Computer Science 400
Parallel Processing
Siena College
Fall 2008

Topic Notes: POSIX Threads

Before considering our first parallelization paradigm, POSIX threads, we will think about what code can be parallelized and how we can find opportunities for concurrency.

Finding Concurrency

We find opportunities for parallelism by looking for parts of the sequential program that can be run in any order.

Before we look at the matrix-matrix multiply, we step back and look at a simpler example:

```
1: a = 10;
2: b = a + 5;
3: c = a - 3;
4: b = 7;
5: a = 3;
6: b = c - a;
7: print a, b, c;
```

Which statements can be run in a different order (or concurrently) but still produce the same answers at the end?

- 1 has to happen before 2 and 3, since they depend on a having a value.
- 2 and 3 can happen in either order.
- 4 has to happen after 2, but it can happen before 3.
- 5 has to happen after 2 and 3, but can happen before 4.
- 6 has to happen after 4 (so 4 doesn't clobber its value) and after 5 (because it depends on its value)
- 7 has to happen last.

This can be formalized into a set of rules called *Bernstein's conditions* to determine if a pair of tasks can be executed in parallel:

Two tasks P_1 and P_2 can execute in parallel if all three of these conditions hold:

1. $I_1 \cap O_2 = \emptyset$
2. $I_2 \cap O_1 = \emptyset$

3. $O_1 \cap O_2 = \emptyset$

where I_i and O_i are the input and output sets, respectively, for task i (Bernstein, 1966). The *input set* is the set of variables read by a task and the *output set* is the set of variables modified by a task.

Back to our example, let's see what can be done concurrently.

```
/* initialize matrices, just fill with junk */
for (i=0; i<SIZE; i++) {
  for (j=0; j<SIZE; j++) {
    a[i][j] = i+j;
    b[i][j] = i-j;
  }
}

/* matrix-matrix multiply */
for (i=0; i<SIZE; i++) {  /* for each row */
  for (j=0; j<SIZE; j++) { /* for each column */
    /* initialize result to 0 */
    c[i][j] = 0;

    /* perform dot product */
    for(k=0; k<SIZE; k++) {
      c[i][j] = c[i][j] + a[i][k]*b[k][j];
    }
  }
}

sum=0;
for (i=0; i<SIZE; i++) {
  for (j=0; j<SIZE; j++) {
    sum += c[i][j];
  }
}
```

The initialization can all be done in any order – each `i` and `j` combination is independent of each other, and the assignment of `a[i][j]` and `b[i][j]` can be done in either order.

In the actual matrix-matrix multiply, each `c[i][j]` must be initialized to 0 before the sum can start to be accumulated. Also, iteration `k` of the inner loop can only be done after row `i` of `a` and column `j` of `b` have been initialized.

Finally, the `sum` contribution of each `c[i][j]` can be added as soon as that `c[i][j]` has been computed, and after `sum` has been initialized to 0.

That *granularity* seems a bit cumbersome, so we might step back and just say that we can initialize `a` and `b` in any order, but that it should be completed before we start computing values in `c`. Then

we can initialize and compute each `c[i][j]` in any order, but we do not start accumulating `sum` until `c` is completely computed.

But all of these dependencies in this case can be determined by a relatively straightforward computation. Seems like a job for a compiler!

In the example, if we add the flag `-xparallel` to the compile command, the Sun compiler will determine what can be done in parallel and generate code to support it. With this executable, we can request a number of parallel processes by setting the environment variable `PARALLEL`. For example:

```
setenv PARALLEL 4
```

One of our goals is to use parallelism to solve a problem more quickly than we could solve it on a single processor executing a sequential program. We would like to see a *speedup* of our program as we add processors.

