

CS 3006 Parallel and Distributed Computer

Fall 2022

1. Learn about parallel and distributed computer architectures.(1)
2. Implement different parallel and distributed programming paradigms and algorithms using Message-Passing Interface (MPI) and OpenMP.(4)
3. Perform analytical modelling, dependence, and performance analysis of parallel algorithms and programs.(2)
4. Use Hadoop or MapReduce programming model to write bigdata applications.(5)

Week # 4 – Lecture # 10, 11, 12

15th , 17th , 18th Safar ul Muzaffar, 1444

12th , 14th , 15th September 2022

Dr. Nadeem Kafi Khan

Lecture # 10 – Topics (**Lab # 4**)

- **#pragma omp for**
- **How it works with #pragma omp parallel**
- **Shorthand: #pragma omp parallel for**
- **Two common mistakes while using #pragma omp for**

```

a();
#pragma omp parallel
{
    c(1);
    c(2);
}
z();

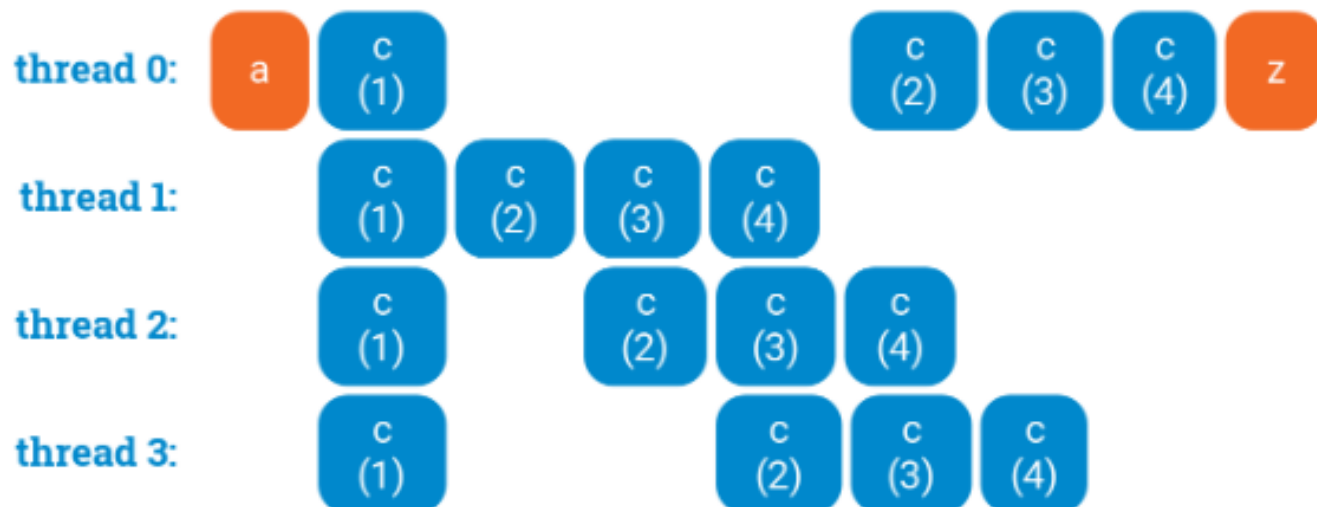
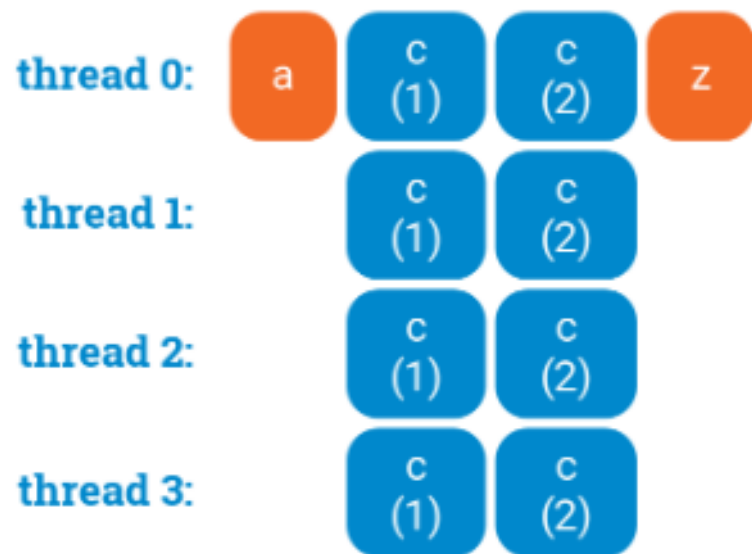
```

```

a();
#pragma omp parallel
{
    c(1);
    #pragma omp critical
    {
        c(2);
    }
    c(3);
    c(4);
}
z();

```

If we execute the above program, the timeline of from left to right):



Shared vs. private data

Any variable that is declared outside a `parallel` region is **shared**: there is only one copy of the variable, and all threads refer to the same variable. Care is needed whenever you refer to such a variable.

Any variable that is declared inside a `parallel` region is **private**: each thread has its own copy of the variable. Such variables are always safe to use.

