Multicore & GPU Programming: OpenMP tasks

Raymond Namyst, Pierre-André Wacrenier

Dept. of Computer Science

University of Bordeaux, France

https://gforgeron.gitlab.io/it224/

Motivation for introducing tasks in OpenMP

- Limits of “all you need is… loops”
  - Loop have long been considered the main way of sharing work between threads
    - Threads were first-class citizens
  - Not all programs exhibit parallelism in the form of loop iterations
    - Graphs, trees, etc.
  - Composition of parallel codes (nested parallelism) leads to poor performance
- Tasking is a concept already present in several runtime systems/libraries
  - Cilk [MIT]
  - Intel TBB

Tasking

- Tasks are code chunks which are implicitly placed in a “pool of task” to be executed in parallel
  - Task are generated using the `#pragma omp task` directive
  - Task execution is potentially postponed until it get picked by a thread
    - Task scheduling is performed by a dynamic runtime system

```
{
  {
    printf ("Start\n");

    printf ("Middle (executed by %d)\n",
            omp_get_thread_num ());

    printf ("End\n");
  }
}
```

Tasking

- Tasks are code chunks which are implicitly placed in a “pool of task” to be executed in parallel
  - Task are generated using the `#pragma omp task` directive
  - Task execution is potentially postponed until it get picked by a thread
    - Task scheduling is performed by a dynamic runtime system

```
{
#pragma omp parallel
  {
    printf ("Start\n");

#pragma omp task
    printf ("Middle (executed by %d)\n",
            omp_get_thread_num ());

    printf ("End\n");
  }
}
```

Tasking

- In this example
  - Each thread generates one task
  - Tasks can be executed by any thread
- All tasks must complete before the next synchronization point
  - Barrier
  - End of parallel region

```
{
#pragma omp parallel
  {
    printf ("Start\n");

#pragma omp task
    printf ("Middle (executed by %d)\n",
            omp_get_thread_num ());

    printf ("End\n");
  }
}
```

See `first-task.c` and `second-task.c`

Tasking

- In the general case, we don't want all these tasks duplicates
  - Only one thread generates tasks
    - `#pragma omp single`
      - Only one thread executes the code, i.e. generates tasks
      - The others wait on an implicit barrier
    - See `single.c`
  - All threads cooperate to empty the pool of ready-tasks

```
#pragma omp parallel
#pragma omp single
{
#pragma omp task
  {
    printf ("Task 1 executed by Thread %d\n", omp_get_thread_num ());
    sleep (1);
    printf ("End of Task 1\n");
  }
#pragma omp task
  {
    printf ("Task 2 executed by Thread %d\n", omp_get_thread_num ());
    sleep (1);
    printf ("End of Task 2\n");
  }
}
```

See `task.c`

Tasking

- Warning
  - The following code behaves quite differently!

```
#pragma omp parallel
{
#pragma omp single
#pragma omp task
  {
    printf ("Task 1 executed by Thread %d\n", omp_get_thread_num ());
    sleep (1);
    printf ("End of Task 1\n");
  }
#pragma omp single
#pragma omp task
  {
    printf ("Task 2 executed by Thread %d\n", omp_get_thread_num ());
    sleep (1);
    printf ("End of Task 2\n");
  }
}
```

See `task.c`

Tasking

- Warning
  - The following code behaves quite differently!
  - Adding nowait allows task creations to take place in parallel

```
#pragma omp parallel
{
#pragma omp single nowait
#pragma omp task
  {
    printf ("Task 1 executed by Thread %d\n", omp_get_thread_num ());
    sleep (1);
    printf ("End of Task 1\n");
  }
#pragma omp single
#pragma omp task
  {
    printf ("Task 2 executed by Thread %d\n", omp_get_thread_num ());
    sleep (1);
    printf ("End of Task 2\n");
  }
}
```

See `task.c`

Tasking

- Now we can generate parallelism from within `while` loops

```
#pragma omp parallel
#pragma omp single
{
  int k = 0;

  while (k < 25) {
#pragma omp task firstprivate (k)
    test_prime (k);

    k++;
 }
}
```

See `primes.c`

Tasking

- More generally, we can handle an arbitrary number of elements
  - Not known *a priori*

```
#pragma omp parallel
#pragma omp single
{
  element_t elt;

  while (elt = get_next ())
#pragma omp task firstprivate (elt)
    treat (elt);
}
```

