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

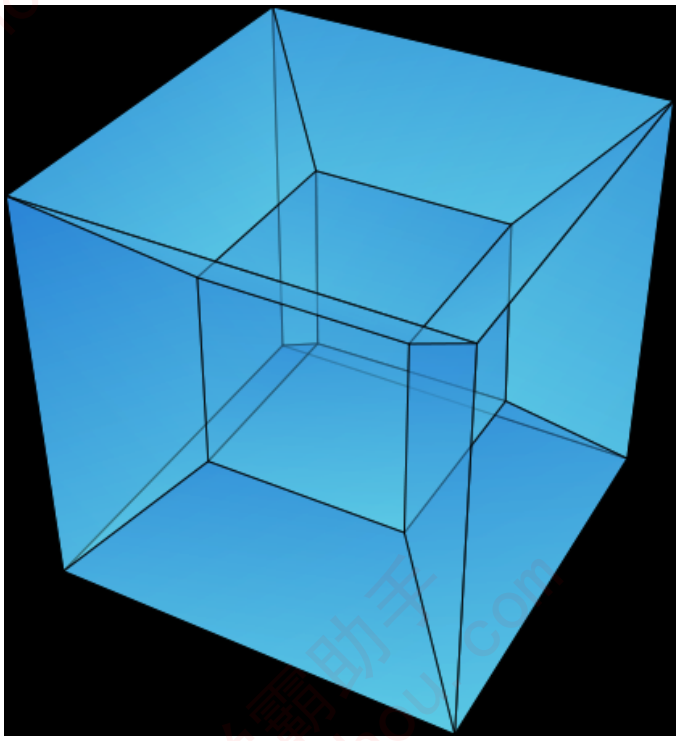

Figure 1: A four-dimensional hypercube

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.

An Introduction to Parallel Programming Solutions, Chapter 1

Jinyoung Choi and Peter Pacheco

July 26, 2011

1. The "+1" is for the '\0(null) character terminating the string. So if we use `strlen(greeting)` instead of `strlen(greeting)+1` the terminating null character won't be transmitted. As with any erroneous program, the exact behavior depends on the implementation. With some implementations the program crashes with a seg fault. With others, the program is apparently correct.

   If we use `MAX_STRING` instead of `strlen(greeting)+1`, the output of the program is identical. However, in this case, we're sending the entire array `greeting` instead of just the text of the greeting (and the terminating null character).

2. If n isn't evenly divisible by `comm_sz`, we won't use the correct number of trapezoids. For example, if $n = 15$ and `comm_sz = 4`, the current version of the program will use $15/4 = 3$ trapezoids on each process, which will mean it will only use a total of $3 \times 4 = 12$ trapezoids. In order to use 15 trapezoids, we need to assign some of the processes 4 trapezoids instead of 3. More precisely $15 \bmod 4 = 3$ processes should get 4 trapezoids, and one should get 3. So we could modify the code so that the first 3 processes get 4 trapezoids, and the last process gets 3.

   One way to do this is to compute the integer quotient and the remainder of n/comm_sz. All the processes will get n/comm_sz trapezoids, while the first n mod comm_sz will get an extra trapezoid:

```
int quotient = n / comm_sz;
int remainder = n % comm_sz;
if (my_rank < remainder) {
   local_n = quotient + 1;
   local_a = a + my_rank*local_n*h;
   local_b = local_a + local_n*h;
} else {
```

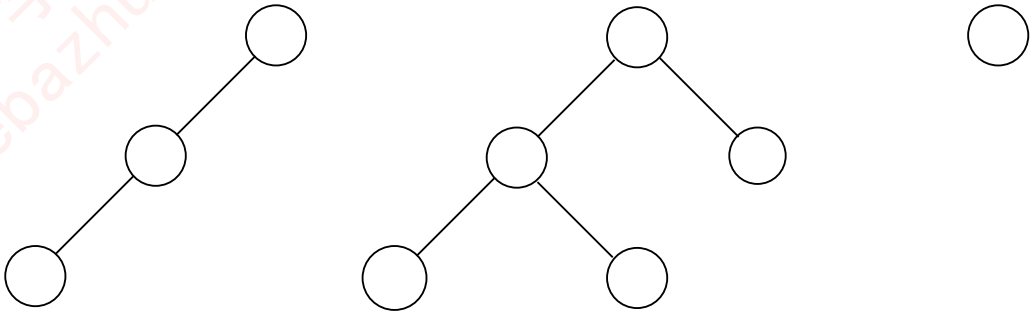

Figure 1: Binary trees

```
   local_n = quotient;
   local_a = a + my_rank*local_n*h + remainder*h;
   local_b = local_a + local_n*h;
}
```

3. The local variables are

   `my_rank, local_n, local_a, local_b, local_int, source.`

   The global variables are

   `comm_sz, n, a, b, h, total_int.`

4. We can send the messages to process 0 for printing. See `ex3.4_mpi_output_inorder.c`

5. We use induction on the number n of leaves. So suppose T is a complete binary tree with $n = 1$ leaf. If T is *any* binary tree with 1 leaf, then T consists of a single path from the root to the leaf. But if T is a complete binary tree, every nonleaf has two children. So if there's more than one node in the tree, there will be more than one leaf. See Figure 1. So the leaf is the root, which has depth 0 and depth $= \log_2(1)$.

   So suppose that if S is a complete binary tree with m leaves, where $1 \leq m < n$, then the depth of the leaves is $\log_2(m)$. Now suppose T is a complete binary tree with $n > 1$ leaves. Consider the parents of the leaves of T. If d is the depth of the leaves, then each parent has depth $d - 1$. So all of the parents have the same depth. Furthermore,

every ancestor of of a parent has two children, since it has two children in T. So the tree S that we obtain by removing all the leaves from T is a complete binary tree, and the leaves of S are the parents of the leaves of T. Since each parent is the parent of two leaves in T, S has $n/2$ leaves, and, by the induction hypothesis, the depth of its leaves is $\log_2(n/2)$. But then, the depth of the leaves of T is

$$1 + \log\left(\frac{n}{2}\right) = 1 + \log(n) - \log(2) = 1 + \log(n) - 1 = \log(n).$$

So by the principle of mathematical induction, in a complete tree with n leaves, each leaf has depth $\log(n)$.

6. (a) We can use the distribution we outlined in 3.1. This would assign components as follows.

$$\begin{aligned}
\text{Process } 0 &: x_0, x_1, x_2, x_3 \\
\text{Process } 1 &: x_4, x_5, x_6, x_7 \\
\text{Process } 2 &: x_8, x_9, x_{10} \\
\text{Process } 3 &: x_{11}, x_{12}, x_{13}
\end{aligned}$$

(b) With a cyclic distribution, we simply assign one component to each successive process — returning to process 0 after assigning to process `comm_sz` — until all the components have been assigned:

$$\begin{aligned}
\text{Process } 0 &: x_0, x_4, x_8, x_{12} \\
\text{Process } 1 &: x_1, x_5, x_9, x_{13} \\
\text{Process } 2 &: x_2, x_6, x_{10} \\
\text{Process } 3 &: x_3, x_7, x_{11}
\end{aligned}$$

(c) With a block-cyclic distribution, we can proceed as we did with the cyclic distribution, except that instead of assigning one component to each successive process, we assign two — the block size:

$$\begin{aligned}
\text{Process } 0 &: x_0, x_1, x_8, x_9 \\
\text{Process } 1 &: x_2, x_3, x_{10}, x_{11} \\
\text{Process } 2 &: x_4, x_5, x_{12}, x_{13} \\
\text{Process } 3 &: x_6, x_7
\end{aligned}$$

Note that the block and block-cyclic distributions are not uniquely determined. For example, in the block-cyclic distribution we might split up the final block(s) among

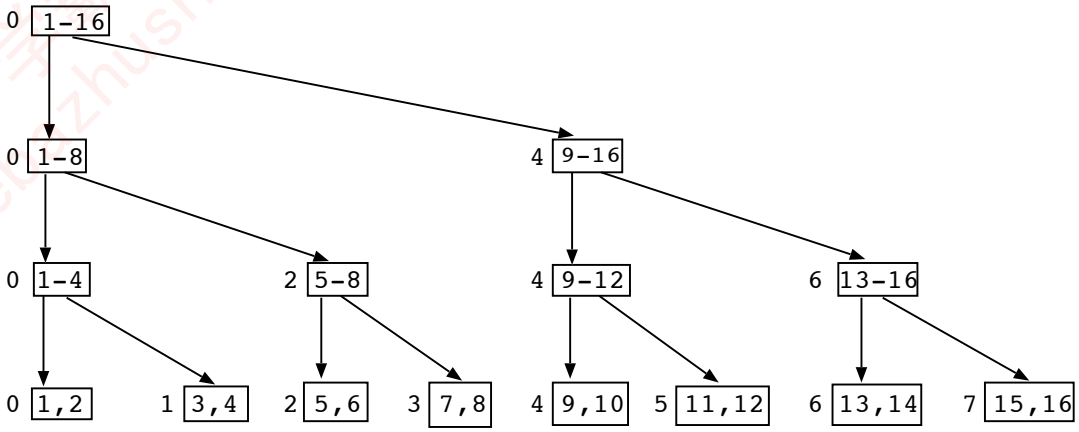

Figure 2: Scatter

the processes. In our example, we might have

$$\begin{aligned} \text{Process } 0 &: x_0, x_1, x_8, x_9 \\ \text{Process } 1 &: x_2, x_3, x_{10}, x_{11} \\ \text{Process } 2 &: x_4, x_5, x_{12} \\ \text{Process } 3 &: x_6, x_7, x_{13} \end{aligned}$$

7. See `ex3.7_mpi_coll_one_proc.c`. When we ran the program `MPI_Bcast` returned with no change to the input buffer. The other collectives simply copied the contents of the input buffer to the output buffer.

8. (a) Figure 2 shows the stages in scattering the integers $1, 2, \ldots, 16$.

   (b) Figure 3 shows the stages in gathering the integers $1, 2, \ldots, 16$ if they're initially distributed using a block distribution.

9. See the source file `ex3.9_mpi_vect_mult.c`.

10. This is OK because both formal arguments corresponding to `local_n` are input argument. The rule prohibiting aliasing only applies if one of the arguments is an output or an input/output argument.

11. (a)
```
prefix_sums[0] = vect[0];
for (i = 0; i < n; i++)
   prefix_sums[i] = prefix_sums[i-1] + vect[i];
```

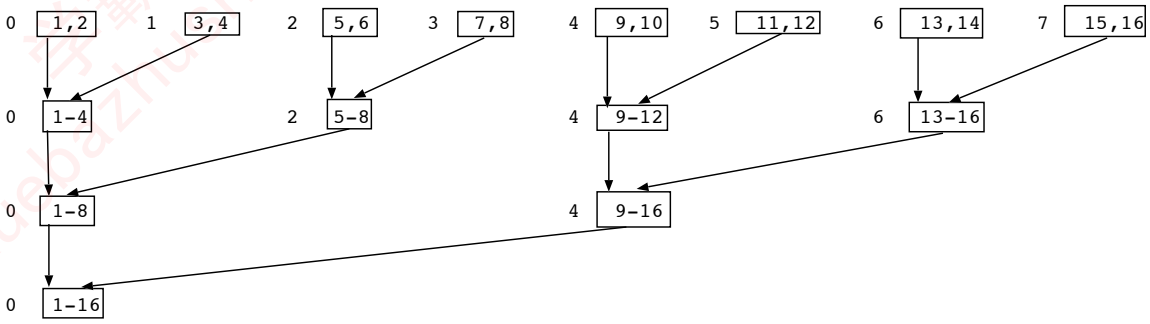

Figure 3: Gather

Also see the source file `ex3.11a_serial_prefix_sums.c`.

