Parallel Programming Languages and Systems

Murray Cole

e-mail: mic  room: 2508
What?

This course is about bridging the gap between the parallel algorithms and applications 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 (‘‘easy’’ to program correctly, adaptable), 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 case, but the performance issue is vital now.
Why?

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

- **high performance computing**: parallelism, at various levels in the system, is the primary means by which the execution time of computationally demanding applications can be reduced;

- **distributed computing**: concurrency is inherent in the nature of distributed systems. We have no choice but to express and control it.

- **timesliced/real-time computing**: it is sometimes conceptually simpler to think of a program as being composed of concurrent components, even though these will actually all be executed by sharing a single processor.
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 algorithms.

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

We will do this first for the two main conventional models, shared variable programming, and distributed (message passing) programming, before considering some of the more novel approaches which have also been suggested.
Terminology

Unfortunately, this field is awash with imprecise, confusing and mutually inconsistent use of jargon (and of course, everyone is convinced that they are right and everyone else is wrong!).

We will not add to the list of “correct” definitions. However, you should be aware that Andrews tends to use the term “multithreaded” to mean conceptually concurrent but typically time-sliced, “distributed” to mean “without shared variables”, and “parallel” to mean having “reduced execution time through concurrency” as a goal, irrespective of model.

He also tends to use “process” to refer to any sequence of activities, whether these interact through shared variables or messages (or some other means). Most real systems use “process” to refer to activities with distinct address spaces, and “thread” to refer to concurrent activities within an address space.
Parallel Architecture for Dummies

Figure 1.2 Structure of Shared-Memory Multiprocessors.

Copyright © 2000 by Addison Wesley Longman, Inc.

Shared Memory Multiprocessor. One physical address space, shared by all processors.

Caches improve performance as usual, but now we have a cache coherence problem. Solved in hardware and OS software. Expensive.
Parallel Architecture for Dummies

Also raises the memory consistency issue, which trades off conceptual simplicity against cost (time or hardware complexity).

Roughly speaking, when, and in what order should updates to memory made by one processor become visible to others?

Sequential consistency: every processor “sees” the same sequential interleaving.

Processor consistency: writes by any one processor will be seen by all in the same order, but different processors may see different interleavings.

Release consistency: memory updates are seen only guaranteed to be visible by program specified synchronisation points (triggered by special machine instructions). Even the ordering as written by one processor may be seen differently by different other processors.
Parallel Architecture for Dummies

Consistency models must be understood and respected by programmers/compilers/libraries to ensure program correctness.

Shared address space architectures also raise complex performance issues.

Uniform Memory Access (UMA) architectures (aka SMPs) have all memories “equidistant” (in access time) from all processors. NUMA (N for “non”) architectures don’t, so it matters (in performance terms) where data resides.

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 in the worst case).
Distributed Memory Multiprocessor. Each processor has its own physical address space, so no coherence/consistency issues. Information is shared by explicit (usually co-operative) copying between memories (address spaces).
Parallel Architecture for Dummies

Many ways to express this. Performance/correctness issues include choice between one or two sided communications, use (and abuse) of buffering, semantics of synchronisation and constraints on message ordering.

Arguably, architectural parallelism is ultimately always about communication, with the distinction between models and architectures concerning the extent to which this is visible and controllable at various levels of abstraction.
Low Level Models: Smoke and Mirrors

Most (all?) general purpose processor/OS combinations support Virtual Memory (i.e. virtual address spaces).

The model of parallelism (and in particular, memory) presented to the application programmer doesn’t have to match the physical architectural model!

A shared (physical) memory machine could be programmed

- with multiple threads, running on distinct processors but sharing a single virtual address space (ie all within one “process”)

- with multiple processes, running on distinct processors, each with its own address space, exchanging data by messages (which are actually implemented as memory copies)
Low Level Models: Smoke and Mirrors

Similarly, a distributed (physical) memory machine could be programmed

- with multiple processes, running on distinct processors, each with its own address space, exchanging data by messages (which are actually implemented as real communications on a physical network device)

- with multiple threads, running on distinct processors but sharing a single virtual address space, implemented by an extension of the virtual memory system with page faults causing a flurry of rectifying messages!
Low Level Models: Smoke and Mirrors

Logical behaviour will be independent of the underlying mechanism, but performance will be greatly affected by it (a page fault every time you want to share an update to an integer variable?!).

We will focus on working with a given logical model to provide programming constructs which support correct control of parallel behaviour, while occasionally noting the physical roots which impact upon performance.
Programming Patterns

To help understand and design parallel programming frameworks we should also understand something about the requirements of typical parallel algorithms.

In principle, parallelism can involve arbitrary, dynamically determined interactions between arbitrary, dynamically created activities (processes, threads).

In practice, the interactions between activities in many parallel applications follow one or more of a number of informally defined patterns.

NB: even the names of the patterns are not commonly agreed!
The Iterative Parallelism Pattern

Derived from sequential programs in which the bulk of the work is done by loops which traverse large arrays.

When inter (and intra) loop dependencies allow, the iterations can be executed concurrently (i.e. all at the same time, rather than in sequence).

The parallelism produced is often static (i.e. it can be determined at compile time).
Matrix Multiplication

double a[n,n], b[n,n], c[n,n];

for [i = 0 to n-1] {
    for [j = 0 to n-1] {
        # compute inner product of a[i,*] and b[*,j]
        c[i,j] = 0.0;
        for [k = 0 to n-1]
            c[i,j] = c[i,j] + a[i,k]*b[k,j];
    }
}

Sequential Matrix Multiplication

Copyright © 2000 by Addison Wesley Longman, Inc.

This is “embarrassingly parallel”. All iterations of the outer two loops are independent in terms of computed values (though they do share input values).
Matrix Multiplication

\[
\begin{align*}
\text{co } [i = 0 \text{ to } n-1] & \{ \quad \# \text{ compute rows in parallel} \\
& \text{for } [j = 0 \text{ to } n-1] \{ \\
& \quad c[i,j] = 0.0; \\
& \quad \text{for } [k = 0 \text{ to } n-1] \\
& \quad \quad c[i,j] = c[i,j] + a[i,k]*b[k,j]; \\
& \quad \} \\
& \} \\
\end{align*}
\]

Parallel Matrix Multiplication by Rows

Use \( n \) concurrent activities (with shared variables) to compute the rows in parallel.
The co Notation

The co notation indicates creation of a set of activities, for the duration of the enclosed block, with synchronisation at the end of the block.

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.

We will also allow co statements with several indices, for iterative parallelism from nested sequential loops.

We will allow co statements with several distinct sub-statements, separated by //) to express parallelism not generated from iteration.

[In Andrews’ book, comments are indicated by #, not //]
Matrix Multiplication

\[
\begin{align*}
&\text{co } [i = 0 \text{ to } n-1, \ j = 0 \text{ to } n-1] \ { # \ all \ rows \ and} \\
&\quad c[i, j] = 0.0; \ # \ all \ columns \\
&\quad \text{for } [k = 0 \text{ to } n-1] \\
&\quad \quad c[i, j] = c[i, j] + a[i, k]*b[k, j]; \\
&\}
\end{align*}
\]

Parallel Matrix Multiplication by Rows and Columns

Copyright © 2000 by Addison Wesley Longman, Inc.

Use \( n \times n \) concurrent activities (with shared variables) to compute the all entries of \( c \) concurrently (though computation of any individual entry is sequential).
The process Notation

The process notation indicates creation of a set of activities, each executing the enclosed block in an (by default) unsynchronised manner.

Statements after a process statement begin execution immediately after the process activities have been created (unlike the corresponding situation with a co statement).

We will need to express any required synchronisation explicitly with other language mechanisms.
Matrix Multiplication

process row[i = 0 to n-1] { # rows in parallel
    for [j = 0 to n-1] {
        c[i,j] = 0.0;
        for [k = 0 to n-1]
            c[i,j] = c[ij] + a[i,k]*b[k,j];
    }
}

Parallel Matrix Multiplication Using a Process Declaration

Copyright © 2000 by Addison Wesley Longman, Inc.

