OpenMP Examples - Tasking

Marco Ferretti, Mirto Musci

Dipartimento di Ingegneria Industriale e dell'Informazione
University of Pavia

December 3, 2018

Outline

Outline

Why task parallelism?

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);
  }
}
```

Without tasks

- Ackward
- Very poor performance
- Not composable

Why task parallelism?

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 );
}
```

Without tasks

- Too many parallel regions
- Extra overheads
- Extra synchronizations
- Not always well supported

Task parallelism

- Better solution for those problems
- Tasks were first introduced to OpenMP in version 3.0
- In general: parallelize irregular problems
  - unbounded loops
  - recursive algorithms
  - producer/consumer schemes
- In OpenMP 4.0, the `depend` clause and the `taskgroup` construct were introduced
  - `taskloops` were introduced in OpenMP 4.5

List traversal (with task)

Example

```
void traverse_list (List l)
{
  Element e;
  for (e=l->first; e; e=e->next)
    #pragma omp task
     process(e);
}
```

What is a task?

- Tasks are code blocks that the compiler wraps up and makes available to be executed in parallel
  - A task is a package of both code and variable
- Execution is completely in the hand of the compiler
  - No need to manually assign to threads!
  - Highly composable. Can be nested inside parallel regions, other tasks and worksharings

```
#pragma omp parallel
{
  #pragma omp task
   printf("hello world from a random thread\n");
}
```

How to create them?

- Like worksharing constructs, tasks must also be created inside of a parallel region.
- In the previous example every thread created a "print" task.
- To only spawn a task once, the `single` construct is used.
  - The execution is then distributed to every thread

```
#pragma omp parallel
{
  #pragma omp single
  {
    #pragma omp task
    printf("hello world\n");

    #pragma omp task
    printf("hello again!\n");
  }
}
```

Task synchronization

- The previous example will only print *hello world* once, but the ordering of *hello world* and *hello again* is undefined.
  - Only one guarantee: both tasks will end at the parallel region `barrier` (or any other barrier)
- Two ways to specify the order: `taskwait` and dependencies
  - `taskwait` waits for tasks spawned by current task and itself

```
#pragma omp parallel
{
  #pragma omp single
  {
    #pragma omp task
    printf("hello world\n");

    #pragma omp taskwait

    #pragma omp task
    printf("hello again!\n");
  }
}
```

Data sharing for tasks

- Like other OpenMP constructs, variables that are used inside a task can be specified:
  - explicitly with clauses (shared, firstprivate, private, etc.)
  - implicitly, if not otherwise specified.
- In general, data accessed by a task is shared.

Example 1

```
#pragma omp parallel
{
  int x = 0;
  #pragma omp single
  {
    #pragma omp task
    {   x++;
      printf("from task 1: x = %d\n", x); }

    #pragma omp taskwait

    #pragma omp task
    { x++;
      printf("from task 2: x = %d\n", x); }
  }
}
```

- Variable x will be 1 and then 2, since both tasks are using the (implicitly) shared variable x.

Example 1

```
#pragma omp parallel
{
  int x = 0;
  #pragma omp single
  {
    #pragma omp task
    {   x++;
      printf("from task 1: x = %d\n", x); }

    #pragma omp taskwait

    #pragma omp task
    { x++;
      printf("from task 2: x = %d\n", x); }
  }
}
```

- Variable x will be 1 and then 2, since both tasks are using the (implicitly) shared variable x.

Example 2

```
#pragma omp parallel
{ int x = 0;
  #pragma omp single
  {
    #pragma omp task firstprivate(x)
    { x++;
      printf("from task 1: x = %d\n", x); 10  }

    #pragma omp taskwait

    #pragma omp task firstprivate(x)
    { x++;
      printf("from task 2: x = %d\n", x); }
  }
}
```

- Variable `x` is made `firstprivate`, so both tasks increment their copy of `x`, and both print `x=1`
  - `private` wouldn't be valid: `x` is not initialized in the task

Example 2

```
#pragma omp parallel
{ int x = 0;
  #pragma omp single
  {
    #pragma omp task firstprivate(x)
    { x++;
      printf("from task 1: x = %d\n", x); 10  }

    #pragma omp taskwait

    #pragma omp task firstprivate(x)
    { x++;
      printf("from task 2: x = %d\n", x); }
  }
}
```

- Variable `x` is made `firstprivate`, so both tasks increment their copy of `x`, and both print `x=1`
  - `private` wouldn't be valid: `x` is not initialized in the task