(b) The following algorithm should work if n, the order of the vector, is evenly divisible by `comm_sz`.

```
/* First compute prefix sums of my local vector */
loc_prefix_sums[0] = loc_vect[0];
for (loc_i = 1; loc_i < loc_n; loc_i++)
   loc_prefix_sums[loc_i] = loc_prefix_sums[loc_i-1] + loc_vect[loc_i];

if (my_rank != 0) {
   /* If I'm not 0 receive sum of preceding components */
   MPI_Recv(&sum_of_preceding, 1, MPI_DOUBLE, my_rank-1, 0, comm,
         MPI_STATUS_IGNORE);

   /* Add in sum of preceding components to my prefix sums */
   for (loc_i = 0; loc_i < n; loc_i++)
      loc_prefix_sums[loc_i] += sum_of_preceding;
}

/* Now send my last element to the next process */
if (my_rank != comm_sz - 1)
   MPI_Send(&loc_prefix_sums[loc_n-1], 1, MPI_DOUBLE, my_rank+1, 0, comm);
```

See the source file `ex3.11b_mpi_prefix_sums.c`.

(c) The following algorithm should also work if n, the order of the vector, is evenly

divisible by comm_sz

```
/* First compute prefix sums of local data */
loc_prefix_sums[0] = loc_vect[0];
for (loc_i = 1; loc_i < loc_n; loc_i++)
   loc_prefix_sums[loc_i] = loc_prefix_sums[loc_i-1] + loc_vect[loc_i];

/* Now use butterfly structured communication */
sum = loc_prefix_sums[loc_n-1];
mask = 1;
while (mask < comm_sz) {
   partner = my_rank ^ mask;
   MPI_Sendrecv(&sum, 1, MPI_DOUBLE, partner, 0,
         &tmp, 1, MPI_DOUBLE, partner, 0,
         comm, MPI_STATUS_IGNORE);
   sum += tmp;
   if (my_rank > partner)
      for (loc_i = 0; loc_i < loc_n; loc_i++)
         loc_prefix_sums[loc_i] += tmp;
   mask <<= 1;
}
```

See the source file ex3.11c_mpi_prefix_sums.c.

(d) The main thing to beware of is that MPI_SUM doesn't carry out prefix sums on a process' local subvector. So simply using MPI_Scan with MPI_SUM will only work with one element per process. One solution is to use MPI_Scan on the sums of the processes' local elements:

```
Perform a prefix sum  of my elements;
my_sum = the sum of my elements;
Perform MPI_Scan of the processes' my_sum's getting pred_sum's;
Add pred_sum - my_sum to each of my elements;
```

This latter approach is implemented in the source file ex3.11d_mpi_scan.c.

12. (a) See the source file ex3.12a_mpi_ringpass_allreduce.c. The butterfly-structured implementation is much faster than the ring-pass. On one of our systems, with 16 processes, the butterfly is more than 10 times faster than the ring-pass.

(b) See the source file ex3.12b_mpi_ringpass_prefix.c.

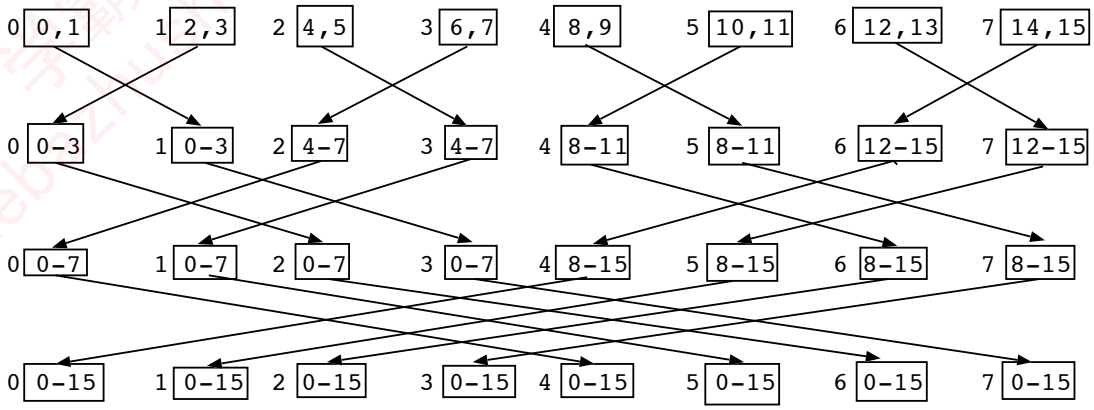

Figure 4: Allgather

13. See the source file `ex3.13_mpi_vect_sum_dot.c`.

14. See the source file `ex3.14_array.c`.

15. The two storage schemes are identical. They both store the elements of an array in the following order in main memory: elements in the first row followed by elements in the second row followed by elements in the third row, etc.

16. See Figure 4.

17. See the source file `ex3.17_mpi_type_contig.c`.

18. See the source file `ex3.18_mpi_type_vect.c`.

19. See the source file `ex3.19_mpi_type_indexed.c`.

20. See `ex3.20_mpi_trap_pack.c`.

21. On the system we used to generate the data in the book, the distributions of the run-times assume *many* different forms. There are combinations of `comm_sz` and matrix order for which the distribution of run-times is unimodal, distributions that are nearly uniform, and distributions that are bimodal. Among the unimodal distributions, there are distributions that are symmetric, distributions that are skewed left, and distributions that are skewed right.

22. It's important to choose the function being integrated and the number of trapezoids so that the elapsed time is a good deal larger than the resolution of the timer. If,

for example, the timer resolution is 1 microsecond, and the actual elapsed time of the trapezoidal rule program is less than 1 microsecond, the time reported by the program could be zero. But even if the actual elapsed time is (say) 2 microseconds, the time reported by the program could be off by as much as 50%.

The resolution of `MPI_Wtime` is returned by `MPI_Wtick`. On the system we used, the resolution is one microsecond.

The following table shows minimum elapsed times on one of our systems. There were ten runs for each problem size and each number of processes. "K" denotes thousands. Times are in milliseconds.

	Number of Trapezoids				
`comm_sz`	50K	100K	150K	200K	250K
1	1.90	3.81	5.70	7.57	9.50
2	1.07	2.09	3.12	4.14	5.18
4	0.58	1.12	1.67	2.21	2.75
8	0.33	0.62	0.92	1.21	1.50
16	0.61	0.75	0.79	0.99	1.11

On this system, the range of elapsed times is larger, relative to the minimum elapsed time, when `comm_sz` is larger and the problem size is smaller.

The following table shows speedups of the trapezoidal rule.

	Number of Trapezoids				
`comm_sz`	50K	100K	150K	200K	250K
1	1.00	1.00	1.00	1.00	1.00
2	1.80	1.82	1.83	1.83	1.84
4	3.31	3.38	3.41	3.43	3.46
8	5.89	6.14	6.21	6.24	6.33
16	3.16	5.08	7.18	7.67	8.57

The following table shows efficiencies of the trapezoidal rule.

	Number of Trapezoids				
`comm_sz`	50K	100K	150K	200K	250K
1	1.00	1.00	1.00	1.00	1.00
2	0.90	0.91	0.91	0.91	0.92
4	0.83	0.85	0.85	0.86	0.86
8	0.74	0.77	0.78	0.78	0.79
16	0.20	0.32	0.45	0.48	0.54

Based on these data, the trapezoidal rule isn't scalable. For example, when we increase from 2 processes to 4 processes, there is no problem size for which the program has an efficiency of 0.9 or better.

23. (a) For the data we collected in the preceding problem, the least-squares estimates for a and b are 3.85×10^{-8} and 1.07×10^{-4}, respectively. With these values the formula in the text predicts the following run-times (in milliseconds).

	Number of Trapezoids				
`comm_sz`	50K	100K	150K	200K	250K
1	1.92	3.85	5.77	7.69	9.62
2	1.07	2.03	2.99	3.95	4.91
4	0.70	1.18	1.66	2.14	2.62
8	0.56	0.80	1.04	1.28	1.52
16	0.55	0.67	0.79	0.91	1.03

(b) The predicted run-times are somewhat accurate. The maximum error in the predicted run-times is 0.27 milliseconds, and the maximum relative error is 0.73. That is, the maximum error in the predicted times is 73% of the actual time. On the other hand, the mean error is 0.092 ms, and the mean relative error is 0.085.

24. Although a zero-length message will contain no data, it will contain "envelope" information such as the tag and communicator. So there shouldn't be any problem with timing a zero-length message. On one of our systems the average time for such a message is about 3.4 microseconds.

25. (a) If the vectors don't fit into cache in the single process run but they do fit into cache in the multiprocess run, it could happen that the multiprocess obtained better than linear speedup, since the average time to add two components might be less: the loads and stores in the multiprocess run might not have to directly access main memory.

(b) The resource limitation in the single process run was the shortage of available cache.

This is the subject of some debate. Here's an example that might be considered superlinear speedup, but doesn't overcome a resource limitation. Suppose we have a serial program that searches a binary tree using depth-first search. If the node we are searching for is near the root, but in the right subtree of the root, the serial program will search the entire left subtree before it finds the desired node. However, if we parallelize the program so that the subtrees of the root are divided among the processes, then, when we run the parallel program with two processes,

process 1 will get the right subtree, and it could find the desired node in much less than half the time it took the serial program.

26. (a) See `ex3.26ab_odd_even.c`.

    (b) See `ex3.26ab_odd_even.c`.

    (c) See `ex3.26c_odd_even.c`. When we ran the program with five hundred random lists each containing 25,000 integers, none were faster with the checks.

27. The following tables show the run-times, speedups and efficiencies of odd-even transposition sort on one of our systems. "M" is millions.

	Run-Times (in seconds)				
	Number of Elements				
comm_sz	1M	2M	4M	8M	16M
1	4.10E-02	8.73E-02	2.22E+00	4.65E+00	9.69E+00
2	2.04E-02	4.32E-02	1.10E+00	2.31E+00	4.81E+00
4	1.10E-02	2.24E-02	5.65E-01	1.18E+00	2.46E+00
8	6.73E-03	1.25E-02	2.98E-01	6.22E-01	1.29E+00
16	5.19E-03	8.45E-03	1.70E-01	3.53E-01	7.31E-01

	Speedups				
	Number of Elements				
comm_sz	1M	2M	4M	8M	16M
1	1.00	1.00	1.00	1.00	1.00
2	2.01	2.02	2.02	2.02	2.02
4	3.73	3.90	3.93	3.94	3.93
8	6.09	6.98	7.46	7.48	7.50
16	7.89	10.34	13.11	13.19	13.26

	Efficiencies				
	Number of Elements				
comm_sz	1M	2M	4M	8M	16M
1	1.00	1.00	1.00	1.00	1.00
2	1.00	1.01	1.01	1.01	1.01
4	0.93	0.97	0.98	0.98	0.98
8	0.76	0.87	0.93	0.93	0.94
16	0.49	0.65	0.82	0.82	0.83

On the basis of the data that were collected, the program is not scalable. For example, with 2 processes, the efficiencies are 1 (or better), while when run with 4 processes the program's maximum efficiency is 0.98.