Essentially the same as before, but statements after this block would need to be prevented from accessing c before it was ready.
Matrix Multiplication

process worker[w = 1 to P] { # strips in parallel
    int first = (w-1) * n/P; # first row of strip
    int last = first + n/P - 1; # last row of strip
    for [i = first to last] {
        for [j = 0 to n-1] {
            c[i,j] = 0.0;
            for [k = 0 to n-1]
                c[i,j] = c[i,j] + a[i,k]*b[k,j];
        }
    }
}

Parallel Matrix Multiplication by Strips (Blocks)

Copyright © 2000 by Addison Wesley Longman, Inc.

Here we recognize that we may have fewer than \( n \) real processors, and assign computation of a strip of rows of \( c \) to each of \( P \) processes.
Mesh Modelling - Jacobi

Modelling of a physical system (formally expressed as a system of partial differential equations) by approximation on a matrix of points.

initialize the matrix, including fixed boundary values;
while (not yet terminated) {
    compute a new value for each point;
    check for termination;
}

The “compute” step usually involves only a small number of neighbouring points.
The termination step usually looks for convergence (i.e. small difference in point values from one iteration to the next, indicating approximate solution of the p.d.e.s)
Mesh Modelling - Jacobi

```c
real grid[n+1,n+1], newgrid[n+1,n+1];
bool converged = false;
process Grid[i = 1 to n, j = 1 to n] {
    while (not converged) {
        newgrid[i,j] = (grid[i-1,j] + grid[i+1,j] + 
                        grid[i,j-1] + grid[i,j+1]) / 4;
        check for convergence as described in the text;
        barrier(i);
        grid[i,j] = newgrid[i,j];
        barrier(i);
    }
}
```

**Figure 3.19** Grid computation for solving Laplace’s equation.
Mesh Modelling - Jacobi

This is a simple (naive) shared variable solution, but it illustrates some useful points

• we need to synchronize at various points within the loop

• we need to think about how to co-operate in the check for termination

It would certainly be a good idea in practice to use a more realistic (smaller) number of processes.
The Recursive Parallelism Pattern

This emerges from divide-and-conquer algorithms, which naturally generate a tree of calls to the same function, with different parameters.

Sometimes the tree can be statically determined, sometimes it only emerges dynamically, in a data-dependent way.

Recursive calls can be executed concurrently.

Typically we need to combine results from the recursive calls to produce a result for the original call, so the tree of work expands then contracts.
Adaptive Quadrature

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, right, fleft, fright, lrarea) {
    double mid = (left + right) / 2;
    double fmid = f(mid);
    double larea = (fleft + fmid) * (mid - left) / 2;
    double rarea = (fmid + fright) * (right - mid) / 2;
    if (abs((larea + rarea) - lrarea) > EPSILON) {
        // recurse to integrate both halves
        larea = quad(left, mid, fleft, fmid, larea);
        rarea = quad(mid, right, fmid, fright, rarea);
    }
    return (larea + rarea);
}

Recursive Procedure for Quadrature Problem

Copyright © 2000 by Addison Wesley Longman, Inc.
Adaptive Quadrature

To compute the whole approximation we call

```plaintext
area = quad (a, b, f(a), f(b), (f(a)+f(b))*(b-a)/2);
```

Noting that the recursive calls to `quad` do not interfere with each other, we can parallelize the program by changing the calls to

```plaintext
c0
  larea = quad(left, mid, fleft, fmid, larea);
  // rarea = quad(mid, right, fmid, fright, rarea);
c0
```

The synchronisation inherent in `c0` ensures that both `larea` and `rarea` have been computed before being added together and returned.
The Pipeline Parallelism Pattern

The **producers and consumers** pattern arises when a number of processes/threads generate data which is consumed by some other processes/threads.

Typically we want to allow production and consumption to be only loosely synchronized. We will need some buffering in the system.

When one group of consumers become the producers for yet another group, we have a **pipeline**.
Parallelizing grep

grep is the Unix utility which searches a file line by line for instances of a given pattern.

```c
string line;
read a line of input from stdin into line;
while (!EOF) { #EOF is end of file
    look for pattern in line;
    if (pattern is in line)
        write line;
    read next line of input;
}
```

Finding Patterns: Sequential Program

Copyright © 2000 by Addison Wesley Longman, Inc.
Parallelizing grep

A simple parallelization might introduce two processes, one of which reads lines, the other of which checks lines for the pattern.

They might interact through a buffer which can contain a single line.

The programming challenge is to ensure that the reading process doesn’t overwrite the buffer before the checker has checked it, and that the checker doesn’t try to “check” a line that doesn’t really exist (or to re-check a line it has already checked).
Parallelizing grep

```c
string buffer;  // contains one line of input
bool done = false;  // used to signal termination

// # process 1: find patterns
string line1;
while (true) {
    wait for buffer to be full or done to be true;
    if (done) break;
    line1 = buffer;
    signal that buffer is empty;
    look for pattern in line1;
    if (pattern is in line1)
        write line1;
}

// # process 2: read new lines
string line2;
while (true) {
    read next line of input into line2;
    if (EOF) {done = true; break;}
    wait for buffer to be empty;
    buffer = line2;
    signal that buffer is full;
}
```

Figure 2.1 Finding patterns in a file.
The Interacting Peers Pattern

The *interacting peers* pattern is usually associated with distributed (no shared variable) programming, often when implementing algorithms from the “iterative parallelism” class.

There is no “master”, “root” or “controller” process, except possibly at the very beginning and end to distribute and gather data and results.

All processes execute more or less the same code on a distinct partition of the data (sometimes also called SPMD, for “single program multiple data”). Exchanging messages, usually in a fixed pattern, to share information and keep loosely synchronized.
process worker[i = 0 to n-1] {
    double a[n];       # row i of matrix a
    double b[n];       # one column of matrix b
    double c[n];       # row i of matrix c
    double sum = 0.0;   # storage for inner products
    int nextCol = i;    # next column of results
    receive row i of matrix a and column i of matrix b;
    # compute c[i,i] = a[i,*] × b[*,i]
    for [k = 0 to n-1]
        sum = sum + a[k] * b[k];
    c[nextCol] = sum;
    # circulate columns and compute rest of c[i,*]
    for [j = 1 to n-1] {
        send my column of b to the next worker;
        receive a new column of b from the previous worker;
        sum = 0.0;
        for [k = 0 to n-1]
            sum = sum + a[k] * b[k];
        if (nextCol == 0)
            nextCol = n-1;
        else
            nextCol = nextCol-1;
        c[nextCol] = sum;
    }
    send result vector c to coordinator process;
}
The Bag of Tasks Pattern

The bag of tasks (or “task farm”, “process farm” or just “farm”) pattern allows a fixed number of worker processes/threads to maintain and process a dynamically varying collection of self-contained “tasks”, each usually of the same type.

Execution of a particular task may lead to the creation of more task instances.

Tasks may achieve their overall effect by returning some result, or by the side-effect of updating some data structure.

We can implement the pattern on top of shared variables or message passing.
type task = (double left, right, fleft, fright, lrarea);
queue bag(task); # the bag of tasks
int size; # number of tasks in bag
int idle = 0; # number of idle workers
double total = 0.0; # the total area

compute approximate area from a to b;
insert task (a, b, f(a), f(b), area) in the bag;
count = 1;

process Worker[w = 1 to PR] {
    double left, right, fleft, fright, lrarea;
    double mid, fmid, larea, rarea;
    while (true) {
        # check for termination
        idle++;
        if (idle == n && size == 0) break;
    }

    # get a task from the bag
    await (size > 0)
        remove a task from the bag;
        size--; idle--; }
    mid = (left+right) / 2;
    fmid = f(mid);
    larea = (fleft+fmid) * (mid-left) / 2;
    rarea = (fmid+fright) * (right-mid) / 2;
    if (abs((larea+rarea) - lrarea) > EPSILON) {
        # put (left, mid, fleft, fmid, larea) in the bag;
        # put (mid, right, fmid, fright, rarea) in the bag;
        size = size + 2;
    } else
        total = total + lrarea; }

    if (w == 1) # worker 1 prints the result
        printf("the total is \%f\n", total);
}

Figure 3.21 Adaptive quadrature using a bag of tasks.
The Bag of Tasks Pattern

The preceding code introduces some new syntax:

- statements enclosed in ⟨ and ⟩ must be executed atomically (without interference from other activities)

- the await statement allows us to indicate that execution of the body must be delayed until some condition arises (note also the atomicity of the check and the resulting action)

Also notice the care we need to exercise in checking for termination: this should occur when there are no tasks left and all workers are inactive but in fact, the code in 3.21 is buggy! We will think about how to fix it in an exercise.
Shared Variable Programming

A collection of concurrent activities (normally “threads”) own a set of shared variables, which they can access with some collection of atomic actions (for example, reads and writes).