If a shared variable is read-only, you can safely refer to it from multiple threads inside the `parallel` region. However, if **any thread ever writes to a shared variable**, then proper coordination is needed to ensure that no other thread is simultaneously reading or writing to it.

```
static void critical_example(int v) {

    // Shared variables
    int a = 0;
    int b = v;

    #pragma omp parallel
    {
        // Private variable - one for each thread
        int c;

        // Reading from "b" is safe: it is read-only
        // Writing to "c" is safe: it is private
        c = b;

        // Reading from "c" is safe: it is private
        // Writing to "c" is safe: it is private
        c = c * 10;


        #pragma omp critical
        {
            // Any modifications of "a" are safe:
            // we are inside a critical section
            a = a + c;
        }
    }

    // Reading from "a" is safe:
    // we are outside parallel region
    std::cout << a << std::endl;
}
```

OpenMP parallel for loops

Let us start with a basic example: we would like to do some preprocessing operation `a()`, then we would like to do some independent calculations `c(0)`, `c(1)`, ..., and finally some postprocessing `z()` once all calculations have finished. In this example, each of the calculations takes roughly the same amount of time. A straightforward for-loop uses only one thread and hence only one core on a multi-core computer:

```
a();  
for (int i = 0; i < 10; ++i) {  
    c(i);  
}  
z();
```

thread 0: 

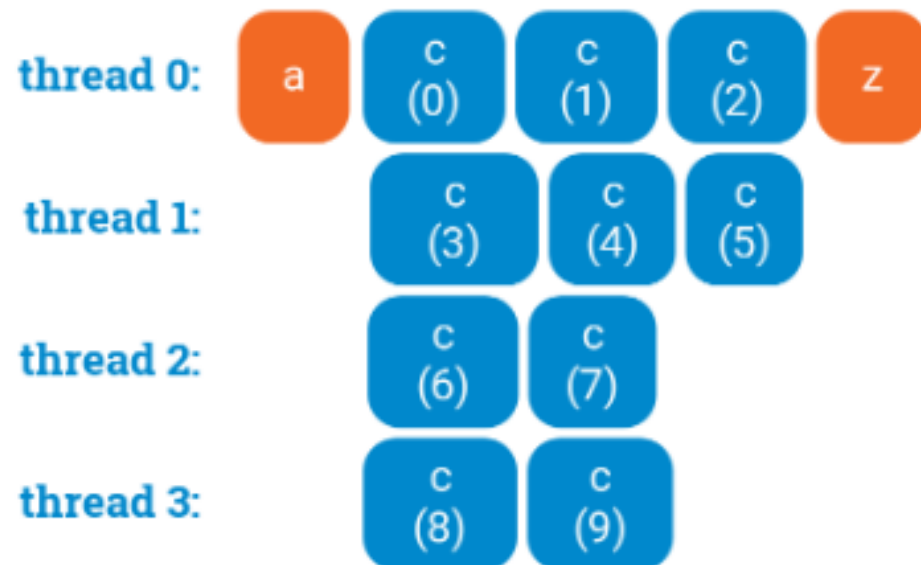
thread 1:

thread 2:

thread 3:

With OpenMP parallel for loops, we can easily parallelize it so that we are making a much better use of the computer. Note that OpenMP automatically waits for all threads to finish their calculations before continuing with the part that comes after the parallel for loop:

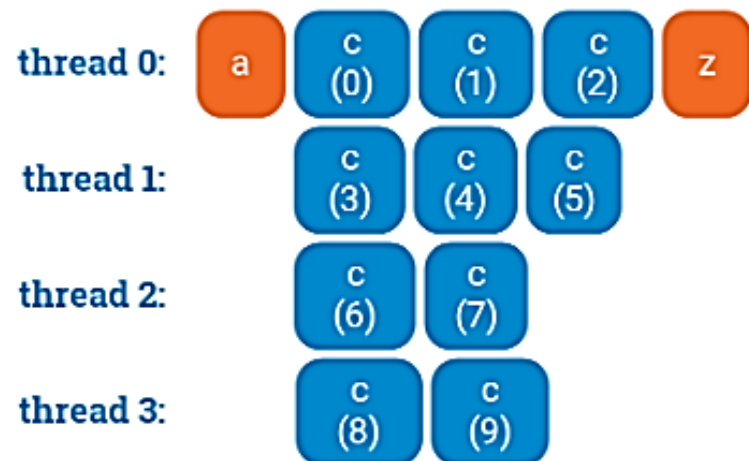
```
a();  
#pragma omp parallel for  
for (int i = 0; i < 10; ++i) {  
    c(i);  
}  
z();
```



