Task (6A)

- Task
-

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using OpenOffice and Octave.

Tasking

- Tasking was introduced in OpenMP 3.0
- Until then it was impossible to efficiently and easily implement **certain types of parallelism**
- the initial functionality was very **simple** by design
- note that tasks can be **nested**

https://www.openmp.org//wp-content/uploads/sc13.tasking.ruud.pdf

Tasking

Developer

- Use a **pragma** to specify where the tasks are
- Assume that all tasks can be executed independently

OpenMP runtime system

- when a thread encounters a **task** construct,
  a new task is generated
- the **moment** of **execution** of the task
  is up to the **runtime system**
- execution can either be **immediate** or **delayed**
- **completion** of a task can be enforced
  through **task synchronization**

https://www.openmp.org//wp-content/uploads/sc13.tasking.ruud.pdf

Tasking

The task pragma can be used to explicitly define a task.

Use the task pragma when you want to identify a block of code
to be executed in parallel with the code outside the task region.

The task pragma can be useful for parallelizing irregular algorithms
such as pointer chasing or recursive algorithms.

The task directive takes effect only if
you specify the SMP compiler option.

https://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-task.html

Tasking – smp option

Symmetric Multi Processing

-qnosmp | -qsmp[=suboption[:suboption] [...]]

specifies if and how parallelized object code is generated,
according to suboption(s) specified:

http://ps-2.kev009.com/wisclibrary/vacpp/batch/ref/ruoptsmp.htm

Tasking – task region

task
a specific instance of executable code and its data environment
that the OpenMP implementation can schedule for execution by threads.

task region
A region consisting of all code encountered
during the execution of a task.

COMMENT: A **parallel region** consists of
one or more **implicit task regions**.

implicit task
A task generated by an **implicit parallel region** or
generated when a **parallel construct** is encountered during execution.

https://www.openmp.org/spec-html/5.0/openmpsu5.html

Tasking

if you specify something as being parallel,
OpenMP will create a ‘block of work‘:
a section of code plus the data environment
in which it occurred.
This block is set aside for execution at some later point.

The task mechanism allows you to do things that are
hard or impossible with the loop and section constructs.

for instance, a loop traversing a linked list can be implemented with tasks:

https://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-task.html

Tasking

```
x = f ();                    // the variable x gets a value
pragma omp task {            // a task is created with the current value of x
   y = g(x);
}
z = h();                     // the variable z gets a value
```

The thread that executes this code segment creates a task,
which will later be executed, probably by a different thread.
The exact timing of the execution of the task is
up to a task scheduler,
which operates invisible to the user.

https://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-task.html

Task example (1)

```
#include <stdlib.h>
#include <stdio.h>
int main(intargc, char *argv[])
{
   printf("A ");
   printf("race ");
   printf("car ");
   printf("\n");
   return(0);
}
```

```
$ cc -fast hello.c
$ ./a.out
A race car
$
```

```
#include <stdlib.h>
#include <stdio.h>
int main(intargc, char *argv[])
{
   #pragma omp parallel  {
      printf("A ");
      printf("race ");
      printf("car ");
   }
   printf("\n");
   return(0);
}
```

```
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2 $
./a.out
A race car A race car        or
   "A A race race car car"   or
   "A race A car race car"   or
   "A race A race car car"
```

https://www.openmp.org//wp-content/uploads/sc13.tasking.ruud.pdf

Task example (2)

```
#include <stdlib.h>
#include <stdio.h>
int main(intargc, char *argv[])
{
  #pragma omp parallel    {
    #pragma omp single    {
      printf("A ");
      printf("race ");
      printf("car ");
    }
  }
  printf("\n");
  return(0);
}
```

```
$ cc -xopenmp –fast hello.c
$ export OMP_NUM_THREADS=2 $
./a.out
A race car
```

```
#include <stdlib.h>
#include <stdio.h>
int main(intargc, char *argv[])
{
  #pragma omp parallel    {
    #pragma omp single    {
      printf("A ");
      #pragma omp task    {  printf("race ");}
      #pragma omp task    {  printf("car "); }
    }
  }
  printf("\n");
  return(0);
}
```

```
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out        A race car
$ ./a.out        A race car
$ ./a.out        A car race
$
```