An execution of a program corresponds to an interleaving of the individual thread operation sequences. [NB this is what computer architects would call “sequential memory consistency” - we have to be careful if the real architecture doesn’t support this directly].

There are many possible interleavings (“histories”), some of which will be “correct” (in terms of what the program is meant to achieve), others not.

The purpose of synchronization is to constrain execution so that only correct interleavings can occur.
Synchronization

There are two fundamental kinds of synchronization:

- **Mutual Exclusion** is more like anti-synchronization! We want to prevent two threads from being active concurrently (because their actions may interfere incorrectly).

- **Condition Synchronization** occurs when we want to delay some action until some condition (on the shared variables, or with respect to the progress of other threads) becomes true.

We will see a range of real primitives which support these in different situations, and will initially use the “pseudo primitives” ⟨ ⟩ and await to specify what we need.
Locks & Critical Sections

The simplest instantiation of mutual exclusion is the critical section problem. This occurs when $n$ processors execute code of the following form, in which it is essential that at most one process is in the critical section at a time (for example, because of potentially unsafe access to shared variables)

```plaintext
process CS[i = 1 to n] {
    while (something) {
        < critical section; >
        non-critical section;
    }
}
```

The challenge is to implement the atomic $< >$. We must design some code to execute before (an entry protocol) and after (an exit protocol) the critical section.
Locks & Critical Sections

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

```c
lock_t l;
process CS[i = 1 to n] {
    while (something) {
        lock(l);
        critical section;
        unlock(l);
        non-critical section;
    }
}
```
Important Properties

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

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

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

Eventual Entry. A process that is attempting to enter its critical section will eventually succeed. This is actually less important for most speed-up oriented parallel programs, whose internal dependencies will tend to naturally delay “faster” processes while others catch up (otherwise why are the slow processes executing at all?).
Implementing Locks

A simple approach is simply to represent a lock by a shared boolean variable.

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

To unlock the lock, the lock-holding process simply sets its value to false.

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

```cpp
bool lock = false;
process CS1 {
    while (true) {
        ⟨await (!lock) lock = true;⟩ /* entry */
        critical section;
        lock = false; /* exit */
        noncritical section;
    }
}

process CS2 {
    while (true) {
        ⟨await (!lock) lock = true;⟩ /* entry */
        critical section;
        lock = false; /* exit */
        noncritical section;
    }
}
```

Figure 3.2 Critical sections using locks.

Parallel Programming Languages and Systems

Copyright © 2000 by Addison Wesley Longman, Inc.
Implementing Locks

To implement the detail, it is normal to rely on some simpler atomic primitive, implemented “in the hardware”. There are many possibilities, including “Fetch-and-Add”, “Test-and-Set” (which we examine here) and the “Load-Linked, Store-Conditional” pairing.

Test-and-Set (TS) implements the following effect (but as 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

```c
bool lock = false; /* shared lock */
process CS[i = 1 to n] {
    while (true) {
        while (TS(lock)) skip; /* entry protocol */
        critical section;
        lock = false; /* exit protocol */
        noncritical section;
    }
}
```

Figure 3.3 Critical sections using Test and Set.

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 processes which fail to gain access immediately.
Simple spin locks don’t make good use of the cache (all 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). In effect, we only “Test” (ie read) until there is at least a chance that Test-and-Set might succeed.

```cpp
bool lock = false;  /* shared lock */
process CS[i = 1 to n] {
    while (true) {
        while (lock) skip; /* entry protocol */
        while (TS(lock)) {
            while (lock) skip;
        }
        critical section;
        lock = false; /* exit protocol */
        noncritical section;
    }
}
```

**Figure 3.4** Critical sections using Test and Test and Set.
The Bakery Algorithm

It is actually possible to implement critical sections without special atomic machine instructions. One such algorithm is due to Lamport. This is really only of intellectual and historical interest, but it is cute nonetheless! Also note that it assumes sequential memory consistency. Finally, Lamport’s algorithm has the strong property of guaranteeing eventual entry (unlike our spin locks).

The idea is that upon arrival at the critical section, a process calculates when its “turn” will be (as an integer), by looking at other processes’ turns. A process sets its turn to be one more than any other turn currently claimed (processes not at the critical section have a turn of 0).

If this can be done atomically, then it is obviously correct. Lamport’s idea is to show how the correct effect can be achieved even if the turns are not determined atomically (and so may not be correct).
Bakery Algorithm

```java
int turn[1:n] = ([n] 0);
## predicate BAKERY is a global invariant -- see text
process CS[i = 1 to n] {
    while (true) {
        turn[i] = max(turn[1:n]) + 1;
        for [j = 1 to n st j != i]
            await (turn[j] == 0 or turn[i] < turn[j]);
        critical section;
        turn[i] = 0;
        noncritical section;
    }
}
```

**Figure 3.10** The bakery algorithm: Coarse-grained solution.

Lamport’s algorithm with (large!) atomic sections. How can we get rid of these?
Bakery Algorithm

If the turn setting is not atomic then there is a possibility that two (or more) processes will claim the same turn.

When this happens we need to decide systematically which process will be allowed to progress first.

In this situation we allow processes to proceed by ascending order of process identifier. We write \((x, y) > (a, b)\) to mean \((x > a) \lor (x == a \land y > b)\).
Bakery Algorithm

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

**Figure 3.11** Bakery algorithm: Fine-grained solution.

Copyright © 2000 by Addison Wesley Longman, Inc.
Barriers

Many (typically iterative) algorithms have the following structure

```plaintext
process Worker[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 one kind of condition synchronization.

Most libraries provide a corresponding operation. How might this be implemented?
Counter Barriers

```java
int count = 0;
process Worker[i = 1 to n] {
    while (something) {
        do some work;
        <count = count + 1;>
        <await (count == n);>
    }
}
```

Simple, and given atomic support from hardware (eg a “Fetch-and-Add” instruction), and efficient, coherent cache, this can be satisfactory, but note that here, in contrast to spin lock implementations, the internal busy-waiting is guaranteed to cause high contention, rather than simply being an unfortunate possibility. A full implementation is complicated by the need to reset the barrier.
Co-ordinated Barriers

An alternative is (probably wastefully) to have a separate process act purely as barrier co-ordinator.

For each other process $i$, we have variables $\text{arrive}[i]$ and $\text{continue}[i]$ (placed on separate cache lines, for efficiency).

The co-ordinator spins waiting for each process to arrive, before allowing all to continue.

Each process notes its arrival then spins waiting to be released (but the spinning is now on “its own” continue value, so less contention than with the repeated updates of the shared counter approach.)
int arrive[1:n] = ([n] 0), continue[1:n] = ([n] 0);

process Worker[i = 1 to n] {
    while (true) {
        code to implement task i;
        arrive[i] = 1;
        ⟨await (continue[i] == 1);⟩
        continue[i] = 0;
    }
}

process Coordinator {
    while (true) {
        for [i = 1 to n] {
            ⟨await (arrive[i] == 1);⟩
            arrive[i] = 0;
        }
        for [i = 1 to n] continue[i] = 1;
    }
}

Figure 3.12  Barrier synchronization using a coordinator process.
Combining Barriers

A further approach is to distribute the work of the co-ordinator between the real processes, arranged conceptually into a tree, with synchronization sweeping up then down.

Figure 3.13  Tree-structured barrier.

Copyright © 2000 by Addison Wesley Longman, Inc.
Combining Barriers

leaf node L:  \( \text{arrive[L]} = 1; \)
\( \langle \text{await (continue[L] == 1);} \rangle \)
\( \text{continue[L]} = 0; \)

interior node I:  \( \langle \text{await (arrive[left] == 1);} \rangle \)
\( \text{arrive[left]} = 0; \)
\( \langle \text{await (arrive[right] == 1);} \rangle \)
\( \text{arrive[right]} = 0; \)
\( \text{arrive[I]} = 1; \)
\( \langle \text{await (continue[I] == 1);} \rangle \)
\( \text{continue[I]} = 0; \)
\( \text{continue[left]} = 1; \text{continue[right]} = 1; \)

root node R:  \( \langle \text{await (arrive[left] == 1);} \rangle \)
\( \text{arrive[left]} = 0; \)
\( \langle \text{await (arrive[right] == 1);} \rangle \)
\( \text{arrive[right]} = 0; \)
\( \text{continue[left]} = 1; \text{continue[right]} = 1; \)

Figure 3.14  Barrier synchronization using a combining tree.
Symmetric Barriers

With a symmetric barrier the principal is similar, but each process executes the same code. We achieve overall barrier synchronization by implication from a carefully chosen sequence of pairwise synchronizations.

To synchronize between processes myid and friend (where each sees the other as its friend), both execute

