Chapter 12
Parallel Processing and Performance

12.1. For simplicity, it is assumed that 32-bit addresses are used and data consists of 32-bit words. It is also initially assumed that each bus has only 32 lines that are multiplexed between address and data.

The conventional bus is held for a total of $6T$ seconds, including the $4T$ seconds for reading the contents of the memory. But useful information appears on the bus for only $2T$ seconds, consisting of $1T$ to send the address to memory and $1T$ to transfer the data to the processor. Multiple read requests on this bus occur one at a time, with each request holding the bus for $6T$ seconds. The effective throughput is therefore $2T/6T$ or 33.3 percent of the maximum bandwidth of the bus.

For a split-transaction bus, however, each read request requires only $2T$ seconds of actual bus utilization. The bus is otherwise available for other requests and responses. If there are multiple memory modules available, and no conflicts occur among several requests that appear on the bus in quick succession, then the effective throughput can ideally be 100 percent of the maximum bandwidth of the bus.

Therefore, $100/33.3 = 3$ conventional buses would give the same effective throughput as the split-transaction bus.

If, instead of multiplexed bus lines, there were separate address/data lines, it would be possible to overlap the address of the next request on the conventional bus with the data for the current request. This would reduce the time between successive data responses to $5T$ seconds. If the effective throughput is considered only for the data, it is just $1T/5T$ or 20 percent of the maximum bandwidth of the bus.

A split-transaction bus with separate address/data lines could, in the best case, have a new data response appear every $1T$ seconds (with the address of a new request overlapping with the data for a previous request). If there are sufficient memory modules available with no conflicts occuring in a sequence of requests appearing in quick succession, the effective throughput for data would ideally be 100 percent.

With separate lines, $100/20 = 5$ conventional buses would be needed to equal the throughput of the split-transaction bus.

If the memory access time increases, the effective throughput for the conventional bus is lower in both cases (multiplexed or separate lines) because the bus is held idle for a longer period of time. However, the split-transaction bus in either case can still ideally provide an effective throughput of 100 percent, if a sufficient number of non-conflicting requests are made over a short period of time. Thus, the number of conventional buses needed to achieve the same effective throughput increases.

Finally, a further generalization that can be considered is the situation of data transfers on the bus requiring more than $1T$ seconds. For example, a single cache block consisting of many words may require several successive transfers on the bus data lines. For a given memory access time to obtain the first portion of the data from the memory, a larger data size has the effect of decreasing the amount of idle time within the total time that the coventional bus must be held (the total is the sum of the initial memory access time and the data transfer time). Thus, the effective throughput of the conventional bus increases. For the split-transaction bus, the ideal throughput of 100 percent is still applicable, even with a larger data size. Consequently, fewer conventional buses would be needed to achieve the same throughput as the split-transaction bus, in comparison to a situation with a smaller data size. Because the data size is larger, the ratio of effective throughputs in the calculation for the required number of conventional buses may not yield an integer result. Hence, the next largest integer would be the final answer.

In all cases, using a larger number of conventional buses than indicated by the calculation to determine the throughput-matching number of buses would result in a situation where the aggregate throughput with multiple conventional buses could—under the right circumstances with multiple non-conflicting requests in quick sucession—exceed the maximum throughput achievable with a single split-transaction bus.

12.2. (*a*) A program that executes on a single processor and performs the specified tasks in the order that respects the dependences is shown below.

```
void    main (void)
{
        T0 ();
        T1 ();
        ⋮
        Tk ();
}
```

(*b*) To execute the same tasks in parallel on k processors, it is necessary to ensure that function T0() is executed first. The remaining functions, T1() to Tk(), can be then executed simultaneously on different processors. There is more than one way to implement a parallel program for this case. The simplest solution involves using the *create_thread()* library routine to create k threads, passing pointers to each of the functions T1() to Tk(). This solution is shown below.

```
#include    "threads.h"          /* Routines for thread creation/synchronization. */

void        main (void)
{
            T0 ();
            create_thread (T1);
            ⋮
            create_thread (Tk);
}
```

There are a total of $k + 1$ threads, but the main thread does not perform any additional computation after creating the other threads. The main thread can terminate, allowing the remaining k threads to use the k available processors. The program remains active until the last of of the created threads terminates.