Tasking

- Caution!
  - Tasks are “tied” by default
    - Tasks are tied to the 1st thread that start their execution
      - Codes using `omp_get_thread_num` are guaranteed to stick to the same thread
    - When a tied task is interrupted, no other thread can continue its execution…

```
#pragma omp parallel
#pragma omp single
{
  element_t elt;

  while (elt = get_next ())
#pragma omp task firstprivate (elt)
    treat (elt);
}
```

Implicit task

Tasking

- Caution!
  - Tasks are “tied” by default
    - Tasks are tied to the 1st thread that start their execution
      - Codes using `omp_get_thread_num` are guaranteed to stick to the same thread
    - When a tied task is interrupted, no other thread can continue its execution…

```
#pragma omp parallel
#pragma omp single
{
  element_t elt;

  while (elt = get_next ())
#pragma omp task firstprivate (elt)
    treat (elt);
}
```

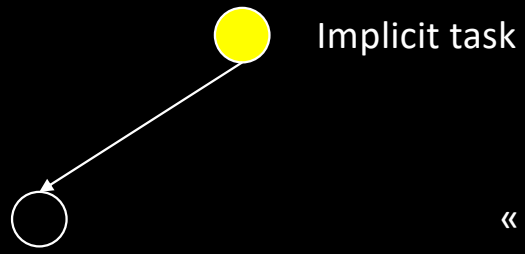
Implicit task

Tasking

- Caution!
  - Tasks are “tied” by default
    - Tasks are tied to the 1st thread that start their execution
      - Codes using `omp_get_thread_num` are guaranteed to stick to the same thread
    - When a tied task is interrupted, no other thread can continue its execution…

```
#pragma omp parallel
#pragma omp single
{
  element_t elt;

  while (elt = get_next ())
#pragma omp task firstprivate (elt)
    treat (elt);
}
```

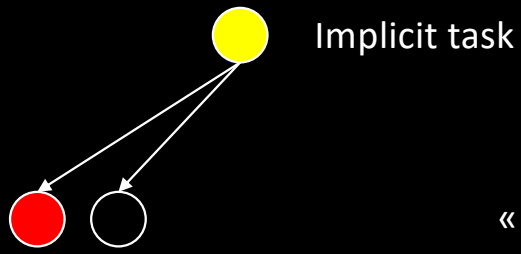


Tasking

- Caution!
  - Tasks are “tied” by default
    - Tasks are tied to the 1st thread that start their execution
      - Codes using `omp_get_thread_num` are guaranteed to stick to the same thread
    - When a tied task is interrupted, no other thread can continue its execution…

```
#pragma omp parallel
#pragma omp single
{
  element_t elt;

  while (elt = get_next ())
#pragma omp task firstprivate (elt)
    treat (elt);
}
```

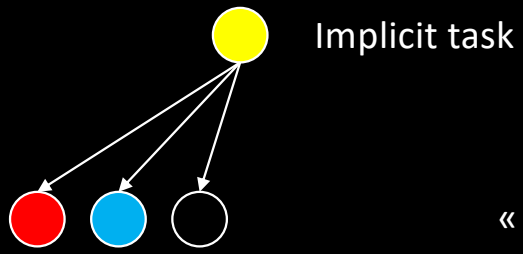


Tasking

- Caution!
  - Tasks are “tied” by default
    - Tasks are tied to the 1st thread that start their execution
      - Codes using `omp_get_thread_num` are guaranteed to stick to the same thread
    - When a tied task is interrupted, no other thread can continue its execution…

```
#pragma omp parallel
#pragma omp single
{
  element_t elt;

  while (elt = get_next ())
#pragma omp task firstprivate (elt)
    treat (elt);
}
```

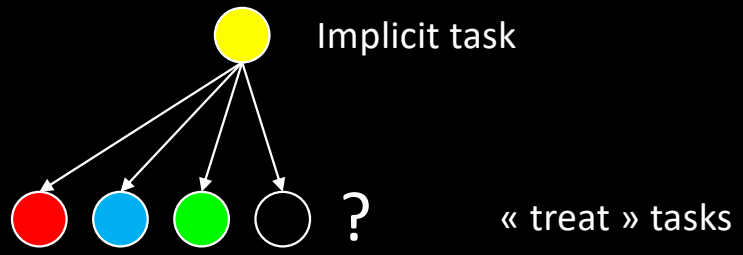


Tasking

- Caution!
  - Tasks are “tied” by default
    - Tasks are tied to the 1st thread that start their execution
      - Codes using `omp_get_thread_num` are guaranteed to stick to the same thread
    - When a tied task is interrupted, no other thread can continue its execution…

```
#pragma omp parallel
#pragma omp single
{
  element_t elt;

  while (elt = get_next ())
#pragma omp task firstprivate (elt)
    treat (elt);
}
```