Advanced data sharing

```
#pragma omp parallel
{
  #pragma omp single
  {
    #pragma omp task
    {
      int x = 0;
      #pragma omp task
      { x++;
        printf("x = %d\n"); }
    }
  }
}
```

- One task spawning another.
  - the inner task accesses a variable local to the outer task.
  - even if only one task modify it, `x` exists on the stack of the outer task: outer task can finish before inner task begins!
  - up to the programmer to synchronize (e.g. `taskwait`)

Outline

The depend clause

- The depend clause takes a type followed by a list of variables

```
#pragma omp task depend(in: x)
  depend(out: y) depend(inout: z)
```

- They are both to correctly order the tasks.
  - IN will make a task dependent on the last task that used the same variable as an out (or inout) dependency.
  - OUT will make a task dependent on the last task that used the same variable as an in, out (or inout) dependency
  - INOUT the same as OUT, only used for readability.
- There is no data movement with respect to external accesses

Example: simple depend

```
#pragma omp parallel
{
  #pragma omp single
  {
    int x, y, z;
    #pragma omp task depend(out: x)
    x = init();

    #pragma omp task depend(in: x) depend(out: y)
    y = f(x);

    #pragma omp task depend(in: x) depend(out: z)
    z = g(x);

    #pragma omp task depend(in: y, z)
    finalize(y, z);
  }
}
```

Example: array sections as dependencies

```
void matmul_depend(int N, int BS, float A[N][N], float B[N][N], float C[N][N])
{

  for(int i=0; i<N; i+=BS)
    for(int j=0; j<N; j+=BS)
      for(int k=0; k<N; k+=BS)

        #pragma omp task depend(in: A[i:BS][k:BS], B[k:BS][j:BS]) \
          depend(inout: C[i:BS][j:BS])

          for(int ii=i; ii<i+BS; ii++)
            for(int jj=j; jj<j+BS; jj++)
              for(int kk=k; kk<k+BS; kk++)
                C[ii][jj] += A[ii][kk] * B[kk][jj];

}
```

- Assumptions:
  - BS divides evenly into N
  - i, j, k, A, B, C are firstprivate by default
  - A, B and C are just pointers: they all refer to the same data.

Taskgroup I

- A `taskgroup` is similar to `taskwait`
  - waits on all descendant created in the block that follows
  - not only for childs of current task, like `taskwait`

```
#pragma omp taskgroup
  #pragma omp task
    task_spawning_function();
```

Taskgroup II

- Taskgroups can also be used to selectively synchronize tasks:
  - `background` can continue to the end of the parallel region
  - `task_spawning` and all of it descendants will be waited at the end of the taskgroup.

```
#pragma omp parallel
{
  #pragma omp single
  {
    #pragma omp task
      background_work();
      #pragma omp taskgroup
      {
        #pragma omp task
        task_spawning_function();
      }
  }
}
```

The taskloop construct

- `taskloop` precedes a for loop, and creates a tasks for one or more iterations of the loop.
  - implicit taskgroup
  - basic usage is similar to a for construct
  - clauses from both `for` and `task` constructs for fine-tuning
- Unique clauses
  - `grainsize` and `num_tasks` are used to specify how to break up the iterations (similar to chunksize and number of threads)
  - `nogroup`: similar to `nowait`, removing the taskgroup behavior

```
#pragma omp taskloop
for(int i=0; i < N; i++)
  //Work
```

