What?

This course is about bridging the gap between the parallel applications and algorithms which we can design and describe in abstract terms and the parallel computer architectures (and their lowest level programming interfaces) which it is practical to construct.

The challenge is to provide programming mechanisms (whether through language constructs or libraries) which provide a good balance between

- **conceptual simplicity**: it should be “easy” to program correctly, and

- **performance retention**: if our algorithm and architecture are good, we shouldn’t lose “too much” in the mapping between them.

This is similar to the sequential computing world, but performance is now central.
Why?

The ability to express parallelism (a.k.a concurrency) concisely, correctly and efficiently is important in several contexts

- **Performance Computing:** when parallelism, at various levels in the system, is the means by which the execution time of computationally demanding applications can be reduced. In the era of static (or even falling) clock speeds and increasing core count, this class is entering the computing mainstream.

- **Distributed Computing:** when concurrency is inherent in the nature of the system and we have no choice but to express and control it.

- **Systems Programming:** when it is conceptually simpler to think of a system as being composed of concurrent components, even though these will actually be executed by time-sharing a single processor (quaint historical concept ;-)).
How?

We begin by briefly reviewing the complex capabilities of realistic parallel architectures, and the conceptual structure and control requirements of a range of typical parallel applications.

We then examine some of the programming primitives and frameworks which have been designed to bridge the gap, considering conceptual purpose, implementation challenges and concrete realisation in real systems.

We will do this first for the two traditional models, shared variable programming, and message passing programming, before considering other approaches and variations.
Parallel Architecture for Dummies

The world of parallel architectures is diverse and complex. We will focus on the mainstream, and note a key division into two architectural classes.

- **Shared Memory** architectures in which all processors can physically address the whole memory, usually with support for cache coherency (for example, a quad or oct core chip, or more expensive machines with tens or hundreds of cores).

- **Multicomputer** architectures in which processors can only physically address their “own” memory (for example, a networked cluster of PCs), which interact with messages across the network.

Increasingly, systems will span both classes (eg cluster of manycore, or network-on-chip manycores like the Intel SCC), and incorporate other specialized, constrained parallel hardware such as GPUs.
Shared Memory Architectures

UMA style

One address space, shared by all CPUs. Green box shows memory addressable by any CPU. Uniform Memory Access (UMA) architectures have all memory “equidistant” from all CPUs. For NUMA (N for “non”) performance varies with data location. NUMA is also confusingly called Distributed Shared Memory as memory is physically distributed but logically shared.

NUMA style
Shared Memory Architectures

Caches improve performance as usual, but raise a memory consistency challenge, which is not present in simple sequential cache systems: roughly speaking, when, and in what order should updates to memory made by one processor become visible to others?

\[ x = 2; \]
\[ y = 1; \]
\[ \text{....} \]
\[ \text{cache write-back of } y \rightarrow \text{memory} \]
\[ \text{....} \]
\[ \text{if } (x \leq y) \]
\[ \text{print("Yes");} \]
Shared Memory Architectures

At the branch, processor sees \( x = 2, y = 1 \), even though main memory contains \( x = 0, y = 1 \).

Does this matter? Not sequentially, but consider this (now parallel).

\[
\text{[shared } x = 0, \text{ shared flag} = 0]\]

\[
\begin{align*}
P0 : & \quad \text{lots of work}(&x); \\
     & \quad \text{flag} = 1;

P1 : & \quad \text{while (flag == 0); //spinning} \\
     & \quad y = f(x);
\end{align*}
\]

Exactly what and when it is permissible for each processor to see is defined by the consistency model.
Shared Memory Architectures

The consistency model (which is effectively a contract between hardware and software) must be respected by application programmers (and compiler/library writers) to ensure program correctness. Different consistency models trade off conceptual simplicity against cost (time/hardware complexity). There have been many schemes, for example:

**Sequential consistency**, every processor “sees” the same sequential interleaving of the basic reads and writes. This is very intuitive, but expensive to implement.

**Release consistency**: writes are only guaranteed to be visible after program-specified synchronization points (triggered by special machine instructions). Even the ordering as written by one processor between such points may be seen differently by other processors. This is less intuitive, but allows faster implementations.
Shared Memory Architectures

Shared memory architectures also raise tricky performance issues. The unit of transfer between memory and cache is a cache-line or block, containing several words. False sharing occurs when two logically unconnected variables share the same cache-line. Updates to one cause remote copies of the line, including the other variable, to be invalidated, creating very expensive, but semantically undetectable, “ping-pong” effects.

```c
shared int x, y;

for 1000000 iterations {
    x = ....not touching y ......
}

P0

for 1000000 iterations {
    y = ....not touching x ......
}

P1
```

These look nicely independent, but may not be physically independent at the level of cache blocks.
Multicomputer Architectures

The same block diagram as for NUMA shared memory! The difference is the lack of any hardware integration between cache/memory system and the interconnect. Each processor only accesses its own physical address space, so no consistency issues. Information is shared by explicit, co-operative message passing.
Multicomputer Architectures

Performance/correctness issues include the semantics of synchronization and constraints on message ordering.

For example, consider the following producer-consumer behaviour

```
while (whatever) {
    x = ...;
    send (x, P1);
}
send(-1, P1);
```

```
recv(y, P0);
while (y!=-1) {
    ...= ....y....;
    recv(y, P0);
}
```

OK as long as the underlying implementation of messaging guarantees order.
Summary: Real parallel machines are complex, with unforeseen semantic and performance traps. We need to provide programming tools which simplify things, but without sacrificing too much performance.
Parallel Applications and Algorithms

To help understand and design good parallel programming models, we must understand the requirements of typical parallel algorithms.

We will shortly introduce, with examples, three well-known parallel patterns: Bag of Tasks, Pipeline and Interacting Peers. All could be implemented for either architectural model, but we will consider only one version of each.

There is a circularity here - want to examine the patterns to understand control requirements, but to explain the patterns we need some control notation! So, we will first introduce some simple notation and an idealised execution model.

NB This is not a real programming language, just a concise way of expressing what we will need mechanisms to say in real languages and libraries.
The co Notation

The co notation indicates creation of a set of activities, for the duration of the enclosed block, with synchronization across all activities at the end of the block. This is sometimes called “fork-join” parallelism. The parallel activities in a simple co statement are separated by // (so we use `##` to indicate comments - sorry!).

```co
co
    a=1; // b=2; // c=3; ## all at the same time
oc
```

We will also use co statements with indices.

```co
co [i = 0 to n-1] {
    a[i] = a[i] + 1; ## all at the same time
}
```
The co Notation

Things get more interesting when the statements within a co access the same locations.

```
c
  a=1; // a=2; // a=3;  ## What is a afterwards?
  ...
```

To resolve this, we need to define our memory consistency model. For our toy language examples we assume sequential memory consistency (SC). We will think about the implications of weakening this model as we go along, and particularly when we turn to real languages which typically don’t support it.
Sequential Memory Consistency Model (SC)

A program executed with an SC model will produce a result which is consistent with the following rules:

1. ordering of atomic actions (particularly reads and writes to memory) from any one thread have to occur in normal program order

2. atomic actions from different threads are interleaved arbitrarily (ie in an unpredictable sequential order, subject only to rule 1), with every thread seeing this same ordering

We will return to the question of what an “atomic action” is in a moment. We will also assume that in an interleaving, no thread is ever permanently excluded from having a turn, but there is no bound on how far apart turns might be.
Sequential Memory Consistency Model (SC)

We can think of SC execution as being like a random switch, allowing processes to access memory one at a time.

NB It is crucial to understand that this doesn’t mean that SC programs have to be executed sequentially! It only means that the results we get must be the same as if the program had been executed in this way.
Sequential Memory Consistency Model (SC)

An SC execution of

```
c o
  a=1; // a=2; // a=3;  // all at the same time  What is a?
```

oc

can result in a having any of the three written values. How about

```
a=0;
```

co

```
  a=1; // a=2; // b=a+a;  // all at the same time  What is b?
```

oc

To answer this we need to decide what the “atomic actions” are. For our toy notation (and exercises), we will think of reads and writes of single variables as being atomic. Each value accessed in an expression is a read. Each assignment is a write. Thus, in our example, b could 0, 1, 2, 3, or 4.
Atomic Actions

Even such a simple example can illustrate the complications introduced for real languages, compilers and architectures. A sensible compiler would implement $b = a + a$ with one read of $a$, so the outcomes which produce an odd value for $b$ would never happen.

We shouldn’t rely on such unknown factors. It is therefore useful to have a means of specifying that certain blocks of code are to be treated as atomic. In our notation, statements enclosed in `< >` must appear to be atomic, i.e. they must appear to execute as a single indivisible step with no visible intermediate states.

```
a = 0;
co
    a = 1; // a = 2; // <b = a + a;> ## all at the same time What is b?
oc
```

Now the only outcomes for $b$ are 0, 2 or 4.
Atomic Actions and Interleavings

As another example, consider this attempt to increment the variable count twice (which distills what might happen inside some more complex code)

count++; // count++;

where each statement corresponds to a sequence of three actions (read, compute, write). Even with sequential consistency, there are twenty possible interleavings, of which only two match the intended semantics.

What we really meant was

<count++;> // <count++;>

Parallel Programming Languages and Systems
Consider this attempt to reverse the contents of an array in parallel. Can you see what might go wrong?

```
co [i = 0 to n-1] {
    a[i] = a[n-i-1]; ## try to reverse a in parallel
}
```

Here is another flawed attempt. What's wrong this time?

```
co [i = 0 to n-1] {
    <a[i] = a[n-i-1];> ## try to reverse a in parallel
}
```

Can you produce a correct version?
The **await Notation**

The await notation `< await (B) S >` allows us to indicate that S must appear to be delayed until B is true, and must be executed within the same atomic action as a successful check of B. For example, the code below results in x having a value of 25.

```plaintext
a=0; flag=0;
co
  {a=25; flag=1;}
//
  <await (flag==1) x=a;>
oc
```

In terms of the SC interleaving, think of an await as being *eligible* for execution when its condition is true.
The await Notation

However, note that an await is not guaranteed to execute immediately simply because its condition is true. If other atomic actions make the condition false again, before the await executes, it will have to wait for another chance (if there is one). For example, the program

```plaintext
a=0; b=0;
co
  {a=1; other stuff; a=0;}
//
  <await (a==1) b=a;>
oc
```

could either terminate, with a being 0 and b being 1, or could fail to terminate because there is a valid SC execution in which a is set to 1 and then back to 0 before the await statement executes.
Adaptive Quadrature

\[ a \quad b \]

\[ x \]

\[ f(x) \]

Figure 1.4 The quadrature problem.

Copyright © 2000 by Addison Wesley Longman, Inc.

Compute an approximation to the shaded integral by partitioning until the trapezoidal approximation is “good enough”.

Parallel Programming Languages and Systems
Adaptive Quadrature

double quad(double left, double right, double fleft, double fright, double lrarea) {

    double mid, fmid, larea, rarea;

    mid = (left + right) / 2;
    fmid = F(mid);
    larea = (fleft + fmid) * (mid - left) / 2;
    rarea = (fmid + fright) * (right - mid) / 2;
    if( fabs((larea + rarea) - lrarea) > EPSILON ) {
        larea = quad(left, mid, fleft, fmid, larea);
        rarea = quad(mid, right, fmid, fright, rarea);
    }
    return (larea + rarea);
}
Adaptive Quadrature

To compute the whole approximation we call

\[
\text{area} = \text{quad} (a, b, f(a), f(b), (f(a)+f(b))*(b-a)/2);
\]

Noting that the recursive calls to \text{quad} do not interfere with each other, we can trivially parallelize the program by changing the calls to

\[
\begin{align*}
\text{co} & \\
\text{larea} & = \text{quad}(\text{left}, \text{mid}, \text{fleft}, \text{fmid}, \text{larea}); \\
& \quad // \\
\text{rarea} & = \text{quad}(\text{mid}, \text{right}, \text{fmid}, \text{fright}, \text{rarea}); \\
\text{oc}
\end{align*}
\]

The synchronization inherent in \text{co} ensures that both \text{larea} and \text{rarea} have been computed before being added together and returned.
The Bag of Tasks Pattern

Getting a little more practical, we note that there is very little work directly involved in each call to quad. The reality of typical systems is that the work involved in creating and scheduling a process or thread is substantial (much worse than a simple function call), and our program may be swamped by this overhead.

The Bag of Tasks pattern suggests an approach which may be able to reduce overhead, while still providing the flexibility to express such dynamic, unpredictable parallelism.