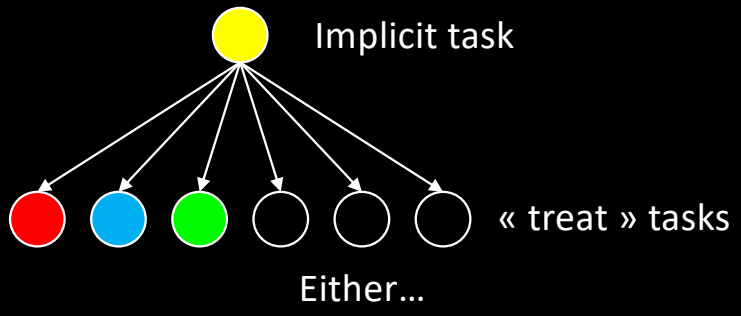


Tasking

- Caution!
  - Tasks are “tied” by default
    - Tasks are tied to the 1st thread that start their execution
      - Codes using `omp_get_thread_num` are guaranteed to stick to the same thread
    - When a tied task is interrupted, no other thread can continue its execution…

```
#pragma omp parallel
#pragma omp single
{
  element_t elt;

  while (elt = get_next ())
#pragma omp task firstprivate (elt)
    treat (elt);
}
```

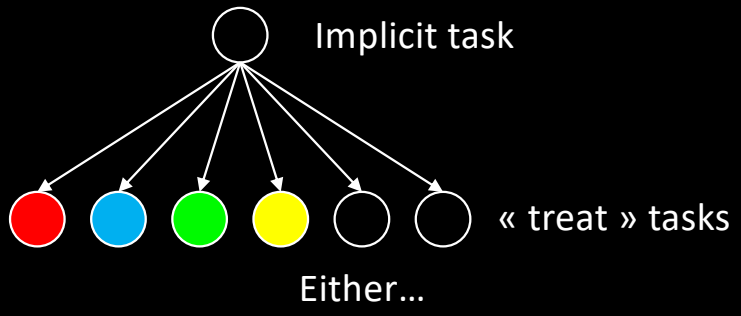


Tasking

- Caution!
  - Tasks are “tied” by default
    - Tasks are tied to the 1st thread that start their execution
      - Codes using `omp_get_thread_num` are guaranteed to stick to the same thread
    - When a tied task is interrupted, no other thread can continue its execution…

```
#pragma omp parallel
#pragma omp single
{
  element_t elt;

  while (elt = get_next ())
#pragma omp task firstprivate (elt)
    treat (elt);
}
```

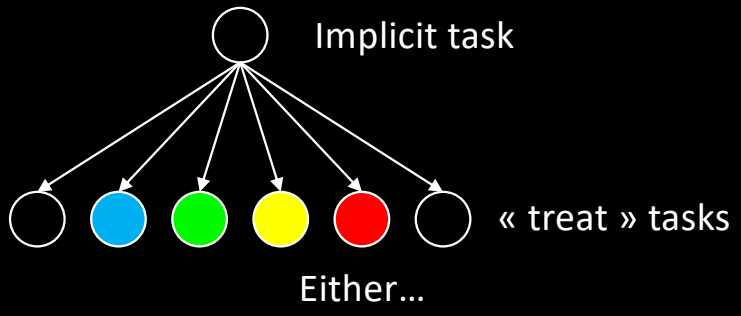


Tasking

- Caution!
  - Tasks are “tied” by default
    - Tasks are tied to the 1st thread that start their execution
      - Codes using `omp_get_thread_num` are guaranteed to stick to the same thread
    - When a tied task is interrupted, no other thread can continue its execution…

```
#pragma omp parallel
#pragma omp single
{
  element_t elt;

  while (elt = get_next ())
#pragma omp task firstprivate (elt)
    treat (elt);
}
```