```c
<await (arrive[myid] == 0);>
arrive[myid] = 1;
<await (arrive[friend] == 1);>
arrive[friend] = 0;
```

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

We then choose a suitable sequence of synchronisation partners, so that every process either synchronizes directly or indirectly with every other process. For example, for \( n \) a power of two, we have the butterfly pattern.

![Butterfly Pattern Diagram](image)

**Figure 3.15** Butterfly barrier for 8 processes.
Symmetric Barriers

There is a possible problem with such a “solution”. Suppose process 1 arrives at its first level barrier “quickly” (well before process 2). Then suppose processes 3 and 4 also synchronize quickly. Process three then “sees” that process 1 is at the pairwise barrier (but the wrong one!), and synchronizes with it. Processes 3 and 1 carry on, and process 2 is stuck.

We can “fix” this with the following approach

```plaintext
for [s = 1 to stages] {
    arrive[myid] = arrive[myid]+1;
    work out who my friend is at this stage;
    while (arrive[myfriend] < arrive[myid]) skip;
}
```

(This assumes counters never wrap around, but requires no atomic actions.)
Symmetric Barriers

If \( n \) isn’t a power of 2, we can employ a dissemination barrier.

<table>
<thead>
<tr>
<th>Workers</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stage 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stage 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stage 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 3.16** Dissemination barrier for 6 processes.

The internals of each local synchronization are slightly altered (\( a \rightarrow b \) means “\( a \) notifies \( b \)”), but the effect is similar.
```c
int a[n], sum[n], old[n];

process Sum[i = 0 to n-1] {
    int d = 1;
    sum[i] = a[i]; /* initialize elements of sum */
    barrier(i);
    ## SUM: sum[i] = (a[i-d+1] + ... + a[i])
    while (d < n) {
        old[i] = sum[i]; /* save old value */
        barrier(i);
        if ((i-d) >= 0)
            sum[i] = old[i-d] + sum[i];
        barrier(i);
        d = d+d; /* double the distance */
    }
}
```

**Figure 3.17** Computing all partial sums of an array.

(Thought exercise: do we really need barriers to synchronize this algorithm?)

Copyright © 2000 by Addison Wesley Longman, Inc.

Parallel Programming Languages and Systems
Structured Primitives

Locks (and barriers) are quite low-level in flavour. It is easy to get things wrong when programming with them.

A number of more structured primitives have been devised and implemented. We will look at two of the most popular, semaphores and monitors.

These include capabilities which facilitate the expression of both types of synchronization.

In contrast to the “spinning” implementations we have seen so far, both semaphores and monitors tend to be implemented within the operating system kernel, where processes/threads can be directly suspended and resumed.
Semaphores

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

\[
P(s): \text{<await (s>0) s=s-1>;}
\]
\[
V(s): \text{<s=s+1;>}
\]

Notice that if a semaphore is initialised to have a non-negative value, then it can never become negative subsequently.

In a typical implementation, a process/thread executing P on a zero valued semaphore will be suspended on a queue until after some other process has executed a V.

A Semaphore whose usage is organised to only ever take the value 0 or 1 is called a binary semaphore.
Using Semaphores

A semaphore provides an easy solution to the critical section problem

```c
sem mutex = 1;
process CS[i = 1 to n] {
    while (true) {
        P(mutex);
        critical section;
        V(mutex);
        noncritical section;
    }
}
```

Figure 4.1  Semaphore solution to the critical section problem.

Copyright © 2000 by Addison Wesley Longman, Inc.
Using Semaphores

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

```c
sem arrive1 = 0, arrive2 = 0;
process Worker1 {
    ...
    V(arrive1); /* signal arrival */
    P(arrive2); /* wait for other process */
    ...
}
process Worker2 {
    ...
    V(arrive2); /* signal arrival */
    P(arrive1); /* wait for other process */
    ...
}
```

**Figure 4.2** Barrier synchronization using semaphores.
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.
Figure 4.3  Producers and consumers using semaphores.

Parallel Programming Languages and Systems

Copyright © 2000 by Addison Wesley Longman, Inc.
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 semaphore as before, initialised according to the buffer size (so no longer binary).

Provided the buffer isn’t empty, we should allow producer and consumer to be active within it simultaneously.
Figure 4.4  Bounded buffer using semaphores.
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).

We use a pair of mutual exclusion semaphores to prevent interference between producers (and between consumers). As before, we are happy to have one consumer and one producer actively simultaneously within a non-empty buffer.
Figure 4.5  Multiple producers and consumers using semaphores.
Monitors

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

- they 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 actively invoking a method at any one time.

The effect is as if the body of each monitor method is implicitly surrounded with \( P() \) and \( V() \) operations on a hidden binary semaphore.

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 wait(cv) operation, where 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 cv.

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

The signalling thread continues uninterrupted (this is called signal and continue (SC)).
The thread actually executing SC has a looping transition in the executing state. SW refers to another policy, “signal-and-wait”, which is not widely used in practice.

Figure 5.1  State diagram for synchronization in monitors.

Copyright © 2000 by Addison Wesley Longman, Inc.
Monitors - Condition Synchronization

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

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

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

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

This has the important consequence that when a (previously blocked on a condition) thread is actually awakened again in the monitor, it almost always 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 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 conventional, never wrong, and indeed necessary in all but the most tightly constrained situations, to wrap each conditional variable wait() calls in a loop which rechecks the condition.
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.
We now examine the ways in which the various concepts for shared variable programming considered so far have been embedded in real programming systems. In particular we will look at the Posix threads (Pthreads) library for C, and Java’s threads and monitors.

Traditionally, OS support for concurrency came through processes, each with its own VM address space (including heap and stack), file descriptors (IO) etc. Interaction was either by message passing, or by (quite expensive) logical overlapping of VM spaces.

More recently, OS support has also provided threads, logically concurrent activities within a process. Threads share a VM address space, having only their own stack within it (for local variables), so heap and global data is shared.
#include <pthread.h>
#define P 100

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

int main (int argc, char *argv[]) {
    int i; pthread_t thread[P];

    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);
    }
}
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 just NULL (since the intended effect will have been 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.

Again, 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.
void *sayhello (void *id) {
    sleep(1);  // just to make sure the race looks bad
    printf("Hello from thread %d\n", *(int *)id);
}

int main (int argc, char *argv[]) {
    int i;  pthread_t thread[P];

    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);
    }
}
Intentionally Sharing Data

volatile int target; // not a register
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);
    }  .....

Parallel Programming Languages and Systems
Coordinating Shared Accesses

Variable target is accessible to all threads. Its increment is not atomic, so concurrent increments can interfere with unpredictable results. This is a familiar problem! POSIX provides several mechanisms to coordinate accesses including semaphores, and the 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 Pthreads name for P(s)

3. `sem_post(s)`, which is the Pthreads 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);
    ...
}

Parallel Programming Languages and Systems
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("for %d iterations, the total is %d\n", numIters, total);
}
Posix “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 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 is a lock already held by this thread.

- signalled with pthread_cond_signal(&cv) by a thread which already holds the associated mutex.

- signalled with pthread_cond_broadcast(&cv) by a thread which already holds the associated mutex, which is simply a “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 simply runs the standard Jacobi step for a given fixed number of iterations. To avoid copying, each iteration performs two Jacobi steps. There is no attempt to detect convergence (how would you add this?).

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