https://www.openmp.org//wp-content/uploads/sc13.tasking.ruud.pdf

Task example (3)

```
#include <stdlib.h>
#include <stdio.h>
int main(intargc, char *argv[])
{
  #pragma omp parallel    {
    #pragma omp single    {
      printf("A ");
      #pragma omp task    {  printf("race ");}
      #pragma omp task    {  printf("car "); }
      printf("is fun to watch ");
    }
  }
  printf("\n");
  return(0);
}
```

```
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out          A is fun to watch race car
$ ./a.out          A is fun to watch race car
$ ./a.out          A is fun to watch car race
$
```

```
#include <stdlib.h>
#include <stdio.h>
int main(intargc, char *argv[])
{
  #pragma omp parallel    {
    #pragma omp single    {
      printf("A ");
      #pragma omp task    {  printf("race "); }
      #pragma omp task    {  printf("car "); }
      #pragma omp taskwait {  printf("is fun to watch "); }
    }
  }
  printf("\n");
  return(0);
}
```

```
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out          A race car is fun to watch
$ ./a.out          A race car is fun to watch
$ ./a.out          A car race is fun to watch
$
```

https://www.openmp.org//wp-content/uploads/sc13.tasking.ruud.pdf

Tasking

With tasks it becomes possible to parallelize processes that did not fit the earlier OpenMP constructs.

For instance, if a certain operation needs to be applied to all elements of a **linked list**, you can have **one thread go down the list**, **generating a task** for each element of the list.

#pragrma omp single

```
#pragma omp parallel
#pragma omp single
{
  p = head of list();                  // one thread traverses the list
  while(!end of list(p)) {
    #pragma omp task                   // a task is created,
      process(p);                      // one for each element
    p = next element(p);               // the generating thread goes on without waiting
  }                                    // the tasks are executed while more are being generated.
```

https://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-task.html

Tasking

```
p = head of list();
while(!end of list(p)) {
  #pragma omp task
    process(p);
  p = next element(p);
}
```

// one thread traverses the list

// a task is created,
// one for each element
// the generating thread goes on without waiting
// the tasks are executed while more are being generated.

The way **tasks** and **threads** interact is different from
the other worksharing constructs
Typically, one thread will generate the tasks,
adding them to a queue, from which all threads
can take and execute them.

https://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-task.html

Tasking

```
#pragma omp parallel
#pragma omp single
{
   p = head of list();
   while(!end of list(p)) {
      #pragma omp task
         process(p);
      p = next element(p);
   }
```

// one thread traverses the list

// a task is created,
// one for each element
// the generating thread goes on without waiting
// the tasks are executed while more are being generated.

```
p = head of list();
while(!end of list(p)) {
   p = next element(p);
}
```

process(p1)

process(p2)

process(p3)

https://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-task.html

Tasking

```
#pragma omp parallel          // A parallel region creates a team of threads;
#pragma omp single
{
  ...
#pragma omp task              // a single thread then creates the tasks,
  { ... }                     // adding them to a queue that belongs to the team,
  …                           // and all the threads in that team
}                             // (possibly including the one that generated the tasks)
```

https://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-task.html

Task Data (1)

Treatment of **data in a task** is somewhat subtle.
a **task** gets created at one time,
and executed at another.

if **shared data** is accessed,
does the task see the **value**
at creation time or at execution time?

In fact, both possibilities make sense
depending on the application

The first rule is that **shared data** is
shared in the task, but **private data**
becomes code fragments.

In the first example:

https://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-task.html

Task Data (2)

```
int count = 100;
#pragma omp parallel
#pragma omp single
{
  while (count>0) {
#pragma omp task
    {
      int countcopy = count;
      if (count==50) {
        sleep(1);
        printf("%d,%d\n",count,countcopy);
      } // end if
    }   // end task
    count--;
  }     // end while
}       // end single
```

the variable **count**
declared outside the parallel region
is therefore shared.

when the print statement is executed,
all **tasks** will have been generated,
and so **count** will be **zero**.

