High Performance Computing: Tools and Applications

Edmond Chow
School of Computational Science and Engineering
Georgia Institute of Technology

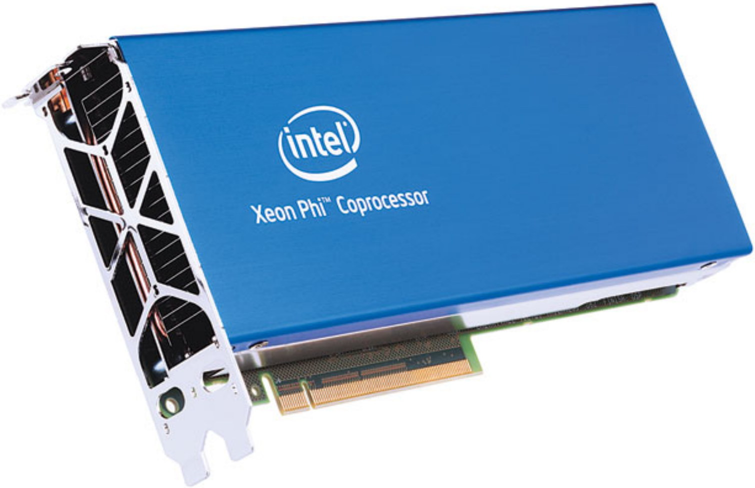
Lecture 10

Thread affinity

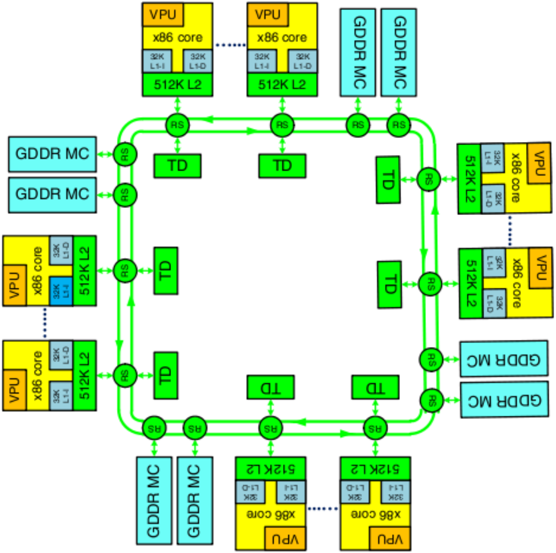
Don't forget about:

```
KMP_AFFINITY=verbose,none
KMP_AFFINITY=verbose,compact
KMP_AFFINITY=verbose,granularity=fine,compact
```

Intel Xeon Phi



Intel Xeon Phi



Logically an UMA architecture.

OpenMP and non-loop-based programs

OpenMP is good at parallelizing loops.

What if we want to parallelize

- while loops where we don't know the number of iterations in advance
- traversing linked lists
- recursive functions
- batch of tasks, where performing a task can create new tasks

OpenMP task-based parallelism

- OpenMP has the facility to define a pool of tasks
- These tasks are executed by threads when threads are free
- Tasks can be added to the pool dynamically

First example

```
#pragma omp parallel
#pragma omp single
{
  printf("Start creating tasks by thread %d\n",omp_get_thread_num());

  #pragma omp task
  printf("1st task executed by thread %d\n",omp_get_thread_num());

  #pragma omp task
  printf("2nd task executed by thread %d\n",omp_get_thread_num());

  #pragma omp task
  printf("3rd task executed by thread %d\n",omp_get_thread_num());

  printf("Done creating tasks\n");

  // taskwait is needed if a barrier is desired
  #pragma omp taskwait
  printf("All tasks completed by now.\n");
}
```

`omp task`

- the task may be run by a thread immediately or added to pool of tasks to be executed later (depends on runtime system)
- tasks are not consumed in any order
- when a task is created, it takes up resources including memory for private variables
- some `omp task` clauses
  - `if`
  - `untied`
  - `default`
  - `mergeable`
  - `private`
  - `firstprivate`
  - `shared`

`private` and `firstprivate` in tasks

- if a variable is private in the enclosing context, then the default is that it is `firstprivate` in the task
- if `task` clause is `private`, then initial copy is not performed

Second example – there is a bug in this code

```
int taskid;

#pragma omp parallel
#pragma omp single
{
  for (int i=0; i<10; i++) {
    taskid = i+1;

    #pragma omp task
    printf("thread %2d: task %2d\n",
      omp_get_thread_num(), taskid);
  }

  printf("Done creating tasks\n");
}

// implied barrier
int threadid = omp_get_thread_num();
if (threadid == 0) printf("All tasks completed by now.\n");
```

Bug fix: `taskid` needs to be (first)private within the task

```
#pragma omp parallel
#pragma omp single
{
  for (int i=0; i<10; i++) {
    int taskid = i+1; // taskid is private

    #pragma omp task // for each task, taskid is threadprivate
    printf("thread %2d: task %2d\n",
      omp_get_thread_num(), taskid);
  }

  printf("Done creating tasks\n");
}

// implied barrier
int threadid = omp_get_thread_num();
if (threadid == 0) printf("All tasks completed by now.\n");
```

Fibonnaci numbers (recursive algorithm)

```
// 1 1 2 3 5 8 13 ...

int fib(int n)
{
    int x, y;
    if (n < 2) return n;
    x = fib(n-1);
    y = fib(n-2);
    return x+y;
}

void main()
{
    int n = 20;
    printf("fib(%d) = %d\n", n, fib(n));
}
```

Fibonnaci numbers with tasks

```
int fib(int n) {
    int x, y;
    if (n < 2) return n;
#pragma omp task shared(x)
    x = fib(n-1);
#pragma omp task shared(y)
    y = fib(n-2);
#pragma omp taskwait
    return x+y;
}

void main() {
    int n = 20;
#pragma omp parallel
#pragma omp single
    printf("fib(%d) = %d\n", n, fib(n));
}
```

Note: without `shared` clause, `x` and `y` would be `firstprivate` in the tasks because `x` and `y` are private in the enclosing context.