In bag of tasks, a fixed number of worker processes/threads maintain and process a dynamic collection of homogeneous “tasks”. Execution of a particular task may lead to the creation of more task instances.
The Bag of Tasks Pattern

place initial task(s) in bag;
co [w = 1 to P] {
    while (all tasks not done) {
        get a task;
        execute the task;
        possibly add new tasks to the bag;
    }
}

The pattern is naturally load-balanced: each worker will probably complete a different number of tasks, but will do roughly the same amount of work overall.

For AQ, a task corresponds roughly to one call of quad in the original algorithm, described by the corresponding parameters and either adds its local area approximation to the total, or creates two more tasks for a better approximation.
shared int size = 1, idle = 0;
shared double total = 0.0;
bag.insert (a, b, f(a), f(b), approxarea);
co [w = 1 to P] {
    while (true) {
        < idle++; >
        < await ( size > 0 || idle == P )
        if (size > 0) {
            bag.remove (left, right ...); size--; idle--;
        } else break;
        mid = (left + right)/2; ..etc..
        if (fabs (larea + rarea - lrarea) > EPSILON) {
            < bag.insert (left, mid, fleft, fmid, larea);
            bag.insert (mid, right, fmid, fright, rarea);
            size = size + 2;
        } else < total = total + larea + rarea; >
    }
}
Implementing the Bag

The challenge is to make accessing the bag much cheaper than creating a new thread. With a shared address space, a simple implementation would make the bag an atomically accessed shared data structure.

A more sophisticated implementation (with less contention) might internally have a collection of bags, perhaps one per worker, with task-stealing to distribute the load as necessary.
Implementing the Bag

Similarly, with message passing, a simple scheme might allocate an explicit “farmer” node to maintain the bag.

Again, a more sophisticated implementation could distribute the bag and the farmer, with task-stealing and termination checking via messages. For AQ, we would also have to rethink our strategy for gathering the result.
The Pipeline Pattern

The Sieve of Eratosthenes provides a simple example of the pipeline pattern.

The object is to find all prime numbers in the range 2 to N. The gist of the original algorithm was to write down all integers in the range, then repeatedly remove all multiples of the smallest remaining number. Before each removal phase, the new smallest remaining number is guaranteed to be prime (try it!)

We will sketch a message passing, pipelined, parallel version with a generator process and a sequence of sieve processes, each of which does the work of one removal phase. The pipeline grows dynamically, creating new sieves on demand, as unsieved numbers emerge from the pipeline.
main () { # the generator
    spawn the first sieve process;
    for (i=2; i<=N; i++) {
        send i to first sieve;
    }
    send -1 to first sieve;  # a "stop" signal
}
sieve () {
    int myprime, candidate;
    receive myprime from predecessor and record it;
    do {
        receive candidate from predecessor;
        if (candidate == -1) {send -1 to successor if it exists}
        else if (myprime doesn’t divide candidate exactly) {
            if (no successor yet) spawn successor sieve process;
            send candidate to successor sieve process;
        }
    } while (candidate != -1)
}
The Pipeline Pattern

Pipelines are composed of a sequence of producer-consumer relationships in which each consumer (except the last) becomes a producer for a further consumer, and so on.

Items of data flow from one end of the pipeline to the other, being transformed by and/or transforming the state of the pipeline stage processes as they go.

Our prime finding program has the interesting property that construction of pipeline stages is dynamic and data-dependent.
Producers-Consumers

The producers-consumers relationships (which make up the pipeline) arise in general when a group of activities generate data which is consumed by another group of activities.

The key characteristic is that the conceptual data flow is all in one direction, from producer(s) to consumer(s).

In general, we want to allow production and consumption to be loosely synchronized, so we will need some buffering in the system.

The programming challenges are to ensure that no producer overwrites a buffer entry before a consumer has used it, and that no consumer tries to consume an entry which doesn’t really exist (or re-use an already consumed entry).
Producers-Consumers

Depending upon the model, these challenges motivate the need for various facilities. For example, with a buffer in shared address space, we may need atomic actions and condition synchronization (ie await).

Similarly, in a distributed implementation we want to avoid tight synchronization between sends to the buffer and receives from it.
The Interacting Peers Pattern

In the *interacting peers* pattern information is exchanged in both directions (unlike producers-consumers), within a multidimensional *fixed structure*.

Typically, all processes execute more or less the *same code*, but on distinct *partitions* of the data. This programming style is sometimes also called *SPMD*, for “Single Program Multiple Data”.

Often there is no “master”, “root” or “controller” process, except possibly at the very beginning and end to distribute and gather data and results.
The Interacting Peers Pattern

repeat
    local = f (neighbours old value);
until (all agree to stop);

Interaction could be through messages or shared variables.
The Interacting Peers Pattern

Models of physical phenomena are often expressed as a system of partial differential equations. These can be approximately solved by “finite difference methods” which involve iteration on a matrix of points, in an interacting peers pattern.

The matrix is surrounded by a fixed fringe, representing the boundary conditions. The “compute” step usually involves only a small number of neighbouring points. The termination test looks for convergence, i.e. small difference in point values from one iteration to the next, indicating approximate solution of the pdes.

In this version, we use a duplicate grid and barriers to enforce correct synchronization between iterations.

In contrast, in a message passing version we could remove the barriers and synchronize “naturally” with local synchronous messages only. The termination test would require us to collate local termination decisions somehow.
Fin 

Finite Difference Methods - Jacobi

shared real grid[n+2, n+2], newgrid[n+2, n+2];
shared bool converged; local real diff;
co [i = 1 to n, j = 1 to n] {
    initialise grid;
do {
    barrier();
    converged = true;
    newgrid[i,j] = (grid[i-1,j] + grid[i+1,j] +
                    grid[i,j-1] + grid[i,j+1])/4;
diff = abs (newgrid[i,j] - grid[i,j]);
    barrier();
    if (diff > EPSILON) converged = false;
    grid[i,j] = newgrid[i,j];
    barrier();
} while (not converged);
}
Summary

Examining just a few simple examples has uncovered a range challenges to be addressed by real programming frameworks.

- creation of activities, both statically and dynamically
- local and global synchronization
- atomic (mutually excluding) actions
- conditional synchronization
- synchronous and asynchronous message exchange
- collective communication patterns and agglomerations
Other Patterns

There is growing interest and literature on gathering collections of parallel programming patterns as a means of better understanding and communicating ideas about parallel software design (Eg McCool at Intel, Keutzer et al. at Berkeley, and the ParaPlop workshop).

Other candidate patterns include MapReduce (championed by Google), Scan, Divide & Conquer, Farm as well as application domain specific operations.

Some emerging programming models try to support patterns directly, through polymorphic library operations (eg Intel’s C++ Threading Building Blocks, Microsoft’s Task Parallel Library, the Skandium Java library).

We will initially focus on the programming layers which underpin such abstractions.
Shared Variable Programming

We now consider a range of programming concepts and constructs which have been suggested to assist the correct programming of machines with a shared address space.

In the first part of this phase of the course we will introduce these within our toy language, assuming an SC memory model. We will use < > and < await (B) S > to specify the intended behaviour, and think about how they might be implemented with simpler primitives, still under SC.

Later in this phase we will see how real shared variable languages and libraries have provided similar constructs and how they allow correct control of parallelism in the absence of SC. After that we will turn to a similar consideration of message passing parallelism.
Shared Variable Synchronization

We have already seen that there are two fundamental kinds of synchronization in shared variable programming:

- **Mutual Exclusion** is more like anti-synchronization! We want to prevent two or more threads from being active concurrently for some period, because their actions may interfere incorrectly. For example, we might require updates to a shared counter (e.g., `count++`) to execute with mutual exclusion.

- **Condition Synchronization** occurs when we want to delay an action until some condition (on the shared variables such as in producer-consumer, or with respect to the progress of other threads such as in a barrier) becomes true.

We consider a range of concepts which help express these in different situations.
Critical Sections

A simple pattern of mutual exclusion occurs in the critical section problem. This occurs when \( n \) threads execute code of the following form, in which it is essential that at most one thread is executing statements from the critical section at a time (because of potentially unsafe access to shared variables)

\[
\text{co [} i = 1 \text{ to } n \text{]} \{ \\
\quad \text{while (something)} \{ \\
\qquad \text{critical section; } \quad \#\# \text{ one thread at a time} \\
\qquad \text{non-critical section;} \\
\quad \}\n\}
\]

We must design code to execute before (entry protocol) and after (exit protocol) the critical section.
Important Properties

Mutual Exclusion. At most one thread is executing the critical section at a time.

Absence of Deadlock (or Livelock). If two or more threads are trying to enter the critical section, at least one succeeds.

Absence of Unnecessary Delay. If a thread is trying to enter its critical section and the other threads are executing their non-critical sections, or have terminated, the first thread is not prevented from entering its critical section.

Eventual Entry (or No Starvation). A thread that is attempting to enter its critical section will eventually succeed.

The first three are always essential. Eventual entry may not matter in some “performance parallel” programs - as long as we are making progress elsewhere.
Critical Sections & Locks

The entry and exit protocol code obviously has to operate upon one or more shared variables. Conventionally we call such variables **locks**, and the protocol code sequences **locking** and **unlocking**. Shared variable libraries will often abstract these as functions.

```
co [i = 1 to n] {
    while (something) {
        lock(l);
        critical section;
        unlock(l);
        non-critical section;
    }
}
```
Implementing Locks

A simple approach is to implement each lock with a shared boolean variable.

If the variable has value false then one locking thread can set it and be allowed to proceed. Other attempted locks must be forced to wait.

To unlock the lock, the lock-holding thread simply sets the lock to false.

We can specify this behaviour with our < await () > notation.
Implementing Locks

```cpp
lock_t l = false;

c o [i = 1 to n] {
    while (something) {
        < await (!l) l = true; >
        critical section;
        l = false;
        non-critical section;
    }
}

(recall that our model assumes that the l = false; write is already atomic)
```
Consider again the code from the last slide

```c
lock_t l = false;
co [i = 1 to n] {
    while (something) {
        < await (!l) l = true; >
        critical section;
        l = false;
        non-critical section;
    }
}
```

This might **fail** if the model is more relaxed than SC. Why? (Hint: what if writes in the “critical section” and l = false; in the exit protocol are re-ordered?)
Implementing Locks

To implement the detail, we rely on some **simpler atomic primitive**, implemented with hardware support. There are many possibilities, including “Fetch-and-Add”, “Test-and-Set” and the “Load-Linked, Store-Conditional” pairing.

A Test-and-Set (TS) instruction implements the following effect. We think of this behaving like a call-by-reference function, so that the variable passed in is read from and written to, but in reality it is a **single machine instruction**.

```c
bool TS (bool v) {
    bool initial = v;
    v = true;
    return initial;
}
```

The key feature is that this happens (or at least, appears to happen) **atomically**.
Implementing Locks

lock_t l = false;
co [i = 1 to n] {
    while (something) {
        while (TS(l)); ## i.e. spin
        critical section;
        l = false;
        non-critical section;
    }
}

This guarantees mutual exclusion, absence of deadlock and absence of delay, but does not guarantee eventual entry. It is called a spin lock because of the behaviour of threads which fail to gain access immediately.
Simple spin locks don’t make good use of the cache (those spinning Test-And-Sets play havoc with contention and coherence performance). A pragmatically better solution is known as Test-and-Test-and-Set (though it still uses Test-and-Set).

```c
lock_t l = false;
co [i = 1 to n] {
    while (something) {
        do {
            while (l) ;  ## spin until lock seems free
        } while (TS(l));  ## actual atomic locking
    critical section;
    l = false;
    non-critical section;
}
}
```
We simply “Test” (i.e. read) until there is a chance that a Test-and-Set might succeed. Using a C style lazy || operator, we express this more concisely:

```c
lock_t l = false;
co [i = 1 to n] {
    while (something) {
        while (l || TS(l));  // only TS if l was false
        critical section;
        l = false;
    non-critical section;
    }
}
Lamport’s Bakery Algorithm

Lamport showed how to implement critical sections using only simple atomic read and simple atomic write instructions (i.e. no need for atomic read-modify-write). It is important to note that the algorithm assumes sequential memory consistency. Finally, Lamport’s algorithm has the strong property of guaranteeing eventual entry (unlike our spin lock versions). The algorithm is too inefficient to be practical if spin-locks are available, but is a great achievement nonetheless!

There are two phases to the entry protocol. Firstly a thread calculates when its “turn” will be (as an integer), by looking at other threads’ turns. A thread sets its turn to be one more than any other turn currently claimed. Threads not at the critical section have a turn of 0. Secondly, the thread waits until its turn comes up, by waiting until it has a lower turn than each of the other competing threads.
Lamport’s Bakery Algorithm

```c
int turn[n] = [0, 0, ... 0];
co [i=1 to n] {
    while (true) {
        < turn[i] = max (turn[1..n]) + 1; >
        for (j = 1 to n except i) {
            <await (turn[j]==0 or turn[i]<turn[j]) ;>
        }
        critical section;
        turn[i] = 0;
        noncritical section;
    }
}
```

This is “obviously” correct, but how can we get rid of the atomic section?
Lamport’s Bakery Algorithm

Just drop the atomic, and use the obvious spinning implementation of await?