Thus, the output will likely be 0,50

https://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-task.html

Task Data (3)

```
#pragma omp parallel
#pragma omp single
{
  int count = 100;
  while (count>0) {
#pragma omp task
    {
      int countcopy = count;
      if (count==50) {
        sleep(1);
        printf("%d,%d\n",count,countcopy);
      } // end if
    }   // end task
    count--;
  }     // end while
}       // end single
```

the count variable
private to the thread creating the tasks,
will be **firstprivate** in the task,
preserving the value that was current
when the task was created.

the **firstprivate** variable
initialized by the original value
when the parallel construct is encountered.

the **lastprivate** variable
updated after the end
of the parallel construct.

https://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-task.html

Task Dependency (1)

It is possible to put a partial ordering on tasks through use of the

```
#pragma omp task
  x = f()
#pragma omp task
  y = g(x)
```

it is conceivable that the second task is executed before the first, possibly leading to an incorrect result. This is remedied by specifying:

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

https://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-task.html

Task Dependency (2)

```
for i in [1:N]:
    x[0,i] = some_function_of(i)
    x[i,0] = some_function_of(i)
```

```
for i in [1:N]:
    for j in [1:N]:
        x[i,j] = x[i-1,j]+x[i,j-1]
```

Observe that the second loop nest is not amenable to OpenMP loop parallelism.

Can you think of a way to realize the computation with OpenMP loop parallelism? Hint: you need to rewrite the code so that the same operations are done in a different order.

Use tasks with dependencies to make this code parallel without any rewriting: the only change is to add OpenMP directives.

https://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-task.html

Task Dependency (3)

Tasks dependencies are used to indicated how two uses of one data item relate to each other. Since either use can be a read or a write, there are four types of dependencies.

[RaW (Read after Write)] The second task reads an item that the first task writes. The second task has to be executed after the first:

```
... omp task depend(OUT:x)
  foo(x)
... omp task depend( IN:x)
  foo(x)
```

[WaR (Write after Read)] The first task reads and item, and the second task overwrites it. The second task has to be executed second to prevent overwriting the initial value:

```
... omp task depend( IN:x)
  foo(x)
... omp task depend(OUT:x)
  foo(x)
```

https://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-task.html

Task Dependency (4)

[WaW (Write after Write)] Both tasks set the same variable.
Since the variable can be used by an intermediate task,
the two writes have to be executed in this order.

```
... omp task depend(OUT:x)
  foo(x)
... omp task depend(OUT:x)
  foo(x)
```

[RaR (Read after Read)] Both tasks read a variable.
Since neither tasks has an `out' declaration, they can run in either order.

```
... omp task depend(IN:x)
  foo(x)
... omp task depend(IN:x)
  foo(x)
```

https://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-task.html

Task Synchronization

even though the above segment looks like a linear set of statements,
it is impossible to say when the code after the task directive will be executed.
This means that the following code is incorrect:

```
  x = f();
#pragma omp task
  { y = g(x); }
  z = h(y);
```

Explanation: when the statement computing z is executed,
the task computing y has only been scheduled;
it has not necessarily been executed yet.

https://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-task.html

Task Synchronization

In order to have a guarantee that a task is finished,
you need the taskwait directive.
The following creates two tasks, which can be executed in parallel,
and then waits for the results:

```
#pragma omp parallel
#pragma omp single
{
  while (!tail(p)) {
    p = p->next();
#pragma omp task
    process(p)
  }
#pragma omp taskwait
}
```

https://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-task.html

Task Synchronization

You can indicate **task dependencies** in several ways:

Using the **task wait** directive
you can explicitly indicate the join of the forked tasks.
The instruction after the **wait** directive will therefore be
dependent on the spawned tasks.

The **taskgroup** directive, followed by a structured block,
ensures completion of all tasks created in the block, even if recursively created.

Each OpenMP task can have a clause,
indicating what **data dependency** of the task.
By indicating what data is produced or absorbed by the tasks,
the **scheduler** can construct the dependency graph for you.

https://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-task.html

taskwait

Completion of a subset of all **explicit tasks** bound to a given **parallel** region may be specified through the use of the **taskwait** directive.

The **taskwait** directive specifies a **wait** on the completion of **child tasks** generated since the beginning of the current (implicit or explicit) task.

Note that the **taskwait** directive specifies a **wait** on the completion of direct **children tasks**, not all descendant **tasks**.

https://docs.oracle.com/cd/E19205-01/820-7883/6nj43o69j/index.html

Tasking example

```
#include <stdio.h>
#include <omp.h>
int fib(int n)
{
  int i, j;
  if (n<2)
    return n;
  else
    {
      #pragma omp task shared(i) firstprivate(n)
      i=fib(n-1);

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

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

```
int main()
{
  int n = 10;

  omp_set_dynamic(0);
  omp_set_num_threads(4);

  #pragma omp parallel shared(n)
  {
    #pragma omp single
    printf ("fib(%d) = %d\n", n, fib(n));
  }
}
```