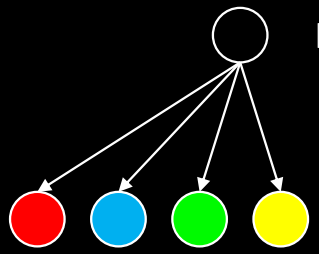


Tasking

- Caution!
  - Tasks are “tied” by default
    - Tasks are tied to the 1st thread that start their execution
      - Codes using `omp_get_thread_num` are guaranteed to stick to the same thread
    - When a tied task is interrupted, no other thread can continue its execution…

```
#pragma omp parallel
#pragma omp single
{
  element_t elt;

  while (elt = get_next ())
#pragma omp task firstprivate (elt)
    treat (elt);
}
```



Implicit task

« treat » tasks

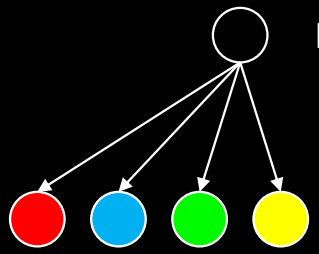
…or

Tasking

- Caution!
  - Tasks are “tied” by default
    - Tasks are tied to the 1st thread that start their execution
      - Codes using `omp_get_thread_num` are guaranteed to stick to the same thread
    - When a tied task is interrupted, no other thread can continue its execution…

```
#pragma omp parallel
#pragma omp single
{
  element_t elt;

  while (elt = get_next ())
#pragma omp task firstprivate (elt)
    treat (elt);
}
```



Implicit task

« treat » tasks

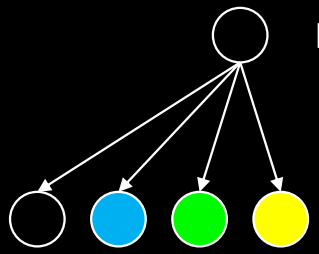
And we’re potentialy stuck for a long time!

Tasking

- Caution!
  - Tasks are “tied” by default
    - Tasks are tied to the 1st thread that start their execution
      - Codes using `omp_get_thread_num` are guaranteed to stick to the same thread
    - When a tied task is interrupted, no other thread can continue its execution…

```
#pragma omp parallel
#pragma omp single
{
  element_t elt;

  while (elt = get_next ())
#pragma omp task firstprivate (elt)
    treat (elt);
}
```



Implicit task

« treat » tasks

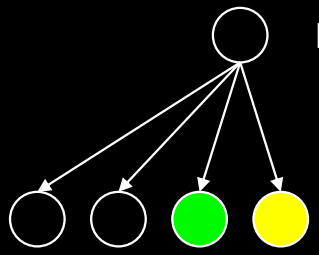
Because only the yellow thread can execute the implicit task

Tasking

- Caution!
  - Tasks are “tied” by default
    - Tasks are tied to the 1st thread that start their execution
      - Codes using `omp_get_thread_num` are guaranteed to stick to the same thread
    - When a tied task is interrupted, no other thread can continue its execution…

```
#pragma omp parallel
#pragma omp single
{
  element_t elt;

  while (elt = get_next ())
#pragma omp task firstprivate (elt)
    treat (elt);
}
```



Implicit task

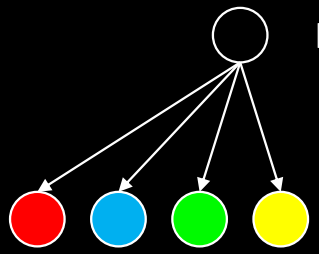
« treat » tasks

Because only the yellow thread can execute the implicit task

Tasking

- Caution!
  - Tasks are “tied” by default
    - Tasks are tied to the 1st thread that start their execution
      - Codes using `omp_get_thread_num` are guaranteed to stick to the same thread
    - When a tied task is interrupted, no other thread can continue its execution…

```
#pragma omp parallel
#pragma omp single
{
  element_t elt;
#pragma omp task untied
  while (elt = get_next ())
#pragma omp task firstprivate (elt)
    treat (elt);
}
```



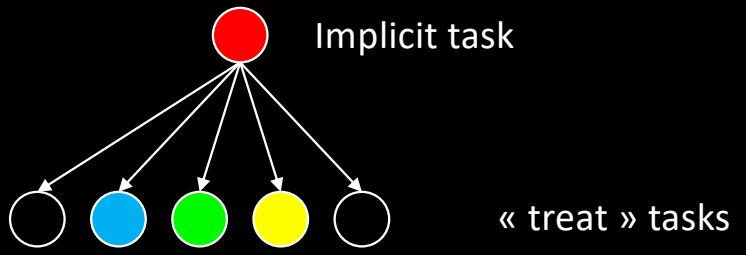
Implicit task

« treat » tasks

Tasking

- Caution!
  - Tasks are “tied” by default
    - Tasks are tied to the 1st thread that start their execution
      - Codes using `omp_get_thread_num` are guaranteed to stick to the same thread
    - When a tied task is interrupted, no other thread can continue its execution…

```
#pragma omp parallel
#pragma omp single
{
  element_t elt;
#pragma omp task untied
  while (elt = get_next ())
#pragma omp task firstprivate (elt)
    treat (elt);
}
```