```c
int turn[n] = [0, 0, ... 0];
co [i=1 to n] {
    while (true) {
        turn[i] = max (turn[1..n]) + 1;
        for (j = 1 to n except i) {
            while ((turn[j]!=0 and (turn[i] > (turn[j]))) skip;
        }
        critical section;
        turn[i] = 0;
        noncritical section;
    }
}
```
Lamport’s Bakery Algorithm

There are two problems with this.

Firstly, there is possibility that a thread can claim a lower turn than another thread which enters the critical section before it!

Secondly, if turn setting is not atomic then there is a possibility that two (or more) threads will claim the same turn.

The following slide shows an instance of the first problem.
Lamport’s Bakery Algorithm

[Thread 3 is in CS, with turn[3] == 9, other turns == 0]

Thr 2 sees turn[1] == 0
Thr 2 sees turn[3] == 9

Thr 3 sets turn[3] = 0

Thr 1 sees turn[2] == 0
Thr 1 sees turn[3] == 0

Thr 2 sets turn[2] = 10
Thr 2 sees turn[1] == 0
Thr 2 sees turn[3] == 0
Thr 2 enters CS

Thr 1 sets turn[1] = 1
Thr 1 sees turn[1] < turn[2]
Thr 1 sees turn[3] == 0
Thr 1 enters CS
Lamport’s Bakery Algorithm

We can fix the first problem by adding the statement \( \text{turn}[i] = 1; \) to the entry protocol.

A \( \text{turn} \) value of 1 now indicates that a thread is in the process of setting its turn. It distinguishes the thread (or to be precise, its turn), from those which are not attempting to enter the critical section. Notice that now, no thread will ever have a “real” turn value of 1.

This fixes the first problem: the artificially low \( \text{turn} \) of 1 will stop any other thread in the second phase from entering the CS until the turn setting of the first thread is complete.

The following slides show the improved code and fixed behaviour.
Lamport’s Bakery Algorithm

```c
int turn[n] = [0, 0, ... 0];
co [i=1 to n] {
    while (true) {
        turn[i] = 1; turn[i] = max (turn[1..n]) + 1;
        for (j = 1 to n except i) {
            while ((turn[j]!=0 and (turn[i] > turn[j])) skip;
        }
        critical section;
        turn[i] = 0;
        noncritical section;
    }
}
```

Solves the first problem (but what about duplicate turns?)
Lamport’s Bakery Algorithm

[Thread 3 is in CS, with turn[3] == 9, other turns == 0]
Thr 2 sets turn[2] = 1
Thr 2 sees turn[1] == 0
Thr 2 sees turn[3] == 9
Thr 3 sets turn[3] = 0

Thr 1 sets turn[1] = 1
Thr 1 sees turn[2] == 1
Thr 1 sees turn[3] == 0
Thr 2 sets turn[2] = 10
Thr 2 sees turn[1] == 1 and can’t enter CS.....

Thr 1 sets turn[1] = 2
Thr 1 sees turn[1] < turn[2]
Thr 1 sees turn[3] == 0
Thr 1 enters CS

Thr 2 is stuck until Thr 1 leaves CS
Lamport’s Bakery Algorithm

The duplicate turn problem occurs when threads end up choosing the same turn (because choosing is no longer atomic).

This is much easier to deal with. We simply need an artificial and systematic way of deciding which of two equal turns will be treated as though it were smaller than the other.

We do this by using the thread ids (which are definitely distinct). In the case of duplicate turns, the thread with the lower id “wins” (i.e. is treated as having a “lower” turn).

In the code, we write \((x, y) > (a, b)\) to mean \((x > a) \lor (x == a \land y > b)\).
Lamport’s Bakery Algorithm

```c
int turn[n] = [0, 0, ... 0];
co [i=1 to n] {
    while (true) {
        turn[i] = 1; turn[i] = max (turn[1..n]) + 1;
        for (j = 1 to n except i) {
            while ((turn[j]!=0 and (turn[i], i) > (turn[j], j)) skip;
        }
        critical section;
        turn[i] = 0;
        noncritical section;
    }
}
```

This is the complete, correct bakery algorithm!
Lamport’s Bakery Algorithm

Here is another example which demonstrates the importance of both the new $\text{turn}[i]=1$ statement and the added tie-breaking mechanism. Without the new statement, but with tie-breaking, the following bad behaviour is possible:

Thr 1 sees $\text{turn}[2] == 0$

Thr 2 sees $\text{turn}[1] == 0$
Thr 2 sets $\text{turn}[2] = 1$  ## ie $0+1$
Thr 2 enters CS because $\text{turn}[1]==0$

Thr 1 sets $\text{turn}[1] = 1$
Thr 1 sees $\text{turn}[2] == 1$
Thr 1 enters CS because $(1,1) < (1,2)$

and both threads are in the CS!
Lamport’s Bakery Algorithm

On the other hand, with the new $\text{turn}[i]=1$ statement in place:

Thr 1 sets $\text{turn}[1] = 1$  ## NEW
Thr 1 sees $\text{turn}[2] == 0$

Thr 2 sets $\text{turn}[2] = 1$  ## NEW
Thr 2 sees $\text{turn}[1] == 1$  ## not 0
Thr 2 sets $\text{turn}[2] = 2$  ## ie 1+1
Thr 2 doesn’t enter, because $\text{turn}[1]<\text{turn}[2]$

Thr 1 sets $\text{turn}[1] = 2$  ## 1+1
Thr 1 sees $\text{turn}[2] == 2$
Thr 1 enters CS because $(2,1) < (2,2)$

Later, thread 1 will reset $\text{turn}[1] = 0$ and thread 2 will enter.
Many algorithms have the following structure

```plaintext
co [i = 1 to n] {
    while (something) {
        do some work;
        wait for all n workers to get here;
    }
}
```

This kind of computation-wide waiting is called barrier synchronization. It is an example of a particular pattern of condition synchronization.
Counter Barriers

```java
shared int count = 0;
co [i = 1 to n] {
    do some work;
    ## now the barrier
    <count = count + 1;>
    <await (count == n);>
}
```

This is fine as a single-use barrier, but things get more complex if (as is more likely) we need the barrier to be reusable.
Reusable Counter Barrier - Wrong!

shared int count = 0;
co [i = 1 to n] {
    while (something) {  ## NB looping now
        do some work;
        ## now the barrier
        <count = count + 1;>
        <await (count == n); count = 0;>
    }
}
Sense Reversing Barrier - Correct

shared int count = 0; shared boolean sense = false;
co [i = 1 to n] {
    private boolean mySense = !sense;  ## one per thread
    while (something) {
        do some work;
        < count = count + 1;
        if (count == n) { count = 0; sense = mySense; }
        >
        while (sense != mySense);  ## wait
        mySense = !mySense;
    }
}
**Sense Reversing Barrier**

The shared variable `sense` is at the core of the synchronization. Its value is flipped after each use of the barrier to indicate that all threads may proceed.

The local variable `mySense` remembers whether to wait for a `true` or `false` value in `sense`.

The key idea is that we have separated the signal that the barrier has been completed (flipping `sense`) from the mechanism used to determine this condition, `count`. There is no longer an inter-iteration race between these two issues.

Flipping the binary `sense` value simultaneously drops the barrier for one iteration while raising it for the next.
Symmetric Barriers

Symmetric barriers are designed to avoid the bottleneck at the counter.

Overall synchronization is achieved transitively from a carefully chosen sequence of pairwise synchronizations. Each thread executes the same code, choosing partners for the pairwise synchs as a function of its own identifier and the internal iteration. For \( n \) a power of two, we have the butterfly pattern.

![Figure 3.15 Butterfly barrier for 8 processes.](image-url)
Symmetric Barriers

To synchronize between a pair of threads \texttt{myid} and \texttt{friend} (where each sees the other as its friend), both could execute

\begin{verbatim}
<await (arrive[myid] == 0)>;
arrive[myid] = 1;
<await (arrive[friend] == 1)>;
arrive[friend] = 0;
\end{verbatim}

The first line avoids race problems caused by previous uses of the barrier.

However, when used as a step within a multistage symmetric barrier, there is an additional problem.
Symmetric Barriers

```c
for [s = 0 to stages-1] {
    <await (arrive[myid] == 0)>;
    arrive[myid] = 1;
    work out who my friend is at stage s;
    <await (arrive[friend] == 1)>;
    arrive[friend] = 0;
}
```

Consider the case with four threads (and thus two stages). Suppose thread 1 arrives at its first level barrier as normal, but that thread 2 will never arrive. Meanwhile threads 3 and 4 synchronize quickly. Thread 3 then “see” that thread 1 is at a pairwise barrier (but unfortunately, not that it is the “wrong” one!), and proceeds, resetting `arrive[1]`. Thread 3 leaves the barrier, even though thread 2 will never arrive!
Symmetric Barriers

We can fix this by having distinct variables for each stage of the barrier.

```plaintext
for [s = 0 to stages-1] {  ## there will be log_2 p stages
  <await (arrive[myid][s] == 0)>;
  arrive[myid][s] = 1;
  work out who my friend is at this stage;
  <await (arrive[friend][s] == 1)>;
  arrive[friend][s] = 0;
}
```

Thread 3 can now only synchronize with thread 1 after thread 1 has synchronized with thread 2. Since this now won't happen in our “bad” scenario, thread 3 will be prevented from leaving the barrier incorrectly.
Dissemination Barriers

What can we do if $n$ isn’t a power of 2? The dissemination barrier approach is similar to the symmetric approach, but instead of pairwise synchs, we have two partners at each stage, one incoming and one outgoing. The code is the same: an arrow in the diagram from $X$ to $Y$ means that $Y$ waits for $X$ to signal that it has arrived (ie $X$ is $Y$’s “friend” in the code).

![Diagram of dissemination barrier for 6 processes]

Figure 3.16 Dissemination barrier for 6 processes.
Structured Primitives

The mechanisms we have designed so far have all been implemented directly in the user-address space (probably hidden inside a library, but nonetheless “invisible” to the OS).

In contrast, a number of more structured primitives have been devised for implementation with the assistance of the operating system, so that threads can be directly suspended and resumed.

We will look at two of the most common, semaphores and monitors. These include capabilities which facilitate the expression of both types of synchronization.
Semaphores

A semaphore is a special shared variable, accessible only through two atomic operations, \( P \) and \( V \), defined by

\[
\begin{align*}
P(s) & : \langle \text{await} \ (s>0) \ s=s-1; \rangle \\
V(s) & : \langle s=s+1; \rangle 
\end{align*}
\]

Notice that if a semaphore is initialised to have a non-negative value, then it can never become negative subsequently. A semaphore whose usage is organised to only ever take the value 0 or 1 is called a \textit{binary semaphore}.

In a typical implementation, a thread executing \( P \) on a 0 valued semaphore will be suspended on a queue until after some other thread has executed a \( V \).

[The names come from Dijkstra’s native Dutch words Probeer (Try) and Verhoog (Increase), but you don’t need to learn that :)]

---

Parallel Programming Languages and Systems
Using Semaphores

A semaphore provides an easy solution to the critical section problem

```
sem mutex = 1;

co [i = 1 to n] {
    while (whatever) {
        P(mutex);
        critical section;
        V(mutex);
        noncritical section;
    }
}
```
Using Semaphores

We can also use semaphores at the core of a symmetric barrier implementation (in which we have an array of arrive semaphores for each stage).

