6 Solutions

6.1 There is no single right answer for this question. The purpose is to get students to think about parallelism present in their daily lives. The answer should have at least 10 activities identified.

6.1.1 Any reasonable answer is correct here.

6.1.2 Any reasonable answer is correct here.

6.1.3 Any reasonable answer is correct here.

6.1.4 The student is asked to quantify the savings due to parallelism. The answer should consider the amount of overlap provided through parallelism and should be less than or equal to (if no parallelism was possible) to the original time computed if each activity was carried out serially.

6.2

6.2.1 For this set of resources, we can pipeline the preparation. We assume that we do not have to reheat the oven for each cake.

Preheat Oven

Mix ingredients in bowl for Cake 1

Fill cake pan with contents of bowl and bake Cake 1. Mix ingredients for Cake 2 in bowl.

Finish baking Cake 1. Empty cake pan. Fill cake pan with bowl contents for Cake 2 and bake Cake 2. Mix ingredients in bowl for Cake 3.

Finish baking Cake 2. Empty cake pan. Fill cake pan with bowl contents for Cake 3 and bake Cake 3.

Finish baking Cake 3. Empty cake pan.

6.2.2 Now we have 3 bowls, 3 cake pans and 3 mixers. We will name them A, B, and C.

Preheat Oven

Mix incredients in bowl A for Cake 1

Fill cake pan A with contents of bowl A and bake for Cake 1. Mix ingredients for

Cake 2 in bowl A.

Finish baking Cake 1. Empty cake pan A. Fill cake pan A with contents of bowl A for Cake 2. Mix ingredients in bowl A for Cake 3.

Finishing baking Cake 2. Empty cake pan A. Fill cake pan A with contents of bowl A for Cake 3.

Finish baking Cake 3. Empty cake pan A.

The point here is that we cannot carry out any of these items in parallel because we either have one person doing the work, or we have limited capacity in our oven.

6.2.3 Each step can be done in parallel for each cake. The time to bake 1 cake, 2 cakes or 3 cakes is exactly the same.

6.2.4 The loop computation is equivalent to the steps involved to make one cake. Given that we have multiple processors (or ovens and cooks), we can execute instructions (or cook multiple cakes) in parallel. The instructions in the loop (or cooking steps) may have some dependencies on prior instructions (or cooking steps) in the loop body (cooking a single cake).

Data-level parallelism occurs when loop iterations are independent (i.e., no loop carried dependencies).

Task-level parallelism includes any instructions that can be computed on parallel execution units, are similar to the independent operations involved in making multiple cakes.

6.3

6.3.1 While binary search has very good serial performance, it is difficult to parallelize without modifying the code. So part A asks to compute the speedup factor, but increasing X beyond 2 or 3 should have no benefits. While we can perform the comparison of low and high on one core, the computation for mid on a second core, and the comparison for A[mid] on a third core, without some restructuring or speculative execution, we will not obtain any speedup. The answer should include a graph, showing that no speedup is obtained after the values of 1, 2, or 3 (this value depends somewhat on the assumption made) for Y.

6.3.2 In this question, we suggest that we can increase the number of cores (to each the number of array elements). Again, given the current code, we really cannot obtain any benefit from these extra cores. But if we create threads to compare the N elements to the value X and perform these in parallel, then we can get ideal speedup (Y times speedup), and the comparison can be completed in the amount of time to perform a single comparison.

6.4. This problem illustrates that some computations can be done in parallel if serial code is restructured. But more importantly, we may want to provide for SIMD operations in our ISA, and allow for data-level parallelism when performing the same operation on multiple data items.

6.4.1 This is a straightforward computation. The first instruction is executed once, and the loop body is executed 998 times.

Version 1—17,965 cycles

Version 2—22,955 cycles

Version 3—20,959 cycles

6.4.2 Array elements D[j] and D[j−1] will have loop carried dependencies. These will `$f4` in the current iteration and `$f0` in the next iteration.