Recursive parallelism

- Fibonacci
  - Computing the n^{th} Fibonacci number the recursive way

```
int fib_seq (int n)
{
  if (n < 2)
    return n;

  int r1, r2;

  r1 = fib_seq (n – 1);

  r2 = fib_seq (n – 2);

  return r1 + r2;
}
```

See `fib.c`

Recursive parallelism

- Fibonacci
  - Computing the n^{th} Fibonacci number the ~~recursive~~ worst way

```
int fib_seq (int n)
{
  if (n < 2)
    return n;

  int r1, r2;

  r1 = fib_seq (n – 1);

  r2 = fib_seq (n – 2);

  return r1 + r2;
}
```

See `fib.c`

Recursive parallelism

- Fibonacci
  - Computing the n^{th} Fibonacci number the ~~recursive~~ worst

```
#pragma omp parallel shared(r)
#pragma omp single
  r = fib_par (n);
```

```
int fib_par (int n)
{
  if (n < 2)
    return n;

  int r1, r2;
#pragma omp task shared (r1)
  r1 = fib_par (n - 1);
#pragma omp task shared (r2)
  r2 = fib_par (n - 2);

  return r1 + r2;
}
```

See `fib.c`

Recursive parallelism

- Fibonacci
  - Computing the n^{th} Fibonacci number the recursive way

```
#pragma omp parallel shared(r)
#pragma omp single
  r = fib_par (n);
```

```
int fib_par (int n)
{
  if (n < 2)
    return n;

  int r1, r2;
#pragma omp task shared (r1)
  r1 = fib_par (n - 1);
#pragma omp task shared (r2)
  r2 = fib_par (n - 2);

  return r1 + r2;
}
```

See `fib.c`

Recursive parallelism

- The `taskwait` directive
  - Waits completion of *child* tasks
    - Ignore childs of childs…
      - In the case of Fibonacci, taskwait is performed at each level, so it does not matter

```
int fib_par (int n)
{
  if (n < 2)
    return n;

  int r1, r2;
#pragma omp task shared (r1)
  r1 = fib_par (n - 1);
#pragma omp task shared (r2)
  r2 = fib_par (n - 2);
#pragma omp taskwait
  return r1 + r2;
}
```

See `fib.c`

Recursive parallelism

- The `taskwait` directive
  - Waits completion of *child* tasks
    - Ignore childs of childs…
      - In the case of Fibonacci, taskwait is performed at each level, so it does not matter
- Note: for big values of n, the function creates a lot of tasks!

```
int fib_par (int n)
{
  if (n < 2)
    return n;

  int r1, r2;
#pragma omp task shared (r1)
  r1 = fib_par (n - 1);
#pragma omp task shared (r2)
  r2 = fib_par (n - 2);
#pragma omp taskwait
  return r1 + r2;
}
```

See `fib.c`

Recursive parallelism

- The `taskwait` directive
  - Waits completion of *child* tasks
    - Ignore childs of childs…
      - In the case of Fibonacci, taskwait is performed at each level, so it does not matter
- Note: for big values of n, the function creates a lot of tasks!
  - Conditional task creation

```
int fib_par (int n)
{
  if (n < 2)
    return n;

  int r1, r2;
#pragma omp task shared (r1) if (n > 11)
  r1 = fib_par (n - 1);
#pragma omp task shared (r2) if (n > 12)
  r2 = fib_par (n - 2);
#pragma omp taskwait
  return r1 + r2;
}
```

See `fib.c`

taskwait vs taskgroup

```
  #pragma omp task
  {
    #pragma omp task
    f ();
    #pragma omp task
    g ();
  }
  #pragma omp task
    h ();
  #pragma omp taskwait
// Only h () is guaranteed to be completed
```

taskwait vs taskgroup

```
  #pragma omp task
  {
    #pragma omp task
    f ();
    #pragma omp task
    g ();
  }
  #pragma omp task
    h ();
  #pragma omp taskwait
// Only h () is guaranteed to be completed
```