```plaintext
for [s = 1 to stages] {
    V(arrive[myid][s]);
    work out who my friend is at stage s;
    P(arrive[friend][s]);
}
```

Notice that with semaphores we no longer need the initial wait for our own semaphore to be zeroed (as earlier), because our V can't be “lost”. The semaphores are being atomically incremented and decremented rather than simply set to 1 or 0, with all work on our friend semaphore captured by the P operation.
Using Semaphores

Semaphores offer neat solutions to various producer-consumer buffering problems.

For example, to control access to a single element buffer, with multiple producers and consumers, we use two semaphores, one to indicate that the buffer is full, the other to indicate that it is empty.

Since only one of the semaphores will ever have the value one, this is sometimes called a split binary semaphore.

More generally a semaphore whose value is counting availability of some resource, is often called a counting semaphore (sometimes “split”).
T buf; sem empty = 1, full = 0;
c
  co [i = 1 to M] {
    while (whatever) {
      ...produce new data locally
      P(empty);
      buf = data;
      V(full);
    }
  }

//
  co [j = 1 to N] {
    while (whatever) {
      P(full);
      result = buf;
      V(empty);
      ... handle result locally
    }
  }

c
Parallel Programming Languages and Systems
Bounded Buffer

Now suppose that we want to have a multi-space buffer, so that the producers can run ahead of the consumers (up to some limit).

We implement the buffer itself with an array, and two integer indices, indicating the current front and rear of the buffer and use arithmetic modulo n (the buffer size), so that the buffer conceptually becomes circular.

For a single producer and consumer, we protect the buffer with a split “counting” semaphore, initialised according to the buffer size (so no longer binary). Think of full as counting how many space in the buffer are full, and empty as counting how many are empty.

Provided the buffer isn’t empty or full, we should allow producer and consumer to be active within it simultaneously.
T buf[n]; int front = 0, rear = 0;
sem empty = n, full = 0;
co  ## Producer
    while (whatever) {
        ...produce new data locally
        P(empty);
        buf[rear] = data; rear = (rear + 1) % n;
        V(full);
    }
//  ## Consumer
co  while (whatever) {
    P(full);
    result = buf[front]; front = (front + 1) % n;
    V(empty);
    ... handle result locally
}
oc

Parallel Programming Languages and Systems
Multiple Producers/Consumers

To allow for multiple producers and consumers, we need two levels of protection.

We use a split counting semaphore to avoid buffer overflow (or underflow), as previously.

We use mutual exclusion semaphores to prevent interference between producers (and another to prevent interference between consumers). This allows up to one consumer and one producer to be actively simultaneously within a non-empty, non-full buffer.
T buf[n]; int front = 0, rear = 0;
sem empty = n, full = 0, mutexP = 1, mutexC = 1;
co
  co [i = 1 to M] {
    while (whatever) {
      ...produce new data locally
      P(empty);
      P(mutexP); buf[rear] = data; rear = (rear + 1) % n; V(mutexP);
      V(full);
    }
  }
  //
  co [j = 1 to N] {
    while (whatever) {
      P(full);
      P(mutexC); result = buf[front]; front = (front + 1) % n; V(mutexC);
      V(empty);
      ... handle result locally
    }
  }
oc
Multiple Producers/Consumers

Can we further relax this solution to allow several producers and/or consumers to be active within the buffer simultaneously? This might be useful if the buffered items are large and take a long time to read/write.

We need to ensure that accesses target distinct buffer locations, so the index arithmetic will certainly need to be kept atomic.

Can you see what's wrong with the proposed solution on the next overhead?
co \[i = 1 \text{ to } M\] \{
    \text{while (whatever) \{ }
    \hspace{1em} \ldots \text{produce new data locally}
    \text{P(} \text{empty} \text{);} \\
    \hspace{1em} \text{P(} \text{mutexP} \text{);} \hspace{0.5em} \text{myrear} = \text{rear}; \hspace{0.5em} \text{rear} = (\text{rear} + 1) \% \text{n}; \hspace{0.5em} \text{V(} \text{mutexP} \text{);} \\
    \hspace{1em} \text{buf[myrear]} = \text{data}; \hspace{1em} \## \text{ where myrear is a private variable} \\
    \hspace{1em} \text{V(} \text{full} \text{);} \\
    \} \} \\
\} \\
//
co \[j = 1 \text{ to } N\] \{
    \text{while (whatever) \{ }
    \hspace{1em} \text{P(} \text{full} \text{);} \\
    \hspace{1em} \text{P(} \text{mutexC} \text{);} \hspace{0.5em} \text{myfront} = \text{front}; \hspace{0.5em} \text{front} = (\text{front} + 1) \% \text{n}; \hspace{0.5em} \text{V(} \text{mutexC} \text{);} \\
    \hspace{1em} \text{result} = \text{buf[myfront]}; \hspace{1em} \## \text{ where myfront is a private variable} \\
    \hspace{1em} \text{V(} \text{empty} \text{);} \\
    \hspace{1em} \ldots \text{ handle result locally}
    \} \} \\
oc
The producers are filling distinct slots, but not necessarily completing these fills in strict order - slot $i+1$ might finish filling before slot $i$.

However, consumers only know that a slot has been filled and assume, possibly incorrectly, that it is the "next" one.

Can you think of a scheme which avoids this? For example, how could you ensure that an entry can’t be read until it has been completely filled?
Monitors

Semaphores are a good idea, but have some drawbacks. For example,

- they still require careful programming: there is no explicit connection in the program source between “matching” semaphore operations. It is easy to get things wrong.

- Similarly, there is no obvious indication of how semaphores are being used - some may be for mutual exclusion, others for condition synchronization. Again confusion is possible.

The monitor is a more structured mechanism.
Monitors - Mutual Exclusion

The monitor concept is quite easy to understand from an object-oriented perspective. A monitor is like an object which encapsulates some data to which access is only permitted through a set of methods.

When the monitor object exists in a threaded concurrent context, the implementation ensures that **at most one thread is active within the monitor at any one time** (though many threads may be suspended within monitor methods).

The effect is as if the body of each monitor method is implicitly surrounded with `P()` and `V()` operations on a single hidden binary semaphore, shared by all methods. Thus, monitors provide structured mutual exclusion “for free”, and implicitly. The mechanism for more complex conditional synchronization requires explicit actions by the program.
Monitors - Condition Synchronization

A condition variable is a special variable, associated with a monitor, which we can think of as controlling a queue of delayed threads.

Once inside a monitor method a thread may call the \texttt{wait(cv)} operation, where \texttt{cv} is a condition variable. This causes the thread both to give up the (implicit) lock it holds on the monitor, and to be blocked upon the queue of \texttt{cv}.

A blocked thread remains so until some other thread, while active inside the monitor, calls the operation \texttt{signal(cv)}. This causes a previously blocked thread (normally chosen by a FIFO discipline) to become ready for scheduling again (ie it becomes blocked on the implicit lock, waiting for this to be released).

The signalling thread continues uninterrupted, hence this scheme is called \texttt{signal and continue} (SC).
Monitors - Condition Synchronization

State transition diagram for threads using a “signal-and-continue” monitor.
Monitors - Condition Synchronization

Finally, an operation \texttt{signal-all(cv)} is usually available. This awakens \textit{all} (rather than just one) of the waiting threads.

They all become eligible to proceed, once the signalling thread releases the lock, but only one will be allowed to enter the monitor at a time, in the usual way.

It is \textit{important not to confuse} these \texttt{wait()} and \texttt{signal()} operations with the similar sounding (and sometimes identically named!) \texttt{operations on semaphores}.

The key difference is that \texttt{signal()} on a condition variable is not “\textit{remembered}” in the way that \texttt{V()} on a semaphore is. If no threads are waiting, then a \texttt{signal()} is “\textit{lost}” or “\textit{forgotten}”, whereas a \texttt{V()} will allow a subsequent \texttt{P()} to proceed.
Monitors - Condition Synchronization

Monitor semantics mean that when a thread which was previously blocked on a condition is actually awakened again in the monitor, it often makes sense to check that the condition it was waiting for is still true.

The point to remember is that when the signal happened, the signalled thread only became available for actual execution again (ie it was allowed to try to acquire the monitor lock again). It could be that some other thread acquires the lock first, and does something which negates the condition again (for example, it consumes the “new item” from a monitor protected buffer).

Thus it is often necessary, in all but the most tightly constrained situations, to wrap each conditional variable `wait()` call in a loop which rechecks the condition. The following bounded buffer works for arbitrarily many producers and consumers.
monitor Bounded_Buffer {

typeT buf[n]; # an array of some type T
int front = 0, # index of first full slot
    rear = 0; # index of first empty slot
    count = 0; # number of full slots
## rear == (front + count) % n
    cond not_full, # signaled when count < n
    not_empty; # signaled when count > 0

    procedure deposit(typeT data) {
        while (count == n) wait(not_full);
        buf[rear] = data; rear = (rear+1) % n; count++;
        signal(not_empty);
    }

    procedure fetch(typeT &result) {
        while (count == 0) wait(not_empty);
        result = buf[front]; front = (front+1) % n; count--;
        signal(not_full);
    }
}

Figure 5.4 Monitor implementation of a bounded buffer.
Real Shared Variable Programming Systems

We now examine the ways in which the various concepts for shared variable programming have been embedded in real programming systems. In particular we look at C’s Posix threads (Pthreads) library and Java’s threads and monitors. Here’s a possible output from the following Pthreads program.

[gateside]mic: ./test
Hello from the main thread
Hello from thread 3
Hello from thread 0
Hello from thread 5
Hello from thread 6
Hello from thread 1
Hello from thread 7
Hello from thread 2
Hello from thread 4
Goodbye from the main thread
```c
#include <pthread.h>
#define P 8
void *sayhello (void *id) {
    printf("Hello from thread \%d\n", (int) id);
}

int main (int argc, char *argv[]) {
    int i; pthread_t thread[P];
    printf("Hello from the main thread\n");
    for (i=0; i<P; i++) {
        pthread_create(&thread[i], NULL, sayhello, (void *)i);
    }
    for (i=0; i<P; i++) {
        pthread_join(thread[i], NULL);
    }
    printf("Goodbye from the main thread\n");
}
```
POSIX Threads

The POSIX threads (Pthreads) standard defines an API for thread programming. Conceptually, a process (whose execution is already a “main” thread) can start, synchronize with and stop other threads of activity within its address space.

Threads (of type pthread_t) begin by executing a given function, and terminate when that function exits (or when killed off by another thread).

```c
int pthread_create (pthread_t *tid,
                 p_thread_attr_t *att, void * (*f) (void *),
                 void *arg);
```

The function run (f) has a “lowest common denominator” C prototype, having a generic pointer as both argument and return type. The actual parameter to the call of f is passed through the final parameter of pthread_create.
POSIX Threads

Often the `arg` parameter is `NULL` (since the intended effect will be achieved directly in shared variable space) or perhaps an integer thread identifier, to assist data partitioning.

```c
int pthread_join (pthread_t t, void ** result);
```

The calling thread waits for the thread identified by the first parameter to finish, and picks up its returned result through the second parameter.

The `result` parameter is often just `NULL` since the intended effect will have been achieved directly in shared variable space. Pthreads also has a range of functions which allow threads to kill each other, and to set properties such as scheduling priority (e.g. through the second parameter to `pthread_create`). We will not discuss these.
Accidentally Sharing Data

void *sayhello (void *id) {
    printf("Hello from thread %d\n", *(int *)id);
}

int main (int argc, char *argv[]) {
    int i; pthread_t thread[P];
    printf("Hello from the main thread\n");
    for (i=0; i<P; i++) {
        // Each thread gets a pointer to i, producing a race
        pthread_create(&thread[i], NULL, sayhello, &i);
    }
    for (i=0; i<P; i++) {
        pthread_join(thread[i], NULL);
    }
    printf("Goodbye from the main thread\n");
}
Intentionally Sharing Data