```c
pthread_mutex_t barrier; /* mutex semaphore for the barrier */
pthread_cond_t go; /* condition variable for leaving */
int numArrived = 0;
// A reusable counter barrier
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);
}
```
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);

    for (i = 0; i < numWorkers; i++)
        if (maxdiff < maxDiff[i])
            maxdiff = maxDiff[i];
}
void *Worker(void *arg) {
    int myid = (int) arg;
    int first, last;

    /* determine first and last rows of my strip of the grids */
    first = myid*stripSize + 1;
    last = first + stripSize - 1;

    for (iters = 1; iters <= numIters; iters++) {
        /* update my points */
        for (i = first; i <= last; 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();
    }
/* update my points again */
for (i = first; i <= last; 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();

/* compute the maximum difference in my strip and set global variable */
maxdiff = 0.0;
for (i = first; i <= last; i++) {
  ....
  if (maxdiff < temp) maxdiff = temp;
}
maxDiff[myid] = maxdiff;
Pragmatic Issues

Correct, efficient Pthreads programming is complex. Issues include

- Avoiding deadlock, which occurs when threads hold resources (usually locks) required by each other to proceed.

- Considering the cache and its effect on performance (already familiar from sequential code). A typical problem is false sharing in which two or more logically independent shared variables map to the same cache block, causing substantial (but logically unnecessary) invalidation/update traffic.

- Using only thread-safe code, which works irrespective of how many threads are active. May be OK for your own code, but what about libraries? Safe code is often called re-entrant (suggesting that it can be entered many times concurrently without error). Typical errors involve the use of non-local data.
Java Threads

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

Threads are created from classes which either extend the java.lang.Thread class or implement the java.langRunnable interface (the latter allows us to extend some other class too).

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

....
Simple s = new Simple();
s.start(); // calls the run() method in s
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 to include 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).
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 Main {  // driver program -- two readers and two writers
    static ReadersWriters RW = new ReadersWriters();
    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;  ReadersWriters RW;
    private static Random generator = new Random();

    public Reader(int rounds, ReadersWriters 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.read();
        }
    }
}
class Writer extends Thread {
    int rounds;  ReadersWriters RW;
    private static Random generator = new Random();

    public Writer(int rounds, ReadersWriters 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();
        }
    }
}

Parallel Programming Languages and Systems
class ReadersWriters {
    protected int data = 0; // the "database"
    int nr = 0;
    private synchronized void startRead() {
        nr++;
    }
    private synchronized void endRead() {
        nr--;
        if (nr==0) notify(); // awaken waiting Writers
    }
    public void read() {
        startRead();
        System.out.println("read: " + data);
        endRead();
    }
}
public synchronized void write() {
    while (nr>0)
        try { wait(); }
            catch (InterruptedException ex) {return;}
    data++;
    System.out.println("wrote: " + data);
    notify();    // awaken another waiting Writer
}
We could express the same effect with *synchronized* blocks

class ReadersWriters {
    protected int data = 0;  // the "database"
    int nr = 0;
    public void read() {
        synchronized(this) {
            nr++;
        }
        System.out.println("read: " + data);
        synchronized(this) {
            nr--;
            if (nr==0) notify();  // awaken waiting Writers
        }
    }
    ....
}
Buffer for Producers-Consumers
(borrowed from Skansholm, Java from the Beginning)

```java
public class Buffer extends Vector {
    public synchronized void putLast (Object obj) {
        addElement(obj);
        notify();
    }
    public synchronized Object getFirst () {
        while (isEmpty())
            try {wait();}
            catch (InterruptedException e) {return null;}
        Object obj = elementAt(0);
        removeElementAt(0);
        return obj;
    }
}
```
Distributed Programming

In Andrews’ sense, distributed programming means programming without shared variables.

This encompasses a wide range of application scenarios, from OS processes time-sharing a single processor, through “single-box” parallel machines trying to maximise performance, and on to “distributed systems” in the more conventional sense.

Processes wishing to interact have to co-operate. Each process remains in sole control of its own virtual address space (so there is no mutual exclusion issue).

The forms which this co-operation can take define a range of possible distributed programming models.
Programming with Message Passing

We begin by examining some fundamental concepts. Later, we will see how these are realized in MPI, the standard library for message passing programming.

Processes use message passing to move data between virtual address spaces.

The address spaces may be implemented on top of the same physical memory (like Unix processes) or on independent distributed memories. Message passing facilities are typically provided through a library, which hides details of real protocols or physical memory sharing.

A common approach is the Single Program Multiple Data (SPMD) model. Each process runs a copy of the same program in its own address space. Variations are achieved by branching with respect to data values or to the process identifier.
Programming Issues

How are messages specified (allowing data to be transferred between address spaces) and what kind of synchronization is supported (is a sender blocked until a matching receive)?

How is non-determinism catered for? For example, a “farmer” process controlling a bag of tasks needs to be ready to receive from any worker at any time.

Which communication patterns are supported? Are these tied to the topology of the architecture? Are collective communications supported?

Are there any facilities for abstracting communication spaces (can libraries be provided safely)?
Channels, Send & Receive

The most basic message passing system must provide at least two operations, send and receive, to be executed by the processes at either end of a communication. The combination of these actions achieves the transfer of data between address spaces.

It is helpful to introduce the concept of the channel, to better define and understand variations in the semantics of even these simple operations.

A channel is rather like a buffer, which exists (conceptually at least) outside the address spaces of the communicating processes. It stores messages which have been sent, but not yet received.
Channels, Send & Receive

Channels may or may not be strongly typed, and may or may not enforce an ordering on the relationship between messages sent and messages received.

For example, a sensible, simple channel would enforce FIFO ordering on messages sent by a particular process.

Channels could be constrained to be one-to-one (with respect to senders and receivers) or many-to-one, and so on.

Some libraries/languages (like occam) make channels explicit, others (like MPI) hide them syntactically, but still have them internally (with impacts on semantics of send and receive).
Asynchronous Message Passing

With asynchronous message passing, the channel can buffer sent messages. send and receive primitives are best thought of as interactions between process and buffer, copying data from/into an address space into/from the buffer.

An asynchronous send does not require the sender to wait until a corresponding receive has been executed (the sender is not blocked). Indeed, the message might never be received!

In contrast, a receive is usually (though not necessarily) blocking (ie it must wait until a corresponding send has executed.

The term asynchronous refers to a property of the relationship between sends and a receives: the interacting processes are not tightly synchronized by the exchange. An asynchronous channel is like a semaphore which carries data.
Matrix Multiplication Pipeline

chan vector[n](double v[n]); # messages to workers
chan result(double v[n]); # rows of c to coordinator

process Coordinator {
    double a[n,n], b[n,n], c[n,n];
    initialize a and b;
    for [i = 0 to n-1] # send all rows of a
        send vector[0](a[i,*]);
    for [i = 0 to n-1] # send all columns of b
        send vector[0](b[*],i);
    for [i = n-1 to 0] # receive rows of c
        receive result(c[i,*]); # in reverse order
}

Figure 9.6 (a) Matrix multiplication pipeline: Coordinator process.

Copyright © 2000 by Addison Wesley Longman, Inc.
process Worker[w = 0 to n-1] {
    double a[n], b[n], c[n];  # my row or column of each
    double temp[n];            # used to pass vectors on
    double total;              # used to compute inner product

    # receive rows of a; keep first and pass others on
    receive vector[w](a);
    for [i = w+1 to n-1] {
        receive vector[w](temp); send vector[w+1](temp);
    }

    # get columns and compute inner products
    for [j = 0 to n-1] {
        receive vector[w](b);  # get a column of b
        if (w < n-1) # if not last worker, pass it on
            send vector[w+1](b);
        total = 0.0;
        for [k = 0 to n-1]  # compute one inner product
            total += a[k] * b[k];
        c[j] = total;       # put total into c
    }

    # send my row of c to next worker or coordinator
    if (w < n-1)
        send vector[w+1](c);
    else
        send result(c);

    # receive and pass on earlier rows of c
    for [i = 0 to w-1] {
        receive vector[w](temp);
        if (w < n-1)
            send vector[w+1](temp);
        else
            send result(temp);
    }
}

Figure 9.6 (b) Matrix multiplication pipeline: Worker processes.
Jacobi Again

We assume that the 2-D grid has been partitioned block row-wise across our processors.

We need to explicitly program the exchange of boundary information.

if (w>1) send up[w-1](new[1,*]);
if (w<PR) send down[w+1](new[HEIGHT,*]);
if (w<PR) receive up[w](new[HEIGHT+1,*]);
if (w>1) receive down[w][new[0,*]);

Worker 1 handles the “maxdiff” calculation.
chan up[1:PR](real edge[0:n+1]);
chan down[1:PR](real edge[0:n+1]);
chan diff(real);

process worker[w = 1 to PR] {
    int HEIGHT = n/PR;    # assume PR evenly divides n
    real grid[0:HEIGHT+1,0:n+1], new[0:HEIGHT+1,0:n+1];
    real mydiff = 0.0, otherdiff = 0.0;
    initialize grid and new, including boundaries;
    for [iters = 1 to MAXITERS by 2] {
        # compute new values for my strip
        for [i = 1 to HEIGHT, j = 1 to n]
            new[i,j] = (grid[i-1,j] + grid[i+1,j] +
                        grid[i,j-1] + grid[i,j+1]) * 0.25;
        exchange edges of new -- see text;
        # compute new values again for my strip
        for [i = 1 to HEIGHT, j = 1 to n]
            grid[i,j] = (new[i-1,j] + new[i+1,j] +
                         new[i,j-1] + new[i,j+1]) * 0.25;
        exchange edges of grid-- see text;
    }
    # compute maximum difference for my strip
    for [i = 1 to HEIGHT, j = 1 to n]
        mydiff = max(mydiff, abs(grid[i,j]-new[i,j]));
    if (w > 1)
        send diff(mydiff);
    else    # worker 1 collects differences
        for [i = 1 to w-1] {
            receive diff(otherdiff);
            mydiff = max(mydiff, otherdiff);
        }
    # maximum difference is value of mydiff in worker 1
}
Observations

We don’t need any explicit **local synchronisations** - all the synchronization we need is provided implicitly by the message passing semantics.

There is no **global synchronisation** (even implicitly) at all, or perhaps more accurately, the necessary global synchronisation is achieved as a side-effect of a set of (implicit) local synchronisations.

The code is arguably quite hard to follow - without knowing the algorithm it could be difficult to work out what’s going on by inspecting the communication calls which are spread through the code.
Synchronous Message Passing

With synchronous message passing, both sender and receiver are forced to wait until the other partner in the communication is ready.

This has the advantage that we don’t need to set aside buffer space (or risk confusion when the system doesn’t do this as we might expect), and therefore that the semantics are simple.

It has the disadvantages that waiting senders can’t get on with other work, and that it is easier to write deadlocking code (or maybe this is an advantage?).

