HPCSE - II

«OpenMP Programming Model - Tasks Part II»

Panos Hadjidoukas

Example: Search

```
void search (int n, int j, bool *state ) {
 int i, res ;
 if (n == j)
 {    /* good solution, count it */
      solutions++;
      return ;
 }

 /* try each possible solution */
 for (i = 0; i < n; i ++)
 {
      state[j] = i ;
      if (ok(j+1, state)){
         search (n, j+1 , state) ;
      }
 }
}
```

Example: Search

```
void search (int n, int j, bool *state ) {
  int i, res ;
  if (n == j)
  {     /* good solution, count it */
        solutions++;
        return ;
  }

  /* try each possible solution */
  for (i = 0; i < n; i ++)
      #pragma omp task          /* incorrect */
      {
         state[j] = i ;
         if (ok(j+1, state)){
             search (n, j+1 , state) ;
         }
      }
}
```

Example: Search

```
void search (int n, int j, bool *state ) {
  int i, res ;
  if (n == j)
  {     /* good solution, count it */
        solutions++;
        return ;
  }

  /* try each possible solution */
  for (i = 0; i < n; i ++)
       #pragma omp task
       {
           bool *new_state = alloca(sizeof(bool)*n); //== bool new_state[n];
           memcpy (new_state, state, sizeof(bool)*n );
           new_state[j] = i ;
           if (ok(j+1, new_state)){
                search (n, j+1, new_state) ;
           }
       }
}
```

Example: Search

```
void search (int n, int j, bool *state ) {
  int i, res ;
  if (n == j)
  {     /* good solution, count it */
        solutions++;  /* shared variable */
        return ;
  }

  /* try each possible solution */
  for (i = 0; i < n; i ++)
        #pragma omp task
        {
           bool *new_state = alloca(sizeof(bool)*n);
           memcpy (new_state, state, sizeof(bool)*n );
           new_state[j] = i ;
           if (ok(j+1, new_state)){
                search (n, j+1, new_state) ;
           }
        }
   #pragma omp taskwait
}
```

- `pragma omp critical`
- `pragma omp atomic`
- Reduction:
  - Not supported for tasks
  - Can be implemented manually

Example: Search

```
int solutions=0;

int mysolutions=0; /* separate counter */
#pragma omp threadprivate (mysolutions)

void start_search()
{
 #pragma omp parallel
  {
    #pragma omp single
    {
        bool initial_state[n];
        search(n, 0, initial_state);
    }
    #pragma omp critical
        solutions += mysolutions;/* accumulate */
  }
}
```

Example: Search

```
void search (int n, int j, bool *state ) {
  int i, res ;
  if (n == j)
  {      /* good solution, count it */
        mysolutions++;          /* problem with untied */
        return ;
  }

  /* try each possible solution */
  for (i = 0; i < n; i ++)
        #pragma omp task     untied             /* easier load balance */
        {
              bool *new_state = alloca(sizeof(bool)*n);
              memcpy (new_state, state, sizeof(bool)*n );
              new_state[j] = i ;
              if (ok(j+1, new_state)){
                    search (n, j+1, new_state) ;
              }
        }
  #pragma omp taskwait
}
```

Example: Search

```
void search (int n, int j, bool *state ) {
  int i, res ;
  if (n == j)
  {      /* good solution, count it */
         #pragma omp task if(0)
         mysolutions++;
   return ;
  }

  /* try each possible solution */
  for (i = 0; i < n; i ++)
         #pragma omp task     untied
         {
               bool *new_state = alloca(sizeof(bool)*n);
               memcpy (new_state, state, sizeof(bool)*n );
               new_state[j] = i ;
               if (ok(j+1, new_state)){
                     search (n, j+1, new_state) ;
               }
         }
  #pragma omp taskwait
}
```

