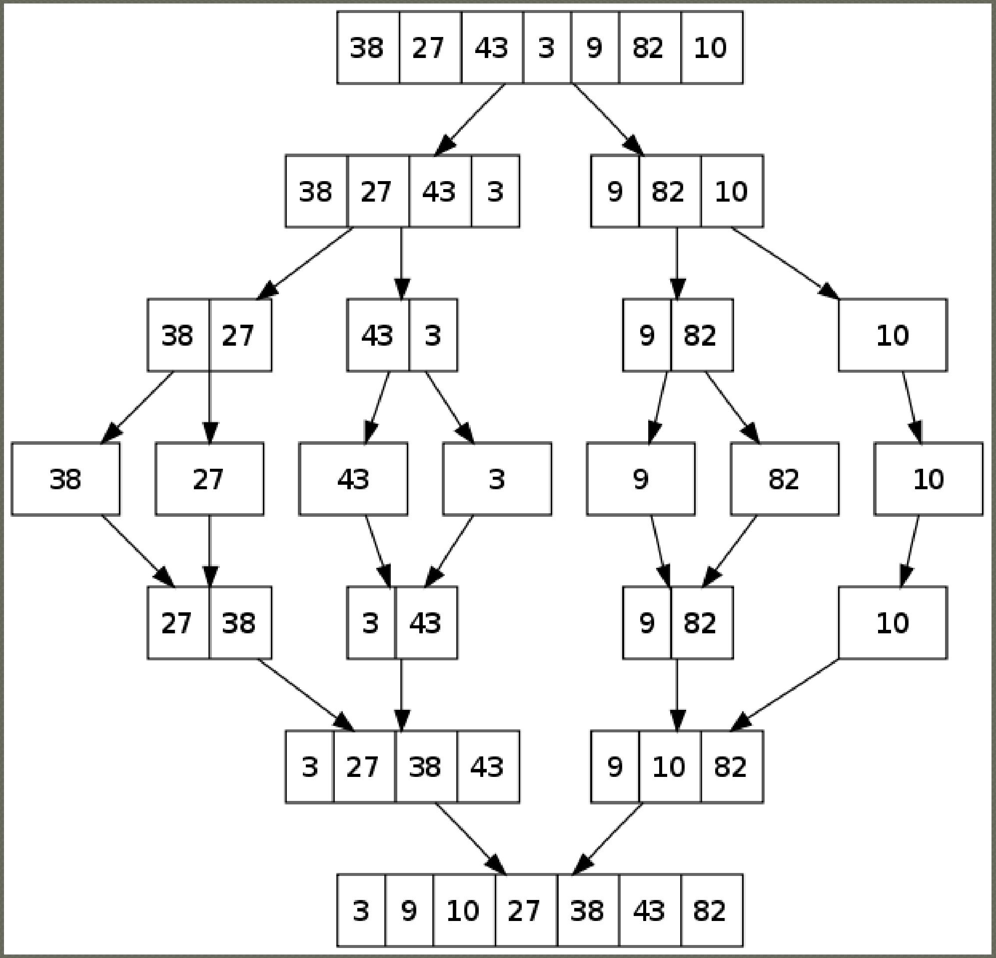
MergeSort over CUDA, MPI, and openMP

As implemented by Brady Tello
CSE710
SUNY at Buffalo
Fall 2009

Presentation overview

- MergeSort review (quick)
- Parallelization strategy
- Implementation attempt 1
- Mistakes in implementation attempt 1
  - What I did to try and correct those mistakes
- Run time analysis
- What I learned