28. See `ex3.28_mpi_odd_even.c`. The following tables show the run-times, speedups and efficiencies of this version of the sort.

	Run-Times (in seconds)				
	Number of Elements				
comm_sz	1M	2M	4M	8M	16M
1	4.11E-02	8.76E-02	2.22E+00	4.65E+00	9.69E+00
2	2.04E-02	4.32E-02	1.10E+00	2.30E+00	4.80E+00
4	1.10E-02	2.23E-02	5.52E-01	1.16E+00	2.41E+00
8	6.66E-03	1.25E-02	2.87E-01	6.00E-01	1.25E+00
16	5.27E-03	8.45E-03	1.64E-01	3.40E-01	7.06E-01

	Speedups				
	Number of Elements				
comm_sz	1M	2M	4M	8M	16M
1	1.00	1.00	1.00	1.00	1.00
2	2.01	2.03	2.02	2.02	2.02
4	3.75	3.93	4.03	4.02	4.02
8	6.17	6.98	7.74	7.75	7.76
16	7.79	10.37	13.59	13.69	13.72

	Efficiencies				
	Number of Elements				
comm_sz	1M	2M	4M	8M	16M
1	1.00	1.00	1.00	1.00	1.00
2	1.01	1.01	1.01	1.01	1.01
4	0.94	0.98	1.01	1.01	1.01
8	0.77	0.87	0.97	0.97	0.97
16	0.49	0.65	0.85	0.86	0.86

In many cases the run-times are similar to the original version. However, as the problem size increases and the number of processes increases, the run-times of this version become significantly better. This isn't surprising since, as the problem size increases the copies become increasingly expensive, and as the number of processes increases, the number of copies increases.

This version still isn't scalable: there's a marked deterioration in efficiencies when 16 processes are used. However, the efficiencies are somewhat better, and it's conceivable that with larger problems, we might be able to find a rate of increase in the problem size that would make the program scalable.

An Introduction to Parallel Programming Solutions, Chapter 4

Krichaporn Srisupapak and Peter Pacheco

May 22, 2011

1. The code will be unchanged if the number of columns isn't evenly divisible by the number of threads. However, if the number of rows isn't evenly divisible by the number of threads we need to decide who will be responsible for the rows left over after integer division of m, the number of rows, by t, the number of threads.

   For example, if $m = 10$ and $t = 4$, then there are two "extra" rows, since `10 % 4 = 2`. Clearly, there are a number of possibilities here. One is to give the first two threads each an extra row. So we'd assign three rows to threads 0 and 1 and two rows to threads 2 and 3.

   Here's an algorithm that implements this:

```
int quotient = m/t;  /* Every thread gets at least m/t rows */
int remainder = m % t;
if (my_rank < remainder) { /* I get m/t + 1 rows */
   local_m = quotient + 1;
   my_first_row = my_rank*local_m;
   my_last_row = my_first_row + local_m - 1;
} else {  /* I get m/t rows */
   local_m = quotient;
   /* Each of the threads 0, 1, . . . , remainder - 1 gets   */
   /* an extra row.  So add in these rows to get my_firs_row */
   my_first_row = my_rank*local_m + remainder;
   my_last_row = my_first_row + local_m - 1;
}
```

2. See `ex4.2_pth_mat_vect.c`. We scheduled the input and output using semaphores. In most cases this version — with A and y distributed — is somewhat faster.

3. - How does the result of the multithreaded calculation compare to the single-threaded calculation with optimization turned off (`-O0`)?

     The single-threaded and the multithreaded estimates of π are identical to about 14 decimals.

     The ratio of the run-time of the single-threaded program to the multithreaded program is equal to the number of threads, as long as the number of threads is no greater than the number of cores.

   - What happens if you try running the program with optimization `O2`?

     The program hangs if it's run with more than one thread.

   - Which variables should be made volatile in the π calculation?

     `flag` and `sum`

   - Change these variables and rerun the program with and without optimization. How do the results compare to the single-threaded program?

     The results are the same with and without optimization.

4. This isn't entirely clear. It might suggest that threads that are descheduled while they're in calls to `pthread_mutex_lock` are not preventing other threads from accessing the critical section. In other words, threads that are descheduled don't seem to be acquiring the mutex — which would prevent other threads from acquiring it until the descheduled thread had been rescheduled.

5. See `ex4.5_pth_pi_mutex.c` for the modified program.

   The performance of the busy-wait version is roughly comparable to this version. On some systems the busy-wait version is slightly faster. On others it's slightly slower.

   In both versions the threads are probably spending much of the execution time waiting to enter the critical section.

6. See `ex4.6_pth_pi_mutex.c`.

   The two programs have similar run-times.

7. For an implementation that uses two threads see `ex4.7_pth_producer_consumer.c`.

   For an implementation that uses an even number of threads with even threads producing and odd threads consuming, see `ex4.7_pth_pc_odd_even.c`.

   For an implementation in which each threads both produces and consumes see `ex4.7_pth_pc_both.c`.

   These programs do use busy-waiting. Until a message becomes available, the consumer threads will repeatedly acquire the mutex, check for a message, and relinquish the mutex.

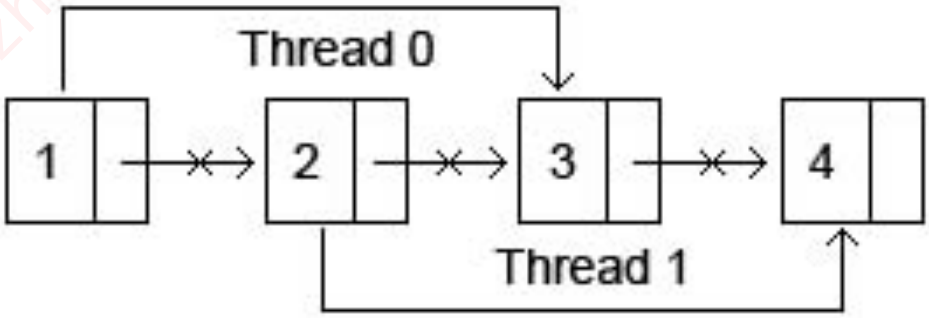

Figure 1: Two deletes that might cause problems

8. (a) The threads will deadlock: thread 0 will be waiting to acquire `mut1` and thread 1 will be waiting to acquire `mut0`, but since thread 0 is waiting for `mut1` it can't relinquish `mut0`, and since thread 1 is waiting to acquire `mut0`, it can't relinquish `mut1`.

   (b) Yes, there would still be a problem. Each thread would be waiting for the other to change a flag variable.

   (c) Yes, there would still be a problem. Both threads would be blocked in calls to `sem_wait`, so neither thread could call `sem_post`.

9. For a program that uses Pthreads barriers see `ex4.9_pth_bar.c`.

   On our systems (AMD and Intel processors running Linux), the barrier implemented with semaphores is always the fastest. The relative speed of the other three depends on the particular systems.

10. The program `ex4.10_pth_trap_time.c` is a modified version of the solution to Programming Assignment 4.3, which implements the trapezoidal rule. The barrier is implemented using a Pthreads barrier, and the maximum elapsed time is found using a mutex.

11. (a) Figure 1 shows two deletes that, when executed simultaneously, might cause problems.

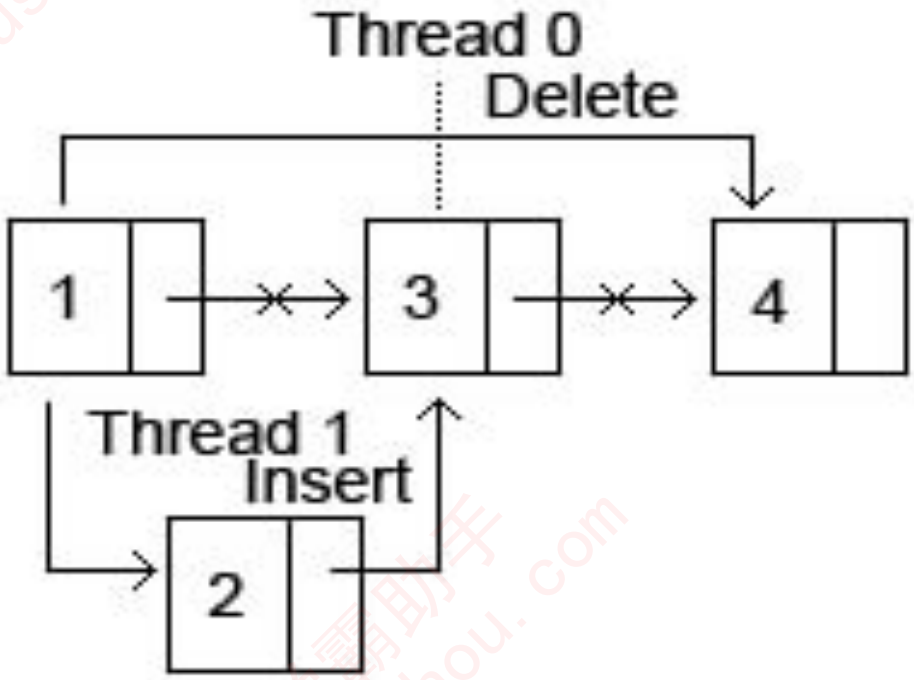

Figure 2: An insert and a delete that might cause problems

(b) Figure 2 shows an insert and a delete that, when executed simultaneously, might cause problems.

(c) If Thread 0 is executing `Member` while Thread 1 is deleting a node, several problems can occur. For example, Thread 0 may return `true` for a value that is no longer in the list because Thread 1 deleted it between the time Thread 0 found it and the time Thread 0 returned. As another example, suppose Thread 0 is visiting a node while Thread 1 is deleting it, and that Thread 0 tries to advance to the next node after Thread 1 has finished the deletion. Then the pointer that Thread 0 dereferences may cause a segmentation violation.

(d) Figure 3 shows two inserts that, when executed simultaneously, might cause problems.

