Computer Science 324
Computer Architecture
Mount Holyoke College
Fall 2009

Topic Notes: Programming Parallel Computers

Why Parallel Computing?

Parallel computing sounds simple enough – when one computer isn't powerful enough to solve your problem, use more than one.

Before we start to think about how to use parallelism on a computer, let's think about a parallel approach to solving a "real-world" problem.

- Taking a census of South Hadley.

  One person doing this would visit each house, count the people, and ask whatever questions are supposed to be asked. This person would keep running counts. At the end, this person has gathered everything.

  If there are two people, they can work concurrently. Each visits some houses, and they need to "report in" along the way or at the end to combine their information. But how to split up the work?

  - Each person could do what the individual was originally doing, but would check to make sure each house along the way had not yet been counted.
  - Each person could start at the town hall, get an address that has not yet been visited, go visit it, then go back to the town hall to report the result and get another address to visit. Someone at town hall keeps track of the cumulative totals. This is nice because neither person will be left without work to do until the whole thing is done. This is the *master-slave* method of breaking up the work.
  - The town could be split up beforehand. Each could get a randomly selected collection of addresses to visit. Maybe one person takes all houses with even street numbers and the other all houses with odd street numbers. Or perhaps one person would take everything west of Route 116 and the other everything east of Route 116. The choice of how to divide up the town may have a big effect on the total cost. There could be excessive travel if one person walks right past a house that has not yet been visited. Also, one person could finish completely while the other still has a lot of work to do. This is a *domain decomposition* approach.

- Grading a stack of exams. Suppose each has several questions. Again, assume two graders to start.

  - Each person could take half of the stack. Simple enough. But we still have the potential of one person finishing before the other.

  - Each person could take a paper from the "ungraded" stack, grade it, then put it into the "graded" stack.
  - Perhaps it makes more sense to have each person grade half of the *questions* instead of half of the exams, maybe because it would be unfair to have the same question graded by different people. Here, we could use variations on the approaches above. Each takes half the stack, grades his own questions, then they swap stacks.
  - Or we form a *pipeline*, where each exam goes from one grader to the next to the finished pile. Some time is needed to start up the pipeline and drain it out, especially if we add more graders. These models could be applied to the census example, if different census takers each went to every house to ask different questions.
  - Suppose we also add in a "grade totaler and recorder" person. Does that make any of the approaches better or worse?

- Adding two $1,000,000 \times 1,000,000$ matrices.

  - Each matrix entry in the sum can be computed independently, so we can break this up any way we like. Could use the master-slave approach, though a domain decomposition would probably make more sense. Depending on how many processes we have, we might break it down by individual entries, or maybe by rows or columns.

In each of these cases, we have taken what we might normally think of as a *sequential* process, and taken advantage of the availability of *concurrent processing* to make use of multiple workers (processing units).

Parallelism adds complexity (as we will see in great detail), so why bother?

- we want to solve the same problem but in a shorter time than possible on one processor – goal: speedup
- we want to solve larger problems than can currently be solved at all on a single processor – goal: scale-up
- some algorithms are more naturally expressed or organized concurrently
- and now: that's where performance gains come from in modern processors!

