

Overview of parallel hardware

Raymond J. Spiteri

Lecture Notes for CMPT 898:
Numerical Software

University of Saskatchewan

January 26, 2013

Objectives

- Paradigms of parallel hardware
- Flynn's taxonomy
- Cache coherence

Paradigms of parallel hardware

In this course, we are concerned with writing software to solve problems in scientific computing.

We do not focus on aspects such as the interconnection network for the processors and the memory modules or the system software.

Nonetheless, effective programming of a parallel algorithm does depend on the target architecture; so before we can take full advantage of parallelism, we must consider how a parallel computer operates.

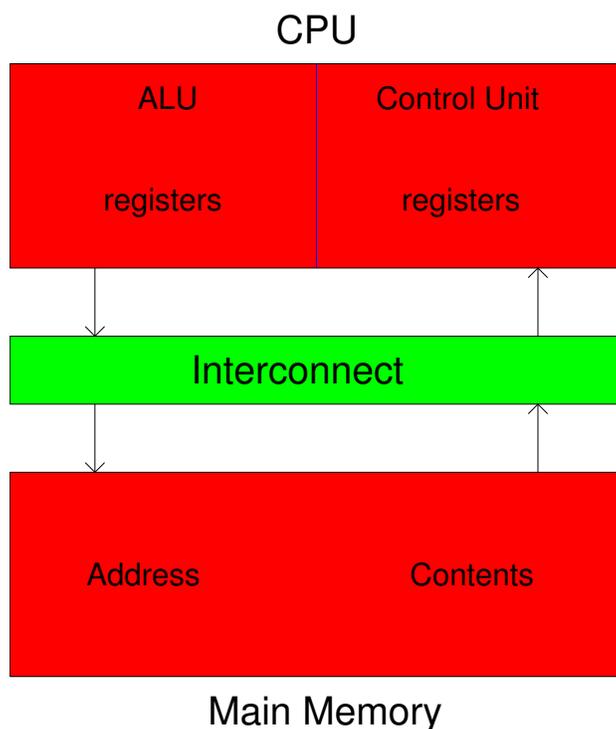
It is easy to write parallel programs if you don't care about performance!

Parallel hardware and software have grown out of conventional *serial* hardware and software, so we begin with a brief review of how serial systems function.

The classical von Neumann machine

The classical von Neumann machine is divided into a central processing unit (CPU) (also called *processor* or *core*) and main memory, connected in some manner.

The CPU is further divided into a control unit and an arithmetic-logic unit (ALU).



The classical von Neumann machine

The control unit directs the execution of programs; the ALU does the calculations called for in the program.

The memory stores both instructions and data.

When being used by the program, they are stored in very fast memory locations, called **registers**.

Naturally, the cost of memory goes up with speed; so there are relatively few registers.

Data and program instructions move between memory and registers via what is called a **bus**.

The von Neumann machine executes exactly one instruction at a time and typically operates on only a few pieces of data.

The classical von Neumann machine

It is important to note that no matter how fast the CPU is, the speed of execution of programs is limited by the rate at which the (inherently sequential) instruction sequence and data between memory and CPU.

This is called the **von Neumann bottleneck**.

Not surprisingly, a lot of effort has gone into modifying the von Neumann machine to alleviate the bottleneck.

The three most common modifications of the von Neumann model in order to reduce the bottleneck are through the use of

- caching,
- virtual memory, and
- low-level parallelism.

Of these, we only examine caching in detail.

Caching

Most modern machines attempt to alleviate von Neumann bottleneck by having a hierarchical memory: in between registers and main memory is an intermediate memory (faster than main memory but slower than registers) called **cache**.

This idea works because programs tend to access both instructions and data sequentially.

So, if we store a small block of instructions and a small block of data in cache, most of the accesses will use this faster memory.

These blocks are called cache blocks or *cache lines*.

A typical cache line stores 8–16 times as much as a single memory location; e.g., 16 floating-point numbers instead of just 1.

Caching

Taking advantage of what is in cache (i.e., minimizing *cache misses* or equivalently maximizing *cache hits*) can have a dramatic effect on program execution time.

For example, if the first 16 elements of an array are in the cache, a block of code that orders its operations such that it accesses these elements sequentially (instead of say elements 1, 17, 33, etc.) will execute 10–100 times faster.

In practice, caches are divided into *levels*.

The first level (L1) is the smallest and the fastest (and most expensive); higher level caches (L2, L3, etc.) are successively larger and slower (and cheaper).

