Using Threads for Parallelism

CS 475

Introduction

- Goal: connecting multiple computers to get higher performance
  - Multiprocessors
  - Scalability, availability, power efficiency
- Job-level (process-level) parallelism
  - High throughput for independent jobs
- Parallel processing program
  - Single program run on multiple processors
- Multicore microprocessors
  - Chips with multiple processors (cores)
Hardware and Software

- Hardware
  - Serial: e.g., Pentium 4
  - Parallel: e.g., quad-core Xeon e5345
- Software
  - Sequential: previous classes
  - Concurrent: this class
- Sequential/concurrent software can run on serial/parallel hardware
  - Challenge: making effective use of parallel hardware

Parallel Programming

- Parallel software is the problem
- Need to get significant performance improvement
  - Otherwise, just use a faster uniprocessor, since it’s easier!
- Difficulties
  - Partitioning
  - Coordination
  - Communications overhead
Amdahl's Law

- Sequential part can limit speedup
- Example: 100 processors, 90x speedup?
  - \( T_{\text{new}} = \frac{T_{\text{parallelizable}}}{100} + \frac{T_{\text{sequential}}}{1 - F_{\text{parallelizable}}} + \frac{F_{\text{parallelizable}}}{100} = 90 \)
  - Solving: \( F_{\text{parallelizable}} = 0.999 \)
- Need sequential part to be 0.1% of original time

Shared Memory

- SMP: shared memory multiprocessor
  - Hardware provides single physical address space for all processors
  - Synchronize shared variables using locks
  - Memory access time
    - UMA (uniform) vs. NUMA (nonuniform)
**Message Passing**
- Each processor has private physical address space
- Hardware sends/receives messages between processors

![Diagram of Message Passing](image)

**Loosely Coupled Clusters**
- Network of independent computers
  - Each has private memory and OS
  - Connected using I/O system
    - E.g., Ethernet/switch, Internet
- Suitable for applications with independent tasks
  - Web servers, databases, simulations, ...
- High availability, scalable, affordable
- Problems
  - Administration cost (prefer virtual machines)
  - Low interconnect bandwidth
    - c.f. processor/memory bandwidth on an SMP
**Grid Computing**

- Separate computers interconnected by long-haul networks
  - E.g., Internet connections
  - Work units farmed out, results sent back

- Can make use of idle time on PCs
  - E.g., SETI@home, World Community Grid

**Using threads for parallelism**

- Shared memory multiprocessors, multicores

- Two common approaches
  - Partition “work” into $t$ portions and then assign each of $t$ different threads to work on its own region
  - “Bag of tasks” approach
    - When partitioning work equally among threads in advance is difficult
Example: Parallel Sum

```c
void *sum(void *vargp);

long psum[MAXTHREADS];

long nelems_per_thread; /* Number of elements summed by each thread */

int main(int argc, char **argv) {
    long i, nelems, log_nelems, nthreads, result = 0;
    pthread_t tid[MAXTHREADS];
    int myid[MAXTHREADS];

    if (argc != 3) {
        printf("Usage: %s <nthreads> <log_nelems>\n", argv[0]);
        exit(0);
    }
    nthreads = atoi(argv[1]);
    log_nelems = atoi(argv[2]);
    nelems = (1L << log_nelems);

    if ((nelems % nthreads) != 0 || (log_nelems > 31)) {
        printf("Error: invalid nelems\n");
        exit(0);
    }

    nelems_per_thread = nelems / nthreads;

    for (i = 0; i < nthreads; i++) {
        myid[i] = i;
        pthread_create(&tid[i], NULL, sum, &myid[i]);
    }
    for (i = 0; i < nthreads; i++)
        pthread_join(tid[i], NULL);

    for (i = 0; i < nthreads; i++)
        result += psum[i];

    if (result != (nelems * (nelems-1))/2)
        printf("Error: result=%ld\n", result);
        exit(0);
}
```

Parallel Sum cont'd

```c
nelems_per_thread = nelems / nthreads;

for (i = 0; i < nthreads; i++) {
    myid[i] = i;
    pthread_create(&tid[i], NULL, sum, &myid[i]);
}
for (i = 0; i < nthreads; i++)
    pthread_join(tid[i], NULL);

for (i = 0; i < nthreads; i++)
    result += psum[i];

if (result != (nelems * (nelems-1))/2)
    printf("Error: result=%ld\n", result);
    exit(0);
```
Parallel Sum cont’d

```c
void *sum(void *vargp) {
    int myid = *((int *)vargp); /* Extract the thread ID */

    long start = myid * nelems_per_thread; /* Start element index */
    long end = start + nelems_per_thread; /* End element index */
    long i, sum = 0;

    for (i = start; i < end; i++) {
        sum += i;
    }
    psum[myid] = sum;
    return NULL;
}
```

Bag of Tasks approach

- Coordinator thread maintains a work queue or data structure from which worker threads remove tasks
- Easier to implement load balancing when amount of computation associated with task is hard to estimate in advance

```c
while (true) {
    get a task from the bag;
    if (no more tasks)
        break;  // exit the while loop
    execute the task, possibly generating new ones;
}
```

Outline of worker processes using the bag-of-tasks paradigm.