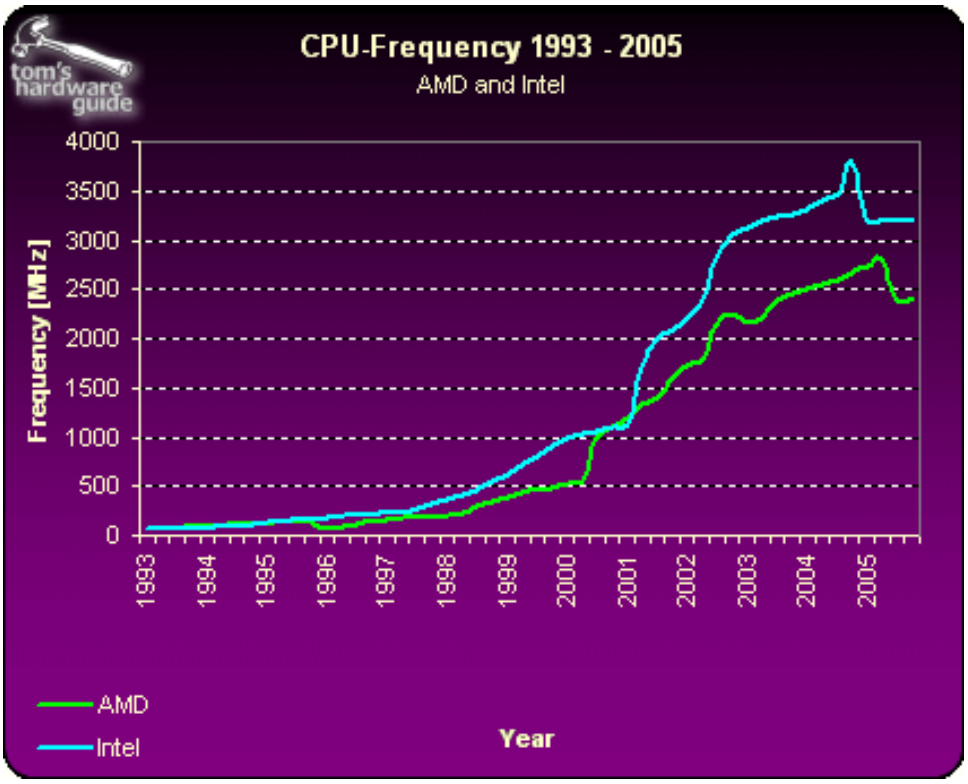


Figure used with permission from article *The Mother of All CPU Charts 2005/2006*, Bert Töpelt, Daniel Schuhmann, Frank Völkel, Tom's Hardware Guide, Nov. 2005, `http://www.tomshardware.com/2005/11/21/the_mother_of_all_cpu_charts_2005/`

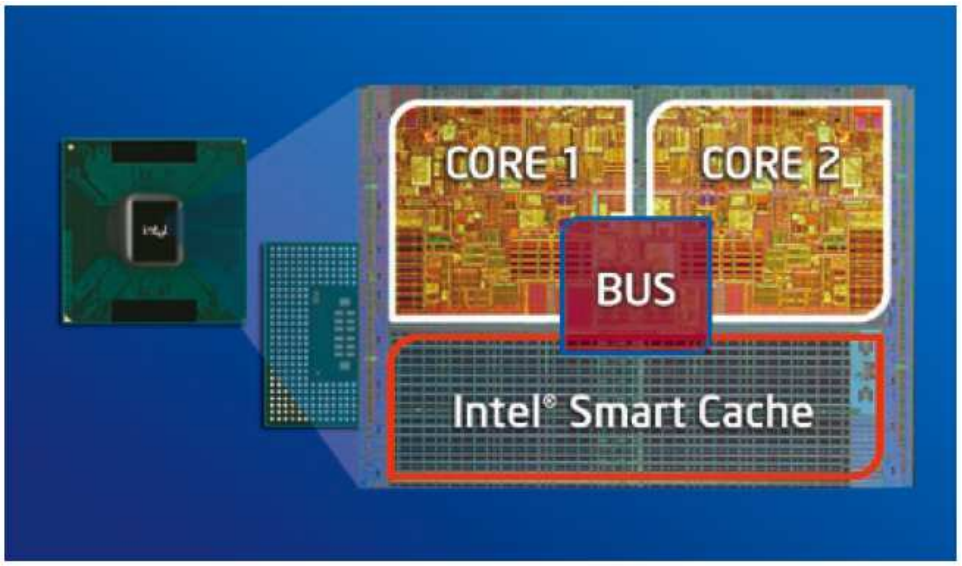


Image from Intel Core Duo Processor product brief.

- The world's fastest computers are all parallel: `http://www.top500.org/`

How to Achieve Parallelism

- We need to determine where concurrency is possible, then break up the work accordingly
- This is easiest if a compiler can do this for you – take your sequential program and extract the concurrency automatically. This is sometimes possible, especially with fixed-size array computations.
- If the compiler can't do it, it is possible to give "hints" to the compiler to tell it what is safe to parallelize.
- But often, the parallelization must be done explicitly: the programmer has to create the threads or processes, assign work to them, and manage necessary communication.