6.4.3 This is a very challenging problem and there are many possible implementations for the solution. The preferred solution will try to utilize the two nodes by unrolling the loop 4 times (this already gives you a substantial speedup by eliminating many loop increment, branch and load instructions). The loop body running on node 1 would look something like this (the code is not the most efficient code sequence):

```
addiu $s1, $zero, 996
l.d $f0, -16($s0)
l.d $f2, -8($s0)
loop:
add.d $f4, $f2, $f0
add.d $f6, $f4, $f2
Send (2, $f4)
Send (2, $f6)
s.d $f4, 0($s0)
s.d $f6, 8($s0)
Receive($f8)
add.d $f10, $f8, $f6
add.d $f0, $f10, $f8
Send (2, $f10)
Send (2, $f0)
s.d. $f8, 16($s0)
s.d $f10, 24($s0)
s.d $f0 32($s0)
Receive($f2)
s.d $f2 40($s0)
addiu $s0, $s0, 48
bne $s0, $s1, loop
add.d $f4, $f2, $f0
add.d $f6, $f4, $f2
add.d $f10, $f8, $f6
s.d $f4, 0($s0)
s.d $f6, 8($s0)
s.d $f8, 16($s0)
```

The code on node 2 would look something like this:

```
 addiu $s2, $zero, 0
 loop:
Receive ($f12)
Receive ($f14)
add.d $f16, $f14, $f12
Send(1, $f16)
Receive ($f12)
Receive ($f14)
add.d $f16, $f14, $f12
Send(1, $f16)
Receive ($f12)
Receive ($f14)
add.d $f16, $f14, $f12
Send(1, $f16)
Receive ($f12)
Receive ($f14)
add.d $f16, $f14, $f12
Send(1, $f16)
addiu $s2, $s2, 1
bne $s2, 83, loop
```

Basically Node 1 would compute 4 adds each loop iteration, and Node 2 would compute 4 adds. The loop takes 1463 cycles, which is much better than close to 18K. But the unrolled loop would run faster given the current send instruction latency.

6.4.4 The loop network would need to respond within a single cycle to obtain a speedup. This illustrates why using distributed message passing is difficult when loops contain loop-carried dependencies.

6.5

6.5.1 This problem is again a divide and conquer problem, but utilizes recursion to produce a very compact piece of code. In part A the student is asked to compute the speedup when the number of cores is small. When forming the lists, we spawn a thread for the computation of left in the MergeSort code, and spawn a thread for the computation of the right. If we consider this recursively, for m initial elements in the array, we can utilize $1 + 2 + 4 + 8 + 16 + \ldots. \log_2 (m)$ processors to obtain speedup.

6.5.2 In this question, $\log_2 (m)$ is the largest value of Y for which we can obtain any speedup without restructuring. But if we had m cores, we could perform sorting using a very different algorithm. For instance, if we have greater than m/2 cores, we can compare all pairs of data elements, swap the elements if the left element is greater than the right element, and then repeat this step m times. So this is one possible answer for the question. It is known as parallel comparison sort. Various comparison sort algorithms include odd-even sort and cocktail sort.

6.6

6.6.1 This problem presents an "embarrassingly parallel" computation and asks the student to find the speedup obtained on a 4-core system. The computations involved are: $(m \times p \times n)$ multiplications and $(m \times p \times (n - 1))$ additions. The multiplications and additions associated with a single element in C are dependent (we cannot start summing up the results of the multiplications for an element until two products are available). So in this question, the speedup should be very close to 4.

6.6.2 This question asks about how speedup is affected due to cache misses caused by the 4 cores all working on different matrix elements that map to the same cache line. Each update would incur the cost of a cache miss, and so will reduce the speedup obtained by a factor of 3 times the cost of servicing a cache miss.

