Recap of Last Lecture

- There are several standard programming models (plus variations) that were developed to support particular kinds of architectures
  - shared memory
  - message passing
  - data parallel
- The programming models are no longer strictly tied to particular architectures, and so offer portability of correctness
- Portability of performance still depends on tuning for each architecture
- In each model, parallel programming has 4 phases
  - decomposition into parallel tasks
  - assignment of tasks to threads
  - orchestration of communication and synchronization among threads
  - mapping threads to processors
Outline

- Performance modeling and tradeoffs
- Shared memory architectures
- Shared memory programming

Cost Modeling and Performance Tradeoffs
Example

° \( s = f(A[1]) + \ldots + f(A[n]) \)

° Decomposition
  • computing each \( f(A[i]) \)
  • \( n \)-fold parallelism, where \( n \) may be \( >> p \)
  • computing sum \( s \)

° Assignment
  • thread \( k \) sums \( s_k = f(A[k*n/p]) + \ldots + f(A[(k+1)*n/p-1]) \)
  • thread 1 sums \( s = s_1 + \ldots + s_p \)
    - for simplicity of this example, will be improved
  • thread 1 communicates \( s \) to other threads

° Orchestration
  • starting up threads
  • communicating, synchronizing with thread 1

° Mapping
  • processor \( j \) runs thread \( j \)

Identifying enough Concurrency

° Parallelism profile
  • area is total work done

Simple Decomposition:
\( f(A[i]) \) is the parallel task
sum is sequential

° Amdahl’s law bounds speedup
  • let \( s \) = the fraction of total work done sequentially

\[
\text{Speedup}(P) \leq \frac{1}{s + \frac{1-s}{P}} \leq \frac{1}{s}
\]

After mapping

CS267 L4 Shared Memory.5  Demmel Sp 1999
**Algorithmic Trade-offs**

- **Parallelize partial sum of the f’s**
  - what fraction of the computation is “sequential”
  
  ![Diagram](image)

- what does this do for communication? locality?
- what if you sum what you “own”

**Problem Size is Critical**

- **Total work= n + P**
- **Serial work: P**
- **Parallel work: n**
- **s = serial fraction**
  \[ s = \frac{P}{n+P} \]
- **Speedup(P)=n/(n/P+P)**
- **Speedup decreases for large P if n small**

In general seek to exploit a fraction of the peak parallelism in the problem.
Algorithmic Trade-offs

- Parallelize the final summation (tree sum)
  - Generalize Amdahl's law for arbitrary “ideal” parallelism profile

\[
\text{Concurrency} \\
p \times \frac{n}{p} \times \text{time}(f) \\
\log_2 p \times \text{time}(\text{sum}(2))
\]

Shared Memory Architectures
Recap Basic Shared Memory Architecture

- Processors all connected to a large shared memory
- Local caches for each processor
- Cost: much cheaper to cache than main memory

Limits of using Bus as Network

Assume 100 MB/s bus
50 MIPS processor w/o cache
=> 200 MB/s inst BW per processor
=> 60 MB/s data BW at 30% load-store

Suppose 98% inst hit rate and 95% data hit rate (16 byte block)
=> 4 MB/s inst BW per processor
=> 12 MB/s data BW per processor
=> 16 MB/s combined BW
\[ \therefore \text{8 processors will saturate bus} \]

Cache provides bandwidth filter
- as well as reducing average access time
**Cache Coherence: The Semantic Problem**

- p1 and p2 both have cached copies of x (as 0)
- p1 writes x=1 and then the flag, f=1, as a signal to other processors that it has updated x
  - writing f pulls it into p1’s cache
  - both of these writes “write through” to memory
- p2 reads f (bringing it into p2’s cache) to see if it is 1, which it is
- p2 therefore reads x, expecting the value written by p1, but gets the “stale” cached copy

- SMPs have complicated caches to enforce coherence

---

**Programming SMPs**

- Coherent view of shared memory
- All addresses equidistant
  - don’t worry about data partitioning
- Caches automatically replicate shared data close to processor
- If program concentrates on a block of the data set that no one else updates => very fast
- Communication occurs only on cache misses
  - cache misses are slow
- Processor cannot distinguish communication misses from regular cache misses
- Cache block may introduce unnecessary communication
  - two distinct variables in the same cache block
  - false sharing
Where are things going

° High-end
  • collections of almost complete workstations/SMP on high-speed network (Millennium)
  • with specialized communication assist integrated with memory system to provide global access to shared data

° Mid-end
  • almost all servers are bus-based CC SMPs
  • high-end servers are replacing the bus with a network
    - Sun Enterprise 10000, IBM J90, HP/Convex SPP
  • volume approach is Pentium pro quadpack + SCI ring
    - Sequent, Data General