MergeSort

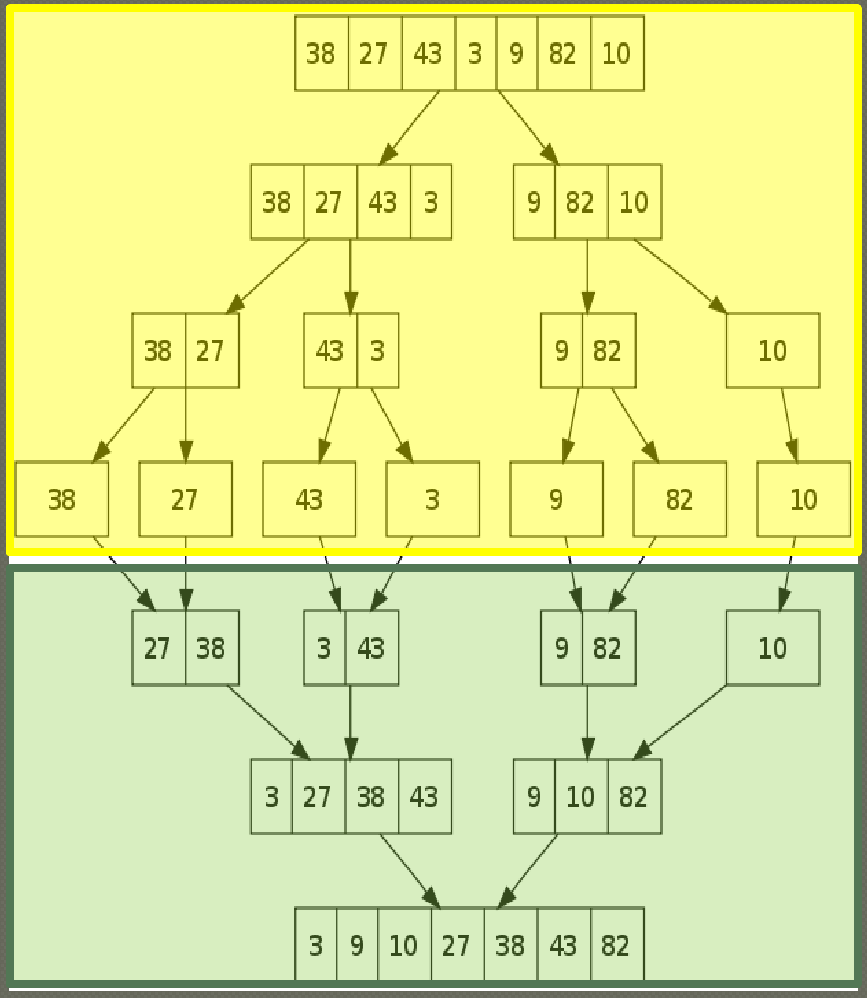


Logical flow of Merge Sort

How can this be parallelized?

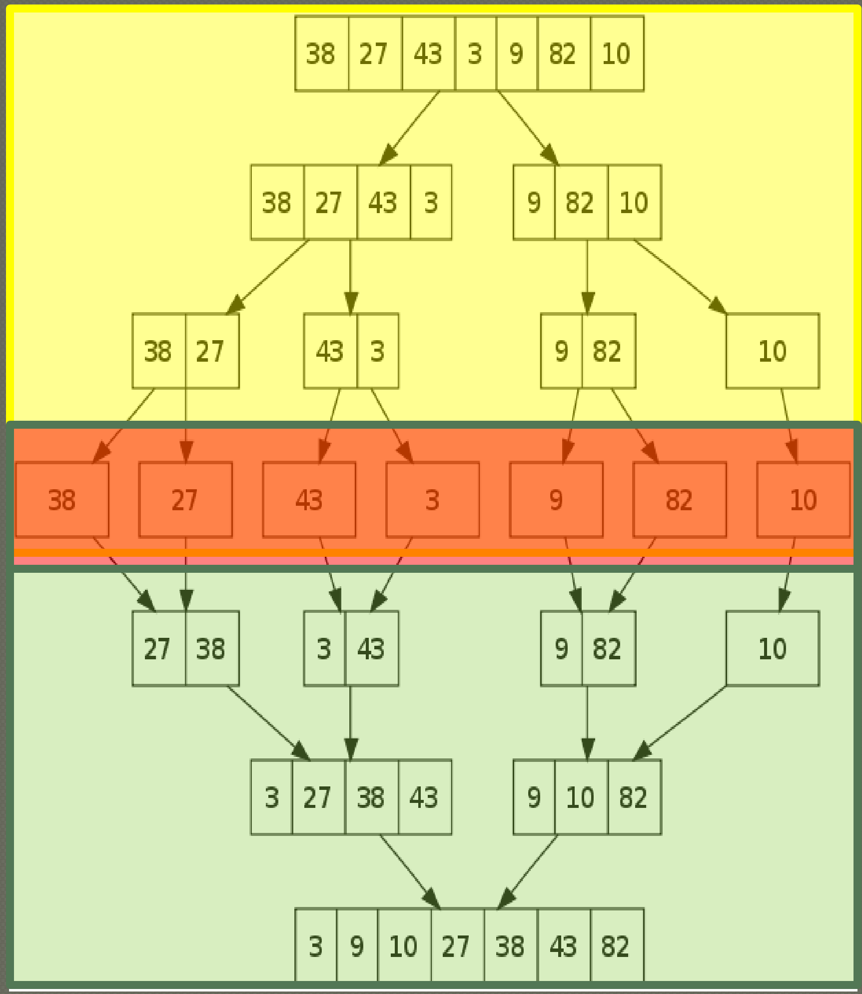
- The algorithm is largely composed of two phases which are readily parallelizable

1. Split Phase
2. Join phase



How can this be parallelized?

