The Concurrent Computing Landscape
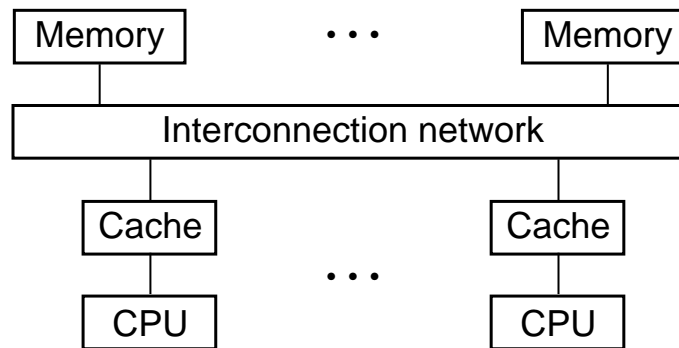
Andrews Chapter 1

```
┌─────────────────┐
│ Primary memory  │
└────────┬────────┘
         │
┌────────┴────────┐
│  Level 2 cache  │
└────────┬────────┘
         │
┌────────┴────────┐
│  Level 1 cache  │
└────────┬────────┘
         │
      ┌──┴──┐
      │ CPU │
      └─────┘
```
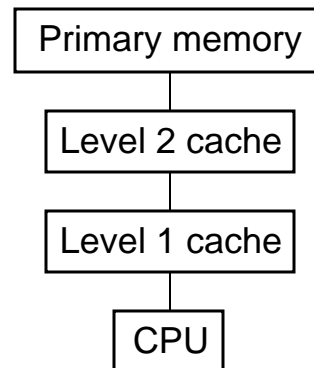
Figure 1.1   Processors, cache, and memory in a modern machine.
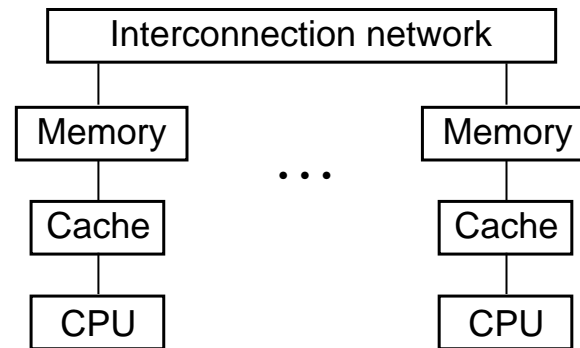
- Temporal locality
- Cache hit
- Cache miss
- Write through cache
- Write back cache
- Cache lines
- Spatial locality

Figure 1.2   Structure of Shared-Memory Multiprocessors.

- Cache consistency problem
    - Snooping
- Memory consistency problem
    - Compilers and libraries
- False sharing
    - Padding

Figure 1.3 Structure of distributed-memory machines.

- Clusters

- Networks

- Message passing

Paradigms

- Iterative parallelism

  – Each process is an iterative program. Scientific computation.

- Recursive parallelism

  – Recursive procedure calls are independent.

- Producers and consumers

  – Each process is a filter. Pipelines.

- Clients and servers

  – World Wide Web. Graphics cards.

- Interacting peers

  – Decentralized decision making.

```
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

Embarrassingly Parallel

- **Read set**: set of variables read by a process

- **Write set**: set of variables written to by a process

- Two operations are **independent** if the write set of each is disjoint from both the read and write sets of the other.

```
co [i = 0 to n-1] {   # compute rows in parallel
  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 Rows

```
co [j = 0 to n-1] {   # compute columns in parallel
  for [i = 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 Columns

```
co [i = 0 to n-1, j = 0 to n-1] { # all rows and
  c[i,j] = 0.0;                    # all columns
  for [k = 0 to n-1]
    c[i,j] = c[i,j] + a[i,k]*b[k,j];
}
```

Parallel Matrix Multiplication by Rows and Columns

```
co [i = 0 to n-1] {        # rows in parallel then
  co [j = 0 to n-1] {   # columns in parallel
    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 Using Nested co Statements

- Can we parallelize the loop over k?

```
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

- Processes are named.

- Processes cannot be nested.

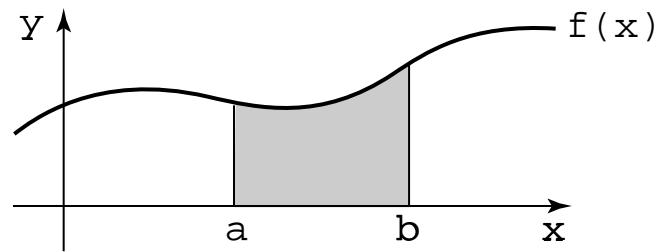- Program does not wait for processes to end.