(e) If Thread 0 is executing `Member` while Thread 1 is trying to insert a new node,

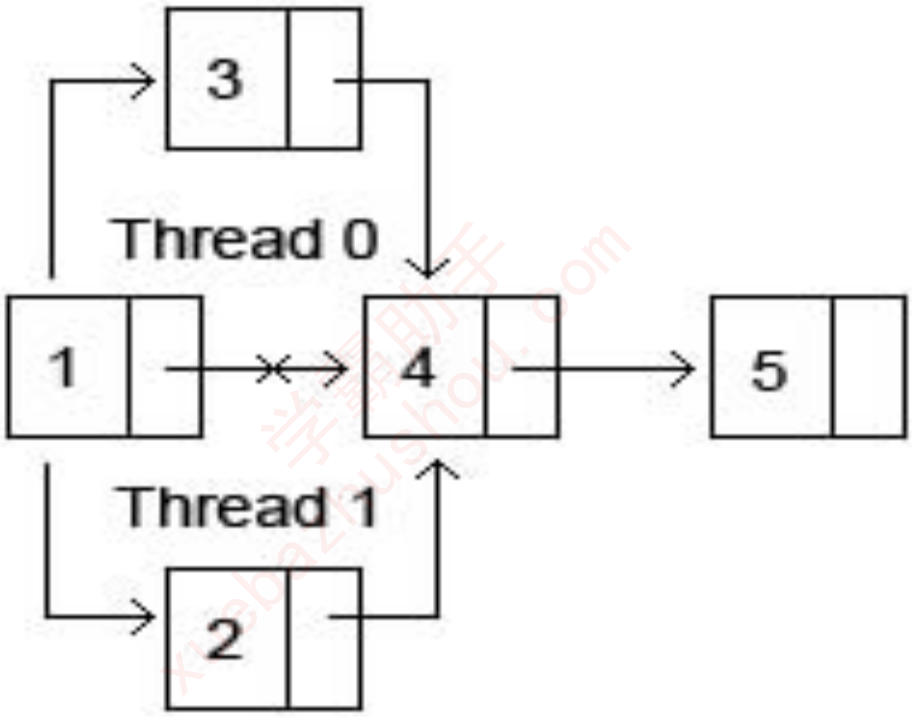

Figure 3: Two inserts that might cause problems

there are several problems that can occur. For example, thread 0 could erroneously return `false`, if thread 1 inserts the searched for value between the time that Thread 0 determines the value isn't in the list and the time it returns.

12. This could be a problem. For example, suppose Thread 0 wants to delete a node from the list. Then it first searches the list with a read-lock. If it finds the node to be deleted, it will need to obtain a write-lock, but in the time that elapses between relinquishing the read-lock and obtaining the write-lock, the state of the list could be changed by another thread. For example, the predecessor of the node to be deleted could be deleted, and the "information" that Thread 0 has about deleting (a pointer to the predecessor node) would no longer be valid.

13. The following table shows the run-times (in seconds) of the three linked-list programs when the initial list has 1000 nodes, and we carry out 100,000 ops. 99.9% of the ops are searches. The times in the "I" column were taken when the remaining 0.1% were insertions, the times in the "D" column were taken when the remaining ops were deletions.

	Number of Threads							
	1		2		4		8	
Implementation	I	D	I	D	I	D	I	D
Read-write locks	0.22	0.21	0.13	0.12	0.099	0.099	0.11	0.11
One mutex	0.22	0.21	0.45	0.42	0.39	0.37	0.46	0.42
One mutex/node	1.8	1.7	6.0	5.6	3.5	3.5	2.7	2.6

The following table shows the run-times of the three linked-list programs when the initial list has 1000 nodes, and we carry out 100,000 ops, 80% of which are searches and 20% are insertions. (We didn't carry out 80% searches and 20% deletions, since there would be about 20,000 deletions and the vast majority of them would be from an empty list.)

	Number of Threads			
Implementation	1	2	4	8
Read-write locks	4.6	8.9	10	10
One mutex	4.8	11	10	11
One mutex/node	35	56	28	19

The run-times with 99.9% searches are roughly comparable to the run-times shown in Table 4.3. When there are 80% searches and 20% insertions, the lists can grow to more than 20 times their original size of 1000. So, not surprisingly, the run-times with 80% searches are *much* greater than the run-times in Table 4.4.

The data with 99.9% searches might seem to indicate that the cost of insertion is slightly greater than the cost of deletion. However, since the average list size for the insertions is slightly greater than the average list size for the deletions, it seems likely that the cost of the two operations is nearly the same.

14. Here's code for the matrix-vector multiplication using a one-dimensional array:

```
void *Pth_mat_vect(void* rank) {
   long my_rank = (long) rank;
   int i, j;
   int local_m = m/thread_count;
   int my_first_row = my_rank*local_m;
   int my_last_row = (my_rank+1)*local_m - 1;

   for (i = my_first_row; i <= my_last_row; i++) {
      y[i] = 0.0;
      for (j = 0; j < n; j++) {
        //y[i] += A[i][j]*x[j];
          y[i] += A[i*n+j]*x[j];
      }
   }

   return NULL;
}  /* Pth_mat_vect */
```

There is little, if any, difference in the run-times of the two versions.

15. **Notes:**

- We've only looked at data cache misses. There are differences in the total number of instructions executed, but the number of instruction cache misses is relatively small and about the same for all three inputs.
- The initialization of A and x will substantially increase the number of misses. It will also make it difficult to determine the state of the cache when the matrix-vector multiplication begins. There are a number of possible solutions to this. We chose the simplest: we omitted initialization and used the default values assigned by the system (0).

We chose $k = 8$. So the orders of the three matrices are 8×8 million, 8000×8000, and 8 million $\times$ 8, respectively. All of the data were taken with one thread. The following table shows the number of *data* cache misses. M = million(s), K = thousand(s). The

numbers in parentheses are percentages of the total number of data-reads or writes. The system on which these data were collected has a 64 Kbyte L1 data-cache and a 1024 Kbyte L2 cache.

Matrix	Data Cache Misses							
	L1-Write		L2-Write		L1-Read		L2-Read	
$8 \times 8M$	16K	(0.0)	65	(0.0)	16M	(12.5)	16M	(12.5)
$8K \times 8K$	2K	(0.0)	1K	(0.0)	16M	(12.5)	8M	(6.2)
$8M \times 8$	1M	(1.4)	1M	(1.4)	8M	(6.2)	8M	(6.2)

c. The largest number of write-misses occur with the $8M \times 8$ system. This makes sense, since for this system the array y has order 8,000,000, while for the other two systems, it has order 8000 and 8, respectively. Furthermore, among the variables A, x, and y, y is the only one that is written by the matrix-vector multiplication code.

f. The largest number of read-misses occur with the $8 \times 8M$ system. This also makes sense. Each element of the array A will be read exactly once with all three inputs, and for each input A has 64,000,000 entries. Also, the updates `y[i] += . . .` will be executed exactly 64,000,000 times for each set of input, and these are unlikely to cause read-misses since before the inner loop is executed, `y[i]` is initialized, and hence is probably already in cache. On the other hand, the reads of `x[j]` may cause cache misses, and in the $8 \times 8M$ system x has 8,000,000 entries as opposed to only 8000 and 8 for the other two systems.

g. From the one-thread run-times given in Table 4.5 on page 192 we see that the program is slowest with the $8 \times 8M$ input and fastest with the $8K \times 8K$ input. Read-misses tend to be more expensive than write-misses. When a program needs data to carry out a computation, it must either try executing another computation or wait for the data. On the other hand, the data created by a write can often be simply queued up and the computation can proceed. So it's not surprising that the program is slowest with the data that results in the most read misses.

On the other hand the program with the $8M \times 8$ input has vastly more write misses than the program with the $8K \times 8K$ data. Furthermore the number of L2 read-misses for the two programs is identical. They do differ in the number of L1 read-misses, but these are substantially less expensive than L2 read-misses. So it's not surprising that the program is fastest with the $8K \times 8K$ input.

16. With 8000 elements `y` will be partitioned as follows

```
Thread 0:     y[0],    y[1], . . . , y[1999]
```

```
Thread 1:  y[2000], y[2001], . . . , y[3999]
Thread 2:  y[4000], y[4001], . . . , y[5999]
Thread 3:  y[6000], y[6001], . . . , y[7999]
```

In order for false-sharing to occur between thread 0 and thread 2, there must be elements of `y` that belong to the same cache line, but are assigned to different threads. On thread 0, the cache line that's "closest" to the elements assigned to thread 2 is the line that contains `y[1999]`. But even if this is the first element of the cache line, the highest possible index for an element of `y` that belongs to this line is 2006:

y[1999]	y[2000]	y[2001]	y[2002]	y[2003]	y[2004]	y[2005]	y[2006]

Since the least index of an element of `y` assigned to thread 2 is 4000, there can't possibly be a cache line that has elements belonging to both thread 0 and thread 2. Similar reasoning applies to threads 0 and 3.

17. If we look at the location of `y[0]` in the first cache line containing all or part of `y` we see that `y` can be distributed across cache lines in eight different ways. If `y[0]` is the first element of the cache line, then we'll have the following assignment of `y` to cache lines:

first line	y[0]	y[1]	y[2]	y[3]	y[4]	y[5]	y[6]	y[7]

If `y[0]` is the second element of the cache line, then we'll have the following assignment:

first line	—	y[0]	y[1]	y[2]	y[3]	y[4]	y[5]	y[6]
second line	y[7]	—	—	—	—	—	—	—

As a final example, if `y[0]` is the last element of the first line, then we'll have the following assignment

first line	—	—	—	—	—	—	—	y[0]
second line	y[1]	y[2]	y[3]	y[4]	y[5]	y[6]	y[7]	—

(a) From our first example, we see it's possible for `y` to fit into a single cache line.

(b) However, in most cases, `y` will be split across two cache lines.

(c) There are eight ways the doubles can be assigned: the eight ways correspond to the eight different possible locations for `y[0]` in the first line.

(d) We can choose two of the threads and assign them to one of the processors: 0 and 1, or 0 and 2, or 0 and 3. Note that this covers all possibilities. For example, choosing 2 and 3 and assigning them to one processor is the same as choosing 0 and 1. So there are three possible assignments of threads to processors.

(e) Yes. Suppose threads 0 and 1 share one processor and threads 2 and 3 share another. Then if `y[0], y[1], y[2],` and `y[3]` are in one cache line and `y[4], y[5], y[6],` and `y[7]` are in another, any write by thread 0 or thread 1 won't invalidate the line storing the data in the cache line of threads 2 and 3. Similarly, writes by 2 and 3 won't invalidate the data in the cache of 0 and 1.

(f) For each of the 3 assignments of threads to processors, there are 8 possible assignments of `y` to cache lines. So we get a total of 24.

(g) Note first that if the execution of the threads is serialized (e.g., first thread 0 runs, then thread 1 runs, etc.), there may not be any false sharing. So we'll assume that all four threads are running simultaneously.

If 0 and 1 are assigned to different processors, then any assignment of `y` that puts `y[1]` and `y[2]` in the same cache line will cause false sharing between 0 and 1. The only way this can fail to happen is if `y[0]` and `y[1]` are in one line and the remainder of `y` is another. But when this happens threads 2 and 3 will be on different processors, and hence there will be false sharing between them.

So 0 and 1 must be assigned to the same processor. Furthermore, if any component of `y` with subscript > 3 is assigned to this processor or if any component with subscript < 4 is assigned to the other processor, there will be false sharing. So the assignment from the previous part is the only one that doesn't result in false sharing.

18. (a) See the file `ex4.18_pth_mat_vec_pad.c`

(b) See the file `ex4.18_pth_mat_vec_private.c`

Note: A third possibility is to simply use a private scalar variable and write to this variable during the inner `for` loop. After the inner for loop is completed, the appropriate element of `y` can be assigned its final value. See file `ex4.18_pth_mat_vec_scalar.c`

(c) The following table shows the run-times (in seconds) of the versions with different input matrices. "Orig" denotes the original implementation; "Pad" is the implementation that pads `y` with extra storage; "Priv Vect" is the implementation that has each thread allocate a private copy of its local part of `y`; and "Priv Scal" is the implementation that has each thread use a private scalar to store the contents of a component of `y` during the computation. In most cases, the private vector implementation performs the worst. No doubt this is due to the costs of

allocating and freeing the temporary storage, and copying the temporary storage into `y`. On the other hand, using a private scalar does as well or better than the other implementations in every case. In particular, it seems to do the best job in avoiding problems with false sharing when the matrix has order $8 \times 8,000,000$.