Quinn defines speedup and *efficiency* on p. 160.

$$\text{Speedup} = \frac{\text{Sequential execution time}}{\text{Parallel execution time}}$$

$$\text{Efficiency} = \frac{\text{Sequential execution time}}{\text{Processors Used} \times \text{Parallel execution time}}$$

We can define these more formally using Quinn's notation:

- We measure performance in terms of a problem size, n, and a number of processors used, p.
- $\sigma(n)$ is the time for the inherently sequential part of the program.
- $\phi(n)$ is the time for the parallelizable part of the program.
- $\kappa(n)$ is the time for the overhead introduced by a parallel execution.
- $\sigma(n) + \phi(n)$ is the sequenial running time.
- $\sigma(n) + \dfrac{\phi(n)}{p} + \kappa(n)$ is the ideal parallel running time.
- $\psi(n, p)$ is the speedup:

$$\psi(n, p) \leq \frac{\sigma(n) + \phi(n)}{\sigma(n) + \frac{\phi(n)}{p} + \kappa(n)}$$

  Note that we use "$\leq$" because "=" would require a perfect distribution of work among the processors, which we will see is very difficult to achieve in many cases.

- $\epsilon(n,p)$ is the efficiency:

$$\epsilon(n,p) \leq \frac{\sigma(n)+\phi(n)}{p\left(\sigma(n)+\frac{\phi(n)}{p}+\kappa(n)\right)}$$

or

$$\epsilon(n,p) \leq \frac{\sigma(n)+\phi(n)}{p\sigma(n)+\phi(n)+p\kappa(n)}$$

An efficient program is one that exhibits *linear* speedup – double the number of processors, halve the running time.

The theoretical upper bound on speedup for p processors is p. Anything greater is called *superlinear speedup* – can this happen?

Try this out. Compile the matrix-matrix multiplication example with the `-xparallel` flag on `bullpen` and run it with the `PARALLEL` environment variable to numbers from 1–4. How does this affect the running time? Now, run this version of the program on a 2-processor Solaris node, again with values of `PARALLEL` ranging from 1–4. Finally, run on a 4-processor node and range `PARALLEL` between 1 and 8. What are the running times you get? Why?

Note: Normally, it would be best to run these through a queueing system to ensure exclusive access to the nodes, but given the small class size, we are unlikely to have a problem.

We will return to this example and parallelize it by hand.

Not everything can be parallelized by the compiler:

See: `/cluster/examples/matmult_serial_init`

The new initialization code:

```
for (i=0; i<SIZE; i++) {
  for (j=0; j<SIZE; j++) {
    if ((i == 0) || (j == 0)) {
      a[i][j] = i+j;
      b[i][j] = i-j;
    }
    else {
      a[i][j] = a[i-1][j-1] + i + j;
      b[i][j] = b[i-1][j-1] + i - j;
    }
  }
}
```

can't be parallelized, so no matter how many processors we throw at it, we can't speed it up.

We can see this by repeating our experiment on the 4-processor node. The initialization time remains the same regardless of the number of processors used. We still get good speedups for our matrix-matrix multiplication.

Amdahl's Law

Any parallel program will have some fraction f that cannot be parallelized, leaving $(1 - f)$ that may be parallelized. This means that at best, we can expect running time on p processors to be $f + \frac{1-f}{p}$.

From this, we can state *Amdahl's Law* in terms of maximum achievable speedup:

$$\psi \leq \frac{1}{f + \frac{1-f}{p}}$$

This is an important equation to keep in mind when determining whether to make the effort to parallelize a program, and how many processors are likely to be worthwhile to use to execute it.

Approaches to Parallelism

Automatic parallelism is great, when it's possible. We got it for free (at least once we bought the compiler)! It does have limitations, though:

- some potential parallelization opportunities cannot be detected automatically – can add directives to help (OpenMP – soon)
- bigger complication – this executable cannot run on distributed-memory systems

Parallel programs can be categorized by how the cooperating processes communicate with each other:

- **Shared Memory** – some variables are accessible from multiple processes. Reading and writing these values allow the processes to communicate.
- **Message Passing** – communication requires explicit messages to be sent from one process to the other when they need to communicate.

