma omp section
              ZAXIS();
   }
}
```

Functional Parallelism II

```
void XAXIS();
void YAXIS();
void ZAXIS();

void a9()
{
   #pragma omp parallel for
   for (int i = 0; i < 3; i++)
      if (i == 0) XAXIS();
      if (i == 1) YAXIS();
      if (i == 2) YAXIS();
   }
}
```

Functional Parallelism III

```
void XAXIS();
void YAXIS();
void ZAXIS();

void a9()
{
   #pragma omp parallel
   {
     #pragma omp single nowait
     XAXIS();

     #pragma omp single nowait
     YAXIS();

     #pragma omp single nowait
     ZAXIS();
   }
}
```

Tasks in OpenMP (3.0)

- We have seen a few ways to parallelize a block
  - `#pragma omp parallel`
  - `#pragma omp sections`
  - `#pragma omp parallel for`
- "`parallel for`" is great for for-loops, but what about unstructured data?
  - Traversal through lists and trees?
  - while loops?
- Spawning threads dynamically is expensive
- Tasks are more lightweight:
  - new tasks get put onto a task queue
  - idle threads pull tasks from the queue

OpenMP Tasks

- Parallelization of irregular problems
  - Loop with dynamic bounds
  - Recursive algorithms
  - Producer-consumer execution schemes
- Work units that are executed asynchronously
  - They can be executed immediately after their creation
- Tasks consist of:
  - Code
  - Data environment: initialized at creation time
  - Internal control variables (ICVs)

Example: Fibonacci sequence

- Parallelize recursive function $F_n = F_{n-1} + F_{n-2}$
- The sequential code first

```
#include <iostream>

int fibonacci(int n)
{
   int i, j;
   if (n<2)
      return n;
   else {
      i = fibonacci(n-1);
      j = fibonacci(n-2);
      return i + j;
   }
}

int main()
{
   int n;
   std::cin >> n;
   std::cout << fibonacci(n) << std::endl;
}
```

Example: Fibonacci sequence

- Parallelize recursive function $F_n = F_{n-1} + F_{n-2}$
- First attempt using sections

```
#include <iostream>

int fibonacci(int n)
{
   int i, j;
   if (n<2)
      return n;
   else {
      #pragma omp parallel sections shared (i,j)
      {
         #pragma omp section
         i = fibonacci(n-1);
         #pragma omp section
         j = fibonacci(n-2);
      }
      return i + j;
   }
}
```

```
int main()
{
   int n;
   std::cin >> n;
   std::cout << fibonacci(n) << std::endl;
}
```

Requirement: `export OMP_NESTED=TRUE`

Problem: uncontrolled spawning of expensive threads

The task directive

- Spawns tasks and puts them into a queue for the threads to work on:

```
#pragma omp task [clause …]\
```

if (scalar_expression)	Only parallelize if the expression is true. Can be used to stop parallelization if the work is too little
private (list)	The specified variables are thread-private
shared (list)	The specified variables are shared among all threads
default (shared I none)	Unspecified variables are shared or not
firstprivate (list)	Initialize private variables from the master thread
mergeable	If specified allows the task to be merged with others
untied	If specified allows the task to be resumed by other threads after suspension. Helps prevent starvation but has unusual memory semantics: after moving to a new thread all private variables are that of the new thread
final (scalar_expression)	If the expression is true this has to be the final task. All dependent tasks are included into it

Data Environment

- Possible options
  - `shared`(list)
  - `private`(list)
  - `firstprivate`(list)
    - The values of variables at task creation
  - `default(shared|none)`
- If not specified, the default rules apply:
  - Global variables are shared
  - Otherwise
    - `firstprivate`
    - `shared`, if defined lexically as such

Example: Data Environment

```
int a ;
void foo() {
 int b,c ;
 #pragma omp parallel shared(c) private(b)
  {
      int d ;
      #pragma omp task
      {
         int e ;
         a = shared
         b = firstprivate
         c = shared
         d = firstprivate
         e = private
      }
  }
}
```

Example: Fibonacci sequence

- Parallelize recursive function $F_n = F_{n-1} + F_{n-2}$
- First attempt using tasks

```
#include <iostream>

int fibonacci(int n)
{
   int i, j;
   if (n<2)
      return n;
   else {
      #pragma omp task shared(i) firstprivate(n)
      i = fibonacci(n-1);

      #pragma omp task shared(j) firstprivate(n)
      j = fibonacci(n-2);

      return i + j;
   }
}
```

```
int main()
{
   int n;
   std::cin >> n;
   std::cout << fibonacci(n) << std::endl;
}
```

Problem 1: no parallel region

Example: Fibonacci sequence

- Parallelize recursive function $F_n = F_{n-1} + F_{n-2}$
- Second attempt using tasks