Threads	Impl	Matrix Order		
		8M × 8	8K × 8K	8 × 8M
1	Orig	0.40	0.36	0.44
	Pad	0.40	0.36	0.44
	Priv Vect	0.49	0.36	0.43
	Priv Scal	0.38	0.33	0.43
2	Orig	0.22	0.19	0.29
	Pad	0.22	0.19	0.26
	Priv Vect	0.28	0.19	0.31
	Priv Scal	0.21	0.18	0.22
4	Orig	0.14	0.12	0.38
	Pad	0.14	0.12	0.24
	Priv Vect	0.19	0.12	0.25
	Priv Scal	0.14	0.12	0.24

19. See the file `ex4.19_pth_tokenize_r.c`

An Introduction to Parallel Programming Solutions, Chapter 5

Krichaporn Srisupapak and Peter Pacheco

June 21, 2011

1. The value of `_OPENMP` is a date having the form *yyyymm,* where *yyyy* is a 4-digit year and *mm* is a 2-digit month. For example, 200505. The OpenMP standard states that when the macro is defined, it will be the year and month of the version of the OpenMP standard that has been implemented. See `ex5.1_omp_macro.c`.

2. The answer here will depend on the system, the number of trapezoids, and the number of threads. On one of our systems, with 1000 trapezoids, we didn't start seeing nondeterministic results until the number of threads was 250, while for another system, we saw nondeterministic results with 100 threads. With only 100 trapezoids, the first system produced unpredictable results with 50 threads, while the second produced unpredictable results with 10. See `ex5.2_omp_trap1_no_crit.c`

3. See `ex5.3_omp_trap1_slow.c` for the version of `omp_trap1.c` that puts the call to `Local_trap` in a critical section. Note that in order to avoid problems with two critical sections (one for the trapezoidal rule result and one for the maximum elapsed time) we simply put the calls to `omp_get_wtime` outside the parallel block.

   When this version is run on one of our systems with 1 thread and $n = 10^6$, the run-time is 2.7 milliseconds, while with 2 threads its run-time is 2.8 milliseconds.

   With the same input both `omp_trap2a_time.c` and `omp_trap2b_time.c` take 2.7 millisecond with 1 thread and 1.4 milliseconds with 2 threads.

   The version that puts the call to `Local_trap` in a critical section forces the threads to execute their parts of the trapezoidal rule sequentially: first one thread executes `Local_trap`, then the other. On the other hand, both of the other two versions allow the threads to do most of their work simultaneously, and they obtain much better performance.

4. The following table shows the identity values for the various operators:

Operator	Identity Value
&&	1
||	0
&	$11\ldots11_2$
|	0
^	0

5. Recall that floating point numbers are stored in the computer in something that's similar to scientific notation. So it's convenient to think of the array `a` as

   ```
   a[] = {2.00e+00, 2.00e+00, 4.00e+00, 1.00e+03}
   ```

   When a value is stored in a register, we can add an additional digit. For example, when `a[0]` is loaded into a register, we can think of it as `2.000e+00`.

   (a) When the values are added using the serial `for` loop, the value stored in `sum` will be

   ```
   After i = 0:  sum = 2.00e+00
   After i = 1:  sum = 4.00e+00
   After i = 2:  sum = 8.00e+00
   ```

   When `i = 3`, the value `1.008e+03` will be stored in a register, and when it's stored in main memory it will be rounded to `1.01e+00`. So the output will be

   ```
   sum = 1010.0
   ```

   (b) When the values are added using the parallel `for`, recall that the run-time system will create a private variable for each thread. This private variable will be used to store that thread's partial sum. Let's call these private variables `local_sum0` on thread 0 and `local_sum1` on thread 1. Then after thread 0 has completed its iterations, `local_sum0 = 4.00e+00`. After thread 1 has completed its iterations `local_sum1 = 1.00e+03` since the register sum `1.004e+03` will be rounded down. Now when the two private variables are added the value stored in the register will be `1.004e+03` and when it's stored in main memory, we'll have `sum = 1.00e+03`. So the output of the code will be

   ```
   sum = 1000.0
   ```

6. See `ex5.6_omp_schedule.c`.

7. When the program is run with one thread, the `parallel for` directive has no effect, and the program is effectively the same as the preceding serial program. In particular, there's no loop-carried dependence, since there's only one thread.

8. Observe that

```
a[0] = 0
a[1] = a[0] + 1 = 0 + 1
a[2] = a[1] + 2 = 0 + 1 + 2
a[3] = a[2] + 3 = 0 + 1 + 2 + 3
a[4] = a[3] + 4 = 0 + 1 + 2 + 3 + 4
```

   etc. In general, then

$$\mathtt{a}[i] = \sum_{j=0}^{i} j.$$

   But

$$\sum_{j=0}^{i} j = \frac{i(i+1)}{2}.$$

   So we can rewrite the code as

```
for (i = 0; i < n; i++)
   a[i] = i*(i+1)/2;
```

   In this loop, the result of any iteration isn't used again. So the code can be parallelized with a `parallel for` directive:

```
#  pragma omp parallel for num_threads(thread_count) \
       default(none) private(i) shared(a, n)
   for (i = 0; i < n; i++)
      a[i] = i*(i+1)/2;
```

9. See the source file `ex5.9_omp_trap3_schedule.c`.

   On our system if no `schedule` clause is used, the default schedule is approximately a block partition. However, the run-time system seems to invariably assign slightly more iterations to thread 0 and slightly fewer to thread `thread_count-1`.

   If the `schedule` clause is included, but the environment variable `OMP_SCHEDULE` isn't defined, then the assignment of iterations to threads seems to be dynamic: the exact assignment changes from run to run.

If the `schedule` clause is included and `OMP_SCHEDULE` is defined to be `guided`, then roughly $n/$`thread_count` iterations are assigned to one thread, and successive blocks of consecutive iterations are roughly half the size of the previous block. In this case also, the exact assignment changes from run to run.

10. See the file `ex5.10_omp_atomic.c`.

    Our implementation doesn't enforce exclusive access across all atomic regions: updates in independent atomic regions can occur simultaneously on different threads. This behavior isn't guaranteed by the standard, however. Also if two different atomic regions update the same variable, then the OpenMP standard requires that a thread executing either update must be given exclusive access to the variable. In our example, if the private variable `my_sum` is replaced by a shared variable `sum`, then only one thread at at time will execute the update to `sum`.

11. The following code will do the job:

```
#  pragma omp parallel for num_thread(thread_count) \
      default(none) private(i, j) shared(A, x, y, m, n)
   for (i = 0; i < m; i++) {
      y[i] = 0.0;
      for (j = 0; j < n; j++)
         // y[i] += A[i][j] * x[j];
            y[i] += A[i*n+j] * x[j];
   }
```

12. **Notes:**

    - Weve only looked at data cache misses. There are differences in the total number of instructions executed, but the number of instruction cache misses is relatively small and about the same for all three inputs.
    - The initialization of A and x will substantially increase the number of misses. It will also make it difficult to determine the state of the cache when the matrix-vector multiplication begins. There are a number of possible solutions to this. We chose the simplest: we omitted initialization and used the default values assigned by the system (0).

    We chose $k = 8$. So the orders of the three matrices are 8 8 million, 8000 8000, and 8 million 8, respectively. All of the data were taken with one thread. The following table shows the number of data cache misses. M = million(s), K = thousand(s). The numbers in parentheses are percentages of the total number of data-reads or writes.

The system on which these data were collected has a 64 Kbyte L1 data-cache and a 1024 Kbyte L2 cache.

Matrix	Data Cache Misses							
	L1-Write		L2-Write		L1-Read		L2-Read	
$8 \times 8M$	16K	(0.0)	570	(0.0)	16M	(12.5)	16M	(12.4)
$8K \times 8K$	4K	(0.0)	2K	(0.0)	16M	(12.5)	8M	(6.2)
$8M \times 8$	1M	(1.4)	1M	(1.3)	8M	(5.2)	8M	(5.2)

c. The largest number of write-misses occur with the $8M \times 8$ system. This makes sense, since for this system the array y has order 8,000,000, while for the other two systems, it has order 8000 and 8, respectively. Furthermore, among the variables A, x, and y, y is the only one that is written by the matrix-vector multiplication code.

f. The largest number of read-misses occur with the $8 \times 8M$ ystem. This also makes sense. Each element of the array A will be read exactly once with all three inputs, and for each input A has 64,000,000 entries. Also, the updates `y[i] += . . .` will be executed exactly 64,000,000 times for each set of input, and these are unlikely to cause read-misses since before the inner loop is executed, `y[i]` is initialized, and hence is probably already in cache. On the other hand, the reads of `x[j]` may cause cache misses, and in the $8 \times 8M$ system x has 8,000,000 entries as opposed to only 8000 and 8 for the other two systems.

g. The table on page 253 has the following run-times (in seconds) when the program is run with one thread on one of our systems:

Threads	$8 \times 8M$	$8K \times 8K$	$8M \times 8$
1	0.33	0.26	0.32

So we see that the program is slowest (although not by much) with the $8 \times 8M$ input and fastest with the $8K \times 8K$ input. Read-misses tend to be more expensive than write-misses. When a program needs data to carry out a computation, it must either try executing another computation or wait for the data. On the other hand, the data created by a write can often be simply queued up and the computation can proceed. So its not surprising that the program is slowest with the data that results in the most read misses.

On the other hand the program with the $8M \times 8$ input has vastly more write misses than the program with the $8K \times 8K$ data. Furthermore the number of L2 read-misses for the two programs is identical. They do differ in the number of L1 read-misses, but these are substantially less expensive than L2 read-misses. So its not surprising that the program is fastest with the $8K \times 8K$ input.

13. With 8000 elements y will be partitioned (approximately) as follows

```
Thread 0: y[0],    y[1],    . . . , y[1999]
Thread 1: y[2000], y[2001], . . . , y[3999]
Thread 2: y[4000], y[4001], . . . , y[5999]
Thread 3: y[6000], y[6001], . . . , y[7999]
```

In order for false-sharing to occur between thread 0 and thread 2, there must be elements of y that belong to the same cache line, but are assigned to different threads. On thread 0, the cache line thats "closest" to the elements assigned to thread 2 is the line that contains `y[1999]`. But even if this is the first element of the cache line, the highest possible index for an element of y that belongs to this line is 2006:

y[1999]	y[2000]	y[2001]	y[2002]	y[2003]	y[2004]	y[2005]	y[2006]

Since the least index of an element of y assigned to thread 2 is 4000, there cant possibly be a cache line that has elements belonging to both thread 0 and thread 2. Similar reasoning applies to threads 0 and 3.

14. If we look at the location of `y[0]` in the first cache line containing all or part of `y` we see that `y` can be distributed across cache lines in eight different ways. If `y[0]` is the first element of the cache line, then we'll have the following assignment of `y` to cache lines:

first line	y[0]	y[1]	y[2]	y[3]	y[4]	y[5]	y[6]	y[7]

If `y[0]` is the second element of the cache line, then we'll have the following assignment:

first line	—	y[0]	y[1]	y[2]	y[3]	y[4]	y[5]	y[6]
second line	y[7]	—	—	—	—	—	—	—

As a final example, if `y[0]` is the last element of the first line, then we'll have the following assignment

first line	—	—	—	—	—	—	—	y[0]
second line	y[1]	y[2]	y[3]	y[4]	y[5]	y[6]	y[7]	—

(a) From our first example, we see it's possible for `y` to fit into a single cache line.