It is just a shorthand

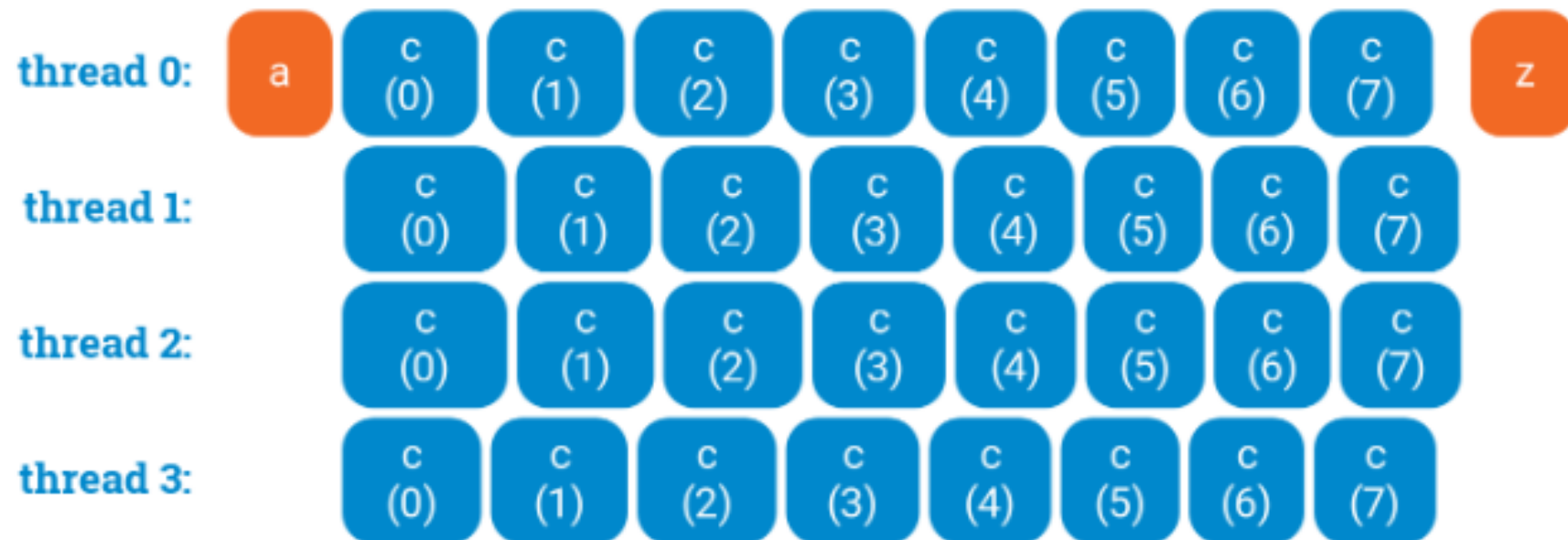
The `omp parallel for` directive is just a commonly-used shorthand for the combination of two directives: `omp parallel`, which declares that we would like to have multiple threads in this region, and `omp for`, which asks OpenMP to assign different iterations to different threads:

```
a();  
#pragma omp parallel  
{  
    #pragma omp for  
    for (int i = 0; i < 10; ++i) {  
        c(i);  
    }  
}  
z();
```



A common mistake is to just use `omp parallel` together with a for loop. This creates multiple threads for you, but it does not do any worksharing – all threads simply run all iterations of the loop, which is most likely not what you want:

```
a();  
#pragma omp parallel  
for (int i = 0; i < 8; ++i) {  
    c(i);  
}  
z();
```



Another common mistake is to use `omp for` alone outside a parallel region. It does not do anything if we do not have multiple threads available:

```
a();  
#pragma omp for  
for (int i = 0; i < 8; ++i) {  
    c(i);  
}  
z();
```

thread 0:



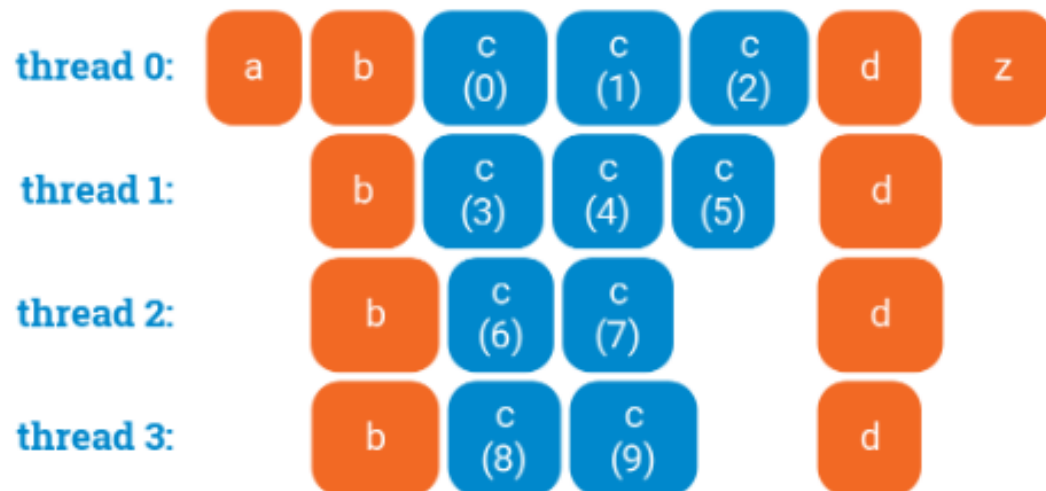
thread 1:

thread 2:

thread 3:

While in many cases it is convenient to combine `omp parallel` and `omp for` in one directive, please remember that you can always split them. This way it is possible to do some thread-specific preprocessing and postprocessing if needed:

```
a();  
#pragma omp parallel  
{  
    b();  
    #pragma omp for  
    for (int i = 0; i < 10; ++i) {  
        c(i);  
    }  
    d();  
}  
z();
```



Lecture # 11 – Topics

- Impact of Memory Bandwidth
 - Understanding the memory hierarchy architecture
 - Dot-Project Example
- Multiplying a matrix with a vector
 - Column-major code explanation
 - Row-major code with explanation

Impact of Memory Bandwidth

- Memory bandwidth is determined by the **bandwidth of the memory bus as well as the memory units.**
- **Memory bandwidth can be improved by increasing the size of memory blocks.**
- The underlying system takes l time units (where l is the latency of the system) to deliver b units of data (where b is the block size).

Impact of Memory Bandwidth: Example

- One commonly used technique to improve memory bandwidth is to increase the size of the memory blocks. For example, a **single memory request** returns a contiguous block of **four words**.
- The single unit of four words in this case is also referred to as a **cache line**. Conventional computers typically fetch two to eight words together into the cache.
- Now, we repeat the dot-product computation with a cache line size of 4 (see Example 2.4 on the next slide).