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.

But on to our running example. We start with the matrix-matrix multiplication example we saw earlier in the semester.

See Example:
`/home/jteresco/shared/cs324/examples/matmult`

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 fact, many of the ideas are similar to those employed by optimizing compilers that reorder instructions to reduce data and control hazards in a pipelined processor.

In the example, if we have a parallelizing compiler and add the appropriate flags to the compile command, (such as `-xparallel` for the Sun compiler), it will determine what can be done in parallel and generate code to support it. The resulting executable can request a number of parallel processes. On the Sun systems, this is accomplished by setting the environment variable `PARALLEL`. For example:

```
setenv PARALLEL 4
```

We do not have a parallelizing compiler handy at the moment, but this program should run in somewhere between $\frac{1}{3}$ and $\frac{1}{4}$ of the time for the single-processor run if we run on a parallel computer with 4 processors.

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.

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

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

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?

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

Not everything can be parallelized by the compiler:

See Example:
`/home/jteresco/shared/cs324/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.

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 – a topic we will not discuss today)
- 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 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 Example:
`/home/jteresco/shared/cs324/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. For FreeBSD, the `-pthread` flag is used.

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 Example:
`/home/jteresco/shared/cs324/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 a single-processor system, the problem is unlikely to show itself - we almost certainly get 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 sequence, resulting in a race condition and an incorrect result. Try the program on a uniprocessor 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 Example:
`/home/jteresco/shared/cs324/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 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 Example:
`/home/jteresco/shared/cs324/examples/pthread_danger_timed`

See Example:
`/home/jteresco/shared/cs324/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.

Here's an alternative:

See Example:
`/home/jteresco/shared/cs324/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.

Data Parallel Computation

Tasks such as many scientific computations can be solved using a *data parallel* programming style. A data parallel program is one in which each process executes the same actions concurrently, but on different parts of shared data.

Contrast this with a *task parallel* (or a pipelined) approach, where different processes each perform a different step of the computation on the same data.

An important consideration in a data parallel computation is *load balancing* – making sure that each process/thread has about the same amount of work to do. Otherwise, some would finish before others, possibly leaving available processors idle while other processors continue to work. Load balancing will be an important topic throughout the course. Parallel efficiency and scalability of a data parallel computation will be highly dependent on a good load balance.

This is our first real example of *thread/process synchronization*, which is the main reason parallelism is so hard. These examples were contrived to "encourage" the problem to show up and to emphasize the overhead of the solution, but in a real problem, the interference may be very subtle and show itself only very rarely.

Bag of Tasks Paradigm

One specific way of deciding which processes/threads do the operations on which parts of the data is the *bag of tasks*. In this case, each thread/process is a *worker* that finds a *task* (unit of work) to do (from the "bag"), does it, then goes back for more:

```
while (true) {
  // get a task from the bag
  if (no tasks remain) break;
  //execute the task
}
```

A nice feature of this approach is that load balancing comes for free, as long as we have more tasks than workers. Even if some tasks cost more than others, or some workers work more slowly than others, any available work can be passed out to the first available worker until no tasks remain.

Back to our matrix multiplication example, we can break up the computation into a bag of tasks. We'll choose a fine-grained parallelization, where the computation of each entry is one of our tasks.

See Example:
`/home/jteresco/shared/cs324/examples/matmult_bagoftasks`

Run this on a four-processor node. You should see that it is still pretty slow. Perhaps the granularity of the computation is too small – too much time picking out a task, not enough time doing it. We created 562,500 tasks. This means we have to acquire the lock 562,500 times. How long does this take to run?

We can easily break up our computation by row or column of the result matrix, as well. Here is a row-wise decomposition:

See Example:
`/home/jteresco/shared/cs324/examples/matmult_smallbagoftasks`

You should find that this is much more efficient! We coarsened the parallelism, but kept it fine enough that all of our threads could keep busy. We still had 750 tasks in the bag. How long does this take to run?