(b) However, in most cases, `y` will be split across two cache lines.

(c) There are eight ways the doubles can be assigned: the eight ways correspond to the eight different possible locations for `y[0]` in the first line.

(d) We can choose two of the threads and assign them to one of the processors: 0 and 1, or 0 and 2, or 0 and 3. Note that this covers all possibilities. For example, choosing 2 and 3 and assigning them to one processor is the same as choosing 0 and 1. So there are three possible assignments of threads to processors.

(e) Yes. Suppose threads 0 and 1 share one processor and threads 2 and 3 share another. Then if `y[0], y[1], y[2],` and `y[3]` are in one cache line and `y[4], y[5], y[6],` and `y[7]` are in another, any write by thread 0 or thread 1 won't invalidate the line storing the data in the cache line of threads 2 and 3. Similarly, writes by 2 and 3 won't invalidate the data in the cache of 0 and 1.

(f) For each of the 3 assignments of threads to processors, there are 8 possible assignments of `y` to cache lines. So we get a total of 24.

(g) Note first that if the execution of the threads is serialized (e.g., first thread 0 runs, then thread 1 runs, etc.), there may not be any false sharing. So we'll assume that all four threads are running simultaneously.

If 0 and 1 are assigned to different processors, then any assignment of `y` that puts `y[1]` and `y[2]` in the same cache line will cause false sharing between 0 and 1. The only way this can fail to happen is if `y[0]` and `y[1]` are in one line and the remainder of `y` is another. But when this happens threads 2 and 3 will be on different processors, and hence there will be false sharing between them.

So 0 and 1 must be assigned to the same processor. Furthermore, if any component of `y` with subscript > 3 is assigned to this processor or if any component with subscript < 4 is assigned to the other processor, there will be false sharing. So the assignment from the previous part is the only one that doesn't result in false sharing.

15. (a) See the file `ex5.15_omp_mat_vect_pad.c`.

(b) See the file `ex5.15_omp_mat_vect_private.c`. Also see the file `ex5.15_omp_mat_vect_scalar.c` for an implementation that uses a temporary private scalar instead of a subvector.

(c) The following table shows the run-times (in seconds) of the versions with different input matrices. "Orig" denotes the original implementation; "Pad" is the implementation that pads `y` with extra storage; "Priv Vect" is the implementation that has each thread allocate a private copy of its local part of `y;` and "Priv Scal" is the implementation that has each thread use a private scalar to store the contents of a component of `y` during the computation. In most cases, the private vector

implementation performs the worst. No doubt this is due to the costs of allocating and freeing the temporary storage, and copying the temporary storage into `y`. On the other hand, using a private scalar does as well or better than the other implementations in almost every case. In particular, it seems to do the best job in avoiding problems with false sharing when the matrix has order $8 \times 8M$.

Threads	Impl	Matrix Order		
		8M × 8	8K × 8K	8 × 8M
1	Orig	0.32	0.26	0.33
	Pad	0.33	0.26	0.33
	Priv Vect	0.42	0.26	0.33
	Priv Scal	0.30	0.25	0.32
2	Orig	0.22	0.19	0.30
	Pad	0.22	0.19	0.26
	Priv Vect	0.28	0.19	0.26
	Priv Scal	0.21	0.18	0.26
4	Orig	0.14	0.12	0.30
	Pad	0.14	0.12	0.23
	Priv Vect	0.18	0.12	0.25
	Priv Scal	0.14	0.12	0.24

16. See the file `ex5.16_omp_tokenize_r.c`.

An Introduction to Parallel Programming Solutions, Chapter 6

Lemuel Mullett and Peter Pacheco

January 17, 2012

1. Recollect that our algorithm computes

```
for each timestep t {
   for each particle i
      compute f(i), the force on i
   for each particle i
      update position and velocity of i using F = ma
   if (output step) Output new positions and velocities
}
```

   The first inner loop uses the current values in positions to find the forces at the current time. The second inner loop updates the positions and velocities using the updated forces. If we put all three computations in a single inner loop, we'd be computing forces using some original positions and some updated positions. So it is necessary to separate the two loops.

2. After compiling the program with optimization and `NO_OUTPUT`, we ran it with 1000 time steps, a stepsize of 0.01, and program-generated initial conditions. With particle counts ranging from 500 to 8000 we got the following run-times:

Particles	500	1000	2000	4000	8000
Run-time (secs)	6.74	27.9	108	431	1720

   We see that doubling the number of particles increases the run-time by roughly a factor of 4. If we continue to double the number of particles and multiply the run-times by four, we see that somewhere between 32,000 and 64,000 particles will result in a run-time of 24 hours (or 86,400 seconds).

   If we infer that the run-times are quadratic in the number of particles and use least-squares to estimate the coefficients, we get that the run-time is approximately

```
2.68e-5*n*n + 4.36e-4*n + 1.18e-1
```

Setting this equal to 86,400 and solving gives approximately 56,770. That is, running the program with 56,770 particles will result in a run-time of approximately 24 hours. Of course, this will depend on the particular system on which the program is run.

3. We used OpenMP since it's relatively easy to modify the serial version. Both inner `for` loops were parallelized with the default schedule. The serial and the parallel programs were compiled with full optimization and `NO_OUTPUT` defined. When we ran the programs with 500 particles for 100 timesteps with a stepsize of 0.01 using the program-generated initial conditions we got the following run-times (in seconds):

Serial	2 threads	4 threads
0.343	2.32	2.06

Interestingly, with cyclic schedules for both loops, the 2-thread run-time was much better (1.50 seconds), but the 4-thread run-time was much worse (2.64 seconds).

See the source code `ex6.3_omp_nbody_red`.

Every time a thread updates the `forces` array, it must obtain access to the critical section, while the updates to the particles' positions and velocities are embarrassingly parallel. So it appears that the contention for access to the critical section is so bad that it is causing the overall run-time to get worse by more than a factor of 4.

4. This program was compiled and run with the same settings and input as the program in the preceding problem.

Serial	2 threads	4 threads
0.343	0.702	0.616

In this case using cyclic schedules performed worse, regardless of the number of number of threads: 0.935 seconds with 2 threads and 1.08 seconds with 4 threads.

It appears that the added cost of a thread's having to acquire and release a lock each time a particle's force is updated is much more than the reduction in cost obtained by having a thread work with a subset of the particles.

5. The change is OK because in the first phase of the calculation of the forces (i.e., when `Compute_force` is called), a thread will only compute forces for particles assigned to threads whose rank is greater than or equal to theirs. For example, if there are $n = 6$ particles and `thread_count` $= 3$ threads, then the following table shows entries in the `loc_forces` array that are zero and nonzero (X):

Thread	Particle					
	0	1	2	3	4	5
0	X	X	X	X	X	X
1	0	0	X	X	X	X
2	0	0	0	0	X	X

The second phase of the forces calculation sums the "columns" of the `loc_forces` array. Thread 0 sums the entries in the first two columns, thread 1 sums the entries in the next two, and thread 2 sums those in the last two. So we see that each thread can exit the sum loop after adding the entries in its columns with "row number" less than or equal to its rank.

Note that if, during the first phase, the particles have a cyclic partition, then this reduction in the second phase isn't possible. In our example, if we use a cyclic partition, then the following table shows the zero and nonzero entries in `loc_forces` after the first phase.

Thread	Particle					
	0	1	2	3	4	5
0	X	X	X	X	X	X
1	0	X	X	X	X	X
2	0	0	X	X	X	X

When we compile the program with full optimization and `NO_OUTPUT`, and then run it with 1000 particles for 1000 timesteps, with a stepsize of 0.01 and program generated initial conditions, we get the following run-times (in seconds):

Threads	Cyclic	Default	To `my_rank`
Serial	24.2	24.2	24.2
2	12.2	18.2	18.2
4	6.2	10.6	10.6
8	3.1	5.8	5.7

The column labelled "Cyclic" contains run-times for the version in which the first loop in the computation of the forces has a cyclic schedule. The remaining loops have the default schedule. The column labelled "Default" contains run-times for the version in which all of the loops have the default schedule, and the inner loop runs to `thread_count`. The third column contains run-times for a version which is the same as the second except that the inner loop runs to `my_rank`.

For these data on this system, there is virtually no difference between the times in the second and third columns, and it would appear that the second pair of nested loops makes very little contribution to the overall run-time.

6. Modifying the two "for each particle" loops with `nowait` clauses *could* cause problems. Without the implied barrier at the end of the second loop

```
#     pragma omp for
      for (part = 0; part < n; part++)
         Update_part(part, forces, curr, n, delta_t);
```

   a thread could call `Output_state` and print positions and velocities for particles that hadn't yet had their positions and velocities updated. For example, if there are two threads and four particles, thread 1 could call `Output_state` before thread 0 had updated the position and velocity of particle 1.

   However, if there is an implied barrier after the second loop and `nowaits` modifying the first loop and the `single` directive, then no thread can use or print invalid data.

7. We compiled the program with full optimization, and ran it with 1024 particles for 1000 timesteps. We used a stepsize of 0.01 and program generated initial conditions. We tried `schedule(static,n)` for $n = 1, 2, 4, 8, \ldots 256$. Each time the program was run with four threads. Here are the run-times (in seconds):

Chunksize	1	2	4	8	16	32	64	128	256
Run-time	3.70	3.70	3.72	3.76	3.84	4.02	4.36	5.04	6.40

   So it appears that for this problem on this system, when the program is run with 4 threads, the chunksize of 1 is optimal.

8. See the source file `ex6.8_omp_daxpy.c`. The following table shows run-times (in milliseconds) on one of our systems after the program was compiled with full optimization.

	Vector Order					
	10^6		10^7		10^8	
Threads	Cyclic	Block	Cyclic	Block	Cyclic	Block
2	34.0	5.9	340	57	3300	540
4	26.0	5.7	220	56	2300	320

   Clearly the block partitioning is far superior to the cyclic. This is probably due to cache. With block partitioning the threads can work independently of each other: if

thread A accesses cache line L, it's very unlikely that any other thread will also access cache line L. On the other hand, with the cyclic partition each thread will need to access every line of both `x` and `y`. This will result in a tremendous amount of false-sharing for `y`.

9. On the system we used for timings the block distribution consistently outperformed the cyclic distribution:

Processes	Local Array Size			
	100		1000	
	Block	Cyclic	Block	Cyclic
2	1.6e-5	2.7e-5	3.4e-5	1.2e-4
4	3.1e-5	5.8e-5	1.1e-4	4.0e-4
8	8.0e-5	1.9e-4	3.7e-4	1.5e-3
16	2.2e-4	5.8e-4	1.6e-3	5.7e-3

See the file `ex6.9_mpi_allgather.c`. The program was compiled with full optimization. Times are in seconds and are averages over 100 iterations.

The results aren't surprising: we expect the cyclic distribution to involve some additional work. For example, a process might receive the data from another process as a contiguous block, and the MPI implementation would then have to copy the contiguous block into noncontiguous locations in memory.