```
#include <iostream>

int fibonacci(int n)
{
   int i, j;
   if (n<2)
      return n;
   else {
      #pragma omp task shared(i) firstprivate(n)
      i = fibonacci(n-1);

      #pragma omp task shared(j) firstprivate(n)
      j = fibonacci(n-2);

      return i + j;
   }
}
```

```
int main()
{
   int n;
   std::cin >> n;

   #pragma omp parallel shared(n)
   {
      std::cout << fibonacci(n) << std::endl;
   }
}
```

Problem 2: now we have too many calls to fibonacci

Example: Fibonacci sequence

- Parallelize recursive function $F_n = F_{n-1} + F_{n-2}$
- Third attempt using tasks

```
#include <iostream>

int fibonacci(int n)
{
   int i, j;
   if (n<2)
      return n;
   else {
      #pragma omp task shared(i) firstprivate(n)
      i = fibonacci(n-1);

      #pragma omp task shared(j) firstprivate(n)
      j = fibonacci(n-2);

      return i + j;
   }
}
```

```
int main()
{
   int n;
   std::cin >> n;

   #pragma omp parallel shared(n)
   {
     #pragma omp single
      std::cout << fibonacci(n) << std::endl;
   }
}
```

Problem 3: i and j get added before the tasks are done
Problem 4: when i and j are written the variables no longer exist

Example: Fibonacci sequence

- Parallelize recursive function $F_n = F_{n-1} + F_{n-2}$
- Fourth attempt using tasks

```
#include <iostream>

int fibonacci(int n)
{
   int i, j;
   if (n<2)
      return n;
   else {
      #pragma omp task shared(i) firstprivate(n)
      i = fibonacci(n-1);

      #pragma omp task shared(j) firstprivate(n)
      j = fibonacci(n-2);

      #pragma omp taskwait
      return i + j;
   }
}
```

```
int main()
{
   int n;
   std::cin >> n;

   #pragma omp parallel shared(n)
   {
     #pragma omp single
      std::cout << fibonacci(n) << std::endl;
   }
}
```

Now it works

Using the final clause

- Parallelize recursive function $F_n = F_{n-1} + F_{n-2}$
- Fifth attempt using tasks

```
#include <iostream>

int fibonacci(int n)
{
   int i, j;
   if (n<2)
      return n;
   else {
      if (n>5)
       {
      #pragma omp task shared(i) firstprivate(n) untied final(n<=5)
      i = fibonacci(n-1);

      #pragma omp task shared(j) firstprivate(n) untied final(n<=5)
      j = fibonacci(n-2);
      }
      else {
      i = fibonacci(n-1);
      j = fibonacci(n-2);

      }

      #pragma omp taskwait
      return i + j;
   }
}
```

Now it will not spawn tasks for n<=5

if and final clauses

- Used for optimization, e.g. avoid creation of small tasks
- If the expression of an if clause on a Task evaluates to false
  - The encountering Task is suspended
  - The new Task is executed immediately
  - The parent Task resumes when the new Task finishes
- If the expression of a final clause on a Task evaluates to true
  - All child tasks will be final and included, that means they will be executed sequentially in the task region, immediately by the encountering thread

Source: Christian Terboven. OpenMP Performance Tuning.

Refinement I

- Parallelize recursive function $F_n = F_{n-1} + F_{n-2}$

- Final refinement

```
int main()
{
   int n;
   std::cin >> n;

   #pragma omp parallel shared(n)
   {
     #pragma omp single  nowait
      std::cout << fibonacci(n) << std::endl;
   }
}
```

Avoid the extra (implicit) barrier.
Are we done?

Refinement II

- Parallelize recursive function $F_n = F_{n-1} + F_{n-2}$
- Final refinement

```
#include <iostream>

int fibonacci(int n)
{
   int i, j;
   if (n<2)
      return n;
   else {
      #pragma omp task shared(i) firstprivate(n) untied final(n<=5)
      i = fibonacci(n-1);


      j = fibonacci(n-2);

      #pragma omp taskwait
      return i + j;
   }
}
```

Task-related directives and functions

- Wait for all dependent tasks:

```
#pragma omp taskwait
```

- Yield the thread to another task

```
#pragma omp taskyield
```

- Check at runtime whether this is a final task

int omp_in_final()	Returns true if the task is a final task

Example: Tree Traversal

```
void traverse(Tree *tree)
{
      if (tree->left)
            traverse(tree->left) ;

      if (tree->right)
            traverse(tree->right);

      process(tree) ;
}
```

Source: Christian Terboven. OpenMP Performance Tuning.

Example: Tree Traversal

```
void traverse(Tree * tree)
{
      #pragma omp parallel sections
      {
        #pragma omp section
        if (tree->left)
           traverse(tree->left);

        #pragma omp section
        if (tree->right)
           traverse(tree->right);
      }
      process(tree);
}
```

Example: Tree Traversal