```plaintext
channel in1(int), in2(int);
process P1 {
    int v1=10, v2;
    synch_send in2(v1);
    receive in1(v2);
}
process P2 {
    int v1=20, v2;
    synch_send in1(v1);
    receive in2(v2);
}
```
Message Passing Interface (MPI)

MPI 1 (and later MPI 2) were defined in an attempt to standardize existing practice.

"We have sought to make use of the most attractive features of a number of existing message passing systems."

Designed by a process involving hundreds of people from tens of organizations.

Specification is language independent, but describes C, C++, Fortran bindings.

We focus on MPI 1 since this is most widely implemented, and use the C binding.
Sieve of Eratosthenes

An very old algorithm which also serves as a concise example for parallel programming language description.

The object is to find the first $p$ prime numbers. The gist of the original algorithm is that we start with a list of “all” integers from 2, and repeatedly remove multiples of the smallest remaining number from this list. After each set of removals, the smallest remaining number is prime. To start, we note that 2 itself is prime.

We will implement a pipelined parallel version by creating a generator process and $p$ sieve processes, each of which computes a prime.
generator () {
    i = 2;
    while (true) {
        send i to first sieve;
        i++;
    }
}

sieve (int prime) {
    get and display prime from predecessor
    while (true) {
        get candidate
        if (prime doesn’t divide candidate)
            send candidate to successor
    }
}
MPI Concepts

All processes are created statically when the program is invoked. The precise method is implementation specific and not part of the standard.

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.

There are no explicit channels. 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 extra values to aid disambiguation. and there are many synchronization modes.

There are a range of collective operations.
int main(int argc, char *argv[]) {
    int rank, p;

    MPI_Init(&argc, &argv); MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    if (rank == 0) generator();
    else sieve(rank, p);
    MPI_Finalize();
}

void generator (void) {
    int next = 2;
    while (1) {
        MPI_Send(&next, 1, MPI_INT, 1, 0, MPI_COMM_WORLD); next++;
    }
}
The Sieve Process

```c
void sieve (int rank, int p)
{
    int prime, n;
    MPI_Status status;

    MPI_Recv(&prime, 1, MPI_INT, rank-1, 0, MPI_COMM_WORLD, &status);
    printf("%d is a prime\n", prime);
    while (1) {
        MPI_Recv(&n, 1, MPI_INT, rank-1, 0, MPI_COMM_WORLD, &status);
        if (n%prime && rank<p-1) {
            MPI_Send(&n, 1, MPI_INT, rank+1, 0, MPI_COMM_WORLD);
        }
    }
}
```
MPI Primitives

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 initial global communicator is called MPI_COMM_WORLD. A process can find the size of a communicator, and its own rank.

int MPI_Comm_Size (MPI_Comm comm, int *np)
int MPI_Comm_rank (MPI_Comm comm, int *me)
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’ (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.
Receive in standard mode

```c
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) introduced with MPI_ANY_SOURCE and/or MPI_ANY_TAG.

**Status information** is returned in a structure with status.MPI_SOURCE and status.MPI_TAG fields (ie. to see where the message came from).
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.

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.
```c
#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++;
    }
}
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);
}
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;
    }
}
Synchronisation in standard mode

MPI uses the term **blocking** in a slightly unconventional way. A **blocking send** returns only when it is safe to reuse the specified output buffer (because the data has been copied by the system). More conventionally, an MPI **synchronous send** returns only when in addition, a matching receive has begun.

Thus, the return of a blocking send can be **dependent** upon the availability of **buffer space within the hidden implementation** (in other words, this is where the channels have gone!).

Standard mode send is **blocking but asynchronous**.

Messages sent between any particular pair of processes are guaranteed to appear to be **non-overtaking**, in the sense that a receive cannot match message B in preference to message A if A was sent before B.
N-Body Problems

These model physical systems in which every object exerts an influence on every other, typically symmetrically (e.g. gravitational systems).

We model time as a discrete sequence, so that state of an item at time $t + 1$ is a function of its own state at time $t$ and of its interactions with other objects as they were at time $t$.

Every body needs to interact with every other body, but we can schedule these interactions to avoid duplication or contention. One solution is to pass the objects around a ring of processes, (using an interacting peers pattern) with an underlying assumption that combination of “forces” is associative.

Notice that the following program might deadlock. How might we make it safe?
void main (int argc, char* argv[]) {
    int myid, np, from, to;
    body_t mybodies[300], otherbodies[300];
    MPI_Status status;

    MPI_Init(&argc, &argv); MPI_Comm_size(MPI_COMM_WORLD, &np);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    from = (myid+np-1)%np; to = (myid+1)%np;

    initialize(mybodies, otherbodies); make mpi_body_t;
    for (i=0; i<np-1; i++) {
        MPI_Send(otherbodies, 300, mpi_body_t, to , 0, MPI_COMM_WORLD);
        MPI_Recv(otherbodies, 300, mpi_body_t, from, 0, MPI_COMM_WORLD, &status);
        update(mybodies,otherbodies);
    }
    printresult(mybodies); MPI_Finalize();
}
Other Communication Modes

They key concept is that MPI makes its implementation partially visible.

Standard mode send (MPI_Send) is blocking and may wait until a matching receive exists (i.e. it might be synchronous!).

Buffered mode send (MPI_Bsend) is blocking and asynchronous, but the programmer must previously have made “enough” buffer space available.

Synchronous mode send (MPI_Ssend) is blocking and synchronous, only returning when a matching receive has been found.

Ready mode send (MPI_Rsend) additionally requires a matching receive to have already been called (or else the behaviour is undefined).
Receiving with MPI_Recv blocks until a matching message has been completely received into buf (so it is effectively blocking and synchronous).

MPI also provides non-blocking sends and receives which return immediately (ie. possibly before it is safe to use/reuse the buffer).

MPI_Isend (I for immediate) is the standard mode immediate send, and MPI_Irecv is the immediate receive. These have an extra parameter, called a ‘request’ which is a handle on the communication, used with MPI_Wait and MPI_Test to wait for or check completion of the communication (in the MPI blocking sense).
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 be able to create a receiving buffer once we do.