Example 2.4 Effect of block size: dot-product of two vectors

Consider again a memory system with a single cycle cache and 100 cycle latency DRAM with the processor operating at 1 GHz. If the block size is one word, the processor takes 100 cycles to fetch each word. For each pair of words, the dot-product performs one multiply-add, i.e., two FLOPs. Therefore, the algorithm performs one FLOP every 100 cycles for a peak speed of 10 MFLOPS as illustrated in [Example 2.2](#).

Now let us consider what happens if the block size is increased to four words, i.e., the processor can fetch a four-word cache line every 100 cycles. Assuming that the vectors are laid out linearly in memory, eight FLOPs (four multiply-adds) can be performed in 200 cycles. This is because a single memory access fetches four consecutive words in the vector. Therefore, two accesses can fetch four elements of each of the vectors. This corresponds to a FLOP every 25 ns, for a peak speed of 40 MFLOPS. Note that increasing the block size from one to four words did not change the latency of the memory system. However, it increased the bandwidth four-fold. In this case, the increased bandwidth of the memory system enabled us to accelerate the dot-product algorithm which has no data reuse at all.

Example 2.4 Effect of block size: dot-product of two vectors

Another way of quickly estimating performance bounds is to estimate the cache hit ratio, using it to compute mean access time per word, and relating this to the FLOP rate via the underlying algorithm. For example, in this example, there are two DRAM accesses (cache misses) for every eight data accesses required by the algorithm. This corresponds to a cache hit ratio of 75%. Assuming that the dominant overhead is posed by the cache misses, the average memory access time contributed by the misses is 25% at 100 ns (or 25 ns/word). Since the dot-product has one operation/word, this corresponds to a computation rate of 40 MFLOPS as before. A more accurate estimate of this rate would compute the average memory access time as $0.75 \times 1 + 0.25 \times 100$ or 25.75 ns/word. The corresponding computation rate is 38.8 MFLOPS. ■

Example 2.5 Impact of strided access

Consider the following code fragment:

```
1  for (i = 0; i < 1000; i++)  
2      column_sum[i] = 0.0;  
3      for (j = 0; j < 1000; j++)  
4          column_sum[i] += b[j][i];
```

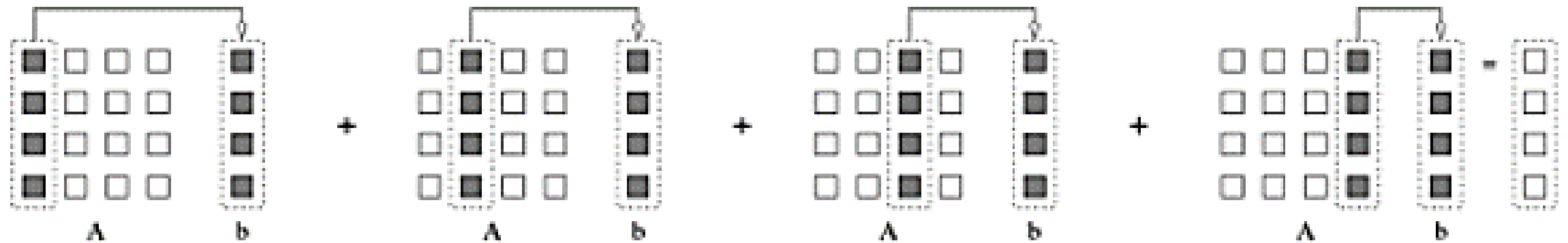


Figure 2.2. (a) Column major data access

Example 2.5 Impact of strided access

The code fragment sums columns of the matrix `b` into a vector `column_sum`. There are two observations that can be made: (i) the vector `column_sum` is small and easily fits into the cache; and (ii) the matrix `b` is accessed in a column order as illustrated in [Figure 2.2\(a\)](#). For a matrix of size 1000 x 1000, stored in a row-major order, this corresponds to accessing every 1000th entry. Therefore, it is likely that only one word in each cache line fetched from memory will be used. Consequently, the code fragment as written above is likely to yield poor performance. ■

Example 2.6 Eliminating strided access

```
1  for (i = 0; i < 1000; i++)
2      column_sum[i] = 0.0;
3  for (j = 0; j < 1000; j++)
4      for (i = 0; i < 1000; i++)
5          column_sum[i] += b[j][i];
```

$$\begin{aligned} Ax &= \begin{bmatrix} 1 & -1 & 2 \\ 0 & -3 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \\ 0 \end{bmatrix} \\ &= \begin{bmatrix} 2 \cdot 1 - 1 \cdot 1 + 0 \cdot 2 \\ 2 \cdot 0 - 1 \cdot 3 + 0 \cdot 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 \\ -3 \end{bmatrix}. \end{aligned}$$

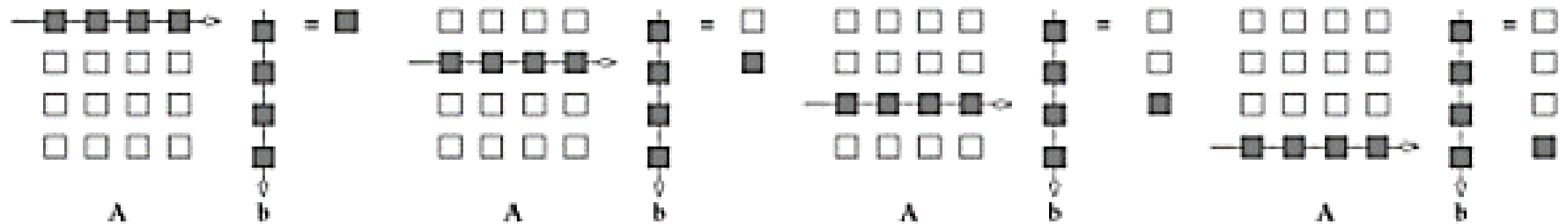


Figure 2.2. (b) Row major data access.