Explicit Domain Decomposition

If we could improve things significantly by coarsening the parallelism in the bag of tasks, perhaps we can do even better by dividing up the entire computation ahead of time to avoid any selection from the bag of tasks whatsoever.

With the matrix-matrix multiply example, this is easy enough – we can just give `SIZE/numworkers` rows to each thread, they all go off and compute, and they'll all finish in about the same amount of time:

See Example:
`/home/jteresco/shared/cs324/examples/matmult_decomp`

Some things to notice about this example:

- During the setup, we compute the range of rows that each thread will be responsible for. We can't simply give every thread the same number of rows, in case the number of threads does not divide the matrix size evenly. The computation as shown also guarantees no more than a one-row imbalance.
- We need to tell each thread its range of rows to compute. But thread functions are restricted to a single parameter, of type `void *`. We can use this pointer to pass in anything we want, by putting all of the parameters into a structure.

  Notice that each thread needs its own copy of this structure – we can't just create a single copy, send it to `pthread_create()`, change the values, and use it again. Why?

Explicit domain decomposition works out well in this example, since there's an easy way to break it up (by rows), and each row's computation is equal in cost.

Is there any advantage to breaking it down by columns instead? How about, in the case of 4 threads, into quadrants?

In more complicated examples, load balancing with a domain decomposition may be more difficult.

What about distributed memory?

So far we have seen two ways to create a parallel program:

1. Let the compiler do whatever it can completely automatically
2. Create threads explicitly using pthreads

A third, that we are not discussing here, is to give hints through compiler directives using a library and compiler that supports OpenMP.

These all suffer from one significant limitation – the cooperating threads must be able to communicate through shared variables.

How can we design and run parallel programs to work when there is no shared memory available?

Message Passing

We will now consider the message passing paradigm.

- Characteristics:
  - **Locality** - each processor accesses *only* its local memory
  - **Explicit parallelism** - messages are sent and received explicitly - programmer controls all parallelism. The compiler doesn't do it.
  - **Cooperation** - every send must have a matching receive in order for the communication to take place. *Beware of deadlock!* One sided communication is possible, but doesn't really fit the pure message-passing model.
- Advantages:
  - **Hardware** - many current clusters of workstations and supercomputers fit well into the message passing model, and shared memory systems can run message passing programs as well.
  - **Functionality** - full access to parallelism. We don't have to rely on a compiler. The programmer can decide when parallelism makes sense. But this is also a disadvantage - the full burden is on the programmer! Advice: If the compiler can do it for you, let it!

- **Performance** - data locality is important - especially in a multi-level memory hierarchy which includes off-processor data. There is a chance for superlinear speedup with added cache as we talked about earlier in the course. Communication is often MUCH more expensive than computation.

Message Passing Libraries

- Message passing is supported through a set of library routines. This allows programmers to avoid dealing with the hardware directly. Programmers want to concentrate on the problem they're trying to solve, not worrying about writing to special memory buffers or making TCP/IP calls or even creating sockets.
- **Examples:** P4, PVM, MPL, MPI, MPI-2, etc. MPI and PVM are the most common.
- **Core Functionality:**
  - *Process Management* - start and stop processes, query number of procs or PID.
  - *Point-to-Point Communications* - send/receive between processes
  - *Collective Communication* - broadcast, scatter, gather, synchronization
- **Terminology:**
  - *Buffering* - copy into a buffer somewhere (in library, hardware)
  - *Blocking communication* - wait for some "event" to complete a communication routine
  - *Nonblocking communication* - "post" a message and return immediately
  - *Synchronous communication* - special case of blocking - send does not return until corresponding receive completes
  - *Asynchronous communication* - pretty much nonblocking

Point-to-Point Communication

All message passing is based on the simple `send` and `receive` operations

```
P0: send(addr,len,dest,tag)

P1: receive(addr,max_len,src,tag,rec_len)
```

These are basic components in any message-passing implementation. There may be others introduced by a specific library.