```
int MPI_Probe(int src, int tag, MPI_Comm comm, MPI_Status *status)
```

is a blocking test for a receive which fills in *status but doesn't actually accept the message.

There is also a non-blocking version

```
int MPI_Iprobe(int src, int tag, MPI_Comm comm,
               int *flag, MPI_Status *status)
```

which leaves a boolean result in *flag).
We can then determine the size of a potential incoming message by inspecting its status information.

```
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.

For example, we can use these functions to receive a message containing an unknown number of integers from an unknown source, but with tag 0 in a given communicator comm.

```
MPI_Probe(MPI_ANY_SOURCE, 0, 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, 0, comm, &status);
```
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.

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

creates a new communicator for each distinct value of colour, labelling processes in each by ascending key (breaking ties by process id in the parent communicator).

Given a \( p \) process hypercube with ranks already computed in `mypid`, create communicators for two sub-cubes (\( 0..p/2 - 1 \) and \( p/2..p - 1 \)) with

```c
MPI_Comm_split (MPI_COMM_WORLD, mypid/(p/2), 0, &subcubecomm);
```
Duplicating Communicators

It is sometime useful to duplicate a communicator.

```c
int MPI_Comm_dup(MPI_Comm old, MPI_Comm *new)
```

makes *new a copy of old (same size and ids)

For example to protect the communications which are internal to some other library function from interference by subsequent or concurrent communications:

```c
MPI_Comm_dup(comm, &newcomm);
some_library_function(... , newcomm);
MPI_Comm_free(&newcomm);
```
Collective Operations

These must be called by all processes in the communicator.

```c
int MPI_Barrier (MPI_Comm comm)
```
block until all processes reach this call.

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

broadcast count items of type t from buf in root to buf in all other procs in comm.
Collective Operations

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).
Collective Operations

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

reduces elements from all send buffers, pointwise to count single values, using op, storing result in all receive buffers.

op is chosen from a predefined set (MPI_SUM, MPI_MAX etc) or constructed (with MPI_Op_create).
Collective Operations

There are several others, including prefix and compound send-receives.

```
int MPI_Sendrecv (void *sendbuf, int sendcount,
                MPI_Datatype sendtype, int dest,
                int sendtag, void *recvbuf,
                int recvcount, MPI_Datatype recvtype,
                int source, int recvtag
                MPI_Comm comm, MPI_Status *status)
```
Jacobi Again (1-dimensional)

```c
int main(int argc, char *argv[]) {
    if (rank == 0) read_problem(&n, work);
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    mysize = n/p;
    local = (float *) malloc(sizeof(float) * (mysize+2);
    MPI_Scatter(work, mysize, MPI_FLOAT, &local[1], mysize,
                 MPI_FLOAT, 0, MPI_COMM_WORLD);
    left = (rank+p-1)%p; right = (rank+1)%p;
    global_error = some_big_float;
```
Jacobi Again (1-dimensional)

```c
while (global_error > acceptable_error) {
    MPI_Sendrecv(&local[1], 1, MPI_FLOAT, left, 0,
                  &local[mysize+1], 1, MPI_FLOAT, right, 0,
                  MPI_COMM_WORLD, &status);
    MPI_Sendrecv(&local[mysize], 1, MPI_FLOAT, right, 0,
                  &local[0], 1, MPI_FLOAT, left, 0,
                  MPI_COMM_WORLD, &status);
    do_one_step(local, local_error);
    MPI_Allreduce(&local_error, &global_error, 1,
                  MPI_FLOAT, MPI_MAX, MPI_COMM_WORLD);
}
MPI_Gather (&local[1], mysize, MPI_FLOAT,
            work, mysize, MPI_FLOAT, 0, MPI_COMM_WORLD);
if (rank == 0) print_results(n, work);
```

Parallel Programming Languages and Systems
More MPI

MPI 1 additionally has functions for assisting with data marshalling (preparing data for communication) by the definition of derived datatypes, and specifying the topology of a process graph.

MPI 2 defines features including dynamic process creation, one sided communication and safe interaction with threaded processes. However, many current MPI implementations do not fully address all of these.
Linda presents an interesting alternative conceptual model for parallelism, based around a small set of primitives which can be added (in principle) to any base sequential language to create a parallel programming framework.

The key concept is that activities interact through tuple space, a global, content-addressable memory. Each tuple is an ordered collection of typed data fields.

(12, TRUE, "Green")
(3.5, "x")
(34, "next")
(3.5, "stop")
(34, "interesting")
Processes run asynchronously and can operate on tuple space with:

```plaintext
out(exp1, exp2, ..., expN), which evaluates the expressions then atomically places a copy of the results as tuple in tuple space. For example,

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

```plaintext
in(tuple-template) which atomically removes a tuple matching the template from tuple space. The template contains actual values and formal parameters (indicated by ?) to be assigned during the match. A match occurs with any tuple for which all actual values match and the types on formals match. in is blocking, in the sense that the caller is suspended until a matching tuple becomes available. For example,