Example 2.6 Eliminating strided access

In this case, the matrix is traversed in a row-order as illustrated in [Figure 2.2\(b\)](#). However, the reader will note that this code fragment relies on the fact that the vector `column_sum` can be retained in the cache through the loops. Indeed, for this particular example, our assumption is reasonable. If the vector is larger, we would have to break the iteration space into blocks and compute the product one block at a time. This concept is also called ***tiling*** an iteration space. The improved performance of this loop is left as an exercise for the reader. ■

Memory System Performance: Summary

- The series of examples presented in this section illustrate the following concepts:
 - Exploiting spatial and temporal locality in applications is critical for amortizing memory latency and increasing effective memory bandwidth.
 - The ratio of the number of operations to number of memory accesses is a good indicator of anticipated tolerance to memory bandwidth.
 - Memory layouts and organizing computation appropriately can make a significant impact on the spatial and temporal locality.

Example 2.1 Superscalar execution

Figure 2.1. Example of a two-way superscalar execution of instructions.

```
1. load R1, @1000
2. load R2, @1008
3. add R1, @1004
4. add R2, @100C
5. add R1, R2
6. store R1, @2000
```

(i)

```
1. load R1, @1000
2. add R1, @1004
3. add R1, @1008
4. add R1, @100C
5. store R1, @2000
```

(ii)

```
1. load R1, @1000
2. add R1, @1004
3. load R2, @1008
4. add R2, @100C
5. add R1, R2
6. store R1, @2000
```

(iii)

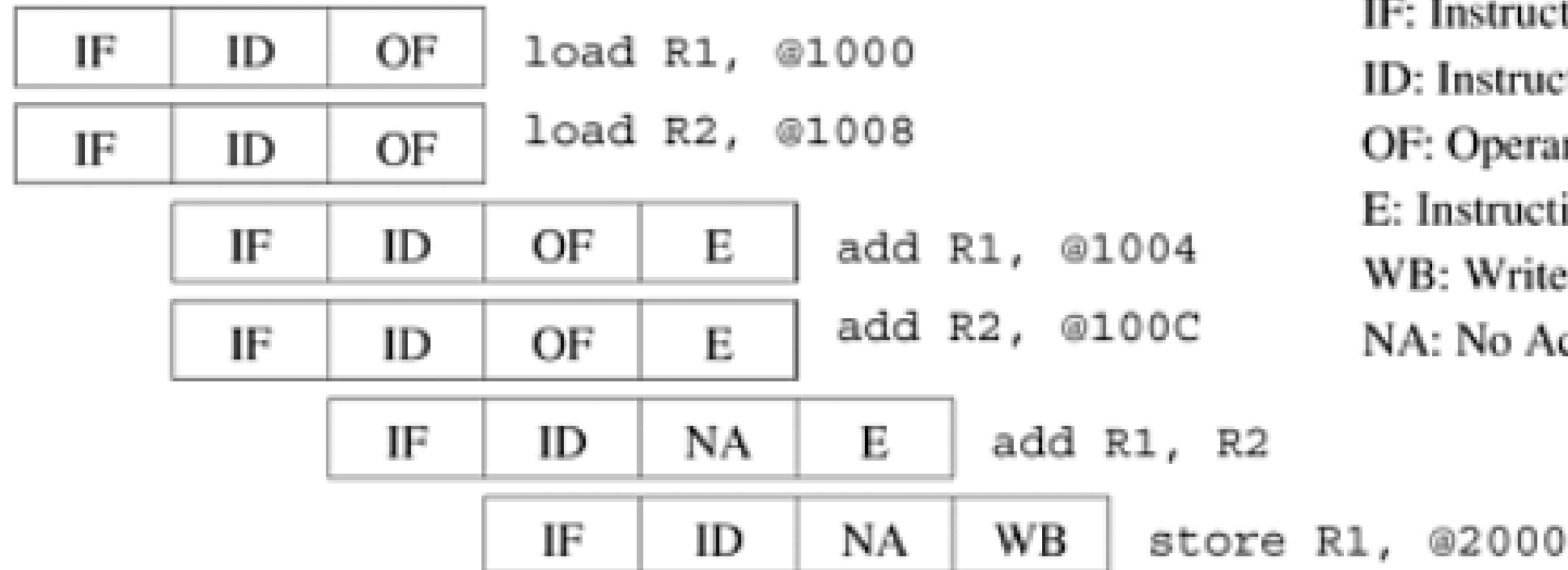
(a) Three different code fragments for adding a list of four numbers.

Note. In all three assembly code snippets shown above, only load and store can access memory. Therefore constant with load and store are memory locations. In all other instructions, they are immediate values which are part of the instructions word.