An alternative approach involves the use of a common function that is executed by all threads. One thread can be designated to call T0() from the common function, then each thread can uniquely call one of the individual functions T1() to Tk(). A barrier can be introduced for synchronization to ensure that the dependences are respected. The solution reflecting this alternative approach is shown below.

```
#include    "threads.h"    /* Routines for thread creation/synchronization. */

Barrier     bar;    /* Shared variable to support barrier synchronization. */

void        ParallelFunction (void)
{
            int my_id;

            my_id = get_my_thread_id ();    /* Get unique identifier for this thread. */

            if (my_id == 0) T0();    /* Only main thread performs this computation. */
            barrier (&bar, k);    /* All threads wait here for T0() to be performed. */
            switch (my_id + 1)    /* Thread i - 1 executes function Ti(). */
            {
               case 1: T1(); break;
               case 2: T2(); break;
               :
               case k: Tk(); break;
            }
}

void        main (void)
{
            int i;

            init_barrier (&bar);
            for (i = 1; i < k; i++)    /* Create k - 1 additional threads. */
               create_thread (ParallelFunction);
            ParallelFunction();    /* Main thread also joins parallel execution. */
}
```

12.3. The details of how either invalidation or updating can be implemented are described in Section 12.4, and the advantages/disadvantages of the two techniques can be deduced directly from that discussion. In general, it would seem that the use of invalidation and write-back for locations in memory that are modified results in less bus traffic and eliminates potentially wasted cache updating operations. However, cache hit rates may be lowered by using this strategy, if locations being accessed by different processors are frequently invalidated. Updating associated with a write-through policy may lead to higher hit rates and may be simpler to implement, but may cause unacceptably high bus traffic and wasted update operations. The details of how reads and writes on shared cached blocks (lines) are normally interleaved from distinct processors in some class of applications will actually determine which coherence strategy is most appropriate.

12.4. A shared-memory multiprocessor can emulate a message-passing multicomputer more easily than the reverse arrangement. The act of message passing can be implemented by the transfer of (message) buffer pointers or the contents of complete (message) buffers between two processes that operate in a shared address space but effectively access data unique to each of them. The exchange of pointers or data involves normal Read and Write requests that are all treated in the same manner, regardless of the source of the request and the location in the shared memory.

A multicomputer system can emulate a multiprocessor by considering the aggregate of all of the local memories of the individual computers as the shared memory of the multiprocessor. However, each computer can make Read and Write requests directly to only its local memory. Any request by a computer to access a nonlocal component of the shared memory must be distinguished. Instead of proceeding with a normal access to the local memory, the computer must translate such a request into a message-passing interaction with another computer. This process can be slow and cumbersome.

12.5. From the discussion in Section 12.2, the number of instructions executed is $9N$ for the original loop and $9N/L$ for the vectorized loop. However, it is appropriate to enlarge the scope of the comparison beyond the instruction counts within the loops. For a somewhat more accurate assessment of the total execution time in each case, the instruction immediately before the loop must also be included. Assuming for simplicity that all instructions require the same amount of time to execute, the resulting expression for the ideal speedup from vectorization is

$$\frac{1 + 9N}{1 + 9N/L}$$

Using $N = 32$, the following table provides the ideal speedup values for vector lengths of 4, 8, 16, and 32.

L	ideal speedup
4	3.9
8	7.8
16	15
32	29

An even more accurate assessment for the execution time in each case would also include three additional instructions to initialize registers with the starting addresses of the three arrays that are accessed in the loop. The revised expression for the speedup from vectorization is

$$\frac{4 + 9N}{4 + 9N/L}$$

The ideal speedup values for $N = 32$ can be recalculated to obtain the table below.

L	ideal speedup
4	3.8
8	7.3
16	13
32	22

12.6. The tables from the solution to Problem 12.5 can be augmented with another column for the efficiency given by the ratio of the speedup to the vector length. The augmented table below reflects the scope of comparison that considers only the given instruction that appears before the loop.