- `addr` is the address of the send/receive buffer
- `len` is the length of the sent message
- `max_len` is the size of the receive buffer (to avoid overflow)
- `rec_len` is the length of the message actually received
- `dest` identifies destination of a message being sent
- `src` identifies desired sender of a message being received (or where it actually came from if "any source" is specified)
- `tag` a user-defined identifier restricting receipt

Point-to-Point Communication - Blocking

Blocking communication has simple semantics:

- `send` completes when send buffers are ready for reuse, after message received or at least copied into system buffers
- `receive` completes when the receive buffer's value is ready to use

But beware of **deadlock** *when using blocking routines!!*

```
       Proc 0          Proc 1
-------------------------------------
    bsend(to 1)    bsend(to 0)
   brecv(from 1)  brecv(from 0)
```

If both processors' send buffers cannot be copied into system buffers, or if the calls are strictly synchronous, the calls will block until the corresponding receive call is made... Neither can proceed... *deadlock*...

Possible solutions - reorder to guarantee matching send/receive pairs, or use *nonblocking routines*...

Point-to-Point Communication - Nonblocking

- `send` or `receive` calls return immediately - but how do we know when it's done? When can we use the value?
- can overlap computation and communication
- must call a `wait` routine to ensure communication has completed before destroying send buffer or using receive buffer

Example:

```
      Proc 0          Proc 1
-------------------------------------
    nbsend(to 1)    nbsend(to 0)
   nbrecv(from 1)  nbrecv(from 0)
   compute......   compute......
     waitall         waitall
    use result      use result
```

During the "compute....." phase, it's possible that the communication can be completed "in the background" while the computation proceeds, so when the "waitall" lines are reached, the program can just continue.

Deadlock is less likely but we still must be careful – the burden of avoiding deadlock is on the programmer in a message passing model of parallel computation.

Collective Communication

Some common operations don't fit well into a point-to-point communication scheme. Here, we may use *collective communication* routines.

- collective communication occurs among a group of processors
- the group can be all or a subset of the processors in a computation
- collective routines are blocking
- types of collective operations
  - *synchronization/barrier* - wait until all processors have reached a given point
  - *data movement* - broadcast (i.e. error condition, distribute read-in values), scatter/gather (exchange boundary on a finite element problem, for example), all-to-all (extreme case of scatter/gather)
  - *reductions* - collect data from all participating processors and operate on it (i.e. add, multiply, min, max)

These kinds of operations can be achieved through a series of point-to-point communication steps, but operators are often provided.

Using collective communication operators provided is generally better than trying to do it yourself. In addition to providing convenience, the message passing library can often perform the operations more efficiently by taking advantage of low-level functionality.

MPI

The Message Passing Interface (MPI) was created by a standards committee in the early 1990's.

- motivated by the lack of a good standard
  - everyone had their own library
  - PVM demonstrated that a portable library was feasible
  - portablity and efficiency were conflicting goals
- The MPI-1 standard was released in 1994, and many implementations (free and proprietary) have since become available
- MPI specifies C and Fortran interfaces (125 functions in the standard), more recently C++ as well
- parallelism is explicit - the programmer must identify parallelism and implement a parallel algorithm using MPI constructs
- MPI-2 is an extention to the standard developed later in the 1990's and there are now some implementations

---

MPI Terminology

- *Rank* – a unique identifier for a process
  - values are 0...$n-1$ when n processes are used
  - specify source and destination of messages
  - used to control conditional execution
- *Group* – a set of processes, associated with a *communicator*
  - processes in a group can take part in a collective communication, for example
  - we often use the predefined communicator specifying the group of all processors in a communication: `MPI_COMM_WORLD`
  - the communicator ensures safe communication within a group - avoid potential conflicts with other messages
- *Application Buffer* - application space containing data to send or received data
- *System Buffer* - system space used to hold pending messages

Major MPI Functions

MPI Simple Program - basic MPI functions

We begin with a simple "Hello, World" program.