Example 2.1 Superscalar execution

Instruction cycles

0 2 4 6 8



IF: Instruction Fetch

ID: Instruction Decode

OF: Operand Fetch

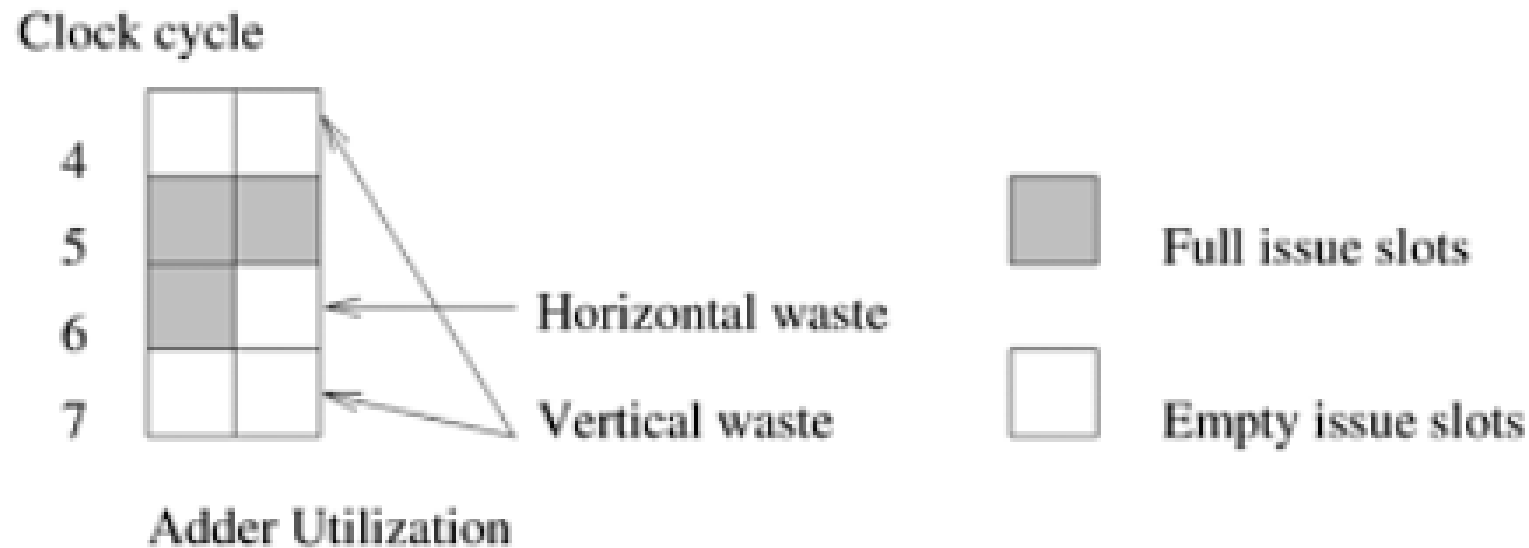
E: Instruction Execute

WB: Write-back

NA: No Action

(b) Execution schedule for code fragment (i) above.

Example 2.1 Superscalar execution



(c) Hardware utilization trace for schedule in (b).

Assuming two execution units (multiply-add units), the figure illustrates that there are several zero-issue cycles (cycles in which the floating point unit is idle). These are essentially wasted cycles from the point of view of the execution unit. If, during a particular cycle, no instructions are issued on the execution units, it is referred to as **vertical waste**; if only part of the execution units are used during a cycle, it is termed **horizontal waste**. In the example, we have two cycles of vertical waste and one cycle with horizontal waste. In all, only three of the eight available cycles are used for computation. This implies that the code fragment will yield no more than three-eighths of the peak rated FLOP count of the processor.

Example 2.2 Effect of memory latency on performance

Consider a processor operating at 1 GHz (1 ns clock) connected to a DRAM with a latency of 100 ns (no caches). Assume that the processor has two multiply-add units and is capable of executing four instructions in each cycle of 1 ns. The peak processor rating is therefore 4 GFLOPS. Since the memory latency is equal to 100 cycles and block size is one word, every time a memory request is made, the processor must wait 100 cycles before it can process the data. Consider the problem of computing the dot-product of two vectors on such a platform. A dot-product computation performs one multiply-add on a single pair of vector elements, i.e., each floating point operation requires one data fetch. It is easy to see that the peak speed of this computation is limited to one floating point operation every 100 ns, or a speed of 10 MFLOPS, a very small fraction of the peak processor rating. This example highlights the need for effective memory system performance in achieving high computation rates. ■

Example 2.3 Impact of caches on memory system performance

As in the previous example, consider a 1 GHz processor with a 100 ns latency DRAM. In this case, we introduce a cache of size 32 KB with a latency of 1 ns or one cycle (typically on the processor itself). We use this setup to multiply two matrices A and B of dimensions 32×32 . We have carefully chosen these numbers so that the cache is large enough to store matrices A and B , as well as the result matrix C . Once again, we assume an ideal cache placement strategy in which none of the data items are overwritten by others. Fetching the two matrices into the cache corresponds to fetching 2K words, which takes approximately 200 μ s. We know from elementary algorithmics that multiplying two $n \times n$ matrices takes $2n^3$ operations. For our problem, this corresponds to 64K operations, which can be performed in 16K cycles (or 16 μ s) at four instructions per cycle. The total time for the computation is therefore approximately the sum of time for load/store operations and the time for the computation itself, i.e., 200+16 μ s. This corresponds to a peak computation rate of 64K/216 or 303 MFLOPS. Note that this is a thirty-fold improvement over the previous example, although it is still less than 10% of the peak processor performance. We see in this example that by placing a small cache memory, we are able to improve processor utilization considerably. ■