6.6.3 In this question, we are asked how to fix this problem. The easiest way to solve the false sharing problem is to compute the elements in C by traversing the matrix across columns instead of rows (i.e., using index-j instead of index-i). These elements will be mapped to different cache lines. Then we just need to make sure we process the matrix index that is computed (i, j) and (i + 1, j) on the same core. This will eliminate false sharing.

6.7

6.7.1 $x = 2, y = 2, w = 1, z = 0$

$x = 2, y = 2, w = 3, z = 0$

$x = 2, y = 2, w = 5, z = 0$

$x = 2, y = 2, w = 1, z = 2$

$x = 2, y = 2, w = 3, z = 2$

$x = 2, y = 2, w = 5, z = 2$

$x = 2, y = 2, w = 1, z = 4$

$x = 2, y = 2, w = 3, z = 4$

$x = 3, y = 2, w = 5, z = 4$

6.7.2 We could set synchronization instructions after each operation so that all cores see the same value on all nodes.

6.8

6.8.1 If every philosopher simultaneously picks up the left fork, then there will be no right fork to pick up. This will lead to starvation.

6.8.2 The basic solution is that whenever a philosopher wants to eat, she checks both forks. If they are free, then she eats. Otherwise, she waits until a neighbor contacts her. Whenever a philosopher finishes eating, she checks to see if her neighbors want to eat and are waiting. If so, then she releases the fork to one of them and lets them eat. The difficulty is to first be able to obtain both forks without another philosopher interrupting the transition between checking and acquisition. We can implement this a number of ways, but a simple way is to accept requests for forks in a centralized queue, and give out forks based on the priority defined by being closest to the head of the queue. This provides both deadlock prevention and fairness.

6.8.3 There are a number or right answers here, but basically showing a case where the request of the head of the queue does not have the closest forks available, though there are forks available for other philosophers.

6.8.4 By periodically repeating the request, the request will move to the head of the queue. This only partially solves the problem unless you can guarantee that all philosophers eat for exactly the same amount of time, and can use this time to schedule the issuance of the repeated request.

6.9

6.9.1

Core 1	Core 2
A3	B1, B4
A1, A2	B1, B4
A1, A4	B2
A1	B3

6.9.2

FU1	FU2
A1	A2
A1	
A1	
B1	B2
B1	
A3	
A4	
B2	
B4	

6.9.3

FU1	FU2
A1	B1
A1	B1
A1	B2
A2	B3
A3	B4
A4	

6.10 This is an open-ended question.

6.11

6.11.1 The answer should include a MIPS program that includes 4 different processes that will compute ¼ of the sums. Assuming that memory latency is not an issue, the program should get linear speed when run on the 4 processors (there is no communication necessary between threads). If memory is being considered in the answer, then the array blocking should consider preserving spatial locality so that false sharing is not created.

6.11.2 Since this program is highly data parallel and there are no data dependencies, a 8× speedup should be observed. In terms of instructions, the SIMD machine should have fewer instructions (though this will depend upon the SIMD extensions).

6.12 This is an open-ended question that could have many possible answers. The key is that the student learns about MISD and compares it to an SIMD machine.

6.13 This is an open-ended question that could have many answers. The key is that the students learn about warps.

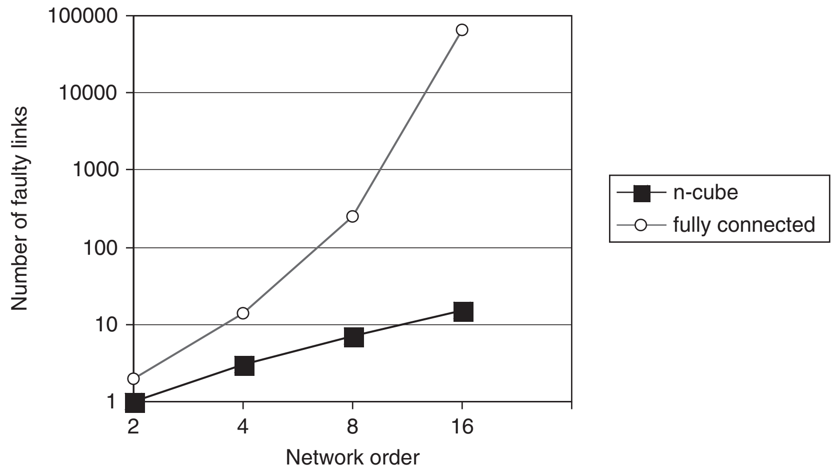
6.14 This is an open-ended programming assignment. The code should be tested for correctness.

