An Introduction to Parallel Programming Solutions, Chapter 2

Jinyoung Choi and Peter Pacheco

February 25, 2011

1. (a) The following table shows times in nanoseconds.

Stage	Time
Fetch	2
Compare	1
Shift	1
Add	1
Normalize	1
Round	1
Store	2
Total	9

   (b) $9 \times 1000 = 9000$ nanoseconds.

   (c) Observe that if we start storing a result at time t, we won't be able to complete the store until time $t+2$. So no matter how fast the other stages of the pipeline operate, we can't produce a result more than once every two nanoseconds. Tracking the progress of data through the pipeline should convince you of this:

Time	F	C	Sh	A	N	R	St
0	1						
1	1						
2	2	1					
3	2		1				
4	3	2		1			
5	3		2		1		
6	4	3		2		1	
7	4		3		2		1
8	5	4		3		2	1
9	5		4		3		2
10	6	5		4		3	2
11	6		5		4		3

Here, "F" is "Fetch," "C" is Compare, etc. The numbers in columns F–St represent which operands/results are being used.

It does take 9 ns for the first result to appear. However, after this, it takes 2 ns before the next result appears, and, more generally, after any result is completed, it will take 2 ns before the next result is completed. So the total time to compute 1000 results will be $9 + 999 \times 2 = 2007ns$.

(d) From the previous part, we see that once the pipeline is full, when a fetch begins, it looks something like this:

Time	F	C	Sh	A	N	R	St
t	n	$n-1$		$n-2$		$n-3$	$n-4$

So if operand n wasn't in Level 1 cache, but it was in Level 2 cache, then the fetch will take 5 nanoseconds. So our pipeline might look something like this:

Time	F	C	Sh	A	N	R	St
t	n	$n-1$		$n-2$		$n-3$	$n-4$
$t+1$	n		$n-1$		$n-2$		$n-3$
$t+2$	n			$n-1$		$n-2$	$n-3$
$t+3$	n				$n-1$		$n-2$
$t+4$	n					$n-1$	$n-2$
$t+5$	$n+1$	n					$n-1$

Of course a Level 2 miss will be even worse: the data will stay in the fetch stage from t to $t+50$. Clearly, there is an opportunity to improve overall performance if there is some useful work for the processor to do while the fetch from main memory is taking place.

2. In a write-through cache, every write to the cache causes a write to main memory. Of course writing to main memory is much slower than processing within the CPU. If we have a queue in the CPU, we can put the contents of the writes in the queue, and the queue can be emptied at the speed of the memory, while the CPU can continue processing. As long as the queue isn't full, the CPU won't need to wait for the writes to main memory to complete.

3. Suppose the matrix has order 8 or 64 elements; the cache line size is still 4; the cache can store 4 lines; and the cache is direct-mapped. Then the following table shows how A is stored in cache lines.

Cache Line	Elements of A			
0	A[0][0]	A[0][1]	A[0][2]	A[0][3]
1	A[0][4]	A[0][5]	A[0][6]	A[0][7]
2	A[1][0]	A[1][1]	A[1][2]	A[1][3]
3	A[1][4]	A[1][5]	A[1][6]	A[1][7]
4	A[2][0]	A[2][1]	A[2][2]	A[2][3]
5	A[2][4]	A[2][5]	A[2][6]	A[2][7]
6	A[3][0]	A[3][1]	A[3][2]	A[3][3]
7	A[3][4]	A[3][5]	A[3][6]	A[3][7]
8	A[4][0]	A[4][1]	A[4][2]	A[4][3]
9	A[4][4]	A[4][5]	A[4][6]	A[4][7]
10	A[5][0]	A[5][1]	A[5][2]	A[5][3]
11	A[5][4]	A[5][5]	A[5][6]	A[5][7]
12	A[6][0]	A[6][1]	A[6][2]	A[6][3]
13	A[6][4]	A[6][5]	A[6][6]	A[6][7]
14	A[7][0]	A[7][1]	A[7][2]	A[7][3]
15	A[7][4]	A[7][5]	A[7][6]	A[7][7]

Assuming that no lines of A are in the cache when the first pair of loops begins, we see that there will be two misses for each row of A. Also, after the first two rows have been read, the cache will be full, and each miss will evict a line. As in the example in the text, an evicted line won't need to be read again. So we see that the total number of misses for the first pair of loops is 16, or

$$\frac{\text{number of elements in A}}{\text{cache line size}}.$$

More generally, then, for the first pair of loops, the number of misses is only affected by the size of A, not the size of the cache.