These are functionally equivalent given appropriate operating system support. For example, one can write message-passing software using shared memory constructs, and one can simulate a shared memory by replacing accesses to non-local memory with a series of messages that access or modify the remote memory.

The automatic parallelization we have seen to this point is a shared memory parallelization, though we don't have to think about how it's done. The main implication is that we have to run the parallelized executable on a computer with multiple processors.

Our first tool for explicit parallelization will be shared memory parallelism using threads.

A Brief Intro to POSIX threads

Multithreading usually allows for the use of shared memory. Many operating systems provide support for threads, and a standard interface has been developed: *POSIX Threads* or *pthreads*.

A good online tutorial is available at `https://computing.llnl.gov/computing/tutorials/pthreads/`.

You read through this and remember that it's there for reference.

A Google search for "pthread tutorial" yields many others.

Pthreads are available on the Solaris nodes in the cluster, and are standard on most modern Unix-like operating systems.

The basic idea is that we can create and destroy threads of execution in a program, on the fly, during its execution. These threads can then be executed in parallel by the operating system scheduler. If we have multiple processors, we should be able to achieve a speedup over the single-threaded equivalent.

We start with a look at a pthreads "Hello, world" program:

See: `/cluster/examples/pthreadhello`

The most basic functionality involves the creation and destruction of threads:

- `pthread_create(3THR)` – This creates a new thread. It takes 4 arguments. The first is a pointer to a variable of type `pthread_t`. Upon return, this contains a thread identifier that may be used later in a call to `pthread_join()`. The second is a pointer to a `pthread_attr_t` structure that specifies thread creation attributes. In the `pthreadhello` program, we pass in `NULL`, which will request the system default attributes. The third argument is a pointer to a function that will be called when the thread is started. This function must take a single parameter of type `void *` and return `void *`. The fourth parameter is the pointer that will be passed as the argument to the thread function.
- `pthread_exit(3THR)` – This causes the calling thread to exit. This is called implicitly if the thread function called during the thread creation returns. Its argument is a return status value, which can be retrieved by `pthread_join()`.
- `pthread_join(3THR)` – This causes the calling thread to block (wait) until the thread with the identifier passed as the first argument to `pthread_join()` has exited. The second argument is a pointer to a location where the return status passed to `pthread_exit()` can be stored. In the `pthreadhello` program, we pass in `NULL`, and hence ignore the value.

Prototypes for pthread functions are in `pthread.h` and programs need to link with `libpthread.a` (use `-lpthread` at link time). When using the Sun compiler, the `-mt` flag should also be specified to indicate multithreaded code.

A slightly more interesting example:

See: `/cluster/examples/proctree_threads`

This example builds a "tree" of threads to a depth given on the command line. It includes calls to `pthread_self()`. This function returns the thread identifier of the calling thread.

Try it out and study the code to make sure you understand how it works.

A bit of extra initialization is necessary to make sure the system will allow your threads to make use of all available processors. It may, by default, allow only one thread in your program to be executing at any given time. If your program will create up to n concurrent threads, you should make the call:

```
pthread_setconcurrency(n+1);
```

somewhere before your first thread creation. The "+1" is needed to account for the original thread plus the n you plan to create.

You may also want to specify actual attributes as the second argument to `pthread_create()`. To do this, declare a variable for the attributes:

```
pthread_attr_t attr;
```

and initialize it with:

```
pthread_attr_init(&attr);
```

and set parameters on the attributes with calls such as:

```
pthread_attr_setscope(&attr, PTHREAD_SCOPE_PROCESS);
```

I recommend the above setting for threads in Solaris.

Then, you can pass in `&attr` as the second parameter to `pthread_create()`.

Any global variables in your program are accessible to all threads. Local variables are directly accessible only to the thread in which they were created, though the memory can be shared by passing a pointer as part of the last argument to `pthread_create()`.

Brief Intro to Critical Sections

As you may have been shown in other contexts, concurrent access to shared variables can be dangerous.