10. We'll also assume that

    - No lines from `x` or `y` are in cache when the code begins execution.
    - The only misses that do occur are the result of accesses to `x` or `y`. In other words, except for `x` and `y`, all needed data and instructions are in cache when the code begins executing.

    Thread 0 will execute the body of the first `for` loop when $i = 0, 1, \ldots 31$. So this will result in four write misses for `x` and four write misses for `y`. Thread 1 will execute the body of the first `for` loop when $i = 32, 33, \ldots 63$. So it will also have four misses for `x` and four misses for `y`.

    The execution of the second loop depends on `chunksize`:

    (a) If `chunksize = n/thread_count = 32`, then the thread 0 will execute the body of the loop for $i = 0, 1, \ldots 31$. So the elements of both `x` and `y` that it needs should already be cached. Similar reasoning applies to thread 1. So when `chunksize` is the same as it was for the first loop, the second loop should result in no further misses.

(b) If `chunksize = 8`, then the threads will execute the body of the second `for` loop as follows:

Thread	Iterations (values of i)			
0	$0, \ldots, 7$	$16, \ldots, 23$	$32, \ldots, 39$	$48, \ldots, 55$
1	$8, \ldots, 15$	$24, \ldots, 31$	$40, \ldots 47$	$56, \ldots, 63$

The elements of `x` and `y` that are accessed in the first two chunks are already cached on thread 0. However, the elements that are accessed in the last two chunks are not. So there will be two additional misses for `x` and two additional misses for `y`. Similar reasoning applies to thread 1. already cached

11. On the system we used for timings `MPI_Allgather` with `MPI_IN_PLACE` consistently outperformed `MPI_Allgather` using separate send and receive buffers:

	Local Array Size			
	100		1000	
Processes	Two bufs	In place	Two bufs	In place
1	2.0e-5	6.0e-6	4.5e-5	6.0e-6
2	6.3e-4	6.1e-4	1.4e-3	1.3e-3
4	2.5e-3	1.5e-3	4.4e-3	3.4e-3
8	3.7e-3	2.3e-3	9.5e-3	8.3e-3
16	6.9e-3	5.1e-3	2.0e-2	1.8e-2

See the file `ex6.11_mpi_in_place.c`. The program was compiled with full optimization. Times are in seconds and are total times to execute 100 calls to `MPI_Allgather`.

The results with one process aren't surprising: when there's only one process and `MPI_IN_PLACE` is used, the call to `MPI_Allgather` can simply return: the data to be sent is already in the receive buffer. On the other hand, the implementation that uses separate buffers will need to copy from the send buffer to the receive buffer.

The other results can be explained — at least in part — using similar reasoning: with `MPI_IN_PLACE` each process is spared the trouble of copying the contents of its send buffer into its receive buffer.

12. (a) We compiled both versions with `NO_OUTPUT` and full optimization. Both versions were run with 500 particles for 500 timesteps with a stepsize of 0.01 and program-generated initial conditions. The following table shows run-times in seconds.

Processes	Original	Separate `loc_pos`
1	3.37	3.37
2	1.70	1.70
4	0.86	0.86
8	0.44	0.44
16	0.23	0.23

There is virtually no difference between the run-times. Since the memory requirements of the original version are somewhat smaller, it is probably preferred.

(b) With a separate array for the masses of the local particles, it's necessary to execute an allgather before computing the forces, since the total force on each particle depends on the masses of all the other particles. Hence in a system with constant masses (e.g., a Newtonian system) the use of separate local storage for the masses of the local particles serves no purpose.

Once again we compiled both versions with `NO_OUTPUT` and full optimization. Both versions were run with 500 particles for 500 timesteps with a stepsize of 0.01 and program-generated initial conditions. The following table shows run-times in seconds.

Processes	Original	With `loc_masses`
1	3.37	3.37
2	1.70	1.70
4	0.86	0.87
8	0.44	0.45
16	0.23	0.26

For small numbers of processes the run-times are virtually identical. However, for larger numbers of processes the cost of the extra communication begins to degrade the overall run-time.

13. See Figure 1.

14. See `ex6.14_mpi_nbody_red.c`

15. The following table shows the correspondence between global indexes and local indexes if $p =$ `comm_sz` $= 4$, $n = 12$, and the program is using a block distribution.

Process	Global Indexes	Local Indexes
0	0, 1, 2	0, 1, 2
1	3, 4, 5	0, 1, 2
2	6, 7, 8	0, 1, 2
3	9, 10, 11	0, 1, 2

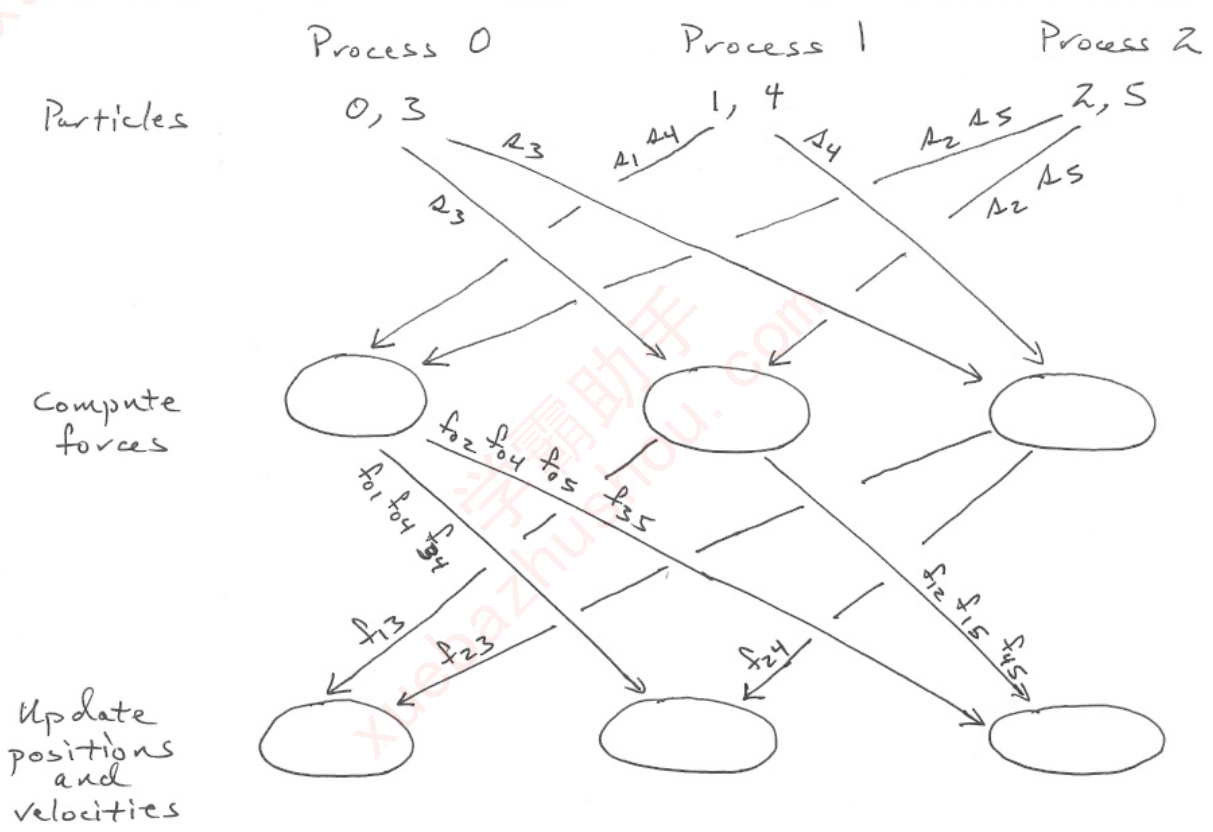

Figure 1: Communications in an "obvious" MPI implementation of the reduced n-body solver

(a) So we see that if `local_n = n/p`, then

```
global_i = local_i + my_rank*local_n
```

(b) We also see that

```
local_i = global_i % local_n
```

The following table shows the correspondence when the program is using a cyclic distribution.

Process	Global Indexes	Local Indexes
0	0, 4, 8	0, 1, 2
1	1, 5, 9	0, 1, 2
2	2, 6, 10	0, 1, 2
3	3, 7, 11	0, 1, 2

(c) So if a program is using a cyclic distribution, then

```
global_i = my_rank + local_i*p
```

(d) We also have that

```
local_i = global_i/p
```

16. The following table shows the global indexes if $p =$ `comm_sz` $= 4$, $n = 12$, and the program is using a cyclic distribution.

Process	Global Indexes		
0	0	4	8
1	1	5	9
2	2	6	10
3	3	7	11

So, for example,

```
First_index(5, 1, 3, 4) = 7
First_index(1, 1, 0, 4) = 4
First_index(7, 3, 1, 4) = 9
```

If the rank of the first process `rk1` is less than the rank of the second `rk2`, then the next index can be computed by simply moving down the appropriate column in the table. So if the global index of the first particle is `gbl1`, `First_index` should return

```
gbl1 + (rk2 - rk1)
```

in this case.

If the rank of the first process is greater than the rank of the second, then the next index is in the column following the column containing `gbl1`. Since there are p entries in each column, we can get to the next column by adding p to `gbl1`. For example, when `gbl1` is 7, $7 + 4 = 11$ is in the next column, but since `rk1` $\geq$ `rk2`, we need to go *up* the new column `rk1 - rk2` slots. So in this case the function should return

```
gbl1 + (rk2 - rk1) + p
```

So the function could be coded as follows.

```
int First_index(int gbl1, int rk1, int rk2, int p) {
   if (rk1 < rk2)
      return gbl1 + (rk2 - rk1);
   else
      return gbl1 + (rk2 - rk1) + p;
```

17. (a) Note that Figure 6.10 shows an edge joining the partial tour $0 \to 1 \to 2$ to the complete tour $0 \to 1 \to 2 \to 3 \to 0$. However, if we examine the code, we'll see that there is an intermediate partial tour, $0 \to 1 \to 2 \to 3$, and it's only after this partial tour is popped off the stack that we stop pushing partial tours, and call the `Best_tour` function. So the stack will contain the maximum number of records when the following partial tours have been pushed:

$$0 \to 1 \to 2 \to 3,\ 0 \to 1 \to 3,\ 0 \to 2,\ 0 \to 3$$

So in this case we see that there is a maximum of 4 records on the stack.

(b) In an attempt to make the diagram more readable, we've omitted the last level of the tree and the arrows joining successive vertices in the partial tours. We've also rotated the tree counter-clockwise by 90 degrees. See Figure 2.

(c) In the five city problem, if we add in the partial tour $0 \to 1 \to 2 \to 3 \to 4$, we see that the stack will contain the maximum number of records when the following partial tours have been pushed:

$$0 \to 1 \to 2 \to 3 \to 4,\ 0 \to 1 \to 2 \to 4,\ 0 \to 1 \to 3,\ 0 \to 1 \to 4,\ 0 \to 2,\ 0 \to 3,\ 0 \to 4$$

So in this case there is a maximum of 7 records on the stack.