Lecture # 12 – Topics

- Alternate Approaches for Hiding Memory Latency
- Multithreading – max processor utilization through task decomposition
- Prefetching data using the principle of Spatial Locality
- Tradeoffs of Multithreading and Prefetching
- Buses vs Crossbar
- Message Passing Costs in Parallel Computers

Alternate Approaches for Hiding Memory Latency

- **Scenario:** Consider the problem of browsing the web on a very slow network connection. We deal with the problem in one of three possible ways:
 - We anticipate which pages we are going to browse ahead of time and issue requests for them in advance; (**Pre-fetching**)
 - We open multiple browsers and access different pages in each browser, thus while we are waiting for one page to load, we could be reading others; or (**Multi-Threading**)
 - We access a whole bunch of pages in one go - amortizing the latency across various accesses (**Spatial Locality**)
- The first approach is called *prefetching*, the second *multithreading*, and the third one corresponds to *spatial locality in accessing memory words*.

Multithreading for Latency Hiding

A thread is a single stream of control in the flow of a program.

We illustrate threads with a simple example:

```
for (i = 0; i < n; i++)  
    c[i] = dot_product(get_row(a, i), b);
```

Each dot-product is independent of the other, and therefore represents a concurrent unit of execution. We can safely rewrite the above code segment as:

```
for (i = 0; i < n; i++)  
    c[i] = create_thread(dot_product, get_row(a, i), b);
```

Multithreading for Latency Hiding

- The execution schedule in the previous example is predicated upon two assumptions:
 - the memory system is capable of servicing multiple outstanding requests, and
 - the processor is capable of switching threads at every cycle.
- It also requires the program to have an explicit specification of concurrency in the form of threads (GPUs have threads managed in hardware).
- Many machines rely on multithreaded processors that can switch the context of execution in every cycle. Consequently, they are able to hide latency effectively.

Prefetching for Latency Hiding

- Misses on loads cause programs to stall.
- Why not advance the loads so that by the time the data is actually needed, it is already there!
- The only drawback is that you might need more space to store advanced loads.
- However, if the advanced loads are overwritten, we are no worse than before!

Tradeoffs of Multithreading and Prefetching

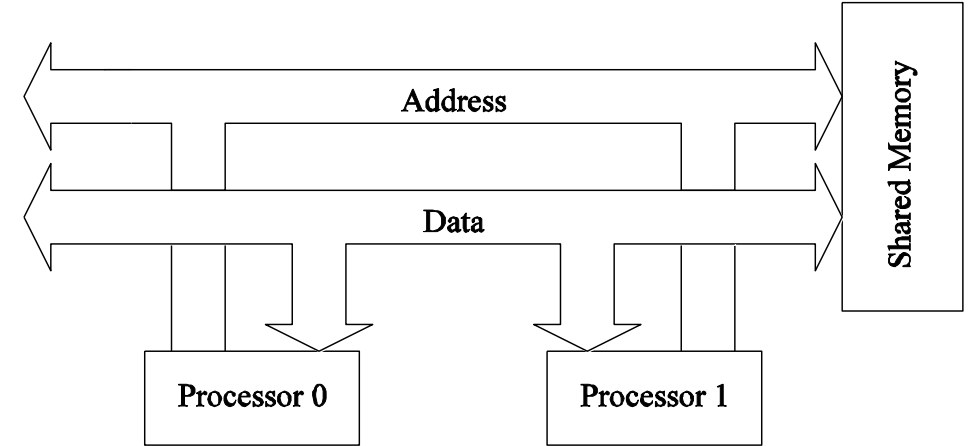
- Bandwidth requirements of a multithreaded system may increase very significantly because of the smaller cache residency of each thread.
 - They become bandwidth bound instead of latency bound.
- Multithreading and prefetching only address the latency problem and may often exacerbate the bandwidth problem.
- Multithreading and prefetching also require significantly more hardware resources in the form of storage.

Network Topologies: Buses

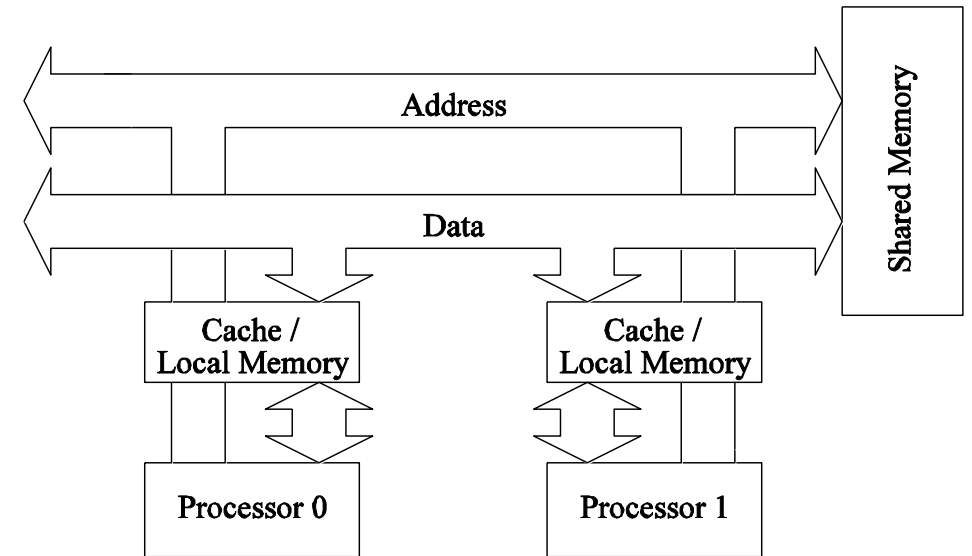
- Some of the simplest and earliest parallel machines used buses.
- All processors access a common bus for exchanging data.
- The distance between any two nodes is $O(1)$ in a bus. The bus also provides a convenient broadcast media.
- the bandwidth of the shared bus is a major bottleneck
- However,
- Typical bus based machines are limited to dozens of nodes. Sun Enterprise servers and Intel Pentium based shared-bus multiprocessor

Network Topologies: Buses

Since much of the data accessed by processors is local to the processor, a local memory can improve the performance of bus-based machines.