L	ideal speedup	efficiency
4	3.9	0.98
8	7.8	0.98
16	15	0.94
32	29	0.91

The augmented table below enlarges the scope of the comparison to also account for three additional instructions for initializing registers with array starting addresses of the three arrays before entering the loop.

L	ideal speedup	efficiency
4	3.8	0.96
8	7.3	0.91
16	13	0.83
32	22	0.70

Having only a few instructions outside of the loop causes the efficiency to diminish as the vector length increases. The number of passes through the loop is reduced with a larger vector length, leading to fewer total instructions being executed. However, the instructions outside the loop are always executed and are not affected by the vector length. This situation is a reflection of Amdahl's Law, where the unenhanced portion of execution time limits the overall speedup.

Increasing the array size for a given vector length improves the efficiency because the number of instructions executed within the loop increases, thereby reducing the contribution of the instructions outside the loop to the total execution time. For example, a larger array size of $N = 256$ for a vector length of 32 gives a speedup of 30 when four instructions outside the loop are considered. The corresponding efficiency is 0.95, which is larger than the 0.70 for $N = 32$.

12.7. The proposed method of incrementally updating the global sum for the dot product involves all threads accessing a shared counter variable. Because threads must wait for the counter value to change, and each such change is caused by another thread, it is necessary to declare the counter variable as being volatile (see Section 4.7).

In addition to requiring all threads to wait for exclusive access to update the global sum for the dot product, the thread that executes the main function must also wait until all threads have completed their updates before printing the final result. The required condition is when the counter value matches the total number of threads.

The modified version of the parallel program in Figure 12.8 that implements the approach based on a shared counter variable is shown on the next page. The barrier is no longer required, nor is the global array to hold the partial results for the threads.

```
#include      <stdio.h>              /* Routines for input/output. */
#include      "threads.h"            /* Routines for thread creation/synchronization. */

#define       N    100               /* Number of elements in each vector. */
#define       P    4                 /* Number of processors for parallel execution. */

double        a[N], b[N];            /* Vectors for computing the dot product. */
double        dot_product;           /* The global sum of partial results computed by the threads. */
volatile int  thread_id_counter;     /* Used to ensure exclusive access to dot_product. */
                                     /* Note that the counter is declared as volatile. */

void          ParallelFunction (void)
{
              int my_id, i, start, end;
              double s;

              my_id = get_my_thread_id ();    /* Get unique identifier for this thread. */
              start = (N/P) * my_id;     /* Determine start/end using thread identifier. */
              end = (N/P) * (my_id + 1) - 1;    /* N is assumed to be evenly divisible by P. */
              s = 0.0;
              for (i = start; i <= end; i++)
                 s = s + a[i] * b[i];

              while (thread_id_counter != my_id);     /* Wait for permission to proceed. */
              dot_product = dot_product + s;     /* Update dot_product. */
              thread_id_counter = thread_id_counter + 1;     /* Give permission to next thread. */
}

void          main (void)
{
              int i;

              <Initialize vectors a[], b[] – details omitted.>
              dot_product = 0.0;     /* Initialize sum of partial results. */
              thread_id_counter = 0;     /* Initialize counter that ensures exclusive access. */
              for (i = 1; i < P; i++)     /* Create P - 1 additional threads. */
                 create_thread (ParallelFunction);
              ParallelFunction();     /* Main thread also joins parallel execution. */
              while (thread_id_counter != P);     /* Wait until last update to dot_product. */
              printf ("The dot product is %g\n", dot_product);
}
```

12.8. The expression of Amdahl's Law for speedup is $1/(f_{unenh} + f_{enh}/p)$. An even distribution of the parallel workload over eight processors means that $p = 8$. The desired speedup is 5. Combining this information with the fact that $f_{unenh} + f_{enh} = 1$ results in the equation

$$1/(1 - f_{enh} + f_{enh}/8) = 5$$

Rearranging the terms in the above equation leads to the solution $f_{enh} = 0.91$. Thus, the fraction of the total (original) execution time that can be parallelized must be 91 percent to achieve the desired speedup with eight processors.