6.15 This question will require the students to research on the Internet both the AMD Fusion architecture and the Intel QuickPath technology. The key is that students become aware of these technologies. The actual bandwidth and latency values should be available right off the company websites, and will change as the technology evolves.

6.16

6.16.1 For an n-cube of order N (2^N nodes), the interconnection network can sustain $N-1$ broken links and still guarantee that there is a path to all nodes in the network.

6.16.2 The plot below shows the number of network links that can fail and still guarantee that the network is not disconnected.



6.17

6.17.1 Major differences between these suites include:
Whetstone—designed for floating point performance specifically
PARSEC—these workloads are focused on multithreaded programs

6.17.2 Only the PARSEC benchmarks should be impacted by sharing and synchronization. This should not be a factor in Whetstone.

6.18

6.18.1 Any reasonable C program that performs the transformation should be accepted.

6.18.2 The storage space should be equal to (R + R) times the size of a single precision floating point number + (m + 1) times the size of the index, where R is the number of non-zero elements and m is the number of rows. We will assume each floating-point number is 4 bytes, and each index is a short unsigned integer that is 2 bytes. For Matrix *X* this equals 111 bytes.

6.18.3 The answer should include results for both a brute-force and a computation using the Yale Sparse Matrix Format.

6.18.4 There are a number of more efficient formats, but their impact should be marginal for the small matrices used in this problem.

6.19

6.19.1 This question presents three different CPU models to consider when executing the following code:

```
if (X[i][j] > Y[i][j])
    count++;
```

6.19.2 There are a number of acceptable answers here, but they should consider the capabilities of each CPU and also its frequency. What follows is one possible answer:

Since X and Y are FP numbers, we should utilize the vector processor (CPU C) to issue 2 loads, 8 matrix elements in parallel from A and 8 matrix elements from B, into a single vector register and then perform a vector subtract. We would then issue 2 vector stores to put the result in memory.

Since the vector processor does not have comparison instructions, we would have CPU A perform 2 parallel conditional jumps based on floating point registers. We would increment two counts based on the conditional compare. Finally, we could just add the two counts for the entire matrix. We would not need to use core B.

6.19.3 The point of the problem is to show that it is difficult to perform an operation on individual vector elements when utilizing a vector processor. What might be a nice instruction to add would be a vector comparison that would allow for us to compare two vectors and produce a scalar value of the number of elements where one vector was larger the other. This would reduce the computation to a single instruction for the comparison of 8 FP number pairs, and then an integer computation for summing up all of these values.

6.20 This question looks at the amount of queuing that is occurring in the system given a maximum transaction processing rate, and the latency observed on average by a transaction. The latency includes both the service time (which is computed by the maximum rate) and the queue time.

6.20.1 So for a max transaction processing rate of 5000/sec, and we have 4 cores contributing, we would see an average latency of .8 ms if there was no queuing taking place. Thus, each core must have 1.25 transactions either executing or in some amount of completion on average.

So the answers are:

Latency	Max TP rate	Avg. # requests per core
1 ms	5000/sec	1.25
2 ms	5000/sec	2.5
1 ms	10,000/sec	2.5
2 ms	10,000/sec	5

6.20.2 We should be able to double the maximum transaction rate by doubling the number of cores.

6.20.3 The reason this does not happen is due to memory contention on the shared memory system.