```
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)

• Useful because we often have only 4 or 8 cores.

Figure 1.4 The quadrature problem.

- Find the area under a function.

- Iterative approach:

  Break region into many equal intervals.

- Recursive approach:

  Break region in half, recursion on each half.

```
double fleft = f(a), fright, area = 0.0;
double width = (b-a) / INTERVALS;
for [x = (a + width) to b by width] {
  fright = f(x);
  area = area + (fleft + fright) * width / 2;
  fleft = fright;
}
```

Iterative Quadrature Program

```
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

- Change the recursion to:

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

- co statements do not end until all branches end.

Independent procedure calls

- If a procedure does not reference global variables and has only value parameters, then every call of the procedure will be independent.

- Functional programming has these features.

- For example, quicksort.

```
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 in parallel
    co larea = quad(left, mid, fleft, fmid, larea);
    // rarea = quad(mid, right, fmid, fright, rarea);
    oc
  }
  return (larea + rarea);
}
```

Recursive Parallel Adaptive Quadrature

Source files → (sed) → (tbl) → (eqn) → (groff) → PostScript
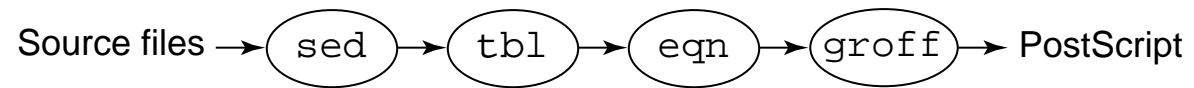
Figure 1.5   A pipeline of processes.

```
sed -f Script $* | tbl | eqn | groff Macros -
```

- Producers and consumers.
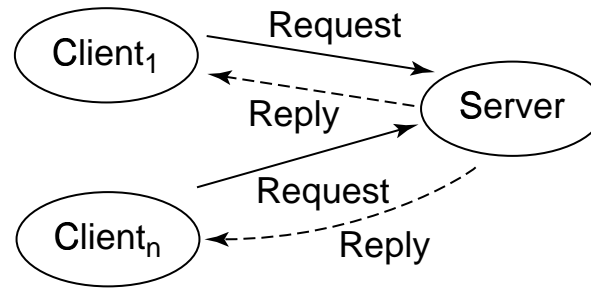
Figure 1.6   Clients and servers.

- Web pages
- Graphics cards
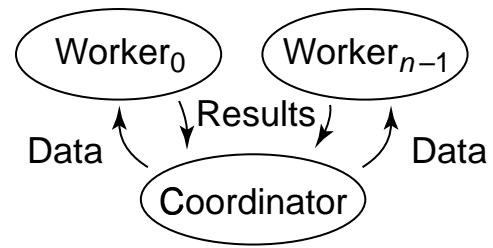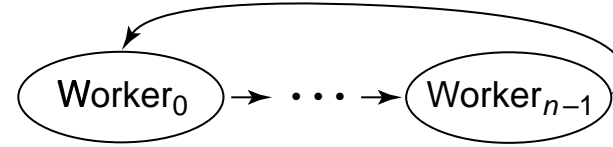
(a) Coordinator/worker interaction   (b) A circular pipeline

Figure 1.7 Matrix multiplications using message passing.

- Interacting peers

```
process worker[i = 0 to n-1] {
  double a[n];     # row i of matrix a
  double b[n,n];   # all of matrix b
  double c[n];     # row i of matrix c
  receive initial values for vector a and matrix b;
  for [j = 0 to n-1] {
    c[j] = 0.0;
    for [k = 0 to n-1]
      c[j] = c[j] + a[k] * b[k,j];
  }
  send result vector c to the coordinator process;
}

process coordinator {
  double a[n,n];   # source matrix a
  double b[n,n];   # source matrix b
  double c[n,n];   # result matrix c
  initialize a and b;
  for [i = 0 to n-1] {
    send row i of a to worker[i];
    send all of b to worker[i];
  }
  for [i = 0 to n-1]
    receive row i of c from worker[i];
  print the results, which are now in matrix c;
}
```

Matrix Multiplication Using Coordinator/Worker Interaction

```
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;
}
```

Matrix Multiplication Using a Circular Pipeline