° Low-end
  • SMP desktop is here

° Major change ahead
  • SMP on a chip as a building block

Programming Shared Memory Machines

° Creating parallelism in shared memory models
° Synchronization
° Building shared data structures
° Performance programming (throughout)
**Programming with Threads**

- Several Threads Libraries
  - PTHREADS is the Posix Standard
    - Solaris threads are very similar
    - Relatively low level
    - Portable but possibly slow
  - P4 (Parmacs) is a widely used portable package
    - Higher level than Pthreads
- OpenMP is new proposed standard
  - Support for scientific programming on shared memory
  - Currently a Fortran interface
  - Initiated by SGI, Sun is not currently supporting this
  - [http://www.openMP.org](http://www.openMP.org)

---

**Creating Parallelism**
Language Notions of Thread Creation

- **cobegin/coend**
  
  ```
  cobegin
  job1(a1);
  job2(a2);
  coend
  ```

  - Statements in block may run in parallel
  - cobegins may be nested
  - Scoped, so you cannot have a missing coend

- **fork/join**
  
  ```
  tid1 = fork(job1, a1);
  job2(a2);
  join tid1;
  ```

  - Forked function runs in parallel with current
  - join waits for completion (may be in different function)

- cobegin cleaner, but fork is more general

Forking Threads in Solaris

**Signature:**

```c
int thr_create(void *stack_base, size_t stack_size,
               void *(*start_func)(void *),
               void *arg, long flags, thread_t *new_tid)
```

**Example:**

```c
thr_create(NULL, NULL, start_func, arg, NULL, &tid)
```

- **start_fun** defines the thread body
- **start_fun** takes one argument of type void* and returns void*
- an argument can be passed as arg
  - j-th thread gets arg=j so it knows who it is
- **stack_base** and **stack_size** give the stack
  - standard default values
- **flags** controls various attributes
  - standard default values for now
- **new_tid** thread id (for thread creator to identify threads)
Basic Types of Synchronization: Barrier

Barrier -- global synchronization

- fork multiple copies of the same function “work”
  - SPMD “Single Program Multiple Data”
- simple use of barriers -- a threads hit the same one
  
  ```
  work_on_my_subgrid();
  barrier;
  read_neighboring_values();
  barrier;
  ```

- more complicated -- barriers on branches
  
  ```
  if (tid % 2 == 0) {
      work1();
      barrier
  } else {
      barrier
  }
  ```

- or in loops -- need equal number of barriers executed
- barriers are not provided in many thread libraries
  - need to build them
**Basic Types of Synchronization: Mutexes**

**Mutexes -- mutual exclusion aka locks**

- threads are working mostly independently
- need to access common data structure

```c
lock *l = alloc_and_init(); /* shared */
acquire(l);
access data
release(l);
```

- Java and other languages have lexically scoped synchronization
  - similar to cobegin/coend vs. fork and join

- Semaphores give guarantees on “fairness” in getting the lock, but the same idea of mutual exclusion

- Locks only affect processors using them:
  - pair-wise synchronization

---

```c
#define _REENTRANT
#include <synch.h>
/* Data Declarations */
typedef struct {
    int     maxcnt ;                /* maximum number of runners */
    struct _sb {
        cond_t  wait_cv;        /* cv for waiters at barrier */
        mutex_t wait_lk;        /* mutex for waiters at barrier */
        int     runners;        /* number of running threads */
    } sb[2];
    struct _sb      *sbp;           /* current sub-barrier */
} barrier_t;

int barrier_init( ... int count, ... ) {
    bp->maxcnt = count;
    ...
}
```
Barrier Implementation Example (Cont)

```c
int barrier_wait( register barrier_t *bp ) {  
    ...  
    mutex_lock( &sbp->wait_lk );  
    if ( sbp->runners == 1 ) { /* last thread to reach barrier */  
        if ( bp->maxcnt != 1 ) { /* reset runner count and switch sub-barriers */  
            sbp->runners = bp->maxcnt;  
            bp->sbp = ( bp->sbp == &sbp[0] )? &sbp[1] : &sbp[0];  
            /* wake up the waiters */  
            cond_broadcast( &sbp->wait_cv );  
        } else {  
            sbp->runners--; /* one less runner */  
            while ( sbp->runners != bp->maxcnt )  
                cond_wait( &sbp->wait_cv, &sbp->wait_lk );  
        }  
    }  
    mutex_unlock( &sbp->wait_lk );  
}
```

Sharks and Fish

http://www.cs.berkeley.edu/~demmel/cs267/Sharks_and_Fish/