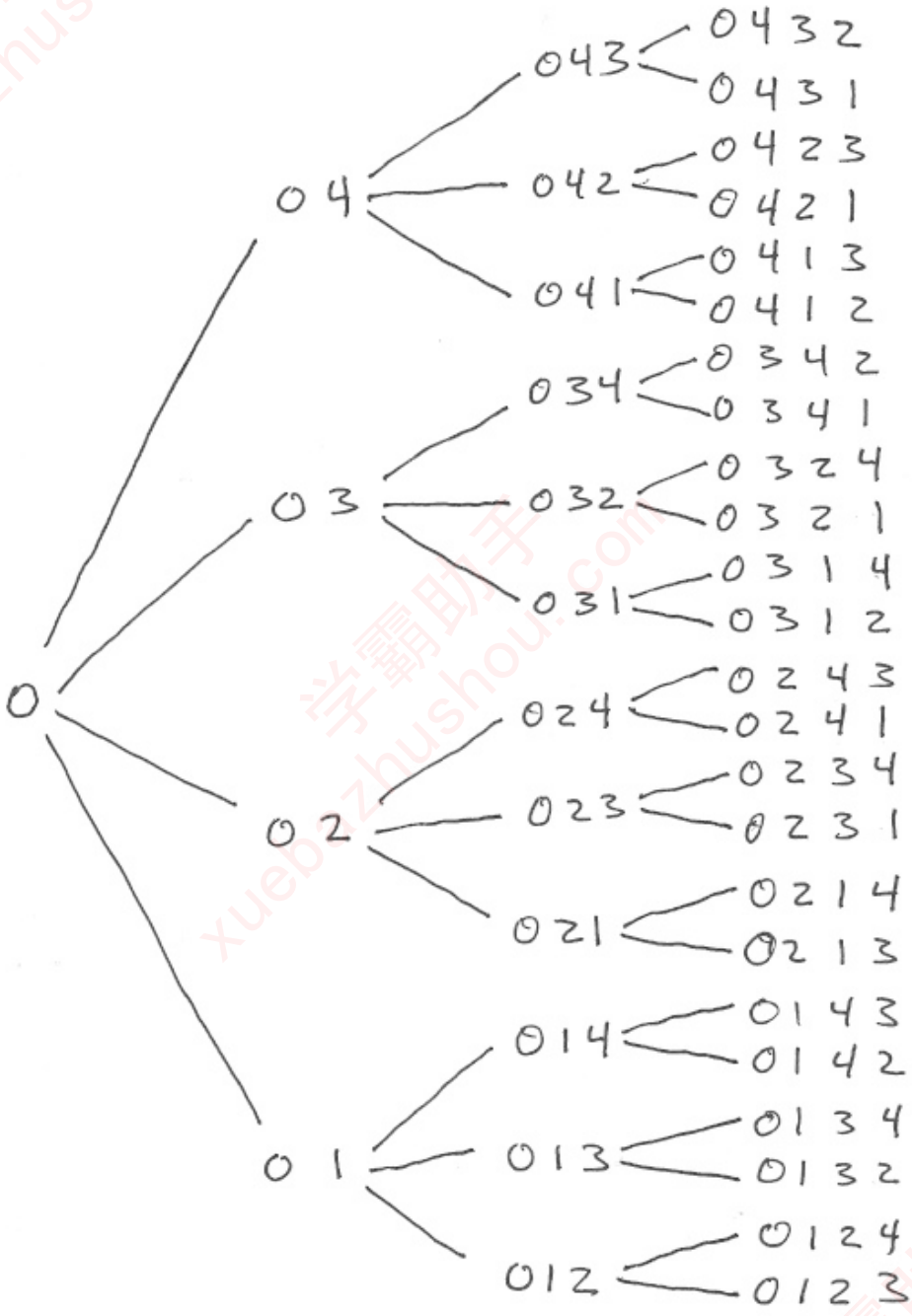

Figure 2: A partial search tree for a five-city TSP

(d) In general, then, we see that there will be a maximum of

$$1 + 1 + 2 + \cdots + n - 2 = 1 + \sum_{k=1}^{n-2} k = 1 + \frac{(n-1)(n-2)}{2}$$

records on the stack.

18. The queue will need to simultaneously store all of the partial tours at a given level in the tree, and the number of tours at a given level will grow very rapidly. If level one contains the 1-city tour, level two contains the 2-city tours, etc., we see that for an n-city problem, level k, $1 < k < n$, will contain

$$(n - 2 + 1)(n - 3 + 1) \cdots (n - k + 1)$$

k-city partial tours, and, this number can be huge. For example, if $n = 15$, and $k = 14$, the product will be $14! = 87,178,291,200$ or about 90 billion.

(a) We can write a function `queue_sz(queue)` which returns the number of partial tours in the queue, and instead of continuing to iterate until the queue is empty, we can iterate until the queue has at least `thread_count` tours:

```
. . .
while (queue_sz(queue) < thread_count) {
   tour = Dequeue(queue);
   for each neighboring city
      if (Feasible(tour, city)) {
         Add_city(tour, city);
         Enqueue(tour);
         Remove_last_city(tour);
      }
   Free_tour(tour);
}
```

(b) As we move from the head to the tail of the queue, the tours will have increasing (really, nondecreasing) length. So it probably makes sense to use a cyclic decomposition of the queue:

```
for (q_elt = my_rank; q_elt < queue_sz(queue); q_elt += thread_count) {
   tour = Link_to(queue, q_elt);
   Push(stack, tour);
}
```

```
Barrier();
if (my_rank == 0) Free_queue(queue);
```

The details will depend on the implementation of the queue and the stack. For example, if `Enqueue` enqueues a copy of its argument, then `Link_to` can simply return a pointer to a tour, and `Push` can push this tour onto the stack — without copying. The `Barrier()` is needed to make sure that the queue's supporting structures aren't freed before every thread has finished acquiring its initial tours.

19. See `ex6.19_pth_tsp_stat.c`. We ran the program with two 15-city problems. The following table show elapsed times in seconds for the original Pthreads version and the version that uses read-write locks:

Threads	First Problem		Second Problem	
	Orig	RWL	Orig	RWL
1	36	55	212	312
2	30	390	99	689
4	28	655	37	1002
8	25	901	32	114

In a word, the performance of the version that uses read-write locks is disastrous. From the one-thread times, we can see that the version that uses read-write locks adds about 50% to the run-times just because of calls to the lock and unlock functions. Except for the 8-thread run of the second problem, the performance of the version that uses read-write locks only gets worse as more threads are added. Presumably, this is due to contention for the lock, in spite of the fact that no run acquired the write lock more than 17 times total.

20. (a) When a new partial tour is pushed onto the stack, it either has the same length as the partial tour that was previously on the top of the stack or it has one more city. So the $k/2$ partial tours on the top of the stack will probably be longer than the $k/2$ tours on the bottom of the stack. Since a shorter tour is the root of a larger subtree than a longer tour, we expect that the $k/2$ tours on the bottom of the stack will require more work than the $k/2$ on the top of the stack, and this strategy will probably do a poor job of load-balancing.

    (b) This strategy seems better than the strategy in the preceding part. However, this approach doesn't directly address the problem discussed in the preceding part: short partial tours are the roots of larger subtrees. For example, if a stack contains an m-city partial tour with cost $c + 1$ and a $2m$-city partial tour with cost c, we wouldn't be surprised if the m-city tour led to considerably more work.

(c) This strategy seems to address the problem in the preceding part. When two partial tours have roughly the same total cost, the shorter tour will have a higher average cost per edge, and we might expect it to require less work. On the other hand, when two partial tours have roughly the same number of edges, then the partial tour with lower overall cost and hence lower average cost per edge is more likely to require more work.

See `ex6.20_pth_tsp_dyn.c`. Table 1 shows the results of using the different splitting strategies in the Pthreads implementation when it's run with two different fifteen city problems. The strategies are:

(0) Alternate tours are assigned to the donating thread and the receiving thread — the strategy outlined in the text.

(1) Strategy (a) above.

(2) Strategy (b) above.

(3) Strategy (c) above.

The run-times are in seconds. The numbers in the "Splits" column are the total number of times the stack was split by all threads. It should be noted that because of nondeterminism in the `Terminated` function, there is considerable variability in the number of splits, and, for the 8 and 16 threads, there can be considerable variability in the overall run-time.

Not surprisingly strategy (a) invariably requires considerably more splits than the other strategies. Presumably the thread that gets the top half will, in most cases, soon run out of work again. The remaining strategies result in virtually identical run-times when there are fewer than 8 threads. When there are 8 or 16 threads, the average cost strategy tends to require fewer splits, but the original, alternating tour, strategy generally gives the best run-times.

21. (a) See `ex6.21_mpi_tsp_stat.c`. This implementation will be much slower than the original static implementation if one process quickly finds the best tour and its other tours can be quickly eliminated, while another process has to search a large subtree of relatively expensive tours. For example, suppose there are two processes. Also suppose that process 0 finds the best tour by always branching left and the other tours assigned to process 0 have an expensive edge going from 0 to the first nonzero city. Also suppose process 1 has a subtree in which all the tours are more expensive than the best tour, but all the edges have roughly the same cost. Then process 0 will finish almost immediately, but in the modified program process 1 will have to search a large subtree. Furthermore, if the same

Threads	Strat	First Problem		Second Problem	
		Run-time	Splits	Run-time	Splits
1	0	41.0	0	240.0	0
	1	41.0	0	240.0	0
	2	41.0	0	240.0	0
	3	41.0	0	240.0	0
2	0	34.0	10	110.0	8
	1	34.0	793	110.0	775
	2	34.0	7	110.0	8
	3	34.0	10	110.0	9
4	0	30.0	32	41.0	40
	1	30.0	4385	41.0	2774
	2	30.0	36	41.0	48
	3	30.0	44	41.0	52
8	0	27.0	171	3.3	141
	1	36.0	10,053	3.6	7616
	2	35.0	231	3.3	120
	3	28.0	141	3.3	185
16	0	5.1	484	2.0	375
	1	6.2	22,225	2.3	13,626
	2	4.2	548	2.0	399
	3	7.6	329	2.0	321

Table 1: Run-times (in seconds) and total numbers of stack splits using different stack-splitting strategies

problem is solved by a single process, it will find the solution just as quickly as process 0.

If, on the other hand, the best tour is found by always branching right, and there are many updates to the best tour, then this implementation may be better than the original static implementation, since it won't incur the cost of all the broadcasts.

(b) The performance of the original static implementation in this setting depends on the amount of work done by process 0. If it quickly finds the best tour, and subsequent tours can be eliminated when only a small part of the tour is examined, the original static implementation should be much faster than the modified implementation. On the other hand, if process 0 has to search extensively in its subtree, then the performance of the two implementations should be relatively close.

The following table shows the performance of the two static implementations and the dynamic implementation for two different input problems. Both problems have 17 cities. The cost of the edges $0 \to x$ is 99 for $x = 5, 6, \ldots, 16$. The cost of all other edges is 3, except both problems have a tour with cost 17 in process 0's subtree. For the first problem, the tour is $0 \to 1 \to 2 \to \cdots \to 16 \to 0$. For the second problem the tour is $0 \to 4 \to 1 \to 2 \to 3 \to 5 \to 6 \to 7 \to \cdots \to 16 \to 0$. So in the first problem process 0 will find the best tour very quickly, while for the second problem it will need to search for some time before it finds the best tour:

Problem	Original Static	Modified Static	Dynamic
First	4.9	46	5.7×10^{-3}
Second	35	53	5.6

Times are in seconds. The run-times in the last column were taken with minimum stack size for a split of two and maximum tour size for reassignment of seventeen.