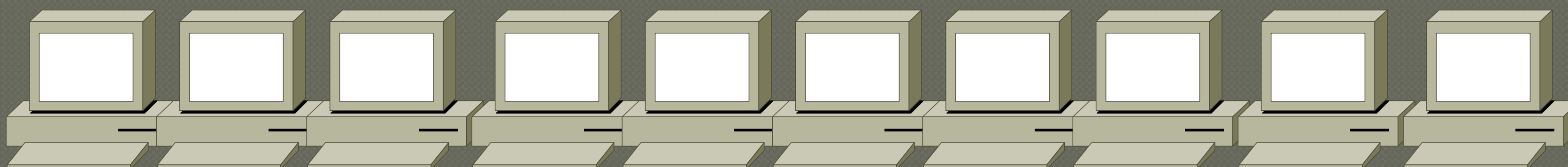
- Normally, mergeSort takes log(n) splits to break the list into single elements
- Using the Magic cluster's CUDA over OpenMP over MPI setup we should be able to do it in 3.



Breaking the list down(MPI)

Data => | 1/10 | 1/10 | 1/10 | 1/10 | 1/10 | 1/10 | 1/10 | 1/10 | 1/10 | 1/10 |

MPI_SEND

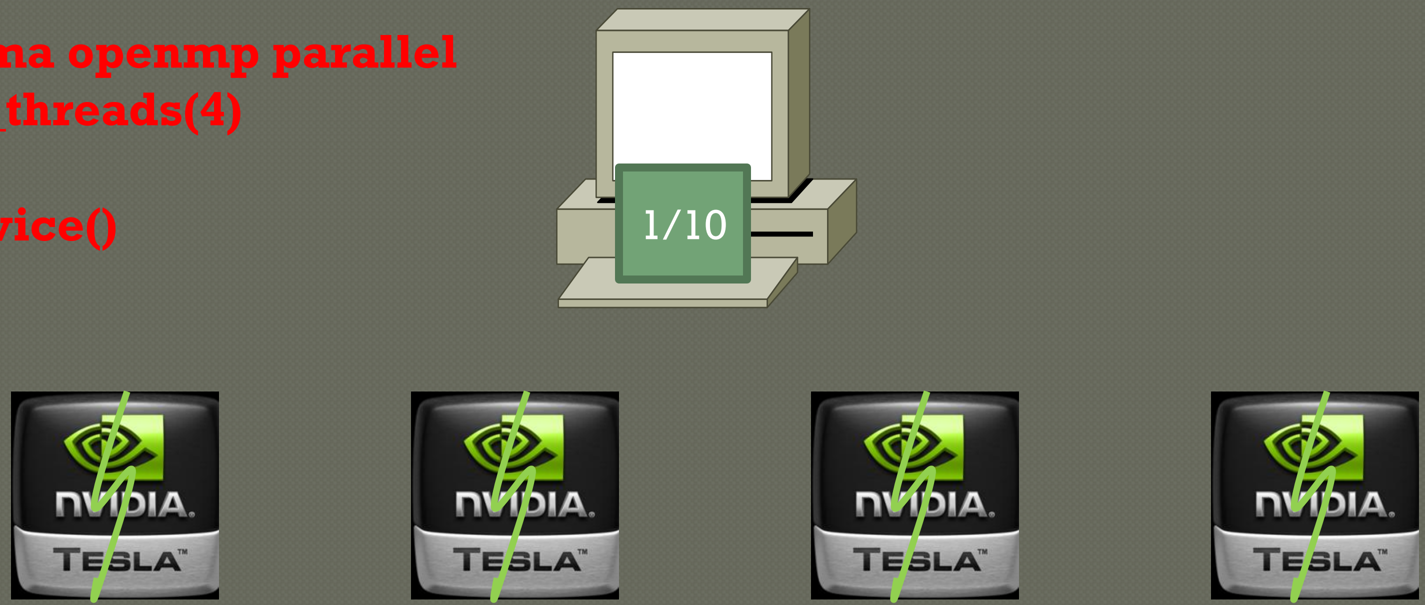


Dell Nodes

- For my testing I used 10 of the 13 Dell nodes (for no reason besides 10 is a nice round number)
- Step 1 is to send 1/10th of the overall list to each dell node for processing using MPI.