See Example:
`/home/jteresco/shared/cs324/examples/mpihello`

MPI calls and constructs in the "Hello, World" program:

- `#include <mpi.h>` - the standard MPI header file
- `MPI_Init(int *argc, char *argv[])` - MPI Initialization
- `MPI_COMM_WORLD` - the global communicator. Use for `MPI_Comm` args in most situations
- `MPI_Abort(MPI_Comm comm, int rc)` - MPI Abort function 3
- `MPI_Comm_size(MPI_Comm comm, int *numprocs)` - returns the number of processes in a given communicator in `numprocs`
- `MPI_Comm_rank(MPI_Comm comm, int *pid)` - returns the rank of the current process in the given communicator
- `MPI_Get_processor_name(char *name, int *rc)` - returns the name of the node on which the current process is running
- `MPI_Finalize()` - clean up MPI

The model of parallelism is very different from what we have seen. All of our processes exist for the life of the program. We are not allowed to do anything before `MPI_Init()` or after `MPI_Finalize()`. We need to think in terms of a number of copies of the *same program* all starting up at `MPI_Init()`.

To run, compile with `mpicc`, run with `mpirun` according to the instructions on the course web page.

MPI Point-to-Point message functions

There are just a few that we'll use frequently:

- `MPI_Send/MPI_Recv` - standard blocking calls (may have system buffer)
- `MPI_Isend/MPI_Irecv` - standard nonblocking calls

- wait calls for nonblocking communications: `MPI_Wait, MPI_Waitall, MPI_Waitsome, MPI_Waitany`

And there are many variations that we won't likely use much:

- `MPI_Ssend/MPI_Issend` - synchronous blocking/nonblocking send
- `MPI_Bsend/MPI_Ibsend` - buffered blocking/nonblocking send - programmer allocates message buffer with `MPI_Buffer_attach`
- `MPI_Rsend/MPI_Irsend` - ready mode send - matching receive *must* have been posted previously
- `MPI_Sendrecv` - combine send/recv into one call before blocking
- also: `MPI_Probe` and `MPI_Test` calls

Blocking Point-to-point Communication

A simple MPI program that sends a single message using blocking communication:

See Example:
`/home/jteresco/shared/cs324/examples/mpimsg`

- All MPI calls return a status value, and it's a good idea to check it as is done in this example for the `MPI_Init` call.
  - Most class examples will not be thorough in this to keep things looking simpler.
  - For our purposes, any MPI error will cause the program to terminate with an error message, so it usually is not that important to us.
  - When developing large-scale software, we often wish to return error codes rather than crash the whole program, so error checking becomes more important there.
  - The error checking includes a messy little chunk of code to print out appropriate messages, so it's probably worth putting this into your own error reporting function if you want to use it.
- `MPI_Status status` - structure which contains additional info following a receive. We often ignore it, but we will see some instances where it comes in handy.
- `MPI_Send(void *buf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm)` - blocking send - does not return until the corresponding receive is completed. sends `count` copies of data of type `type` located in `buf` to the processor with pid `dest`.

- `MPI_Recv(void *buf, int count, MPI_Datatype type, int src, int tag, MPI_Comm comm, MPI_Status status)` - blocking receive - does not return until the message has been received. `src` may be specific PID or `MPI_ANY_SOURCE` which matches, well, a message from any source.
- `MPI_Datatype` examples: `MPI_CHAR, MPI_INT, MPI_LONG, MPI_FLOAT, MPI_DOUBLE, MPI_BYTE, MPI_PACKED`

Non-blocking Point-to-point Communication

A slightly more interesting MPI program that sends one message from each process with non-blocking messages:

See Example:
`/home/jteresco/shared/cs324/examples/mpiring`

- `MPI_Request request` - structure which contains info needed by nonblocking calls to check on their status or to wait for their completion.
- `MPI_Isend(void *buf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm, MPI_Request *req)` - nonblocking send - returns immediately. `buf` must not be modified until a wait function is called using this request.
- `MPI_Irecv(void *buf, int count, MPI_Datatype type, int source, int tag, MPI_Comm comm, MPI_Request *req)` - nonblocking receive - returns immediately. `buf` must not be used until a wait function is called using this request.
- `MPI_Wait(MPI_Request *req, MPI_Status *status)` - wait for completion of message which had `req` as its request argument. Additional info such as source of a message received as `MPI_ANY_SOURCE` is contained in `status`.