    in("Green", ?y, ?r, FALSE);
```
rd (tuple-template) which is like in but without removing the matching tuple from tuple space.

eval(expr, expr, ...) which is like out, but dynamically creates new processes to evaluate any field of the tuple which has been expressed as a function call. The process calling eval proceeds immediately. The resulting tuple enters tuple space when all the newly sparked processes have terminated.

There are also non-blocking forms inp, rdp which complete “immediately”, returning a boolean indicating whether or not a match occurred.

We could model a semaphore s in Linda with out("s"); for V(s) and in("s"); for P(s), calling an appropriate number of out operations to “initialise” s as required.

(However, often Linda's blocking, atomic semantics will let us implement the required synchronisation directly on the real data.)
**Example: Bag of Tasks**

Here is bag of tasks style Linda program for sieving primes. It finds the first LIMIT primes.

“Task” tuples are identified by the string "candidate" in their first field. The work associated with a task involves checking the primality of the accompanying value.

Results are communicated back to the main process as "result" tuples, with those which are actually prime subsequently returned to tuple space as "prime" tuples.

Each worker keeps its own local table of known primes.
int primes[LIMIT] = {2,3}; /* my table of primes */
int numPrimes = 2, value = 5;

for (i = 1; i <= numWorkers; i++)
    eval("worker", worker());
out("candidate", value);
while (numPrimes < LIMIT) {
    in("result", value, ?isprime);
    if (isprime) { /* put value in table and TS */
        primes[numPrimes] = value;
        out("prime", numPrimes, value);
        numPrimes++;
    }
    value = value + 2;
}
out("stop");
for (i = 0; i < LIMIT; i++) printf("%d\n", primes[i]);
void worker() {
    int primes[LIMIT] = {2,3}, numprimes=2;
    while(true) {
        if (rdp("stop")) return;
        in("candidate", ?candidate); /* get candidate */
        out("candidate", candidate+2); /* output next one */
        i = 0; isprime = TRUE;
        while (primes[i]*primes[i] <= candidate) {
            if (candidate%primes[i] == 0) {isprime = FALSE; break;}
            i++;
            if (i >= numPrimes) { /* need another prime */
                rd("prime", numPrimes, ?primes[numPrimes]);
                numPrimes++;
            }
        }
        out("result", candidate, isprime);
    }
}
Example: Pipeline

By way of contrast, here is a Linda version of the prime sieve pipeline (as seen in MPI), 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);
    }
}

Parallel Programming Languages and Systems
int sieve (int me) {
    int n, p, in_seq=1, out_seq=1, stop=FALSE;

    in("number", me, in_seq, ?p);
    while (!stop) {
        in_seq++;
        in("number", me, in_seq, ?n);
        if (n==LIMIT) {
            stop = TRUE; out("number", me+1, out_seq, n);
        } else if (n%p !=0) {
            if (out_seq == 1) eval("prime", me+1, sieve(me+1));
            out("number", me+1, out_seq, n);
            out_seq++;
        }
    }
    return p;
}
Implementing Tuple Space

A powerful model sets a demanding implementation challenge! “Real” associative memory is expensive to implement in hardware (e.g. caches). In ordinary memory, we have to decide where to store, and look for tuples, as well as to respect the atomic semantics of the operations.

Advanced Linda implementations perform considerable compile-time analysis of program specific tuple usage.

Possible tuples (in a given program) can be categorised into a set of classes by type signature. Tuples in each class are stored through a carefully chosen hash function, to reduce searching.
Implementing Tuple Space

When mapping to distributed memory, classes can be assigned to distinct nodes (effectively a node becomes a “tuple-server” as well as a worker).

This mapping can be adjusted dynamically, if and when regular patterns emerge in the actual use of tuples.

Tuples which are going to be read but not removed can be pre-broadcast.

The ultimate aim is that, for programs which have natural, direct message-passing counterparts, performance should be similar, but expressed with this “simpler” programming model.
Footnote: JavaSpaces

The Linda model has recently had something of a revival, in distributed (rather than performance parallel) programming, with the introduction of JavaSpaces.

A JavaSpace is essentially a tuple space object, executed by a node in a system which runs a server for this purpose, and accepting the invocation of `in()`, `out()` and `rd()` operations (though the names are different), from other nodes which know about it (there is a “look-up” concept for finding such servers).

There is no equivalent of `eval()`.

There is a concept of `lease`, a duration for which a tuple will be maintained in tuple space - the model is that tuples may be persistent, outliving the processes which create them.
Distributed Shared Memory

Shared variable programming is widely perceived as being easier than message passing (or similar).

Distributed memory hardware, built from commodity components (i.e. boxes full of conventional PC boards) are much cheaper.

Can we have the best of both worlds? Can we implement shared variable programming models on distributed memory hardware?

Yes, but can we do so with a reasonable price-complexity trade-off?
Distributed Shared Memory (DSM)

The key to understanding DSM is to remember that memory as seen by application processes in conventional systems is already an illusion.

Addresses generated by programs are virtual and are translated to physical addresses dynamically by a combination of hardware and OS software.

The traffic between main memory and disc is in terms of pages (blocks of contiguous virtual addresses mapped to blocks of contiguous physical addresses while in main memory), triggered when an application process generates a page fault (i.e. accesses a virtual address which lies in a virtual page which is not currently mapped into a physical page).

DSM observes that many processors could share the same virtual address space, mapped page-wise to many physical shared memory spaces.
Distributed Shared Memory (DSM)

The memory management mechanisms will need to be upgraded to keep track of this more complex virtual to physical page mapping. For cheapness, the physical components (ie processors and memories) should be unchanged - the extra work will be done in the virtual memory software in the OS.

Consider the following, with x and y on the same page.

```c
int x = 0, y = 0;
process P1 {
    x = 1;
}
process P2 {
    y = 1;
}
```
Distributed Shared Memory (DSM)

Node 1

initial state \( x = 0, y = 0 \)

1. write \( x \), no fault

2. write \( y \), page fault

3. send request to Node 1

4. send page to Node 2

5. receive page

6. write \( y \), no fault

final state \( x = 1, y = 1 \)

**Figure 10.16** Page fault handling in a DSM.
The simplest scheme would only allow one copy of each virtual page to exist at any time. This would be hopelessly inefficient in practice.

An obvious improvement would allow many read only copies to exist, invalidated as soon as any owner needs to write.

This is still too inefficient for most practical uses, being vulnerable to false sharing (as in cache systems), It also fails to recognize that most real shared variable algorithms use synchronization to avoid race conditions. This programming style can be exploited to improve performance.
Consider an application in which processes share some data structure. In the simplest case this could be a single counter variable.

To avoid races, a good (correct) programmer will use some synchronization mechanism to enforce mutual exclusion on accesses to the data.

Suppose some process has gained access and made an update. Do all processes which share the corresponding page need to know about this immediately?

No! Only the next process to successfully claim access to the variable needs to know, and then only when it gains access.

This observation is only correct if all accesses to the shared data “play by the rules” of using synchronization primitives correctly.
The TreadMarks system is one of the pioneering (and best known) systems which implements this approach, through a small library of special operations.

(\texttt{void *}) \texttt{Tmk\_malloc (unsigned int size)} is a special version of \texttt{malloc} which creates shared data, accessible through a pointer common to all processes.

\texttt{void Tmk\_barrier (unsigned int id)} implements barrier synchronization, across all processes in the system.

\texttt{void Tmk\_lock\_acquire (unsigned int id)} and \texttt{void Tmk\_lock\_release (unsigned int id)} are the usual locking operations, where locks are simply identified by integers.

The TreadMarks programmer must resolve all potential races with explicit calls to these mechanisms.
float **grid; float scratch[N][N];

Tmk_startup();
if (Tmk_proc_id == 0) grid = (float **) Tmk_malloc(N*N*sizeof(float));

Tmk_Barrier(0);
for (some iterations) {
    for (i=begin; i<end; i++)
        for (j=0; j<n; j++)
            scratch[i][j] = (grid[i-1][j] + grid[i+1][j] +
                            grid[i][j-1] + grid[i][j+1])/4;

    Tmk_Barrier(1);
    for (i=begin; i<end; i++)
        for (j=0; j<n; j++)
            grid[i][j] = scratch[i][j];
    Tmk_Barrier(2);
}
int *counter;

Tmk_startup();
if (Tmk_proc_id == 0) {
    counter = (int *) Tmk_malloc(sizeof(int));
    *counter = 0;
}
Tmk_Barrier(0);
.....
if (blah) {
    Tmk_lock_acquire(0);
    *counter += whatever;
    Tmk_lock_release(0);
}
.....
TreadMarks Implementation

By forcing the programmer to explicitly indicate synchronization requirements, TreadMarks is able to substantially weaken its memory consistency model.

Whereas sequential consistency would require every update to be disseminated immediately and consistently (ie so that every process sees the same updates in the same order), Treadmarks’ lazy release consistency means that updates are sent only to the next process which requires to know about them (because of its success with synchronisation), and only when it actually accesses the changed page.

This is a considerable saving. Many changes to same page can be sent as a single message, rather than being broadcast individually.
TreadMarks Implementation

A further trick improves performance even more, by avoiding the false sharing problem. Treadmarks allows many processes to own and update writeable copies of a page simultaneously!

At synchronization points the current owners of the page need only exchange lists of changes they have made (“diff lists”, rather than copies of the whole page.

The receiver merges these with its existing copy of the page. Its own writes to independent parts of the page are unaffected (they would normally have generated false sharing problems).

If two processes have written to the same part of the page concurrently (generating a race), then they must have broken TreadMarks synchronization rules, and get the punishment they deserve (i.e. non-deterministic chaos).
TreadMarks Performance

Roughly speaking, the performance goal of DSM systems would be, for a given algorithm, to match the speed up which could be achieved by programming the required data exchanges explicity (e.g. with MPI), but within the simpler application programming interface of shared memory with synchronizations.

In such cases, the messages passed by the diff sharing mechanism would correspond exactly in content and timing to the explicit messages in an MPI version.

DSM implementations have reported impressive results on suitably chosen applications.