CSC2/458 Parallel and Distributed Systems
High-level Parallel Programming Models

Sreepathi Pai

March 6, 2018

URCS

Outline

Higher Level Parallel Programming

OpenMP

Cilk

Outline

Parallel Programming Models

- What can be executed in parallel?
  - Parallelization
- Where and when should be it executed?
  - Scheduling
  - Synchronization

Low-level Parallel Programming

- Parallelization
  - Threading
  - Multiprocessing
- Scheduling
  - Manual (thread pinning, etc.)
- Synchronization
  - Manual (locks, barriers, etc.)

Higher-level Parallel Programming Models

- Make parallel programming easy
  - for parallel programmers
- Without "overburdening"
  - Programming language designers
  - Compiler writers
  - Computer architects

Building your own parallel programming models

- Identify parallel patterns
  - Embarrassingly parallel
  - Reductions
  - Producer–Consumer
  - etc.
- Provide structures for these parallel patterns
  - e.g. parallel `for` loop – `forall`
  - Programmers use these structures
- Execute in parallel
  - Implement structures using lower-level programming models

Outline

OpenMP

- Aimed at shared memory multithreaded programming
  - Within a single process and single computer node
- Usually coupled with Message Passing Interface (MPI)
  - MPI is used for parallelism across processes and machines
  - Not required for OpenMP programs, will discuss MPI in distributed systems

OpenMP programming model

- C/C++ and Fortran support
  - Must be supported by compiler
  - gcc/clang both support OpenMP
- Supports programmer-created threads
  - But do not need to use them

Vector Addition

```
void vector_add(int *a, int *b, int *c, int N) {
   for(int i = 0; i < N; i++) {
       c[i] = a[i] + b[i]
   }
}
```

How would you parallelize this loop using pthreads?

Parallel Programming Pattern in Vector Addition

- Create T threads
- Distribute N/T work to each thread
  - N is size of array (or work, or loop iterations, etc.)
- Pattern
  - Embarrassingly Parallel
  - Data Decomposition
  - Also called "map" pattern

Vector Addition using OpenMP

```
 #include <omp.h>

void vector_add(int *a, int *b, int *c, int N) {
    #pragma omp parallel for
    for(int i = 0; i < N; i++) {
       c[i] = a[i] + b[i]
    }
    /* implicit barrier after loop*/
}
```

Vector Sum

```
void vector_sum(int *a, int N) {
   int sum = 0;

   for(int i = 0; i < N; i++)  {
       sum += a[i];
   }
}
```

How would you parallelize this loop using pthreads?

Parallel Programming Pattern in Vector Sum

- Create T threads
- Distribute N/T work to each thread
  - N is size of array (or work, or loop iterations, etc.)
- Wait until each thread computes sum
- Compute the sum of these individual sums (recursively and in parallel)
- Pattern
  - Reduction
  - Requires associative operator
  - Also called "reduce"

Vector Sum using OpenMP?

```
void vector_sum(int *a, int N) {
   int sum = 0;

   #pragma omp parallel for
   for(int i = 0; i < N; i++)  {
       sum += a[i];
   }
}
```

Correct?

Vector Sum using OpenMP

```
void vector_sum(int *a, int N) {
   int sum = 0;

   #pragma omp parallel for reduction(+:sum)
   for(int i = 0; i < N; i++)  {
       sum += a[i];
   }
   /* OpenMP aggregates sum across all threads */
}
```

Other operators also supported: +, *, -, max, min, &, &&, etc.

Low-level constructs in OpenMP

- Thread creation
  - `#pragma omp parallel`
  - (without `for`)
- Critical Sections
  - `#pragma omp critical`
- Many others
  - Usually better integrated with C/C++ than pthreads
- Other "task parallel" constructs too

Outline

Cilk programming model

- Also C/C++
- Two keywords:
  - `cilk_spawn`
  - `cilk_sync`

Fibonacci calculation example

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

Fibonacci calculation in Cilk

```
int fib(int n)
{
    if (n < 2)
        return n;
    int x = cilk_spawn fib(n-1);
    int y = fib(n-2);

    cilk_sync;

    return x + y;
}
```

- `cilk_spawn` will allow `fib(n-1)` to execute in parallel with `fib`
- `cilk_sync` will wait for all spawns from *current* function to finish
  - All functions contain an implicit `cilk_sync` at exit
- `cilk_spawn` is like `fork`, `cilk_sync` is like `join`
  - However, these are suggestions, may not actually execute in parallel

Scheduling in Cilk

- `cilk_spawn` creates two tasks
  - Actually one task, and one *continuation*
- How are these tasks mapped to cores?
  - Tasks can produce other tasks

Work stealing in Cilk - I

- Assume each core has a deque (double-ended queue)
- Tasks spawned (and their continuations) are added to the deque of a core
    - And pulled off the **head** of deque and executed
- Will this keep all cores busy?

Work stealing in Cilk - II

- Assume core runs out of tasks
- It randomly picks another core
    - And pulls off a task from the **tail** of the dequeue and executes it
- Will this keep all cores busy?

Embarrassingly parallel loops in Cilk

```
for (int i = 0; i < 8; ++i)
{
    cilk_spawn do_work(i);
}
cilk_sync;
```

- Apparently not a good idea in Cilk
  - Why?

Embarrassingly parallel loops in CilkPlus

```
cilk_for (int i = 0; i < 8; ++i)
{
    do_work(i);
}
cilk_sync;
```

Acknowledgements

Cilk/Cilkplus material from the Cilkplus tutorial