```c
int target;

void *adderthread (void *arg) {
    int i;
    for (i=0; i<N; i++) {
        target = target+1;
    }
}

int main (int argc, char *argv[]) {
    int i; pthread_t thread[P];
    target = 0;
    for (i=0; i<P; i++) {
        pthread_create(&thread[i], NULL, adderthread, NULL);
    } ......
```
Coordinating Shared Accesses

Variable target is accessible to all threads. Its increment is not atomic, so we may get unpredictable results.

POSIX provides mechanisms to coordinate accesses including semaphores and building blocks for monitors. Posix semaphores have type sem_t. Operations are

1. `sem_init(&sem, share, init)`, where init is the initial value and share is a “boolean” (in the C sense) indicating whether the semaphore will be shared between processes (true) or just threads within a process (false).

2. `sem_wait(s)`, which is the Posix name for P(s)

3. `sem_post(s)`, which is the Posix name for V(s)
sem_t lock;
void *adderthread (void *arg)
{
    int i;

    for (i=0; i<N; i++) {
        sem_wait(&lock);
        target = target+1;
        sem_post(&lock);
    }
}

int main (int argc, char *argv[])
{
    target = 0;
    sem_init(&lock, 0, 1);
    ......
Producers & Consumers

sem_t empty, full; // the global semaphores
int data; // shared buffer

int main (int argc, char *argv[]) {
    pthread_t pid, cid;
    ....

    sem_init(&empty, 0, 1); // sem empty = 1
    sem_init(&full, 0, 0); // sem full = 0

    pthread_create(&pid, &attr, Producer, NULL);
    pthread_create(&cid, &attr, Consumer, NULL);
    pthread_join(pid, NULL);
    pthread_join(cid, NULL);
}
void *Producer (void *arg) {
    int produced;
    for (produced = 0; produced < numIters; produced++) {
        sem_wait(&empty);
        data = produced;
        sem_post(&full);
    }
}

void *Consumer (void *arg) {
    int total = 0, consumed;
    for (consumed = 0; consumed < numIters; consumed++) {
        sem_wait(&full);
        total = total+data;
        sem_post(&empty);
    }
    printf("after %d iterations, the total is %d (should be %d)\n", numIters, total, numIters*(numIters+1)/2);
}
Pthreads “Monitors”

Pthreads doesn't provide the monitor as a built-in programming construct, but it does provide the building blocks needed to achieve monitor-like effects. It provides locks, which are of type pthread_mutex_t. These can be

- initialized with `pthread_mutex_init(&m, attr)`, where `attr` are attributes concerning scope (as with semaphore creation).

- locked with `pthread_mutex_lock(&m)`, which blocks the locking thread if already `m` is already locked. There is also a non-blocking version `pthread_mutex_trylock(&m)`.

- unlocked with `pthread_mutex_unlock(&m)`. Only a thread which holds a given lock, should unlock it!
Pthreads Condition Variables

Pthreads provides condition variables, which are of type `pthread_cond_t`. As well as the usual initialization, these can be

- **waited on** with `pthread_cond_wait(&cv, &mut)` where `cv` is a condition variable, and `mut` must be a lock already held by this thread, and which is implicitly released.

- **signalled** with `pthread_cond_signal(&cv)` by a thread which should (but doesn’t strictly have to) hold the associated mutex. The semantics are “Signal-and-Continue” as previously discussed.

- **signalled** with `pthread_cond_broadcast(&cv)`. This is “signal-all”.

It is the programmer’s responsibility to adhere to the discipline on lock ownership.
Simple Jacobi

We round off our examination of Pthreads with a simple Jacobi grid-iteration program.

This runs the standard Jacobi step for a given fixed number of iterations. To avoid copying between “new” and “old” grids, each iteration performs two Jacobi steps. Convergence testing could be added as before.

The code includes the definition of a simple counter barrier, and its use to keep new point calculation and update safely separated.
A Re-usable Counter Barrier

```c
pthread_mutex_t barrier;        // mutex semaphore for the barrier
pthread_cond_t go;             // condition variable for leaving
int numArrived = 0;

void Barrier() {
    pthread_mutex_lock(&barrier);
    numArrived++;
    if (numArrived == numWorkers) {
        numArrived = 0;
        pthread_cond_broadcast(&go);
    } else {
        pthread_cond_wait(&go, &barrier);
    }
    pthread_mutex_unlock(&barrier);
}
```

Parallel Programming Languages and Systems
int main(int argc, char *argv[]) {

    pthread_t workerid[MAXWORKERS];
    pthread_mutex_init(&barrier, NULL);
    pthread_cond_init(&go, NULL);

    InitializeGrids();

    for (i = 0; i < numWorkers; i++)
        pthread_create(&workerid[i], &attr, Worker, (void *) i);

    for (i = 0; i < numWorkers; i++)
        pthread_join(workerid[i], NULL);
}
void *Worker(void *arg) {
    int myid = (int) arg, rowA = myid*rowshare+1, rowB = rowA+rowshare-1;
    for (iters = 1; iters <= numIters; iters++) {
        for (i = rowA; i <= rowB; i++) {
            for (j = 1; j <= gridSize; j++) {
                grid2[i][j] = (grid1[i-1][j] + grid1[i+1][j] +
                                grid1[i][j-1] + grid1[i][j+1]) * 0.25;
            }
        }
        Barrier();
        for (i = rowA; i <= rowB; i++) {
            for (j = 1; j <= gridSize; j++) {
                grid1[i][j] = (grid2[i-1][j] + grid2[i+1][j] +
                                grid2[i][j-1] + grid2[i][j+1]) * 0.25;
            }
        }
        Barrier();
    }
}
Memory Consistency

As previously noted, weak consistency models can wreck naive DIY synchronization attempts! What does Pthreads have to say about this? To enable portability, Pthreads mutex, semaphore and condition variable operations implicitly act as memory fences, executing architecture specific instructions.

In effect, the C + Pthreads combination guarantees a weak consistency memory model, with the only certainties provided at uses of Pthreads primitives. For example, all writes by a thread which has released some mutex, are guaranteed to be seen by any thread which then acquires it. Nothing can be assumed about the visibility of writes which cannot be seen to be ordered by their relationship to uses of Pthread primitives.

We can be sure that our Jacobi program will execute correctly because the critical phases are separated by barriers which are implemented with mutex locks.
Pragmatic Issues

The programmer must also be careful to use only thread-safe code, which works irrespective of how many threads are active.

Taking care to make your own code thread-safe is one thing, but what about code from libraries?

Typical problems involve the use of non-local data. For example, imagine a non-thread safe malloc. Unluckily interleaved calls might break the underlying free space data structure.

Some libraries will provide thread-safe versions (but of course, which pay an unnecessary performance penalty when used in a single threaded program).
Java Threads

Java was designed from the start to be multithreaded with a synchronization model based around the monitor concept.

We will begin by looking at the thread programming mechanisms which are built into the Java language itself (rather than just added through a class library).

Then we will look briefly at some of the packages which have been added to provide additional support for shared variable parallelism.

Java comes with its own architecture-independent memory consistency model, defined around the use of various primitives (including the volatile keyword) and a “happens-before” partial-order on actions.
Java Threads

Threads can be created from classes which extend java.lang.Thread

class Simple extends Thread {
    public void run() {
        System.out.println(''this is a thread'');
    }
}

new Simple().start();  // implicitly calls the run() method

or implement java.lang.Runnable (so we can extend some other class too).

class Bigger extends Whatever implements Runnable {
    public void run() {
        ...
    }
}

new Thread( new Bigger (...)).start();
Java Threads

As in Pthreads, we can wait to join with another thread.

class Friend extends Thread {
    private int me;

    public Friend (int i) {
        me = i;
    }

    public void run() {
        System.out.println("Hello from thread " + me);
    }
}

Parallel Programming Languages and Systems
Java Threads

class Hello throws java.lang.InterruptedException {
    private static final int n = 5;
    public static void main(String[] args) {
        int i; Friend t[] = new Friend[n];
        System.out.println("Hello from the main thread");
        for (i=0; i<n; i++) {
            t[i] = new Friend(i);
            t[i].start();
        }
        for (i=0; i<n; i++) {
            t[i].join(); // might throw java.lang.InterruptedException
        }
        System.out.println("Goodbye from the main thread");
    }
}

Parallel Programming Languages and Systems
Java “Monitors”

Java provides an implementation of the monitor concept (but doesn’t actually have monitor as a keyword).

Any object in a Java program can, in effect, become a monitor, simply by declaring one or more of its methods to be synchronized, or by including a synchronized block of code.

Each such object is associated with one, implicit lock. A thread executing any synchronized code must first acquire this lock. This happens implicitly (ie there is no source syntax). Similarly, upon leaving the synchronized block the lock is implicitly released.
Java “Condition Variables”

Each synchronizable object is associated with a single implicit condition variable.

This is manipulated with methods `wait()`, `notify()` and `notifyAll()` (where “notify” is just Java-speak for “signal”).

Notice that this means, unlike Pthreads, that we can only have one conditional variable queue per monitor (hence the absence of any explicit syntax for the condition variable itself).

Like Pthreads, Java’s condition variable mechanism uses **Signal-and-Continue** semantics.
Readers & Writers

This problem requires us to control access to some shared resource (imagine a database, for example), such that there may be many concurrent readers, but only one writer (with exclusive access) at a time.

class ReadWrite {  // driver program -- two readers and two writers
    static Database RW = new Database();  // the monitor
    public static void main(String[] arg) {
        int rounds = Integer.parseInt(arg[0], 10);
        new Reader(rounds, RW).start();
        new Reader(rounds, RW).start();
        new Writer(rounds, RW).start();
        new Writer(rounds, RW).start();
    }
}
class Reader extends Thread {
    int rounds;  Database RW;
    private Random generator = new Random();
    public Reader(int rounds, Database RW) {
        this.rounds = rounds;
        this.RW = RW;
    }
    public void run() {
        for (int i = 0; i<rounds; i++) {
            try {
                Thread.sleep(generator.nextInt(500));
            } catch (java.lang.InterruptedException e) {} 
            System.out.println("read: "+ RW.read());
        }
    }
}
class Writer extends Thread {
    int rounds; Database RW;
    private Random generator = new Random();

    public Writer(int rounds, Database RW) {
        this.rounds = rounds;
        this.RW = RW;
    }

    public void run() {
        for (int i = 0; i<rounds; i++) {
            try {
                Thread.sleep(generator.nextInt(500));
            } catch (java.lang.InterruptedException e) {} 
            RW.write();
        }
    }
}
Readers & Writers

We now implement the “database” itself. Simply making both read and write operations synchronized is over restrictive - we would like it to be possible for several readers to be actively concurrently.

The last reader to leave will signal a waiting writer.

Thus we need to count readers, which implies atomic update of the count. A reader needs two protected sections to achieve this.

Notice that while readers are actually reading the data they do not hold the lock.
class Database {
    private int data = 0; // the data
    int nr = 0;
    private synchronized void startRead() {
        nr++;
    }
    private synchronized void endRead() {
        nr--;
        if (nr==0) notify(); // awaken a waiting writer
    }
    public int read() {
        int snapshot;
        startRead();
        snapshot = data;
        endRead();
        return snapshot;
    }
}
public synchronized void write() {
    int temp;
    while (nr>0)
        try { wait(); } catch (InterruptedException ex) {return;}

    temp = data;       // next six lines are the ‘‘database’’ update!
data = 99999;       // to simulate an inconsistent temporary state
    try {
        Thread.sleep(generator.nextInt(500));     // wait a bit
    } catch (java.lang.InterruptedException e) {} 
    data = temp+1;     // back to a safe state
    System.out.println("wrote: "+ data);

    notify();          // awaken another waiting writer
}

We could express the same effect with synchronized blocks

class Database {
    ....
    public int read() {
        int snapshot;
        synchronized (this) { nr++; }
        snapshot = data;
        synchronized (this) {
            nr--;
            if (nr==0) notify(); // awaken a waiting writer
        }
        return snapshot;
    }
}

Would it be OK to use notifyAll() in read()?
Buffer for One Producer - One Consumer

(borrowed from Skansholm, Java from the Beginning)

```java
public class Buffer extends Vector {
    public synchronized void putLast (Object obj) {
        addElement(obj); // Vectors grow implicitly
        notify();
    }

    public synchronized Object getFirst () {
        while (isEmpty())
            try {wait();} catch (InterruptedException e) {return null;}
        Object obj = elementAt(0);
        removeElementAt(0);
        return obj;
    }
}
```
Useful Packages

The java.util.concurrent package defines a number of useful classes, including a re-usable barrier and semaphores (with P() and V() called acquire() and release()). It also has some thread-safe concurrent data structures (queues, hash tables).

The java.util.concurrent.atomic package provides implementations of atomically accessible integers, booleans and so on, with atomic operations like addAndGet, compareAndSet.

The java.util.concurrent.locks package provides implementations of locks and condition variables, to allow a finer grained, more explicit control than that provided by the built-in synchronized monitors.
Programming without Shared Variables

It is possible to provide the illusion of shared variables, even when the underlying architecture doesn’t support physically shared memory (for example, by distributing the OS and virtual memory system).

Alternatively, we can make the disjoint nature of the address spaces apparent to the programmer, who must make decisions about data distribution and invoke explicit operations to allow interaction across these.

There are several approaches to abstracting and implementing such a model. We will focus on message passing, which dominates the performance-oriented parallel computing world.

We begin by examining some key issues. Later, we will see how these are realized in MPI, the standard library for message passing programming.
Programming with Message Passing

At its core, message passing is characterized as requiring the explicit participation of both interacting processes, since each address space can only be directly manipulated by its owner.

The basic requirement is thus for send and receive primitives for transferring data out of and into local address spaces.

The resulting programs can seem quite fragmented: we express algorithms as a collection of local perspectives. These are often captured in a single program source using Single Program Multiple Data (SPMD) style, with different processes following different paths through the same code, branching with respect to local data values and/or to some process identifier.
SPMD Compare-Exchange

co [me = 0 to P-1] { // assumes P is even
    int a, temp; // these are private to each process now
    ......
    // typical step within a parallel sorting algorithm
    if (me%2 == 0) {
        send (me+1, a); // send from a to process me+1
        recv (me+1, temp); // receive into temp from process me+1
        a = (a<=temp) ? a : temp;
    } else {
        send (me-1, a);
        recv (me-1, temp);
        a = (a>temp) ? a : temp;
    }
    ......
}
What’s in a Message?

In designing a message passing mechanism we must consider a number of issues.

Synchronization: Must a sending process pause until a matching receive has been executed (synchronous), or not (asynchronous)? For example, the compare-exchange code above will deadlock if we use synchronous semantics. Can you fix it? Asynchronous semantics require the implementation to buffer messages which haven’t yet been, and indeed may never be, received.

Addressing: When we invoke a send (or receive) do we have to specify a unique destination (or source) process or can we use wild-cards? Do we require program-wide process naming, or can we create process groups and aliases?

Collective Operations: Do we restrict the programmer to single-source, single-destination point-to-point messages, or do we provide abstractions of more complex data exchanges involving several partners?
Collectives: Broadcast

Everyone gets a copy of the same value.
Collectives: Scatter

Data is partitioned and spread across the group.
Collectives: Gather

Data is gathered from across the group.
Collectives: Reduction

Combine the gathered values with an associative operation.
Collectives: Scan (Prefix)

Reduce and also compute all the ordered partial reductions.
Message Passing Interface (MPI) Concepts

Processes can be created statically when the program is invoked (e.g. using the `mpirun` command) or spawned dynamically.

All communications take place within the context of “communication spaces” called communicators, which denote sets of processes. A process can belong to many communicators simultaneously. New communicators can be defined dynamically.

Simple send/receives operate with respect to other processes in a communicator. Send must specify a target but receive can wild card on matching sender.

Messages can be tagged with an extra value to aid disambiguation.

There are many synchronization modes and a range of collective operations.
Hello World in MPI