Information stored in caches is also stored in main memory; information in higher-level caches may or may not be stored in lower-level ones.

Caching

There are two subtle issues that we now mention but do not treat in great detail.

First, there may be inconsistencies between what is stored in cache and main memory.

When something is newly written to cache, it must still be written to main memory.

Some systems write the cache values out immediately; others simply mark the data as *dirty* and only write to main memory when the whole cache line is replaced.

Second, there needs to be a mechanism to replace or *evict* cache lines.

The most commonly used scheme is based on evicting the *least recently used* data in a given cache.

Caching: an example

The cache is controlled by hardware, but the programmer can take advantage of knowing how caches work to write more efficient code.

An important fact is that although we think about two-dimensional arrays as rectangular blocks, they are effectively stored linearly (as one-dimensional vectors).

C stores arrays in *row-major order*, i.e., the one-dimensional storage vector is created by concatenating the rows of the matrix.

Thus, the $(1, 1)$ element is stored next to the $(1, 2)$ in memory, but it will generally be far away from the $(2, 1)$ element.

Fortran, on the other hand, stores arrays in *column-major order*.

This means a program that minimizes cache misses in one language will almost certainly maximize them in the other!

Caching: an example

To see this, we consider two methods for computing a matrix-vector product.

```
double A[MAX][MAX], x[MAX], y[MAX]
...
/* initialize A, x; set y = 0 */
...
/* Method 1 */
for (i=0; i < MAX; i++)
    for j=0; j < MAX; j++)
        y[i] += A[i][j]*x[j];
...
/* reset y = 0 */
...
/* Method 2 */
for j=0; j < MAX; j++)
    for (i=0; i < MAX; i++)
        y[i] += A[i][j]*x[j];
```

Caching: an example

Suppose $MAX=4$ and A is stored as follows; assume x and y are in cache.

Cache line	Elements of A
0	$A[0][0]$ $A[0][1]$ $A[0][2]$ $A[0][3]$
1	$A[1][0]$ $A[1][1]$ $A[1][2]$ $A[1][3]$
2	$A[2][0]$ $A[2][1]$ $A[2][2]$ $A[2][3]$
3	$A[3][0]$ $A[3][1]$ $A[3][2]$ $A[3][3]$

Assuming the cache starts off empty, both methods suffer a cache miss when trying to access $A[0][0]$.

Now suppose the cache loads the first row of A and only has capacity to store one row at a time.

Method 1 scores hits on $A[0][1]$, $A[0][2]$, $A[0][3]$ and not suffer a miss until it tries to access $A[1][0]$.

Method 2, however, misses on $A[1][0]$, then misses again on $A[2][0]$, etc.

For example, with $MAX = 1000$, the Method 1 might be a few times faster than Method 2.

Flynn's taxonomy

The original classification of parallel computers is known as **Flynn's taxonomy**¹.

Flynn classified systems according to the number of instruction streams and the number of data streams.

The classical von Neumann machine has a single instruction stream and a single data stream; hence it is a single-instruction single-data (SISD) machine.

At the other extreme is the multiple-instruction multiple-data (MIMD) machine, where a number of autonomous processors operate on their own data streams. This is the most general architecture in Flynn's taxonomy.

Intermediate to these two extremes are single-instruction multiple-data (SIMD) and multiple-instruction single-data (MISD) systems.

¹Michael Flynn, *Very high-speed computing systems*, Proceedings of the IEEE, **54**:1901–1909, December, 1966.

SIMD systems

A pure SIMD system has one CPU dedicated exclusively to control and a large number of subordinate ALUs, each with its own small amount of memory.

During each instruction cycle, the control processor broadcasts an instruction to all the ALUs, each of which either executes the instruction or is idle.

For example, suppose we have 3 arrays X, Y, and Z distributed such that each processor contains one element of each array, and we want to execute the following code:

```
      N = 1000
      DO 10 I=1,N
          IF (Y(I).NE.0.0) THEN
              Z(I) = X(I)/Y(I)
          ELSE
              Z(I) = X(I)
          ENDIF
10     CONTINUE
```

SIMD systems

Each ALU would execute something like

- Cycle 1: Test $y_{Local} \neq 0$.
- Cycle 2:
 - If $y_{Local} \neq 0$, $Z(I) = X(I)/Y(I)$.
 - If $y_{Local} = 0$, do nothing.
- Cycle 3:
 - If $y_{Local} \neq 0$, do nothing.
 - If $y_{Local} = 0$, $Z(I) = X(I)$.

Note the **synchronous** execution of statements.