Consider this example:

See: `/cluster/examples/pthread_danger`

Run it with one thread, and we get 100000. What if we run it with 2 threads? On a multiprocessor, it is going to give the wrong answer! Why?

The answer is that we have concurrent access to the shared variable `counter`. Suppose that two threads are each about to execute `counter++`, what can go wrong?

`counter++` really requires three machine instructions: (i) load a register with the value of `counter`'s memory location, (ii) increment the register, and (iii) store the register value back in `counter`'s memory location. Even on a single processor, the operating system could switch the process out in the middle of this. With multiple processors, the statements really could be happening concurrently.

Consider two threads running the statements that modify `counter`:

	Thread A		Thread B
A_1	`R0 = counter;`	B_1	`R1 = counter;`
A_2	`R0 = R0 + 1;`	B_2	`R1 = R1 + 1;`
A_3	`counter = R0;`	B_3	`counter = R1;`

Consider one possible ordering: $A_1\ A_2\ B_1\ A_3\ B_2\ B_3$, where `counter=17` before starting. Uh oh.

What we have here is a *race condition* that can lead to *interference* of the actions of one thread with another. We need to make sure that when one process starts modifying `counter`, that it finishes before the other can try to modify it. This requires *synchronization* of the processes.

When we run it on `bullpen`, a single-processor system, the problem is unlikely to show itself - we almost certainly the correct sum when we run it. However, there is no guarantee that this would be the case. The operating system could switch threads in the middle of the load-increment-store, resulting in a race condition and an incorrect result. Try the program on `bullpen` with dozens of threads and you might start to run into problems.

We need to make those statements that increment `counter` *atomic*. We say that the modification of `counter` is a *critical section*.

There are many solutions to the critical section problem and this is a major topic in an operating systems course. But for our purposes, at least for now, it is sufficient to recognize the problem, and use available tools to deal with it.

The pthread library provides a construct called a *mutex* (short for the *mutual exclusion* that we want to enforce for the access of the `counter` variable) allows us to ensure that only one thread at a time is executing a particular block of code. We can use it to fix our "danger" program:

See: `/cluster/examples/pthread_nodanger`

We declare a mutex like any other shared variable. It is of type `pthread_mutex_t`. Four functions are used:

- `pthread_mutex_init(3THR)` – initialize the mutex and set it to the unlocked state.
- `pthread_mutex_lock(3THR)` – request the lock on the mutex. If the mutex is unlocked, the calling thread acquires the lock. Otherwise, the thread is blocked until the thread that previously locked the mutex unlocks it.
- `pthread_mutex_lock(3THR)` – unlock the mutex.
- `pthread_mutex_destroy(3THR)` – destroy the mutex (clean up memory).

A few things to consider about this:

Why isn't the access to the mutex a problem? Isn't it just a shared variable itself? – Yes, it's a shared variable, but access to it is only through the pthread API. Techniques that are discussed in detail in an operating systems course (and that we may discuss more here) are used to ensure that access to the mutex itself does not cause a race condition.

Doesn't that lock/unlock have a significant cost? – Let's see. We can time the programs we've been looking at:

See: `/cluster/examples/pthread_danger_timed`

See: `/cluster/examples/pthread_nodanger_timed`

Try these out. What are the running times of each version? Perhaps the cost is too much if we're going to lock and unlock that much. Maybe we shouldn't do so much locking and unlocking. In this case, we're pretty much just going to lock again as soon as we can jump back around through the `for` loop again.

This is a good example of the parallel overhead we mentioned earlier (the $\kappa(n, p)$ term in Quinn's formulas).

Here's an alternative:

See: `/cluster/examples/pthread_nodanger_coarse`

In this case, the coarse-grained locking (one thread gets and holds the lock for a long time) should improve the performance significantly. How fast does it run now? But at what cost? We've completely serialized the computation! Only one thread can actually be doing something at a time, so we can't take advantage of multiple processors. If the "computation" was something more significant, we would need to be more careful about the granularity of the locking.