Solutions to Questions.

1. (a) The execution of these statements has no concurrent operations and thus will run sequentially. The final values will be: \( X = -15, Y = -11 \).

   (b) These operations are to be run concurrently meaning that any one of them can execute in any order. Given the initial values, the final values are shown below:
   
   IF the execution follows the sequence: \( S1 \rightarrow S2 \rightarrow S3 \), then the final values are as shown in the above problem. \( X = -15, Y = -11 \).
   
   IF the execution follows the sequence: \( S3 \rightarrow S2 \rightarrow S1 \), then the final values are: \( X = 7Y = 3 \).
   
   For all other iterations (\( 3! = 6 \) total), my testing had all the following identical results of: \( X = 3Y = -4 \).

   (c) For this execution, both \( S1 \) and \( S2 \) are bounded by an await for \( X \rightarrow Y \). With initial values of \( X = 3Y = 7 \), this wait condition is not satisfied. Therefore, even though this part and the \( S3 \) part are called concurrently, the first part (arm 1) will actually not run. After \( S3 \) executes, the value of \( X \) becomes \( X = 10 \), thus satisfying the await condition. Now the first part of the code (Arm 1) can run. This part of the code is actually not concurrent. So \( S1 \) will run and then \( S2 \) will execute after. Therefore, our history of execution becomes: \( S3 \rightarrow S1 \rightarrow S2 \). As shown in the previous problem, this is one of the iterations that results in \( X = 3Y = -4 \).

2. (a) A disadvantage of this topology is that all 3 of the processors share the same network, bus, and memory. Therefore, they must take turns accessing memory. If you are doing matrix multiplication with numbers in arrays, then this will create bottlenecks and latencies as each processor attempts to read/write to the memory. There is also only one connection between the network and memory. So if all the processors try to read/write, not only must they wait for the current memory access to finish but also endure the network latency of sending all the requests to memory over the network using one line.

   I suppose an advantage of this topology is that all of the processors are on the same shared network, each with their own line to the network. This means that they should be able to talk to each other. So if they are sharing the state of an array or matrix arithmetic that relies on other subproblems, they can communicate those answers to each other without needing to look it up in memory.

   A program to utilize this topology would be finding the maximum between each cell in 2 matrices:

   ```plaintext
   //Thread 1 and 2
   for i is 1..n cells:
       max = MAX(max, array[i]);
       send_to_thread3(max);
   
   //Thread 3
   final_max = MAX(thread1Max, thread2Max);
   ```

   This would work because 2 processors can calculate each of the 2 arrays and send the results to the third processor which does the final compare between the 2 sub calculations.

   (b) A disadvantage of this topology is that if your processes rely on shared memory, that memory must either be kept and updated in both memory locations or sent back and forth. If you are sharing memory, then you can only do so between the 3 processors that are associated with that memory block. If you need
to communicate between each set of the 3 processors then you would have to send that over the network (introduces network latency) and then have it written to the other memory location and then the end processors would have to read their memory to read the message. This introduces massive slowdowns compared to the previous topology.

An advantage of this topology would be efficiency in a program that segments threads that don’t need to communicate with each other. So for instance, a music player such as iTunes. One set of the processors could be dedicated to playing music, updating the seek bar, displaying artwork and other meta data. The other set could be in charge of network connections to their servers in case the user wants to also buy stuff while listening to their existing library. These two things can happen at the same time but don’t need to use any shared memory since they are doing totaly seperate objetives.

```c
//Process 1
P1: play_music();
P2: update_seek_bar();
P3: display_meta_data();

//Process 2
P4: Maintain_net_socket();
P5: TCP_read_write();
P6: Render_response();
```

Again, this is well suited since each of the 2 processes need to access shared information amongs each of their own threads, however, the processes themselves have no overlap and can remain independent of each other.

(c) A disadvantage of this topology would be the severe network latency. A processor is much faster than a memory access. When you introduce a network latency, the amount of time the CPU must wait is exponentially increased. Depending on the network, it may also only process one transmission at a time. Meaning this would also introduce a large bottleneck on top of the network traversal time.

An advantage of this topology would be that if there were multiple concurrent transmissions supported on the network, then any node on the network could communicate with any other node. So processor1 could talk to memory2 and processor2 could message processor1.

```c
//Process1
for i = 0..n:
x = sqrt(array(i))
send_to_processor(p2, x)

//process2
if x > currentMax:
array2(i) = x;
```

This would work well on this topology because there are multiple interactions between processors and memories within this program.
3. The table is shown below:

<table>
<thead>
<tr>
<th>Matrix Size</th>
<th>Threaded Run 1</th>
<th>Threaded Run 2</th>
<th>Non-Threaded Run 1</th>
<th>Non-Threaded Run 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000x5000</td>
<td>.026</td>
<td>.051</td>
<td>.0505</td>
<td>.0501</td>
</tr>
<tr>
<td>10000x10000</td>
<td>.222</td>
<td>.225</td>
<td>.202</td>
<td>.201</td>
</tr>
<tr>
<td>15000x15000</td>
<td>.45</td>
<td>.43</td>
<td>.458</td>
<td>.458</td>
</tr>
<tr>
<td>20000x20000</td>
<td>.78</td>
<td>.76</td>
<td>.811</td>
<td>.810</td>
</tr>
</tbody>
</table>