This examples illustrates the disadvantages of SIMD systems: many conditionals or long branches may cause many processes to be idle.

But SIMD is easy to program, and communication is essentially no more expensive than computation (unlike in MIMD systems).

SIMD systems

The role of SIMD systems has morphed over time.

In the early 1990s, Thinking Machines made SIMD systems and was also the largest manufacturer of parallel supercomputers.

However, by the late 1990s, the only widely produced SIMD systems were vector processors.

Nowadays, graphics processing units (GPUs) and common CPUs use aspects of SIMD computing.

Due to their increasing popularity, it is worthwhile to say a few words about GPUs and general-purpose computing on GPUs (GPGPU).

GPUs and GPGPU

GPUs are specialized chips that were traditionally designed to efficiently prepare graphics data (e.g., points, lines, triangles, etc.) for visual display.

Their special structure makes them more effective than general-purpose CPUs for certain types of calculations, particularly ones that involve independent manipulation of large amounts of data.

GPUs are not pure SIMD systems; they excel at SIMD parallelism, but they can do more.

GPGPU is becoming increasingly popular because of the commonality of GPUs on standard PCs.

The strategy of accelerated computing is also seen as the most likely path to achieve exascale performance in the near future due to its cost effectiveness in terms of manufacturing and energy consumption.

General MIMD systems

The key difference between MIMD and SIMD systems is that **the processors in MIMD systems are autonomous**; i.e., each processor has its own CPU.

So each processor executes its own program **asynchronously** (at its own pace).

There is generally no relationship between what each processor is doing, even when running the same code.

The processors can be forced to synchronize with each other, but this is generally inefficient.

There are two main ways in which MIMD computers can be organized:

- shared memory
- distributed memory

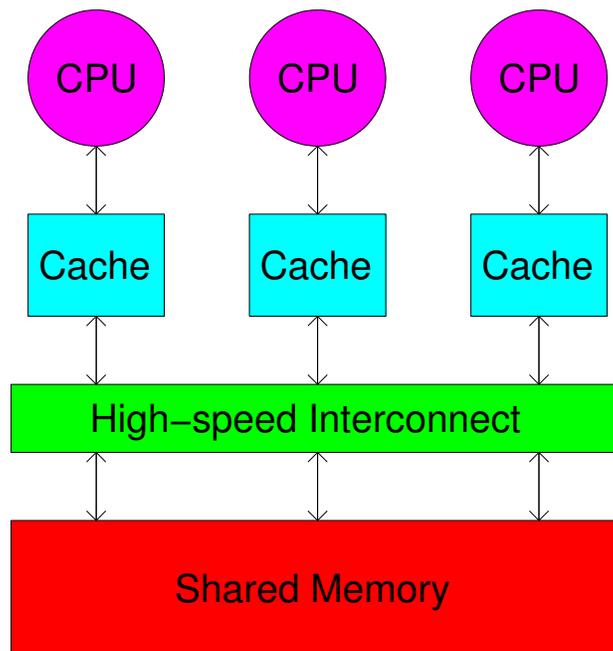
Of late, it has become standard to combine these two ways to form a hybrid third way.

Shared-memory paradigm

Shared-memory computers have multiple processors that share access to a global memory space via a high-speed memory bus.

This global memory space allows the processors to efficiently exchange or share access to data.

The number of processors used in shared-memory architectures is usually limited because the amount of data that can be processed is limited by the bandwidth of the memory bus connecting the processors.



Shared-memory paradigm

Most shared-memory systems use one or more *multicore* processors (multiple CPUs on one chip).

Typically each core has its own private level-1 cache; other caches may or may not be shared between cores.

The interconnect can work in two ways. Each processor is connected directly to a block of main memory.

1. When processors can access all memory at the same rate, this is called *uniform memory access (UMA)*.
2. When processors can access their own memory fastest, this is called *nonuniform memory access (NUMA)*.

UMA systems are generally easier to program because there are no issues with accessing *stale* data.