Cilk Plus features of Intel compilers

- Cilk Plus defines three keywords
  - `cilk_spawn`
  - `cilk_sync`
  - `cilk_for`
- Philosophy is for the programmer to expose parallelism, and let the runtime decide how to optimize thread scheduling, vectorization, etc.
- Cilk Plus is arguably more task-based than OpenMP. Task-stealing, which is transparent to the user, is used for balancing load.
- Cilk Plus also defines array notation (facilitates both multithreading and vectorization) and reducers (parallel data types that help avoid the use of locks)
- On Intel compilers, simply include Cilk Plus header files, generally `cilk/cilk.h` for nicer keywords. No special compilation flags are needed

Fibonnaci numbers with Cilk Plus

```
int fib(int n)
{
    int x, y;
    if (n < 2) return n;
    x = cilk_spawn fib(n-1);
    y = fib(n-2); // No cilk_spawn needed here.
    cilk_sync;    // Block here until all
                  // spawned functions are complete.
    return x+y;
    // Implied cilk_sync at end of any function
    // that contains cilk_spawn.
}

void main()
{
    int n = 20;
    printf("fib(%d) = %d\n", n, fib(n));
}
```

What happens with `cilk_spawn`

- Only functions can be spawn, not sections of code. This makes it easier (than OpenMP) to define the data environemnt (e.g., no need for `threadprivate`). This also makes it unnecessary to use pragmas to define sections of code.
- Each thread has its own work queue
- When a thread encounters `cilk_spawn`, then a task, which is the *continuation* of the original task is added to the end of its own work queue; the thread executes the code that is *spawn*
- When a thread has finished a task, it takes a new task from the *end* of the queue (this is better for cache usage)
- When threads do not have work, they *steal* tasks from other threads from the *front* of their queues
- In this sense, Cilk Plus does not force parallel execution, but provides the opportunity for parallel execution
- Like OpenMP, removing the keywords should give a correct serial program

cilk_for

```
int i;
cilk_for (i=0; i<10; i++)
{
    int id = __cilkrts_get_worker_number();
    printf("iteration %2d: worker %2d\n", i, id);
}
```

- Using the worker number is discouraged in Cilk Plus
- Programmer should not worry about how threads are scheduled (trade-off between easier programming and performance)

But how is `cilk_for` parallelized?

- Divide and conquer
- The thread encountering does the following:
  - makes a task that is the first half of the iterations; adds task to its queue
  - for the remaining iterations, repeat the above, by making a task that is the first half of the remaining iterations, and adding this task to the queue
  - etc.
- Free threads will steal tasks from the front of another task's queue
- How much to steal?
  - stealing half of the total work is good for balancing load
  - this means stealing *one* item (more efficient than stealing many items)

Cilk Plus reducers

- To sum an array of numbers with multiple threads, locks or similar mechanisms are needed
- *Reducers* in Cilk Plus are C++ classes that perform reduction operations without the need for the programmer to use locks or critical sections

reducer-opadd-demo.cpp from Cilk Plus Tutorial

```
// Sum the numbers 1-1000 in parallel,
// adding a pause to allow the continuation to be stolen
cilk::reducer< cilk::op_add<int> > parallel_sum(0);
cilk_for(int i = 0; i < 1000; i++)
{
    if (0 == i % 10)
        stall();
    *parallel_sum += i;
}
printf("Parallel sum: %d\n", parallel_sum.get_value());
```

Reference: www.cilkplus.org

How is reduction implemented in Cilk Plus?

How is reduction implemented in Cilk Plus?

- Reduction is performed between two threads when they join.
- Thus the reduction is performed like a binary tree.

Cilk Plus array notation

Array notation: `A[start:length:stride]`

Array notation implies independent operations and vector code will be generated

```
A[:] = 5;       // set all elements of static array
A[7:3] = 4;     // set elements 7, 8, 9
A[1:5:2] = 4;   // set elements 1, 3, 5, 7, 9

A[:] = B[:] * C[:] + 5; // Cilk assumes no overlap
C[X][:] = A[:]; // X is an expression

C[:] = A[B[:]]; // gather
A[B[:]] = C[:]; // scatter

C[:][:] = 12;   // two-dimensional arrays
func(A[:]);     // pass elements one-by-one

__sec_reduce_add(A[:])  // returns scalar
```

Reference: `www.cilkplus.org`

Cilk Plus array notation

```
int a[array_size];
const char *results[array_size];

if (5 == a[:])
    results[:] = "Matched";
else
    results[:] = "Not Matched";
```

is equivalent to

```
int a[array_size];
const char *results[array_size];

for (int i = 0; i < array_size; i++)
{
    if (5 == a[i])
        results[i] = "Matched";
    else
        results[i] = "Not Matched";
}
```

Reference: `www.cilkplus.org`

Setting number of threads

- environment variable: `CILK_NWORKERS`
- at run time: `__cilkrts_set_param("nworkers","N")`

Administrative note

- Class on Tuesday, Sept. 27 is cancelled.