```
% CC -xopenmp -xO3 task_example.cc
% a.out
fib(10) = 55
```

https://docs.oracle.com/cd/E19205-01/820-7883/6nj43o69j/index.html

Tasking example

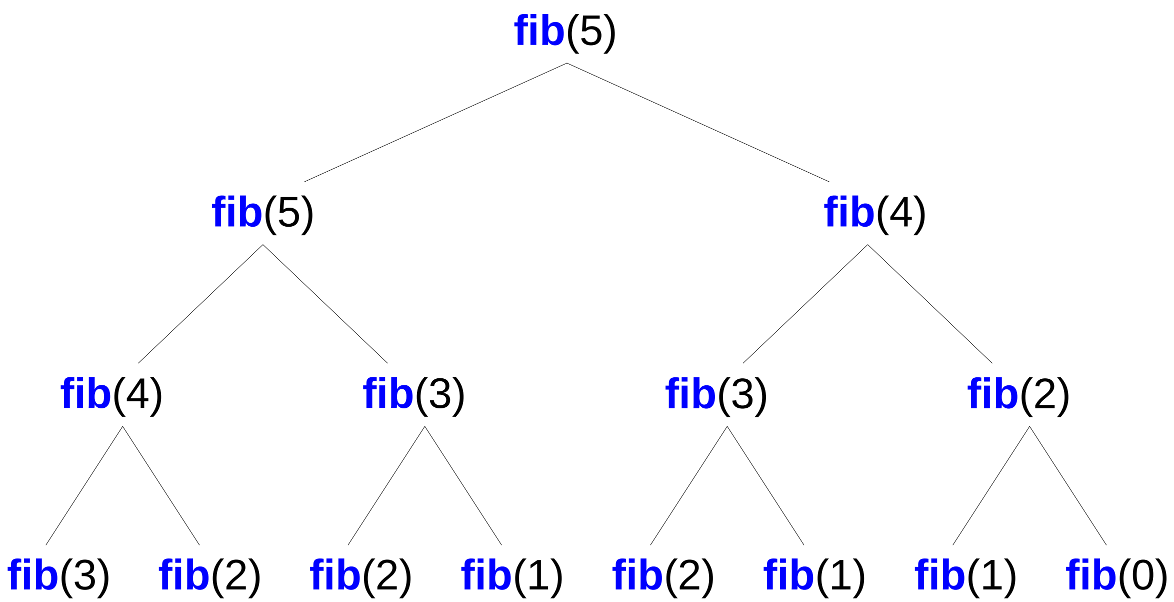
The following C/C++ program illustrates
how the OpenMP **task** and **taskwait** directives
can be used to compute Fibonacci numbers recursively.

In the example, the **parallel** directive denotes
a **parallel region** which will be executed by four threads.

In the **parallel** construct, the **single** directive is used
to indicate that only one of the **threads**
will execute the print statement that calls **fib**(n).

https://docs.oracle.com/cd/E19205-01/820-7883/6nj43o69j/index.html

Tasking example



https://docs.oracle.com/cd/E19205-01/820-7883/6nj43o69j/index.html

Tasking example

The call to **fib**(n) generates two **tasks**, indicated by the task directive.

One of the tasks computes **fib**(n-1) and the other computes **fib**(n-2), and the **return values** are added together to produce the value returned by **fib**(n).

Each of the calls to **fib**(n-1) and **fib**(n-2) will in turn generate two **tasks**.

tasks will be recursively generated until the argument passed to **fib**() is less than 2.

```
#pragma omp task shared(i) firstprivate(n)
i=fib(n-1);
```

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

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

https://docs.oracle.com/cd/E19205-01/820-7883/6nj43o69j/index.html

Tasking example

The **taskwait** directive ensures that
the two **tasks** generated
in an invocation of **fib**(n) are completed
(that is. the tasks compute i and j)
before that invocation of **fib**(n) returns.

Note that although only one **thread**
executes the **single** directive
and hence the call to **fib**(n),
all four threads will participate
in executing the tasks generated

```
#pragma omp task shared(i) firstprivate(n)
i=fib(n-1);
```

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

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

https://docs.oracle.com/cd/E19205-01/820-7883/6nj43o69j/index.html

References

[1] en.wikipedia.org
[2] M Harris, http://beowulf.lcs.mit.edu/18.337-2008/lectslides/scan.pdf