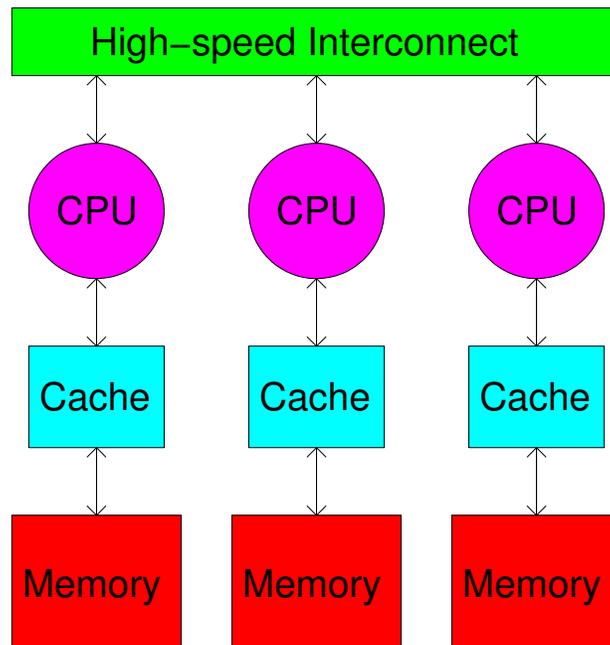
However, CPUs on NUMA systems can access their own memory faster than on UMA systems, and NUMA systems can support more memory than UMA systems.

Distributed-memory paradigms

Distributed-memory computers are essentially a collection (or *cluster*) of serial computers (nodes) working together to solve a problem.

Each node has rapid access to its own local memory and access to the memory of other nodes via some sort of communications network, usually a high-speed communications network (or “interconnect”).

Data are exchanged between nodes as messages over the network.

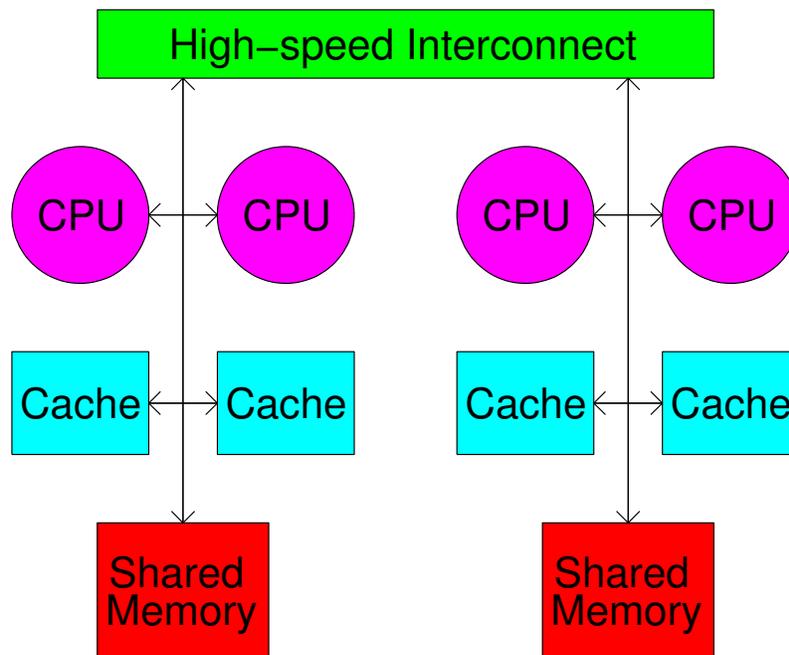


Hybrid memory paradigm

It is now standard for parallel computers to use a mixed shared/distributed memory architecture.

Each node typically consists of a group of up to (say) 16 processors connected via local shared memory, and the multiprocessor nodes are, in turn, connected via a high-speed interconnect.

Geographically distributed computers are connected by a *grid* infrastructure and are typically *heterogeneous*.



Latency and bandwidth

A key to the success of effective parallel programming is an understanding of how long it takes data to reach their destination.

This applies irrespective of whether we are considering data going from hard disk to main memory to cache to register or from one compute node to another.

The two pieces of data used to describe the performance of an(y) interconnect are the [latency](#) and the [bandwidth](#).

The latency is the time between when the source begins to transmit data and the time and the destination begins to receive them.

The bandwidth is rate at which the destination receives the data.

Latency and bandwidth

If the latency of an interconnect is l s and the bandwidth is b bytes per s, then time τ it takes to fully transmit and receive a message of size n bytes is

$$\tau = l + n/b.$$

Note that these terms are used differently!

Sometimes people refer to τ as latency.

Sometimes latency is used to describe any fixed overhead costs associated with sending messages.

For example, a message on a distributed-memory system consists of more than just the raw data; it may also contain the destination address, the message size, etc.

In this situation, latency would include the time it takes to assemble the message on the sending side and the time to disassemble the message at the destination.

Cache coherence

Recall that CPU caches are controlled by the system hardware and software, not the programmer.

This might sound like a good thing, but it has implications on shared-memory systems.