```c
int main(int argc, char *argv[]) {
    int rank, p;

    MPI_Init(&argc, &argv);

    // Explore the world
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    // Say hello
    printf("Hello world from process %d of %d\n", rank, p);

    MPI_Finalize();
}
```

Parallel Programming Languages and Systems
MPI Primitives

```c
int MPI_Init(int *argc, char ***argv)
int MPI_Finalize()
```

These must be called once by every participating process, before/after any other MPI calls. They return MPI_SUCCESS if successful, or an error code.

Each process has a unique identifier in each communicator of which it is a member (range 0..members-1). The built-in global communicator, to which all processes belong, is called MPI_COMM_WORLD. A process can find the size of a communicator, and its own rank within it.

```c
int MPI_Comm_Size (MPI_Comm comm, int *np)
int MPI_Comm_rank (MPI_Comm comm, int *me)
```
MPI Task Farm

A task farm is bag-of-tasks in which all the tasks are known from the start. The challenge is to assign them dynamically to worker processes, to allow for the possibility that some tasks may take much longer to compute than others.

To simplify the code, we assume that there are at least as many tasks as processors and that tasks and results are just integers. In a real application these would be more complex data structures.

Notice the handling of the characteristic non-determinism in the order of task completion, with tags used to identify tasks and results. We also use a special tag to indicate an “end of tasks” message.
#define MAX_TASKS 100
#define NO_MORE_TASKS MAX_TASKS+1
#define FARMER 0

int main(int argc, char *argv[]) {
    int np, rank;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    if (rank == FARMER) {
        farmer(np-1);
    } else {
        worker();
    }
    MPI_Finalize();
}
void farmer (int workers) {
    int i, task[MAX_TASKS], result[MAX_TASKS], temp, tag, who;
    MPI_Status status;

    for (i=0; i<workers; i++) {
        MPI_Send(&task[i], 1, MPI_INT, i+1, i, MPI_COMM_WORLD);
    }

    while (i<MAX_TASKS) {
        MPI_Recv(&temp, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        who = status.MPI_SOURCE; tag = status.MPI_TAG;
        result[tag] = temp;
        MPI_Send(&task[i], 1, MPI_INT, who, i, MPI_COMM_WORLD);
        i++;
    }
}

Parallel Programming Languages and Systems
for (i=0; i<workers; i++) {
    MPI_Recv(&temp, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    who = status.MPI_SOURCE; tag = status.MPI_TAG;
    result[tag] = temp;
    MPI_Send(&task[i], 1, MPI_INT, who, NO_MORE_TASKS, MPI_COMM_WORLD);
}

Notice that the final loop, which gathers the last computed tasks, has a pre-determined bound. We know that this loop begins after dispatch of the last uncomputed task, so there must be exactly as many results left to gather as there are workers.
void worker () {
    int task, result, tag;
    MPI_Status status;
    MPI_Recv(&task, 1, MPI_INT, FARMER, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    tag = status.MPI_TAG;

    while (tag != NO_MORE_TASKS) {
        result = somefunction(task);
        MPI_Send(&result, 1, MPI_INT, FARMER, tag, MPI_COMM_WORLD);
        MPI_Recv(&task, 1, MPI_INT, FARMER, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        tag = status.MPI_TAG;
    }
}

A worker is only concerned with its interaction with the farmer.
Send in standard mode

int MPI_Send(void *buf, int count,
             MPI_Datatype datatype,
             int dest, int tag,
             MPI_Comm comm)

Send ‘count’ items of given type starting in position ‘buf’, to process ‘dest’
in communicator ‘comm’, tagging the message with ‘tag’ (which must be non- negative).

There are corresponding datatypes for each basic C type, MPI_INT, MPI_FLOAT
etc, and also facilities for constructing derived types which group these together.

Are MPI_Send and MPI_Recv synchronous or asynchronous? We’ll come back to
this question soon!
Receive in standard mode

int MPI_Recv(void *buf, int count,
             MPI_Datatype datatype,
             int source, int tag,
             MPI_Comm comm,
             MPI_Status *status)

Receive ‘count’ items of given type starting in position ‘buf’, from process ‘source’ in communicator ‘comm’, tagged by ‘tag’.

Non-determinism (within a communicator) is achieved with “wild cards”, by naming MPI_ANY_SOURCE and/or MPI_ANY_TAG as the source or tag respectively.
Matching Receives

A receive can match any available message sent to the receiver which has the specified communicator, tag and source, subject to the constraint that messages sent between any particular pair of processes are guaranteed to appear to be non-overtaking. In other words, a receive cannot match message B in preference to message A if A was sent before B by the same process.

Status information is returned in a structure with status.MPI_SOURCE and status.MPI_TAG fields. This is useful in conjunction with wild card receives, allowing the receiver to determine the actual source and tag associated with the received message.
MPI uses the term **blocking** in a slightly unconventional way, to refer to the relationship between the caller of a communication operation and the implementation of that operation (ie nothing to do with any matching operation).
Synchronization in MPI

Thus, a **blocking send** completes only when it is safe to reuse the specified output buffer (because the data has been copied somewhere safe by the system).

```
Process A
x=25;
Send (&x, ... B) ---------> Recv (&y, ... A);  // blocking send
x=26;

We know y==25 for sure
```

In contrast, a process calling a **non-blocking send** continues immediately with unpredictable effects on the value actually sent.

```
Process A
x=25;
NBSend (&x, ... B) -------> Recv (&y, ... A);  // non-blocking send
x=26;

y could be 25 or 26!
```
Synchronization in MPI

Similarly, there is a non-blocking receive operation which allows the calling process to continue immediately, with similar issues concerning the values which appear in the buffer.

To manage these effects, there are MPI operations for monitoring the progress of non-blocking communications (effectively, to ask, “is it OK to use this variable now?”).

The idea is that with careful use these can allow the process to get on with other useful work even before the user-space buffer has been safely stored.
Blocking Communication Semantics in MPI

MPI provides four different blocking send operations (though we consider only the main three).

These vary in the level of synchronization they provide. Each makes different demands on the underlying communication protocol (ie the implementation).

**Synchronous mode** send (MPI_Ssend) is blocking and synchronous, only returning when a matching receive has been found.
**Blocking Communication Semantics in MPI**

**Standard mode** send (MPI_Send) is blocking. Its synchronicity depends upon the state of the implementation buffers, in that it will be asynchronous unless the relevant buffers are full, in which case it will wait for buffer space (and so may appear to behave in a “semi” synchronous fashion).

**Buffered mode** send (MPI_Bsend) is blocking and asynchronous, but the programmer must previously have made enough buffer space available (otherwise an error is reported). There are associated operations for allocating the buffer space.

**Receiving** with MPI_Recv blocks until a matching message has been completely received into the buffer (so it is blocking and synchronous).
Non-blocking Communication Semantics in MPI

MPI also provides non-blocking sends and receives which return immediately (i.e. possibly before it is safe to use/reuse the buffer). There are immediate versions of all the blocking operations (with an extra “I” in the name).

For example, MPI_Isend is the standard mode immediate send, and MPI_Irecv is the immediate receive.

Non-blocking operations have an extra parameter, called a ‘request’ which is a handle on the communication, used with MPI_Wait and MPI_Test to wait or check for completion of the communication (in the sense of the corresponding blocking version of the operation).
Probing for Messages

A receiving process may want to check for a potential receive without actually receiving it. For example, we may not know the incoming message size, and want to create a suitable receiving buffer.

```
int MPI_Probe(int src, int tag, MPI_Comm comm, MPI_Status *status)
```
behaves like `MPI_Recv`, filling in `*status`, without actually receiving the message.

There is also a version which tests whether a message is available immediately

```
int MPI_Iprobe(int src, int tag, MPI_Comm comm, int *flag, MPI_Status *status)
```
leaving a (C-style) boolean result in `*flag` (ie message/no message).
We can then determine the size of the incoming message by inspecting its status information.

```c
int MPI_Get_count(MPI_Status *status, MPI_Datatype t, int *count)
```
sets *count to the number of items of type t in message with status *status.

We could use these functions to receive (for example) a message containing an unknown number of integers from an unknown source, but with tag 75, in a given communicator comm.

```c
MPI_Probe(MPI_ANY_SOURCE, 75, comm, &status);
MPI_Get_count(&status, MPI_INT, &count);
buf = (int *) malloc(count*sizeof(int));
source = status.MPI_SOURCE;
MPI_Recv(buf, count, MPI_INT, source, 75, comm, &status);
```
Spawning New MPI Processes (non-examinable!)

```c
int MPI_Comm_spawn (char *command, char *argv[], int p,
                      MPI_Info info,
                      int root, MPI_Comm comm,
                      MPI_Comm *intercomm, int errcodes[])
```

This spawns \( p \) new processes, each executing a copy of program \( \text{command} \), in a new communicator returned as \( \text{intercomm} \).

To the new processes, \( \text{intercomm} \) appears as \( \text{MPI_COMM_WORLD} \). It must be called by all processes in \( \text{comm} \) (it is “collective”), with process root computing the parameters. \( \text{info} \) and \( \text{errcodes} \) are used in system dependent ways to control/monitor process placement, errors etc.

\( \text{MPI_Comm_get_parent} \) gives the new processes a reference to the communicator which created them.
Prime Sieve: The Generator

```c
int main(int argc, char *argv[]) {
    MPI_Comm nextComm; int candidate = 2, N = atoi(argv[1]);
    MPI_Init(&argc, &argv);
    MPI_Comm_spawn("siever", argv, 1, MPI_INFO_NULL, 0,
                   MPI_COMM_WORLD, &nextComm, MPI_ERRCODES_IGNORE);

    while (candidate<N) {
        MPI_Send(&candidate, 1, MPI_INT, 0, 0, nextComm);
        candidate++;
    }
    candidate = -1;
    MPI_Send(&candidate, 1, MPI_INT, 0, 0, nextComm);
    MPI_Finalize();
}
```
Prime Sieve: The Sieve

We use MPI_Comm_spawn to dynamically create new sieve processes as we need them, and MPI_Comm_get_parent to find an inter-communicator to the process group which created us.

```c
int main(int argc, char *argv[]) {
    MPI_Comm predComm, succComm; MPI_Status status;
    int myprime, candidate;

    int firstoutput = 1; // a C style boolean
    MPI_Init (&argc, &argv);

    MPI_Comm_get_parent (&predComm);
    MPI_Recv(&myprime, 1, MPI_INT, 0, 0, predComm, &status);
    printf("%d is a prime\n", myprime);
}
```
Prime Sieve: The Sieve (continued)

MPI_Recv(&candidate, 1, MPI_INT, 0, 0, predComm, &status);
while (candidate!=-1) {
    if (candidate%myprime) { // not sieved out
        if (firstoutput) { // create my successor if necessary
            MPI_Comm_spawn("siever", argv, 1, MPI_INFO_NULL, 0, MPI_COMM_WORLD,
                &succComm, MPI_ERRCODES_IGNORE);
            firstoutput = 0;
        }
        MPI_Send(&candidate, 1, MPI_INT, 0, 0, succComm) // pass on the candidate
    }
    MPI_Send(&candidate, 1, MPI_INT, 0, 0, succComm) // pass on the candidate
    MPI_Recv(&candidate, 1, MPI_INT, 0, 0, predComm, &status); // next candidate
    if (!firstoutput) MPI_Send(&candidate, 1, MPI_INT, 0, 0, succComm); // shut down
}
Collective Operations

MPI offers a range of more complex operations which would otherwise require complex sequences of sends, receives and computations.

These are called collective operations, because they must be called by all processes in a communicator.

For example, MPI_Bcast broadcasts count items of type t from buf in root to buf in all other processes in comm.

