Motivation for the **task** Primitive

- We have seen
  - `parallel` (creates a team of threads)
  - `for` inside `parallel`
    - And `parallel for` as a short-form
  - `sections` inside parallel
- The `for` construct works on parallelizing bodies of loop
  - Identical code (identical work, mostly) on different data
  - The number of iterations (i.e., number of similar tasks) must be fixed
- What about situations when the number of pieces of work is not known a-priory?
  - E.g., traverse list, graph, or tree and do some computation for each node
  - E.g., generate parallelizable work as you execute existing parallel work
- And you want the system to automate work-sharing
The Task Construct

```c
#pragma omp task [clauses...]
structured-block
```

- Conceptually, this enqueues an entry corresponding to the “task” of executing the structured-block into a pool/queue of tasks, from which any thread in the team can execute it.

```c
#pragma omp taskwait
```

- Wait for the completion of child tasks of current task.
  - The current task suspends until then.
Tasking

• OpenMP task is generated when task directive is encountered
  • and the “if” clause evaluates to true, if it exists

• Task directive defines the code being executed and the data environment (shared/firstprivate, etc.)

• Task execution can be
  • Immediate
  • Deferred

• A deferred task is not necessarily executed by the thread that creates it
  • Any member of the current team may execute it
int main( )
{
    int x = 0;
    int n = 30;
    #pragma omp parallel shared(n, x)
    {
        #pragma omp single
        x = fib(n);
    }

    printf("fib(%d) = %d\n", n, x);
    return 0;
}
```c
int main( )
{
    int x = 0;
    int n = 30;
    #pragma omp parallel shared(n, x)
    {
        #pragma omp single
        x = fib(n);
    }
    printf("fib(%d) = %d
", n, x);
    return 0;
}

int fib(int n) {
    int i, j;
    if (n == 0)
        return 0;
    else if (n <= 2)
        return 1;
    else {
        #pragma omp task shared(i) if (n > 20)
        i = fib(n-1);
        #pragma omp task shared(j) if (n > 20)
        j = fib(n-2);
        #pragma omp taskwait
        return i+j;
    }
}
```

L.V.Kale
Clauses for the `task` Directive

- **The usual** – `firstprivate, private, shared, default`
- **if** – allows user-specified condition for whether to spawn task or just do the work
- **untied** – the task created can be executed by different threads over a period of time
  - (i.e., it can “yield” the thread to allow it do other things)
- **priority(value)** – hint to the system about picking which task to work on
  - Higher number is higher priority
    - (This may be reverse of what some of you may expect, from Unix priorities, for example)
- **depend(type:list)** – in, out, inout
**depend Clause and Creating a DAG of Tasks**

- Examples from the OpenMP 4.5 Example document:

```c
#include <stdio.h>
int main(){
    int x = 1;
    #pragma omp parallel
    #pragma omp single
    {
        #pragma omp task shared(x) depend(out: x)
        x = 2;
        #pragma omp task shared(x) depend(in: x)
        printf("x + 1 = %d. ", x+1);
        #pragma omp task shared(x) depend(in: x)
        printf("x + 2 = %d\n", x+2);
    }
    return 0;
}
```

From
OpenMP Application Programming Interface: Examples
[https://www.openmp.org/specifications/](https://www.openmp.org/specifications/)

Copyright © 1997-2016 OpenMP Architecture Review Board

L.V.Kale
Exercise

```c
int f(a) {
    x = g(a);
    y = h(a);
    z = foo(x, y);
    t = bar(x);
    u = last(z, t);
}
```

Add task primitives with `depend` clauses for each of the statements, so that they compute the same results as the sequential code would, assuming none of the functions change any global variables.
// Assume BS divides N perfectly
void matmul_depend(int N, int BS, float A[N][N], float B[N][N], float C[N][N])
{
    int i, j, k, ii, jj, kk;
    for (i = 0; i < N; i+=BS) {
        for (j = 0; j < N; j+=BS) {
            for (k = 0; k < N; k+=BS) {
                // Note 1: i, j, k, A, B, C are firstprivate by default
                // Note 2: A, B and C are just pointers
                #pragma omp task private(ii, jj, kk) \n
                depend ( in: A[i:BS][k:BS], B[k:BS][j:BS] ) \n                depend ( inout: C[i:BS][j:BS] )
                for (ii = i; ii < i+BS; ii++)
                    for (jj = j; jj < j+BS; jj++)
                        for (kk = k; kk < k+BS; kk++)
            }
        }
    }
}
Exercise: Gauss-Seidel Using Tasks

• Recall the formulation with row decomposition and tiles of width $w$ that we did earlier (using flush primitive)
• Redo that with tasks and dependences on the tiles
• Analyze cache performance issues