```
void traverse(Tree* tree)
{
        #pragma omp task
        if (tree->left)
           traverse(tree->left);

        #pragma omp task
        if (tree->right)
           traverse(tree->right);

     process(tree);
}
```

Assume a Parallel region to exist outside the scope of this routine

Example: List Traversal

```
void traverse_list(List l)
{
Element e ;

#pragma omp parallel private(e)
for (e=l->first; e; e=e->next)
        #pragma omp single nowait
        process(e);
}
```

Example: List Traversal

```
void traverse_list(List l)
{
Element e ;

for (e=l->first; e; e=e->next)
        #pragma omp task
        process(e); /* firstprivate */

 /* ... */
}
```

Example: List Traversal

```
void traverse_list(List l)
{
Element e;

for (e=l->first; e; e=e->next)
        #pragma omp task
        process(e);

#pragma omp taskwait
/* ... */
}
```

Example: List Traversal

```
List l;

#pragma omp parallel
traverse_list(l);

/* !!! */
```

Example: List Traversal

```
List l;


#pragma omp parallel
#pragma omp single nowait
traverse_list(l);
```

Example: Multiple Lists

```
List l[N];


#pragma omp parallel
#pragma omp for nowait
for (i = 0; i < N; i++)
 traverse_list(l[i]);
```

Task Scheduling

- Scheduling and synchronization points
  - `#pragma omp taskwait`
    - The encountering task suspends its execution until all the child tasks complete their execution
    - Only the tasks the parent created, not their child tasks!
  - Barriers (implicit / explicit)
    - All the tasks created by any thread of the current team will be completed after the barrier

Execution Model

- An explicit task is executed by a thread of the team that belongs to
    - It can be executed immediately by the thread that creates it
- Parallel regions correspond to task spawning!
    - An implicit task for each thread of the team
    - All task-related operations are meaningful within a parallel region
- Threads can suspend the execution of a task and start or resume another one

Tied and Untied Tasks

- By default, tasks are spawned as **tied**
- Tied tasks
  - Executed only by the same task
  - Have scheduling restrictions
  - Can affect performance
- Untied tasks are more flexible, but special care is needed

Tied and Untied Tasks

- They can migrate between threads at any time, thread specific data structures can lead to unexpected results
- Untied tasks should not be combined with:
  - `threadprivate` variables
  - thread numbers (ids)
- Careful use of critical sections and locks is also required

Data Scope

- Data in the stack of the parent task may be unavailable when new tasks try to access them
- Solutions
  - Use of `firstprivate` whenever this is possible
  - Memory allocation from the heap and not the stack
    - Not always easy
    - Memory deallocation is required
  - Synchronization
    - May affect degree of parallelism

Examples

- Single + OpenMP tasks
- Avoiding Extra Tasks
- Tasks vs For
- Tasks & Reductions

Single and OpenMP tasks

```
#pragma omp parallel
{
   #pragma omp single nowait
   {
      /* this is the initial root task */

       #pragma omp task
       {
          /* this is first child task */
       }

       #pragma omp task
       {
          /* this is second child task */
       }
   }
}
```

Avoiding Extra Tasks

```
void foo ()
{
    A();
    B();
}

void foo ()
{
#pragma omp task
    A();

/*#pragma omp task*/
    B();
}
```

Tasks vs For

```
/* An OpenMP worksharing for loop */
#pragma omp for
for (i=0; i<n; i++) {
    foo(i);
}
```

```
/* The above loop converted to use tasks */
  #pragma omp single
  for (i=0; i<n; i++) {
       #pragma omp task firstprivate(i)
       foo(i);
   }
```

Tasks and Reductions (I)

```
int count_good (item_t *item) {
  int n = 0;
  while (item) {
       if (is_good(item))
          n++;
       item = item->next;
  }
  return n;
}
```

Tasks and Reductions (II)

```
int count_good (item_t *item) {
   int n = 0;
   #pragma omp parallel
   {
      #pragma omp single nowait
      {
          while (item) {
               #pragma omp task firstprivate(item)
               {
                   if (is_good(item)) {
                      #pragma omp atomic
                      n++;
                   }
               }
               item = item->next;
          }
      }
   }
   return n;
}
```

Tasks and Reductions (III)

```
int count_good (item_t *item) {
   int n = 0, pn[P]; /* P is the number of threads used. */
   #pragma omp parallel
   {
      pn[omp_get_thread_num()] = 0;
      #pragma omp single /*nowait*/
      {
          while (item) {
                #pragma omp task firstprivate(item)
                {
                    if (is_good(item)) {
                       pn[omp_get_thread_num()]++;
                    }
                }
                item = item->next;
          }
      }
      #pragma omp atomic
      n += pn[omp_get_thread_num()];
   }
   return n;
}
```

The implicit barrier of `single` is necessary here

One More Example