```c
int MPI_Bcast (void *buf, int count, MPI_Datatype t, int root, MPI_Comm comm)
```
Collective Operations

MPI_Scatter is used to divide the contents of a buffer across all processes.

```c
int MPI_Scatter (void *sendbuf, int sendcount,
                 MPI_Datatype sendt,
                 void *recvbuf, int recvcount,
                 MPI_Datatype recvvt,
                 int root, MPI_Comm comm)
```

$i^{th}$ chunk (of size sendcount) of root’s sendbuf is sent to recvbuf on process $i$ (including the root process itself).

The first three parameters are only significant at the root. Counts, types, root and communicator parameters must match between root and all receivers.
Collective Operations

The MPI_Allreduce operation computes a reduction, such as adding a collection of values together.

```c
int MPI_Allreduce (void *sendbuf, void *recvbuf, int count,
                  MPI_Datatype sendt, MPI_Op op, MPI_Comm comm)
```

reduces elements from all send buffers, point-wise, to count single values, using op, storing result(s) in all receive buffers.

The op is chosen from a predefined set (MPI_SUM, MPI_MAX etc) or constructed with user code and MPI_Op_create.
Jacobi Again (1-dimensional wrapped)
Jacobi Again (1-dimensional wrapped)

```c
int main(int argc, char *argv[]) {

    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0) read_problem(&n, work);

    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    mysize = n/p;        // assume p divides n, for simplicity
    local  = (float *) malloc(sizeof(float) * (mysize+2));  //include halo
    MPI_Scatter(work, mysize, MPI_FLOAT, &local[1], mysize,
                 MPI_FLOAT, 0, MPI_COMM_WORLD);

    left   = (rank+p-1)%p;  // who is my left neighbour?
    right  = (rank+1)%p;    // who is my right neighbour?

```
Jacobi Again (1-dimensional wrapped)

do {
    MPI_Sendrecv(&local[1], 1, MPI_FLOAT,left, 0, // send this
                  &local[mysize+1], 1, MPI_FLOAT,right,0, // and receive this
                  MPI_COMM_WORLD, &status); // anti-clockwise
    MPI_Sendrecv(&local[mysize], 1, MPI_FLOAT, right, 0,
                  &local[0], 1, MPI_FLOAT, left, 0,
                  MPI_COMM_WORLD, &status); // clockwise
    do_one_step(local, &local_error);
    MPI_Allreduce(&local_error, &global_error, 1,
                  MPI_FLOAT, MPI_MAX, MPI_COMM_WORLD);
} while (global_error > acceptable_error);

MPI_Gather (&local[1], mysize, MPI_FLOAT,
            work, mysize, MPI_FLOAT, 0, MPI_COMM_WORLD);

if (rank == 0) print_results(n, work);

MPI_Sendrecv combines a send and a receive, for convenience.
Communicators

Communicators define contexts within which groups of processes interact. All processes belong to MPI_COMM_WORLD from the MPI initialisation call onwards.

We can create new communicators from old ones by collectively calling

```
MPI_Comm_split(MPI_Comm old, int colour, int key, MPI_Comm *new)
```

to create new communicators for each distinct value of colour.

Within each new communicator, processes are assigned a new rank in the range $0..p' - 1$, where $p'$ is the size of the new communicator. Ranks are ordered by (but not necessarily equal to) the value passed in as the key parameter, with ties broken by considering process rank in the parent communicator.
Manipulating Communicators

This can be helpful in expressing algorithms which contain nested structure. For example, many divide-and-conquer algorithms split the data and machine in half, process recursively within the halves, then unwind to process the recursive results back at the upper level.

```c
void some_DC_algorithm ( ..., MPI_Comm comm) {
    MPI_Comm_size(comm, &p); MPI_Comm_rank(comm, &myrank);
    ... pre-recursion work ...
    if (p>1) {
        MPI_Comm_split (comm, myrank<(p/2), 0, &subcomm); // two sub-machines
        some_DC_algorithm ( ..., subcomm); // recursive step
        // in both sub-machines
    } else do_base_case_solution_locally();
    ... post-recursion work ...
}
```
Divide & Conquer Communicators

Parallel Programming Languages and Systems
Task and Pattern Based Models

Programming explicitly with threads (or processes) has some drawbacks.

- Natural expression of many highly parallel algorithms involves creation of far more threads than there are cores. Thread creation and scheduling have higher overheads than simpler activities like function calls (by a factor of 50-100).

- The OS has control over the scheduling of threads to processor cores, but it doesn’t have the application specific knowledge required to make intelligent assignments (for example to optimize cache re-use). Traditional OS concerns for fairness may be irrelevant or even counter-productive.

To avoid this, programmers resort to complex scheduling and synchronization of a smaller number of coarser grained threads. Can this be avoided?
Task and Pattern Based Models

A number of languages and libraries have emerged which

- separate the responsibility for identifying potential parallelism, which remains the application programmer’s job, from detailed scheduling of this work to threads and cores, which becomes the language/library run-time’s job

- provide abstractions of common patterns of parallelism, which can be specialized with application specific operations, leaving implementation of the pattern and its inherent synchronization to the system

These are sometimes called task based approaches, in contrast to traditional threaded models. Examples include OpenMP, which is a compiler/language based model, and Intel’s Threading Building Blocks library.
Threading Building Blocks

Threading Building Blocks is a shared variable model, C++ template-based library, produced by Intel.

It uses generic programming techniques to provide a collection of “parallel algorithms”, each of which is an abstraction of a parallel pattern. It also provides a direct mechanism for specifying task graphs and a collection of concurrent data structures and synchronization primitives.

It handles scheduling of tasks, whether explicit programmed or inferred from pattern calls, to a fixed number of threads internally. In effect, this is a hidden Bag-of-Tasks, leaving the OS with almost nothing to do.

(TBB slides are borrowed from https://parlab.eecs.berkeley.edu/sites/all-parlab-files-Slides_3.pdf)
The compiler creates the needed versions

template <typename T> T max (T x, T y) {
    if (x < y) return y;
    return x;
}

int main() {
    int i = max(20, 5);
    double f = max(2.5, 5.2);
    MyClass m = max(MyClass("foo"), MyClass("bar"));
    return 0;
}
Game of Life Original Code for a Step

```c
enum State {DEAD, ALIVE}; // cell status
typedef State **Grid;

void NextGen(Grid oldMap, Grid newMap) {
    int row, col, ncount;
    State current;

    for (row = 1; row <= MAXROW; row++)
        for (col = 1; col <= MAXCOL; col++) {
            current = oldMap[row][col];
            ncount = NeighborCount(oldMap, row, col);
            newMap[row][col] = CellStatus(current, ncount);
        }
}
```

Parallel Programming Languages and Systems
void NextGen(Grid oldMap, Grid newMap) {
    parallel_for (blocked_range<int>(1, maxrow+1), // Range
                   CompNextGen(oldMap, newMap),       // Body
                   affinity_partitioner());             // Partitioner
}

**Range** defines a task space, and its sub-division (partition) technique

**Body** defines the code which processes a range

**Partitioner** (optional parameter) influencing partitioning and scheduling strategy
Initial range is subdivided by the splitting constructor, undivided subranges are passed to the body code for processing.
The parallel_for Template

template<typename Range, typename Body>
void parallel_for(const Range& range, const Body &body);

• Requires definition of:
  – A range type to iterate over
    • Must define a copy constructor and a destructor
    • Defines is_empty()
    • Defines is_divisible()
    • Defines a splitting constructor, \( R(R &r, \text{split}) \)
  – A body type that operates on the range (or a subrange)
    • Must define a copy constructor and a destructor
    • Defines operator()
Game of Life parallel for Range Class

blocked_range is a built-in range class provided by TBB.

It represents a contiguous sequence of integers, between the given parameters, and can be queried for the beginning (r.begin()) and end (r.end()) of the range.

The TBB runtime can break a blocked_range into two smaller ranges, each (roughly) half the size.

Note that a blocked_range carries no problem data. The values in the range can be used as we choose, for example to index into arrays.
Range is Generic

• Requirements for `parallel_for` Range

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>R::R (const R&amp;)</code></td>
<td>Copy constructor</td>
</tr>
<tr>
<td><code>R::~R()</code></td>
<td>Destructor</td>
</tr>
<tr>
<td><code>bool R::is_empty() const</code></td>
<td>True if range is empty</td>
</tr>
<tr>
<td><code>bool R::is_divisible() const</code></td>
<td>True if range can be partitioned</td>
</tr>
<tr>
<td><code>R::R (R&amp; r, split)</code></td>
<td>Splitting constructor; splits r into two subranges</td>
</tr>
</tbody>
</table>

• Library provides predefined ranges
  – `blocked_range` and `blocked_range2d`

• You can define your own ranges
Game of Life parallel_for Body Class

class CompNextGen {
    Grid oldMap, newMap;

    public:
    CompNextGen (Grid omap, Grid nmap) : oldMap(omap), newMap(nmap) {} 

    void operator()( const blocked_range<int>& r ) const {
        for (int row = r.begin(); row < r.end(); row++)
            for (int col = 1; col <= maxcol; col++) {
                nState current = oldMap[row][col];
                int ncount = NeighborCount(oldMap, row, col);
                newMap[row][col] = CellStatus(current, ncount);
            }
    }
};
Body is Generic

• Requirements for `parallel_for` Body

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Body::Body(const Body&amp;)</code></td>
<td>Copy constructor</td>
</tr>
<tr>
<td><code>Body::~Body()</code></td>
<td>Destructor</td>
</tr>
<tr>
<td><code>void Body::operator()(Range&amp; subrange) const</code></td>
<td>Apply the body to subrange.</td>
</tr>
</tbody>
</table>

• `parallel_for` partitions original range into subranges, and deals out subranges to worker threads in a way that:
  – Balances load
  – Uses cache efficiently
  – Scales
TBB Partitioners

TBB lets us choose a partitioning strategy.

\[ \text{tbb::parallel_for( range, body, tbb::simple_partitioner() );} \]

forces all ranges to be fully partitioned (i.e. until \text{is_divisible()} \) fails).

\[ \text{tbb::parallel_for( range, body, tbb::auto_partitioner() );} \]

allows the TBB runtime to decide whether to partition the range (to improve granularity)

\[ \text{tbb::parallel_for( range, body, tbb::affinity_partitioner );} \]

is like \text{auto_partitioner()} but also, when the \text{parallel_for} is inside a loop, tries to allocate the same range to the same processor core across iterations to improve cache behaviour.
void NextGen(Grid oldMap, Grid newMap) {
    parallel_for (blocked_range2d<int, int> (1, maxrow+1, 1, maxcol+1), // Range
                  CompNextGen(oldMap, newMap)); // Body
    affinity_partitioner(); // Partitioner
}

blocked_range2d is the natural 2D extension of the blocked_range class.

It is partitioned in alternating dimensions, level by level.
How splitting works on

`blocked_range2d`

- Tasks available to be scheduled to other threads (thieves)
- Split range...
  - Recursively...
  - Until \( \leq \) grainsize.
class CompNextGen {
    Grid oldMap, Grid newMap;

public:

    CompNextGen (Grid omap, Grid nmap) : oldMap(omap), newMap(nmap) {}

    void operator()( const blocked_range2d<int, int>& r ) const {
        for (int row = r.rows().begin(); row < r.rows().end(); row++)
            for (int col = r.cols().begin(); col < r.cols().end(); col++) {
                State current = oldMap[row][col];
                int ncount = NeighborCount(oldMap, row, col);
                newMap[row][col] = CellStatus(current, ncount);
            }
    }
};

Game of Life Using a 2D decomposition
Game of Life 1D with C++11 Lambda Function

```cpp
void NextGen(Grid oldMap, Grid newMap) {

    parallel_for (blocked_range<int>(1, maxrow+1),
        [&](const blocked_range<int>& r){
            for (int row = r.begin(); row < r.end(); row++)
                for (int col = 1; col <= MAXCOL; col++) {
                    State current = oldMap[row][col];
                    int ncount = NeighborCount(oldMap, row, col);
                    newMap[row][col] = CellStatus(current, ncount);
                }
        });