Outline

Quicksort Algorithm

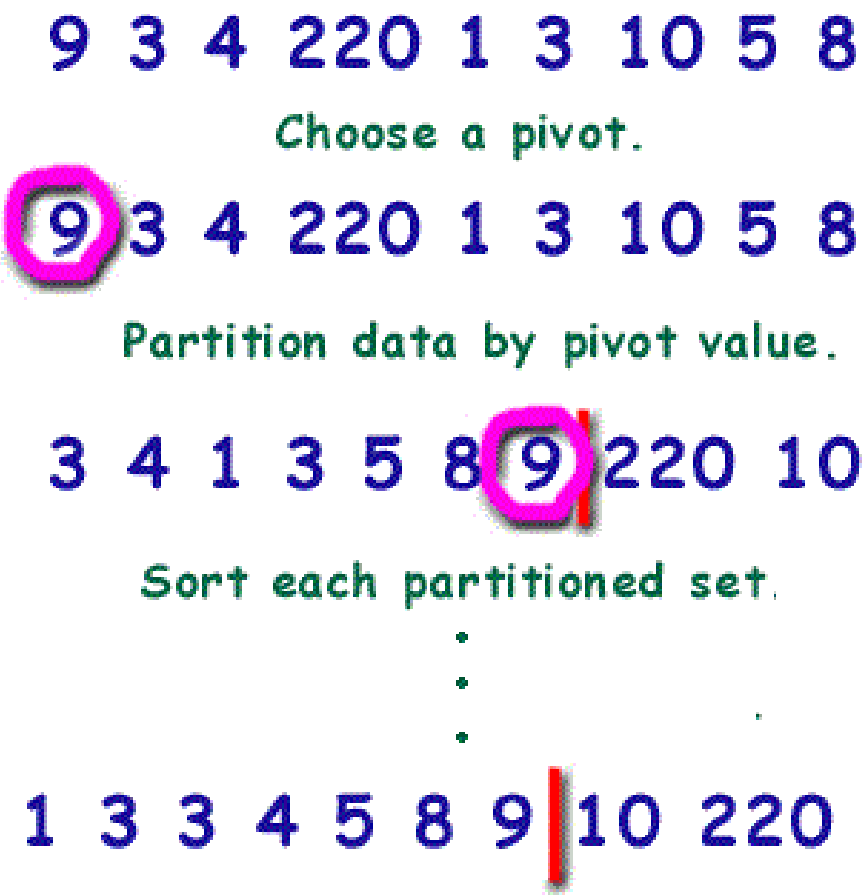
Algorithm

- Given an array of n elements (e.g., integers).
- If array only contains one element: return.
- Else: Pick one element to use as pivot.
- Partition elements into two sub-arrays:
  - Elements less than or equal to pivot
  - Elements greater than pivot
- Recursively quicksort two sub-arrays
- Return results

Notes

- There are a number of ways to pick the pivot element
  - Commonly first or last element, but bad performance if the array is already ordered
  - Random index or middle-point index solve the problem
- After partitioning, the sub-arrays can be stored in the original data array to increase memory efficiency
  - Partitioning loops through, swapping elements

Example


9 3 4 220 1 3 10 5 8
Choose a pivot.
9 3 4 220 1 3 10 5 8
Partition data by pivot value.
3 4 1 3 5 8 9 220 10
Sort each partitioned set.
1 3 3 4 5 8 9 10 220

Serial Code

```
void quicksort ( int a[ ], int lower, int upper )
{
  int i ;
  if ( upper > lower )
  {
    i = partition ( a, lower, upper ) ;
    quicksort ( a, lower, i - 1 ) ;
    quicksort ( a, i + 1, upper ) ;
  }
}
```

Assigment

- Refine the serial implementation provided
- Try to parallelize the code using OpenMP... is not easy!
  - Try with section constructs, or experiment with task
- Remember the code is recursive!
  - You could try to call omp_set_nested(1)
  - Or somehow limit thread spawning
- Also could be necessary to reduce task spawning... `if` clause (check documentation)
- Always measure performance with omp_get_wtime

Outline

The long way to the final project...

- This final exercise will not contain any help, just the source code of a serial implementation
  - Has a level of complexity on par of a final project
- In the final report you should:
  - analyze the serial algorithm
  - implement it serially
  - analyze all of the possible parallel implementations
  - actually implement one or more of them
  - correctly measure the performances scaling the number of threads, the volume of data...

Source code

In numerical linear algebra, the Jacobi method is an algorithm for determining the solutions of a diagonally dominant system of linear equations.

- Here we only report the core of the serial algorithm
- As typical in parallelism, we don't need to understand the meaning; just the structure

```
for(int i=0; i<size; i++) {
    for(int j=i+1; j<size; j++) {
        A[j*size+i] /= A[i*size+i];
        for(int k=i+1; k<size; k++) {
            A[j*size+k] -= A[j*size+i] * A[i*size+k];
        }
    }
}
```

Exercise

- Measure serial performance.
  - How can you parallelize it?
  - Be careful of dependencies.
- Try using the for construct... but where?
- Try using tasking and the depend clause
  - Hint: you have to protect three IN values and one OUT value
  - Strive for a small granularity with task!
- There are solutions provided... but do not look at them!
  - The second task solution is useful for huge dataset.. why?
  - Hint: it's about granularity...

For Further Reading

Blaise Barney

OpenMP Exercise, 2015

https://computing.llnl.gov/tutorials/openMP/exercise.html

NERCS

OpenMP Tasking Tutorial

http://www.nersc.gov/users/software/programming-models/openmp/openmp-tasking/