```
void task(double *x, double *y) {
      *y = x[0]+x[1];
}

int main(int argc, char *argv[]) {
      double result[100];

      for (int i=0; i<100; i++) {
          double d[2];
          d[0] = drand48();
          d[1] = drand48();
          task(d, &result[i]);
      }

   /* print results */
   return 0;
}
```

OpenMP Code

```
void task(double *x, double *y) { *y = x[0] + x[1]; }

int main(int argc, char *argv[]) {
      double result[100];

      #pragma omp parallel
      #pragma omp single nowait
      {
          for (int i=0; i<100; i++) {
              double d[2];
              d[0] = drand48();
              d[1] = drand48();
              #pragma omp task firstprivate(d, i) shared(result)
              {
                task(d, &result[i]);
              }
          }
          #pragma omp taskwait
          /* print results */
      }
    return 0;
}
```

OpenMP Specifications:
Data-Sharing Attribute Clauses, firstprivate clause

"For variables of non-array type, the initialization occurs by copy assignment. For an array of elements of non-array type, each element is initialized as if by assignment from an element of the original array to the corresponding element of the new array"

Translated OpenMP Code

```
/* (l13) #pragma omp single nowait */
if (ort_mysingle(1))
{
  for ((*i) = 0; (*i) < 5; (*i)++)
  {
    double d[ 2];
    d[0] = (*i);
    d[1] = 100 + (*i);

    /* (l19) #pragma omp task firstprivate(d, i) shared(result) */
    struct __taskenv__ {
      double (* result)[ 5];
      int i;
      double d[ 2];
    };
    struct __taskenv__ * _tenv;

    _tenv = (struct __taskenv__ *)ort_taskenv_alloc(sizeof(struct __taskenv__), _taskFunc0_);
    /* byref variables */
    _tenv->result = &(*result);
    /* byvalue variables */
    _tenv->i = (*i);
    memcpy((void *) _tenv->d, (void *) d, sizeof(d));
    ort_new_task(_taskFunc0_, (void *) _tenv, 0, 0);
  }
  /* (l27) #pragma omp taskwait  */
 ort_taskwait(0);
}
ort_leaving_single();
```

Management of Pointers

- `firstprivate` does not perform copy of values accessed through pointers
- Solutions
  - Explicit copy of values
  - Copy to intermediate array and passing of it with `firstprivate`

OpenMP Code (incorrect)

```
void task(double *x, double *y) { *y = x[0] + x[1]; }

void old_main(double *d) {
        double result[100];

        #pragma omp parallel
        #pragma omp single nowait
        {
             for (int i=0; i<100; i++) {
                  d[0] = drand48();
                  d[1] = drand48();
                  #pragma omp task firstprivate(d, i) shared(result)
                  {
                    task(d, &result[i]);
                  }
             }
             #pragma omp taskwait
        }
}

int main(int argc, char *argv[]) {
         double d[2];
         old_main(d);
         return 0;
}
```

Exam Question I

Identify and explain any issues in the following OpenMP code. Propose a solution.

```
int a[20], s;
#pragma omp parallel num_threads(10)
{
        for (int i = 0; i < 20; i++)
        {
                #pragma omp task
                {
                        a[i] = func();
                }
        }

        s = a[0];
        for (int j = 1; j < 20; j++) s += a[j];
}
```

Exam Question II

The following code snippet includes two nested loops that cannot be collapsed and have been parallelized with OpenMP.

```
// A: matrix of size NxN

// nested parallelism is enabled
#pragma omp parallel for
for (int i = 0;  i < N; i++)
{
    // code here

    // TODO: use OpenMP tasks for this loop
    int chunksize = 10;
#pragma omp parallel for schedule(dynamic,chunksize)
    for (int j = 0; j < N; j++)
    {
        A[i][j] = func(i, j);
    }

    // code here
}
```

a) Provide an equivalent parallel implementation of the above code using OpenMP tasks for the innermost loop.

b) Discuss which parallelization approach (original or task-based one) is more efficient and explain why.

Exam Question II

```
#pragma omp parallel for
for (int i = 0;  i < N; i++)
{
    ...
    // #pragma omp parallel for schedule(dynamic,10)
    int ntasks = N / 10;
    for (int t = 0; t < ntasks; t++)
    {
        #pragma omp task shared(a, N) firstprivate(i, t)
        {
            int j0 = t *10;
            int j1 = (t + 1) * 10;
            if (j1 > N) j1 = N;

            for (int j = j0; j < j1; j++)
                a[i][j] = func(i, j);
        }
    }
    #pragma omp taskwait
    ...
}
```

Resources

- OpenMP Specifications & Quick Reference Card
  - www.openmp.org
- The Barcelona OpenMP Task Suite (BOTS) Project
  - https://pm.bsc.es/gitlab/benchmarks/bots