    An alternative interface to parallel_for allows us to use a C++ lambda expression to avoid writing a body class.
```
TBB parallel_reduce Template

TBB parallel_reduce has similar structure to parallel_for but additionally allows bodies to gather results internally as they go along.

We could parallelize the following example with a parallel_for, but we would need a critical section of some kind to accumulate the partial results.

parallel_reduce structures and hides this, with one further generic operation, called join.
A parallel\_reduce Example

In this example, we use parallel\_reduce to compute a numerical approximation to $\Pi$.

The approximation is based on

$$\Pi = \int_0^1 \frac{4}{1 + x^2} dx$$

but you don’t have to understand why (or even know what the above means!). Just take the sequential code on the next slide as the computation we need to parallelize.
Numerical Integration Example

```c
static long num_steps = 100000;
double step, pi;

void main(int argc, char* argv[])
{
    int i;
    double x, sum = 0.0;

    step = 1.0/(double) num_steps;
    for (i = 0; i < num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0 + x*x);
    }
    pi = step * sum;
    printf("Pi = %f\n",pi);
}
```
The parallel_reduce Template

template <typename Range, typename Body>
void parallel_reduce (const Range& range, Body &body);

- **Requirements for parallel_reduce**

  | Body::Body( const Body&, split ) | Splitting constructor |
  | Body::~Body() | Destructor |
  | void Body::operator() (Range& subrange) const | Accumulate results from subrange |
  | void Body::join( Body& rhs ); | Merge result of rhs into the result of this. |

- Reuses Range concept from parallel_for
# Include necessary libraries

```cpp
#include "tbb/parallel_reduce.h"
#include "tbb/task_scheduler_init.h"
#include "tbb/blocked_range.h"
```

```cpp
using namespace tbb;
```

```cpp
int main(int argc, char* argv[]) {
    double pi;
    double width = 1./(double)num_steps;
    MyPi step((double *const)&width);
    task_scheduler_init init;

    parallel_reduce(blocked_range<int>(0,num_steps),
                    step,
                    auto_partitioner());

    pi = step.sum*width;

    printf("The value of PI is %15.12f\n",pi);
    return 0;
}
```
parallel_reduce Example

class MyPi {
    double *const my_step;
public:
    double sum;
    MyPi(double *const step) : my_step(step), sum(0) {}
    MyPi(MyPi& x, split) : my_step(x.my_step), sum(0) {}
    void operator()( const blocked_range<size_t>& r ) {
        double step = *my_step;
        double x;
        for (int i = r.begin(); i < r.end(); i++)
            {
                x = (i + .5)*step;
                sum = sum + 4.0/(1.+ x*x);
            }
    }
    void join( const MyPi& y ) {sum += y.sum;}
};

accumulate results
join
int main(int argc, char* argv[]) {
   double pi;
   double width = 1.0/(double)num_steps;

   pi = parallel_reduce(
      blocked_range<size_t>(0, num_steps),
      double(0),
      [&]( blocked_range<size_t>& r, double current_sum ) -> double {
         for (size_t i = r.begin(); i < r.end(); ++i) {
            double x = (i+0.5)*step;
            current_sum += 4.0/(1.0 + x*x);
         }
         return current_sum;
      },
     [](double s1, double s2 ) {
         return s1+s2;       // "joins" two accumulated values
      });

   pi = pi *= step;
   printf("The value of PI is %15.12f\n",pi);
   return 0;
}
**TBB DIY Range Example**

This example shows how we can build our own ranges. We compute Fibonacci numbers (NB this is not a sensible algorithm, just a short clear example!).

```cpp
class FibRange {
public:
    int n_; // represents the ‘‘range’’ corresponding to fib(n)
    FibRange(int n) : n_(n) { }

    FibRange(FibRange& other, split)
    : n_(other.n_ - 2) // initialize the new object
    { other.n_ = other.n_ - 1; } // reuse the other range object

    bool is_divisible() const { return (n_ > 10); } // sequential threshold

    bool is_empty() const { return n_ < 0; }
};
```

Parallel Programming Languages and Systems
The Fib Body Class (with operator())

class Fib {
public:
    int fsum_;
    Fib() : fsum_(0) { }
    Fib(Fib& other, split) : fsum_(0) { }

    // NB use of += since each body may accumulate more than one range
    void operator() (FibRange& range) { fsum_ += fib(range.n_); }

    int fib(int n) {if (n < 2) return 1; else return fib(n-1)+fib(n-2);}

    void join(Fib& rhs) { fsum_ += rhs.fsum_; }
};
Fibonacci with DIY Range

int main( int argc, char* argv[] ) {

    Fib f;

    parallel_reduce(FibRange(FIBSEED), f, simple_partitioner());

    cout << "Fib " << FIBSEED << " is " << f.fsum_ << "\n";
    return 0;
}

As we have seen, using a simple_partitioner forces full splitting of the ranges. We could use auto_partitioner to let the TBB run-time system control this.
Manipulating Tasks Directly

All of TBB’s parallel pattern constructs are implemented via the same underlying task scheduler, which executes a task graph representing the pattern.

TBB also allows the programmer to (carefully!) create task graphs directly. This allows expression of unstructured task graphs, or the implementation and abstraction of further patterns.

There are functions to create new tasks as children of existing tasks and to specify the control dependencies between them.

We now see how to code Fibonacci using tasks directly. The key method is now task::execute, which we override with our application specific behaviour.
Example: Naive Fibonacci Calculation

- Recursion typically used to calculate Fibonacci number
  - Easy to code
  - Has unbalanced task graph

```c
long SerialFib( long n ) {
    if( n < 2 )
        return 1;
    else {
        int x = SerialFib(n-1);
        int y = SerialFib(n-2);
        return x + y;
    }
}
```
Fibonacci - Task Spawning Solution

- Use TBB tasks to thread creation and execution of task graph
- Create new root task
  - Allocate task object
  - Construct task
- Spawn (execute) task, wait for completion

```cpp
#include "tbb/task.h"

long ParallelFib(long n) {
    long sum;
    FibTask& a = *new(task::allocate_root()) FibTask(n, &sum);
    task::spawn_root_and_wait(a);
    return sum;
}
```
Fibonacci - Task Spawning Solution

class FibTask: public task {
public:
    const long n;
    long* const sum;
    FibTask( long n_, long* sum_ ) : n(n_), sum(sum_){}

task* execute() {     // Overrides virtual function \texttt{task::execute}
    if( n < \texttt{Cutoff} ) {
        *sum = SerialFib(n);
    } else {
        long x, y;
        FibTask& a = *new( \texttt{allocate\_child()} ) FibTask(n-1,&x);
        FibTask& b = *new( \texttt{allocate\_child()} ) FibTask(n-2,&y);
        set\_ref\_count(3); // 3 = 2 children + 1 for wait
        spawn( b );
        spawn\_and\_wait\_for\_all( a );
        *sum = x+y;
    }
    return NULL;
}
};
TBB Scheduler

At any point in execution, the collection of known tasks is maintained as a shared graph. Each thread maintains its own double-ended queue of ready tasks (roughly, as pointers into the shared graph).

Newly spawned tasks are added to the front of the local queue.

A thread requiring a task looks:

- firstly, at the front of its local queue, which encourages locality within one thread’s work;

- failing this, at the back of a randomly chosen other thread’s queue, which encourages stealing of big tasks, and discourages locality across threads.
TBB Scheduler

- Normal thread
- Thief thread (Empty)
- Victim thread

- Old
- New

Get/Put

Steal

Task Graph
**Linda**

Linda presents an alternative conceptual model for parallelism, based around a small library of operations. The Linda model saw something of a revival in distributed java systems programming, under the name JavaSpaces.

The key concept is that processes interact through tuple space, a global, content addressable memory. Each tuple is an ordered collection of typed data fields. Duplicate tuples are allowed.
Linda Operations

Processes run asynchronously and can operate on tuple space with six operations.

To add a new tuple to tuple space we call `out(exp1, exp2, ...., expN);`

This evaluates the expressions in the parameter list before atomically placing a copy of the results as a new tuple in tuple space.

For example,

```
out("Green", x*y, square(2));
```

create a new three value tuple.

Recalling our discussion of message-passing, one way of looking at `out` is as an asynchronous send with a wild-card destination.
Linda Operations

To take a tuple from tuple space we call \texttt{in(tuple-template)};

This \textit{atomically removes} from tuple space a tuple which \textit{matches the template}.

The template contains actual values and formal parameters (indicated by \texttt{?}) to be assigned during the match. A match occurs with any tuple for which all actual values match and the types on formal parameters match. \texttt{in} is blocking, in the sense that the caller is suspended until a matching tuple becomes available. For example,

\begin{verbatim}
  in("Green", ?y, ?r, FALSE);
\end{verbatim}

As above, we could think of \texttt{in} as a blocking, asynchronous receive, with wild-card source, but with additional constraints implied by the pattern matching.
Linda Operations

It is also possible to \textit{atomically} match (and copy the values from) a tuple without removing it from tuple space with \texttt{rd (tuple-template)};

Tuples may also be created with \texttt{eval(expr, expr, \ldots)} which is like \texttt{out}, but \textbf{dynamically creates new processes} to evaluate each field of the tuple which has been expressed as a function call. The calling process continues immediately, and the resulting tuple enters tuple space atomically when all the newly sparked processes have terminated.

Finally, there are \textbf{non-blocking} forms \texttt{inp, rdp (p for “predicate”)} which complete “immediately”, returning a boolean indicating whether or not a match occurred.
Linda Example: Bag of Tasks

We use a "counts" tuple, in effect as a shared variable, to count the number of tasks and number of idle workers. The final field in a task tuple indicates whether this is a real task or a “no more tasks” signal.

```c
int main () {
    out("total", 0.0); out("counts", 1, P);
    out ("task", a, b, f(a), f(b), approxarea, FALSE); // make initial task
    for (i = 0; i<P; i++) eval (worker());
    in ("counts", 0, P); // no tasks left
    // and P workers idle
    in ("total", ?total); // get the result
    out ("task", 0.0, 0.0, 0.0, 0.0, 0.0, TRUE); // indicate no more tasks
    ... use total ...
}
```

Parallel Programming Languages and Systems
```c
int worker () {
    while (true) {
        if (gameOver) {
            out ("task", 0.0, 0.0, 0.0, 0.0, 0.0, TRUE); // for others to see
            break;
        }
        in("counts", ?size, ?idle); out("counts", size-1, idle-1);
        ... usual task calculations ...
        if (abs (larea + rarea - lrarea) > EPSILON) { // create new tasks
            out("task", left, mid, fleft, fmid, larea, FALSE);
            out("task", mid, right, fmid, fright, rarea, FALSE);
            in("counts", ?size, ?idle); out("counts", size+2, idle+1);
        } else {
            in ("total", ?total); out ("total", total+larea+rarea);
            in("counts", ?size, ?idle); out("counts", size, idle+1);
        }
    }
}
```
Linda Example: Pipeline

By way of contrast, here is a Linda version of the prime sieve pipeline, which finds all the primes in the range 2..LIMIT.

We use eval() to create the sieve processes dynamically as we need them.

The sieve processes eventually turn into part of an “array” of primes in tuple space.

We ensure pipelined message flow by tagging tuples with their destination and position in the sequence.
void main (int argc, char *argv[]) {

    int i;

    eval("prime", 1, sieve(1));
    for (i=2; i<LIMIT; i++) {
        out("number", 1, i-1, i);
    }
}
int sieve (int me) {
    int n, p, in_seq=1, out_seq=1, stop=FALSE;

    in("number", me, in_seq, ?p);   // first arrival is prime
    while (!stop) {
        in_seq++;
        in("number", me, in_seq, ?n);  // get the next candidate
        if (n==LIMIT) {
            stop = TRUE; out("number", me+1, out_seq, n); // pass on the signal
        } else if (n%p !=0) {
            if (out_seq == 1) eval("prime", me+1, sieve(me+1)); // new sieve
            out("number", me+1, out_seq, n);                     // and its first input
            out_seq++;
        }
    }
    return p;
}
Implementing Tuple Space

Linda’s powerful matching model sets a demanding implementation challenge, way beyond the associative memory hardware used in on-chip caches.

Indexing and hashing techniques adapted from relational database technology can help (e.g. think about the relationship between Linda’s “rd” and SQL’s “select”).

Advanced Linda implementations perform considerable compile-time analysis of program specific tuple usage. For example, possible tuples (in a given program) can be categorised into a set of classes by type signature, and stored separately.
Review

The first slide said

“This course is about bridging the gap between the parallel applications and algorithms which we can design and describe in abstract terms and the parallel computer architectures (and their lowest level programming interfaces) which it is practical to construct.”

We have examined some of the prominent approaches which address this challenge, but there are others, and this remains a hot research area.

At most, we are at the end of the beginning ....