The `MPI_ANY_SOURCE` option is used in this modified version of the example:

See Example:
`/home/jteresco/shared/cs324/examples/mpiring_anysource`

Collective Communication

We often need to perform operations at a higher level than simple sends and receives.

See Example:
`/home/jteresco/shared/cs324/examples/mpicoll`

- `MPI_Barrier(MPI_Comm comm)` - synchronize procs
- `MPI_Bcast(void *buf,int count,MPI_Datatype type,int root,MPI_Comm comm)` - broadcast - sends `count` copies of data of type `type` located in `buf` on proc `root` to `buf` on all others.

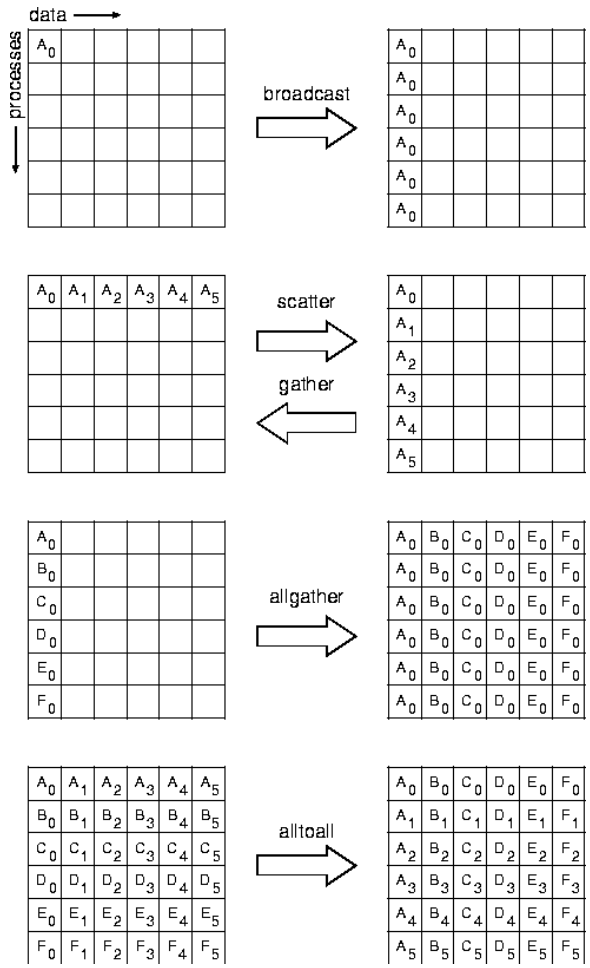
- `MPI_Reduce(void *sendbuf,void *recvbuf,int count, MPI_Datatype type,MPI_Op op,int root,MPI_Comm comm)` - combines data in `sendbuf` on each proc using operation `op` and stores the result in `recvbuf` on proc `root`
- `MPI_Allreduce()` - same as reduce except result is stored in `recvbuf` on all procs
- `MPI_Op` values - `MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, MPI_LAND, MPI_BAND, MPI_LOR, MPI_BOR, MPI_LXOR, MPI_BXOR, MPI_MAXLOC, MPI_MINLOC` plus user-defined
- `MPI_Scan(void *sendbuf, void *recvbuf, int count, MPI_Datatype type, MPI_Op op, MPI_Comm comm)` - parallel prefix scan operations

Scatter/Gather – Higher-level Collective Communication

See Example:
`/home/jteresco/shared/cs324/examples/mpiscatgath`

- `MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)` - `root` sends `sendcount` items from `sendbuf` to each processor. Each processor receives `recvcount` items into `recvbuf`
- `MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)` - each proc sends `sendcount` items from `sendbuf` to `root`. `root` receives `recvcount` items into `recvbuf` from each proc
- `MPI_Scatterv/MPI_Gatherv` work with variable-sized chunks of data
- `MPI_Allgather/MPI_Alltoall` variations of scatter/gather