For the second pair of nested loops, let's also assume that no lines of A are in the cache when the loops begin. When we're working with column 0 ($j = 0$), each time we multiply `A[i][0]*x[0]` we'll need to first load a new cache line, since the previously read lines will only contain elements from rows $< i$. So executing the loop with $j = 0$, will result in 8 misses. After reading in the four lines containing `A[0][0]`, `A[1][0]`, `A[2][0]`, and `A[3][0]`, respectively, subsequent reads of elements of column 0 will evict these four lines. So after completing the multiplications involving column 0 of A, no elements in the first 4 rows of A will be stored in the cache. So every read of an element of A in rows 0–3 of column 1 will result in a miss. In fact, we see that every multiplication will result in a miss and there will be 64 misses.

Observe, however, in the second pair of loops if we have an 8 line cache, then the multiplications involving columns 1–3 of A won't result in misses, and we won't have additional misses until we start the multiplications by elements in column 4. Once the lines containing elements of column 4 are loaded, there won't be any additional misses, and we see that with an 8 line cache, the total number of misses is reduced to 16.

4. $2^{20} = 1,048,576$ pages.

5. The most basic implementations of cache and virtual memory won't change either the number of instructions that can be executed at one time or the amount of data that can be operated on at one time. However, more sophisticated systems can provide some concurrency: when there's a cache miss or a page fault the CPU may attempt to execute instructions not involving the unavailable data or instructions. Such a system might be described as having limited MIMD capabilities: load/store instructions can be executed at the same time as other instructions.

   We can view pipelining as applying one complex instruction applied to multiple data items. Hence it's sometime considered to be SIMD.

   Multiple issue and hardware multithreading attempt to apply possibly different instructions to different data items. So they can be considered to be examples of MIMD.

6. 10 memory banks.

7. A vector processor might divide all three arrays into several blocks of `vector_length` elements, and operate on the first block from each of the three arrays in one stage, then operate on the next block from each of the three arrays in the second stage, and so on. Since different cores of a GPU can operate independently of each other, one core might be used to operate on `x` and `y` while another could be used to simultaneously operate on `z`.

8. If there are many threads, it may be impossible to store the states of all of them in cache, in which case, some it may take a long time to switch from an active thread to

an idle thread. This could be especially costly in a fine-grained system, since active threads might rarely stall because of the large caches.

9. A pipeline in which the stages are full-fledged instructions might be construed as an MISD system: multiple instructions are applied to a single stream of data item as it moves through the pipeline.

10. (a) Since the number of processors is 1000, the number of instructions (other than sending messages) that each processor must execute is $10^12/1000 = 10^9$. Since each processor executes 10^6 instructions per second, the total time each processor will spend executing instructions is $10^9/10^6 = 1000$ seconds.

    Since each processor must send $10^9 \times 999$ messages and it takes 10^{-9} seconds to send a single message, the total time each processor will spend sending messages is 999 seconds.

    So the total time each processor will spend is about 1999 seconds or about 33 minutes.

    (b) The amount of time the processors spend executing instructions is unchanged: 1000 seconds. However, now the total time to send the $10^9 \times 999$ messages is $10^6 \times 999$ seconds. So the total time to run the program will be about

$$10^6 \times 999 + 1000 \text{ seconds} \approx 32 \text{ years!}$$

11. The following table doesn't include links joining nodes to switches. For direct interconnects, the links are bidirectional, while for indirect interconnects, the links are unidirectional.

Interconnect	Nodes	Links
Ring	p	p
Toroidal Mesh	$p = q^2$	$2q^2 = 2p$
Fully Connected	p	$p(p-1)/2$
Hypercube	$p = 2^d$	$d2^{d-1} = p\log_2(p)/2$
Crossbar	p	$2p(p-1)$
Omega Network	$p = 2^d$	$(d-1)2^d = p[\log_2(p) - 1]$

12. (a) Suppose the mesh has $p = q \times q$ nodes, where q is even. Then we can bisect the mesh by simply removing the "middle" horizontal links (as in Figure 2.10). Since there are q of these links, the bisection width is $q = \sqrt{p}$.