For example, consider a shared-memory system with two cores (0 and 1), each with a private data cache.

Suppose $x = 2$ is shared, y_0 is local to Core 0, and y_1, z_1 are local to Core 1.

If cores only *read* shared data, there is no problem.

Cache coherence

But consider the following sequence of events:

Time	Core 0	Core 1
0	$y_0 = x;$	$y_1 = 3*x;$
1	$x = 7;$	Code not involving x
2	Code not involving x	$z_1 = 4*x;$

It is clear that (eventually) y_0 will get the value 2.

We can safely assume y_1 will get the value 6.

But what about z_1 ?

We might like to think z_1 should get the value 28.

However, for this to happen, the value of x will have to be updated on Core 1 based on the change made at time 1 on Core 0.

This takes time! There is no guarantee it will be done by time 2, so z_1 might get the value 8!

This is the dreaded **cache coherence problem**.

Cache coherence

There are two main ways to ensure cache coherence: *snooping* and *directory-based* cache coherence.

Snooping originated in bus-based systems: when cores share a bus, they can see all the signals sent on it.

In terms of our example, basically when Core 0 updates its value x , it *broadcasts* the fact that it has done so.

If Core 1 snoops the bus, it can know x has been updated and mark its local value as invalid.

Of course, this idea is not limited to buses; it requires only the ability of a core to broadcast to all other cores.

Unfortunately, however, broadcasts are expensive, especially in large networks; so snooping is not scalable.

Cache coherence

Directory-based cache coherence attempts to remedy this problem by using a data structure called a directory.

The directory stores the status of each cache line.

It is typically distributed; e.g., each core/memory pair might store the part of the structure associated with its own local memory.

For example, if a line is read into the cache of Core 0, the directory entry is updated to indicate Core 0 has an updated copy of the line.

When a variable is updated, the directory is examined, and the cache controllers of the cores that have the cache line of that variable in their caches mark their lines as invalid.

This solution requires substantial additional storage, but when a cache variable is updated, only the cores storing that variable need to be contacted.

Cache coherence

A key point is that CPU caches are implemented in hardware; they work with cache lines, not variables.

This can have disastrous effects on performance.

Consider the following program to compute a function $f(i, j)$ and add its value to a vector y :

```
int i, j, m, n;
double y[m];

/* assume y = 0 */

for (i=0; i < m, i++)
    for (j=0; j < n, j++)
        y[i] += f(i, j);
```

Cache coherence

We can parallelize this by distributing the outer loop among p cores, e.g., the first m/p iterations are assigned to the first core, etc.

```
/* private variables */
int i, j, blockSize;

/* shared variables */
int m, n, p;
double y[m];

/* assume y = 0 */

iter = m/p;

/* Core 0 does this */
for (i=0; i < blockSize, i++)
    for (j=0; j < n, j++)
        y[i] += f(i,j);

/* Core 1 does this */
for (i=blockSize; i < 2*blockSize, i++)
    for (j=0; j < n, j++)
        y[i] += f(i,j);

/* etc. */
```

Cache coherence

Let $p = 2$, $m = 8$. Then if doubles are 8 bytes, cache lines are 64 bytes, and $y[0]$ is stored at the beginning of a cache line, y can be stored in one cache line.

What happens when the cores simultaneously execute their codes?

Because y is completely contained in one cache line, every time the statement $y[i] += f(i,j)$ is executed, the cache line is invalidated, and the other core will have to re-read the entire line before execution even though the cores are always writing to separate array entries.

In general, if n is large, the values may have to be brought into cache from main memory!

This will kill the performance of the parallel program, possibly making it even slower than serial execution.

Cache coherence

This phenomenon is known as **false sharing** because the system behaves like y was being shared by the cores (when in fact it does not have to be).

The problem is not with the correctness of the results (the results will be correct); it is poor performance.

A simple but effective remedy is to use temporary local storage and only copy results to the shared storage at the end of the computation.

Shared or distributed?

Given the seemingly added complication of using distributed-memory systems, it may be reasonable to ask why all MIMD systems are not shared memory.

The main reason is the scalability of interconnect cost.

Buses are cheap, but they are not efficient as the number of cores and memory modules increases.

On the other hand, certain interconnects such as hypercube and toroidal are relatively cheap; systems with thousands of cores have been successfully built using them.

In practice, distributed-memory systems are often better suited to solve problems with vast amounts of data or computation because of the possibility of a significantly greater number of cores.

Summary

- Paradigms of parallel hardware
- Flynn's taxonomy
- Cache coherence