To understand what is going on with the various broadcast and scatter/gather functions, consider this figure, taken from the MPI Standard, p.91



Sample MPI Applications

Conway's Game of Life

The Game of Life was invented by John Conway in 1970. The game is played on a field of cells, each of which has eight neighbors (adjacent cells). A cell is either occupied (by an organism) or not. The rules for deriving a generation from the previous one are:

- Death: If an occupied cell has 0, 1, 4, 5, 6, 7, or 8 occupied neighbors, it dies (of either boredom or overcrowding, as the case may be)
- Survival: If an occupied cell has 2 or 3 occupied neighbors, it survives to the next generation
- Birth: If an unoccupied cell has 3 occupied neighbors, it becomes occupied.

The game is very interesting in that complex patterns and cycles arise. Do a google search to find plenty of Java applets you can try out.

I like the one here:

`http://www.math.com/students/wonders/life/life.html`

My implementation is not graphical, so it's a lot less fun. It plays the game, but only computes statistics.

Serial version: **See Example:**
`/home/jteresco/shared/cs324/examples/life`

MPI version: **See Example:**
`/home/jteresco/shared/cs324/examples/mpilife`

- Since our memory is not shared, we only allocate enough memory on each process to hold the rows that will be computed by that process, plus a "ghost" row on each side that will allow simple computation of our rows.
- When we need to get a global count of some statistic, such as the count of live cells at the start, we use a reduction.
- The communication is done with two pairs of sends and receives. Here, we use nonblocking calls, then wait for their completion with the waitall call.

Matrix-Matrix Multiplication

Matrix-matrix multiplication using message passing is not as straightforward as matrix-matrix multiplication using shared memory and threads. Why?

- Since our memory is not shared, which processes have copies of the matrices?
- Where does the data start out? Where do we want the answer to be in the end?
- How much data do we replicate?
- What are appropriate MPI calls to make all this happen?

The MPI version of Conway's Game of Life used a distributed data structure. Each process maintains its own subset of the computational domain, in this case just a number of rows of the grid. Other processes do not know about the data on a given process. Only that data that is needed to compute the next generation, a one-cell overlap, is exchanged between iterations.

Think about that – no individual process has all of the information about the computation. It only works because all processes are cooperating.

The "slice by slice" method of distributing the grid was chosen only for its simplicity of implementation, both in the determination of what processes are given what rows, and the straightforward communication patterns that can be used to exchange boundary data. We could partition in more complicated patterns, but there would be extra work involved.

The possiblities for the matrix-matrix multiply are numerous. Now the absolute easiest way to do it would be to distribute the matrix `A` by rows, have `B` replicated everywhere, and then have `C` by rows. If we distributed our matrices this way in the first place, everything is simple:

See Example:
`/home/jteresco/shared/cs324/examples/matmult_mpi_toosimple`

This program has very little MPI communication – this is by design, as we distributed our matrices so that each process would have exactly what it needs.

Unfortunately, this is not likely to be especially useful. More likely, we will want all three matrices distributed the same way.

To make the situation more realistic, but still straightforward, let's assume that our initial matrices `A` and `B` are distributed by rows, in the same fashion as the Life simulator. Further, the result matrix `C` is also to be distributed by rows.

The process that owns each row will do the computation for that row. What information does each process have locally? What information will it need to request from other processes?

Matrix multiplication is a pretty "dense" operation, and we to send all the columns of `B` to all processes.

See Example:
`/home/jteresco/shared/cs324/examples/matmult_mpi_simple`

Note that we only initialize rows of `B` on one process, but since it's all needed on every process, we need to broadcast those rows.

Can we do better? Can we get away without storing all of `B` on each process? We know we need to send it, but we we do all the computation that needs each row before continuing on to the next?

See Example:
`/home/jteresco/shared/cs324/examples/matmult_mpi_better`

Yes, all we had to do was rearrange the loops that do the actual computation of the entries of `C`. We can broadcast each row, use it for everything it needs to be used for, then we move on. We save memory!

Even though we do the exact same amount of communication, our memory usage per process goes from $O(n^2)$ to $O(\frac{n^2}{p})$.