Breaking the list down (OpenMP)

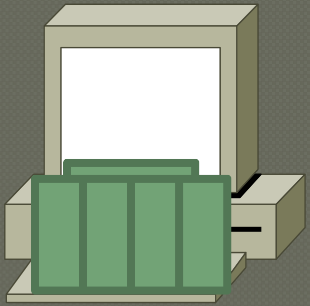
```
#pragma openmp parallel
  num_threads(4)

initDevice()
```



- Now on each Dell node, we start up the 4 Tesla co-processors on separate OpenMP threads

Breaking the list down (CUDA)

cudaMemCpy(…,cudaMe
mcpyHostToDevice)

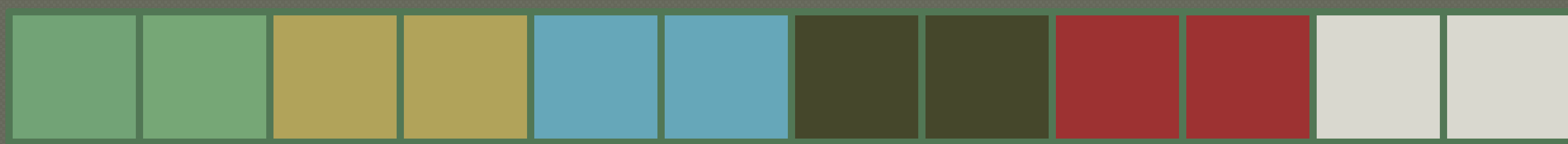


- Now we can send ¼ of the 1/10th of the original list to each Tesla via cudaMemCpy
- At this point CUDA threads can access each individual element and thus we can begin merging!

Merging(CUDA)

- On each Tesla we can merge the data in successive chunks of size 2^i

Merging(CUDA)

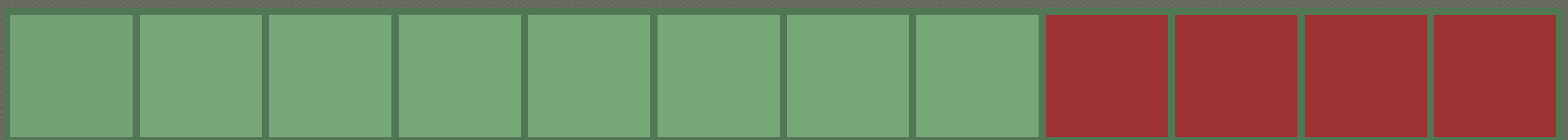


- On each Tesla we can merge the data in successive chunks of size 2^i

Merging(CUDA)

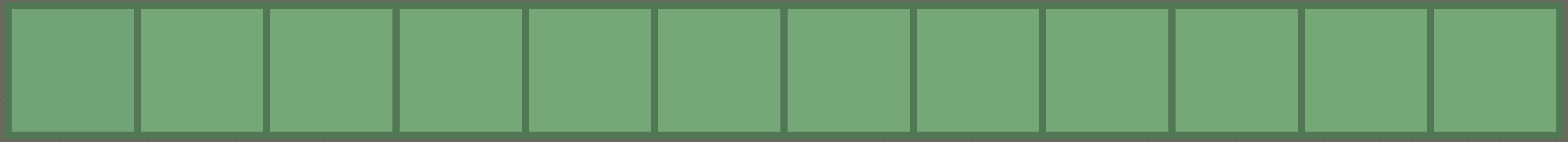
- On each Tesla we can merge the data in successive chunks of size 2^i

Merging(CUDA)