Example: Search

```
void search (int n, int j, bool *state, int depth ) {
  int i, res;
  if (n == j)
  {      /* good solution, count it */
         #pragma omp task if(0)
         mysolutions++;
   return ;
  }

  /* try each possible solution */
  for (i = 0; i < n; i ++)
         #pragma omp task    untied if(depth < MAX_DEPTH)
         {
               bool *new_state = alloca(sizeof(bool)*n);
               memcpy (new_state, state, sizeof(bool)*n );
               new_state[j] = i ;
               if (ok(j+1, new_state)){
                     search (n, j+1, new_state, depth+1) ;
               }
         }
  #pragma omp taskwait
}
```

Example: Search

```
void search (int n, int j, bool *state, int depth ) {
  int i, res;
  if (n == j) {
        #pragma omp task if(0)
        mysolutions++;
   return ;
  }

  for (i = 0; i < n; i ++)
        #pragma omp task untied
        {
              bool *new_state = alloca(sizeof(bool)*n);
              memcpy (new_state, state, sizeof(bool)*n );
              new_state[j] = i ;
              if (ok(j+1, new_state)){
                    if(depth < MAX_DEPTH)
                          search (n, j+1, new_state, depth+1) ;
                    else
                          search_serial (n, j+1, new_state) ;
              }
        }
  #pragma omp taskwait
}
```

Tips

- Use `default(none)` if unsure of data scoping
- Careful when using `firstprivate` on pointers
- Careful with out-of-scope data
- Use `untied` tasks carefully
- Control granularity
- Do not abuse of tasks

Additional Task Features

- taskyield
- taskgroup
- depend
- taskloop

taskyield

- The `taskyield` directive specifies that the current task can be suspended in favor of execution of a different task.
  - Hint to the runtime for optimization and/or deadlock prevention

taskyield

```
#include <omp.h>

void something_useful();
void something_critical();

void foo(omp_lock_t * lock, int n)
{
      for(int i = 0; i < n; i++)
         #pragma omp task
         {
            something_useful();
            while( !omp_test_lock(lock) ) {
               #pragma omp taskyield
            }
            something_critical();
            omp_unset_lock(lock);
         }
}
```

The waiting task may be suspended here and allow the executing thread to perform other work; may also avoid deadlock situations.

taskgroup

- The `task group` directive specifies a wait on completion of child tasks and their descendant tasks
  - "deeper" synchronization than taskwait, but
  - with the option to restrict to a subset of all tasks (as opposed to a barrier)

taskgroup

```
/* Compute f2 (A, f1 (B, C)) */

int foo () {
    int a, b, c, x, y;

    a = A();
    b = B();
    c = C();
    x = f1(b, c);
    y = f2(a, x);

    return y;
}
```

taskgroup

```
/* Compute f2 (A, f1 (B, C)) */
void foo () {
    int a, b, c, x, y;

    #pragma omp task shared(a)
    a = A();

    #pragma omp taskgroup
    {
        #pragma omp task shared(b)
        b = B();

        #pragma omp task shared(c)
        c = C();
    }
    x = f1 (b, c);
    #pragma omp taskwait
    y = f2 (a, x);
}
```

taskgroup

```
/* Compute f2 (A, f1 (B, C)) */
void foo () {
    int a, b, c, x, y;

    #pragma omp task shared(a)
    a = A();

    #pragma omp task if (0) shared (b, c, x)
    {
        #pragma omp task shared(b)
        b = B();

        #pragma omp task shared(c)
        c = C();

        #pragma omp taskwait
    }
    x = f1 (b, c);
    #pragma omp taskwait
    y = f2 (a, x);
}
```

Equivalent approach if taskgroup not supported

depend clause

```
#pragma omp task depend(dependency-type:list)
... structured block ...
```