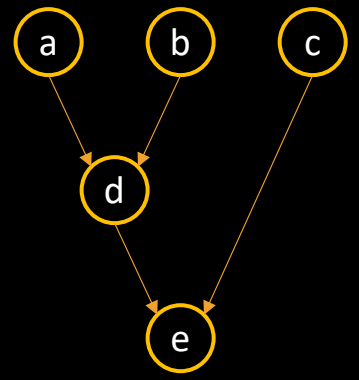
```
#pragma omp taskgroup
{
  #pragma omp task
  {
    #pragma omp task
    f ();
    #pragma omp task
    g ();
  }
  #pragma omp task
    h ();
}
// f(), g() and h () are guaranteed to be
// completed
```

Task dependencies

- In some situations, we need a tighter control on synchronizations
- Say we want to *taskify* the following code
  - Where to insert `taskwait` directives?

```
{
  int a, b, c, d, e;
  {
    a = fa ();
    b = fb ();
    c = fc ();
    d = fadd (a, b);
    e = fmul (c, d);
  }

  printf ("result = %d\n", e);
}
```



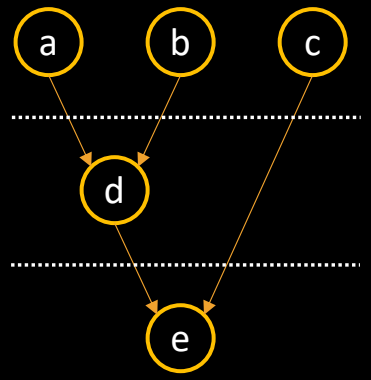
See `flow.c`

Task dependencies

- In some situations, we need a tighter control on synchronizations
- Say we want to *taskify* the following code
  - Where to insert `taskwait` directives?

```
{
  int a, b, c, d, e;
  {
    a = fa ();
    b = fb ();
    c = fc ();
    d = fadd (a, b);
    e = fmul (c, d);
  }

  printf ("result = %d\n", e);
}
```



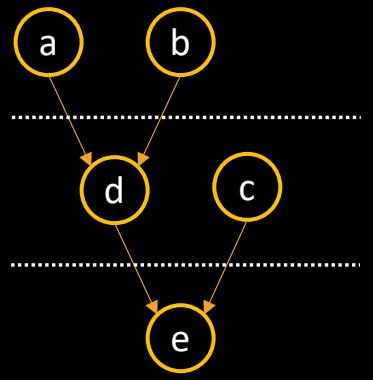
See `flow.c`

Task dependencies

- In some situations, we need a tighter control on synchronizations
- Say we want to *taskify* the following code
  - Where to insert `taskwait` directives?

```
{
  int a, b, c, d, e;
  {
    a = fa ();
    b = fb ();
    c = fc ();
    d = fadd (a, b);
    e = fmul (c, d);
  }

  printf ("result = %d\n", e);
}
```



See `flow.c`

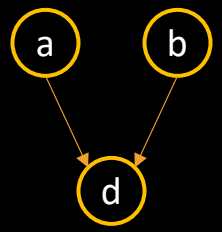
Task dependencies

- Implicit task dependencies can be inferred by OpenMP
  - By specifying in/out/inout accesses to “variables”
- depend clause
  - depend (out: v)
    - The task modifies v
  - depend (in: v)
    - The task reads v
  - depend (mutexinoutset: v)
    - Only one task accessing v can run at a time, but no specific order is required

```
#pragma omp task shared (a) depend (out: a)
  a = fa ();

#pragma omp task shared (b) depend (out: b)
  b = fb ();

#pragma omp task shared (d) depend (out: d) depend (in: a, b)
  d = fadd (a, b);
```



See `flow-depend.c`

Task dependencies

- Dependencies only apply to tasks which have the same parent task
- Depend clauses only use the address of variables internally
  - OpenMP uses addresses as keys to match in/out/inout clauses
  - Variables are not accessed
- OpenMP drops depend(in: v) if no depend(out: v) was previously encountered…

More to come about OpenMP

- Support for hierarchical memory
  - Non-Uniform Memory Access architectures (NUMA)
- Support for accelerators
  - Offloading
- Support for SIMD processors
- Dependencies between loop indexes
  - Ordered clause

Additional resources
available on
http://gforgeron.gitlab.io/it224/