    (b) Suppose the mesh has $p = q^3$ nodes, where, once again, q is even. Then we can imagine the three-dimensional mesh as being built by stacking q two-dimensional

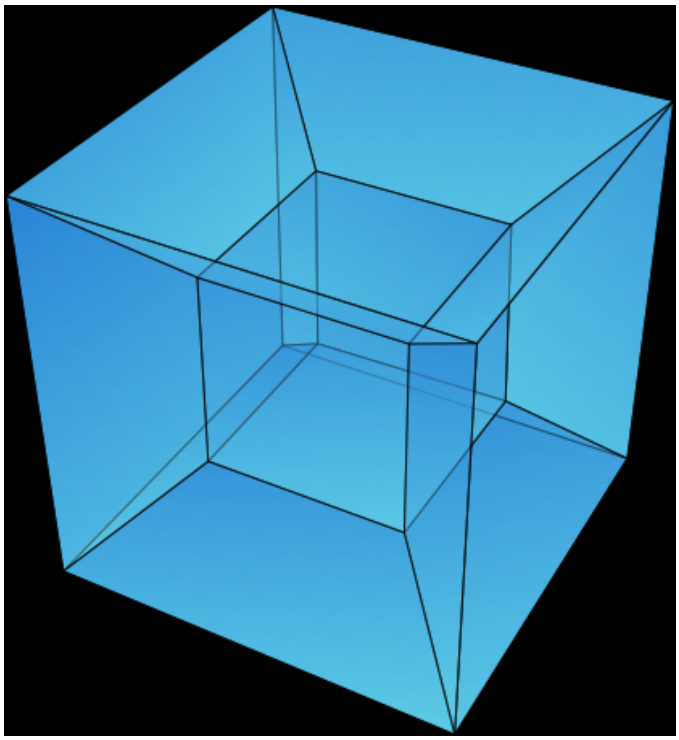

Figure 1: A four-dimensional hypercube

meshes. If we lean the stack of two-dimensional meshes against a wall, each of the two-dimensional meshes is more or less vertical, and we can bisect the three-dimensional mesh by removing by removing the middle horizontal links from each of our two-dimensional meshes. Each of the two-dimensional bisections will remove q links, and we'll have removed a total of q^2 links. So the bisection width of a three-dimensional mesh is $q^2 = p^{2/3}$.

13. (a) See Figure 1[1]. The nodes are located at the vertices or "corners" in the diagram, and links are the edges.

    (b) We can build a $(d+1)$-dimensional hypercube from two d-dimensional hypercubes by joining corresponding nodes with links. So we can bisect a $(d+1)$-dimensional hypercube by cutting each of these new links. Since the d-dimensional hypercube has 2^d nodes, we'll cut 2^d links. But if our $(d+1)$-dimensional hypercube has p nodes, then $p = 2^{d+1}$, and $2^d = p/2$.

[1]This image was downloaded from the Wikipedia article "Hypercube" `http://en.wikipedia.org/wiki/Hypercube` on September 8, 2011.

14. If we cut the outgoing links and the incoming links joining the top $p/2$ nodes to the crossbar, we'll have cut p links, and clearly none of the top $p/2$ nodes can communicate with any node. (See Figure 2.14.) So the bisection width of the 8×8 crossbar is at most 8.

15. (a) Since x is not in core 1's cache, the invalidation that core 0 sends won't have any effect on its cache. Furthermore, since the system uses write-back cache, when core 1 loads the line containing x it may load a line containing the old value of x. So the assignment $y = x$ might assign a value x had before the assignment $x = 5$ was executed.

    (b) In a directory-based system, when core 0 executes the assignment it will invalidate the line containing x in main memory by notifying the directory. The problem here is that core 1 may load the line containing x before the directory has entry has been invalidated.

    (c) The programmer could explicitly synchronize the two cores: core 1 won't attempt to use x until core 0 has notified it that the update has been completed. There are several alternatives that could be used. For example, among the synchronization methods discussed in Chapter 2, either semaphores or busy-waiting could be used.

16. (a) See `ex2.16a_spdup.c`. If we hold n fixed and increase p, the behavior of the speedups and efficiencies depends on the size of n. For $n = 10$, the speedups increase at nearly the rate of increase of p when p is small. But as p increases, the increase in the speedups slows until it actually decreases in going from 64 to 128. The behavior of the efficiencies reflect this: For small p, the efficiencies are close to 1, but as p gets larger, they drop precipitously.

    For $n = 320$, speedups increase by nearly a factor of 2 when we double p — regardless of how large p is — and, as a consequence efficiencies are near 1 for all values of p.

    If we hold p fixed but increase n, then for small p the speedups and efficiencies remain nearly constant. For example, for $p = 2$, the efficiencies are very close to 1 for all values of n. For large p, the speedups and efficiencies increase as n is increased, but the rate of increase starts to diminish as n approaches its maximum of 320.