- The task dependence is fulfilled when the predecessor task has completed
  - *`in`* dependency-type: the generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an `out` or *`inout`* clause.
  - *`out`* and *`inout`* dependency-type: The generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an *`in`*, *`out`*, or *`inout`* clause.
  - The list items in a `depend` clause may include array sections

depend

- T1 has to be completed before T2 and T3 can be executed.
- T2 and T3 can be executed in parallel

```
void process_in_parallel() {
  #pragma omp parallel
  #pragma omp single
  {
    int x = 1;
    ...
    for (int i = 0; i < T; ++i) {
      #pragma omp task shared(x, ...) depend(out: x) // T1
        preprocess_some_data(...);
      #pragma omp task shared(x, ...) depend(in: x) // T2
        do_something_with_data(...);
      #pragma omp task shared(x, ...) depend(in: x) // T3
        do_something_independent_with_data(...);
    }
  } // end omp single, omp parallel
}
```

depend

- The code allows for more parallelism, as there is one i per thread. Thus, two tasks may be active per thread.

```
void process_in_parallel() {
  #pragma omp parallel
  {
    #pragma omp for
    for (int i = 0; i < T; ++i) {
      #pragma omp task depend(out: i)
        preprocess_some_data(...);
      #pragma omp task depend(in: i)
        do_something_with_data(...);
      #pragma omp task depend(in: i)
        do_something_independent_with_data(...);
    }
  } // end omp parallel
}
```

depend

- Even more parallelism: two tasks may be active per thread per i-th iteration.

```
void process_in_parallel() {
  #pragma omp parallel
  #pragma omp single
  {
   for (int i = 0; i < T; ++i) {
     #pragma omp task firstprivate(i)
     {
       #pragma omp task depend(out: i)
         preprocess_some_data(...);
       #pragma omp task depend(in: i)
         do_something_with_data(...);
       #pragma omp task depend(in: i)
         do_something_independent_with_data(...);
      } // end task
    }
  } // end omp parallel
}
```

depend

```
/*
    A0   A1   A2   A3   A4
    |    |    |    |    |
    v    v    v    v    v
    B0->B1->B2->B3->B4
 */
```

```
int main(int argc, char *argv[])
{
  #pragma omp parallel
  #pragma omp single nowait
  {
    for (int i = 0; i < 5; i++) {
      int x[10], y;
      #pragma omp task firstprivate(i) depend(out:x[i])
      {
        do_work_A(i);
      }
      #pragma omp task firstprivate(i) depend(in:x[i]) depend(inout:y)
      {
        do_work_B(i);
      }
    }
    #pragma omp taskwait
  }
  return 0;
}
```

taskloop

- The ‘taskloop’ construct was added in version 4.5
  - it allows dividing iterations of a loop into tasks.
  - It can optionally wait on completion of those tasks
  - Each created task is assigned one or more iterations of the loop.

```
#pragma omp taskloop num_tasks(32)
for (long l = 0; l < 1024; l++)
  do_something(l);
```

- The above code will create 32 tied tasks in a new taskgroup
- A reasonable implementation will assign 32 iterations to each task.
- Due to the implicit taskgroup, the encountering task will wait for completion of all these tasks.
- Instead of specifying the number of tasks, the grainsize clause can be used to specify how many iterations each task should have

Exam Question I

Consider the following code:

```
int count_good (int *item, int size)
{
    int n = 0;
    for (int i = 0; i < size; i++) {
        if (is_good(item[i]))
            n++;
    }
    return n;
}

// int items[N][S];
// int count[N];

int compute_counts()
{
    for (int t = 0; t < N; t++)
        count[t] = count_good(items[t], S);
}
```

a) Parallelize the function `count_good` with OpenMP tasks.

b) Parallelize also the function `compute_counts` with OpenMP.

Exam Question II

Consider the following MPI code:

```
// int myrank: MPI rank
// int nprocs: number of MPI processes
// double A: array of size nprocs-1

if (myrank < nprocs-1)
{
  double a = prepare_data();     // variable execution time
  MPI_Send(&a, 1, MPI_DOUBLE, size-1, myrank+100, MPI_COMM_WORLD);
}
else
{
   for (int i = 0; i < nprocs-1; i++)
   {
      MPI_Status status;
      MPI_Recv(&A[i], 1, MPI_DOUBLE, i, i+100, MPI_COMM_WORLD, &status);
      process(A[i]);
  }
}
```

a) Parallelize the for loop executed by the last rank.

Resources

- OpenMP Specifications & Quick Reference Card
  - www.openmp.org
- OpenMP Tasking In-Depth, C. Terboven, IT Center der RWTH Aachen University