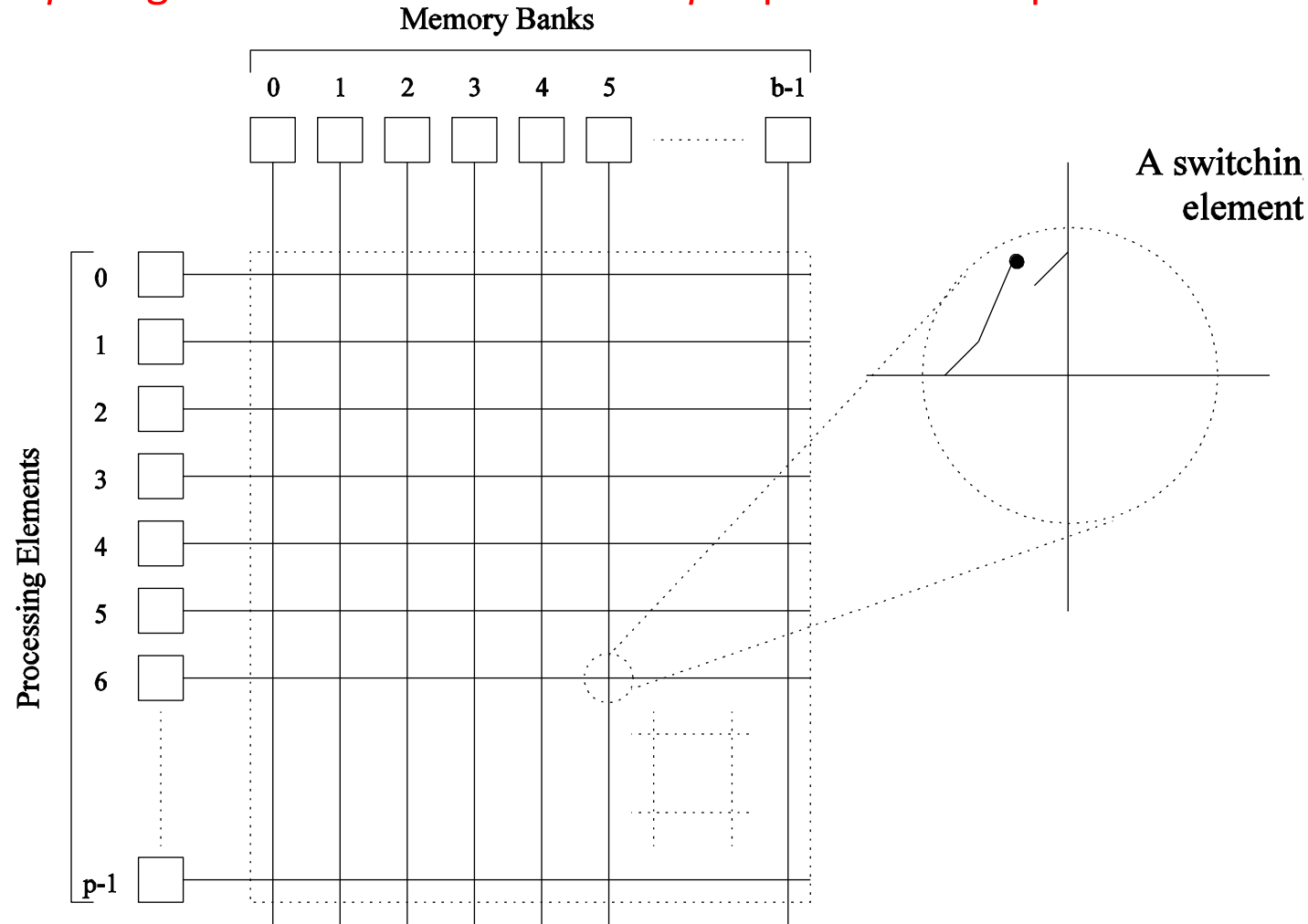
(a)



Bus-based interconnects (a) with no local caches; (b) with local memory/caches. (b)

Network Topologies: Crossbars

A crossbar network uses an $p \times m$ grid of switches to connect p inputs to m outputs in a non-blocking manner.



A completely non-blocking crossbar network connecting p processors to b memory banks.

Message Passing Costs in Parallel Computers

- The total time to transfer a message over a network comprises of the following:
 - **Startup time (t_s):** Time spent at sending and receiving nodes (executing the routing algorithm, programming routers, etc.).
 - **Per-hop time (t_h):** This time is a function of number of hops and includes factors such as switch latencies, network delays, etc.
 - **Per-word transfer time (t_w):** This time includes all overheads that are determined by the length of the message. This includes bandwidth of links, error checking and correction, etc.