    (b) Since $T_{\text{parallel}} = T_{\text{serial}}/p + T_{\text{overhead}}$, we can write the efficiency as

$$E = \frac{T_{\text{serial}}}{pT_{\text{parallel}}} = \frac{T_{\text{serial}}}{T_{\text{serial}} + pT_{\text{overhead}}}.$$

If we multiply both numerator and denominator by $1/T_{\text{serial}}$, we'll have

$$E = \frac{1}{1 + p\,T_{\text{overhead}}/T_{\text{serial}}}$$

If T_{overhead} grows more slowly than T_{serial}, then the fraction $T_{\text{overhead}}/T_{\text{serial}}$ will get smaller as n increases. So the denominator in our formula for E will decrease, and E will increase as n increases.

On the other hand, if T_{overhead} grows faster than T_{serial}, then the fraction $T_{\text{overhead}}/T_{\text{serial}}$ will get larger as n increases. This will cause the denominator in our formula for E to increase, and E will decrease as n increases.

17. In a large system, it might be possible for all of the program data to fit into the combined caches of the cores. If this is the case, then the parallel program might incur considerably fewer cache misses than the serial program running on a single core. In this setting, it's possible that the speedup of the parallel program is greater than the number of cores.

18. Essay.

19. The parallel efficiency is

$$E = \frac{n}{n + p\log(p)}.$$

If we increase p by a factor of k, we want to find a factor r so that if increase n by this factor, then E will be unchanged. So we want to solve the equation

$$\frac{n}{n + p\log(p)} = \frac{rn}{rn + kp\log(kp)}$$

for r. After applying various rules of algebra, this gives us

$$r = \frac{k\log(kp)}{\log(p)}.$$

So we see that the program is scalable: if we increase p at a given rate, this formula tells us how much we need to increase n in order to maintain a constant efficiency. For example, suppose $p = 4$, and the problem size is 1024. Then if we double p, that is, $k = 2$, then according to our formula, we should increase n by a factor of

$$r = \frac{2\log(2 \cdot 4)}{\log(4)} = 3.$$

in order to maintain a constant efficiency. Checking in our formula for E, we see that

$$\frac{1024}{1024 + 4\log(4)} = \frac{3 \cdot 1024}{3 \cdot 1024 + 8\log(8)}.$$

20. A program that obtains linear speedup is strongly scalable, because no matter what the number of processes/threads, the efficiency is 1. If the problem size is n and the number of processes/threads is p, then the efficiency is 1. If the problem size remains n, and the number of processes/threads is increased to $q > p$, then the efficiency remains 1.

21. The reported time using `input_data1` is barely equal to the clock resolution: it's entirely possible that the next time Bob tries running the program with `input_data1`, the `time` command will return time 0. So this command is unsuitable for the first set of input.

    The time using `input_data2` is *much* greater than the clock resolution. So it's possible that repeated timings with this data will result in similar times. However, the `time` command includes process startup and shutdown and I/O, which may be large parts of the reported time, but may not be important parts of what Bob is trying to time. So he should still approach the results with caution.

22. (a) $u \leq r$ and $s \leq r$.

    (b) Before starting message-passing code and after completing message-passing code, Bob can call the three functions:

```
u1 = utime(); s1 = stime(); r1 = rtime();
/* Message Passing Code */
. . .
u2 = utime(); s2 = stime(); r2 = rtime();
udiff = u2 - u1;
sdiff = s2 - s1;
rdiff = r2 - r1;
```

    Now Bob can compare `udiff + sdiff` and `rdiff`. If `rdiff` is much greater, then it's possible that the message-passing code is spending a significant amount of time waiting for messages.

    (c) No. Bob's functions will apparently include all of the time spent message-passing (including wait-time) in user-time. So there won't be any way for her to extract time spent waiting from the numbers Bob's functions will give her.

23. One problem with this approach is that we don't know which elements of `data` belong to which bins. For example, if process/thread q is responsible for all increments to `bin_counts[i]`, we have no way of insuring that process/thread q is assigned the elements of `data` that are assigned to `bin_counts[i]`. So when the processes/threads

determine the bins to which the elements of `data` are assigned, every time a process/thread r other than q gets an element that belongs to bin i, there will have to be communication between processes/threads q and r. In the case of shared memory this will require some kind of mutual exclusion lock for each element of `bin_counts`, and this could result in a serious deterioration in performance as the processes/threads compete for access to element of `bin_counts`. In the case of distributed memory, process/thread r will have to send a message to process/thread q, and if the elements of `data` have been poorly distributed, the program could require communications for most of the elements of `data`.

24. See the solutions to Exercises 1.3, 1.4, and 1.5. In shared memory, we could have a shared array of sums with one element for each thread. The remaining variables can be private. In the communication phase, the receiving thread can add the sending thread's sum into its sum. Some sort of producer-consumer synchronization will be necessary between the sending thread and the receiving thread.