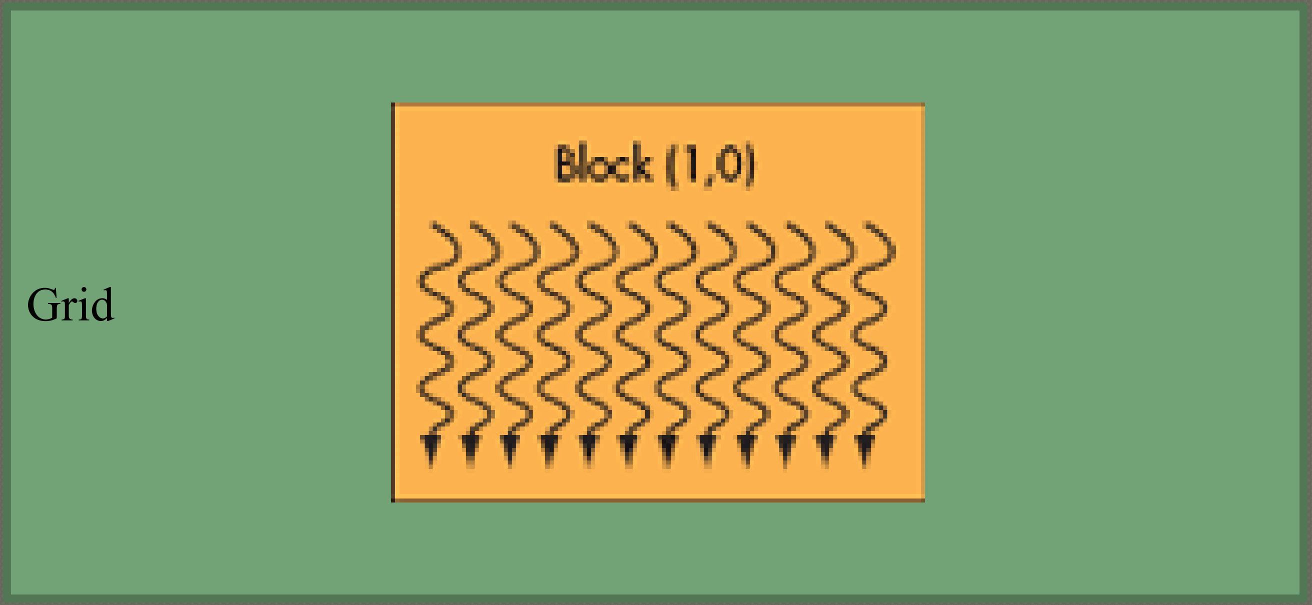
- On each Tesla we can merge the data in successive chunks of size 2^i

Merging(CUDA)



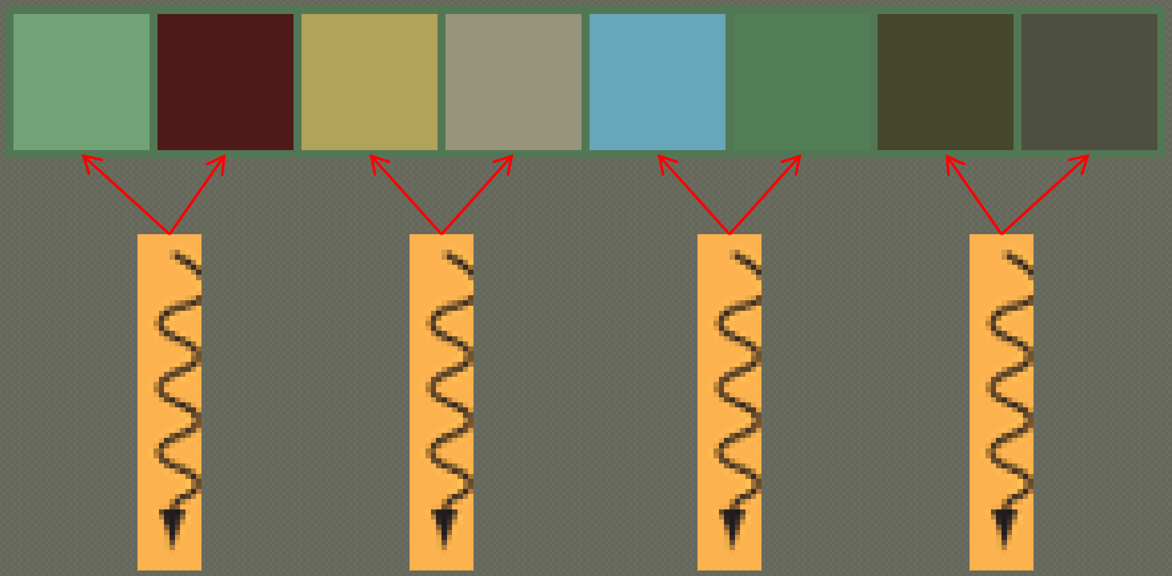
- On each Tesla we can merge the data in successive chunks of size 2^i

Merging(CUDA)



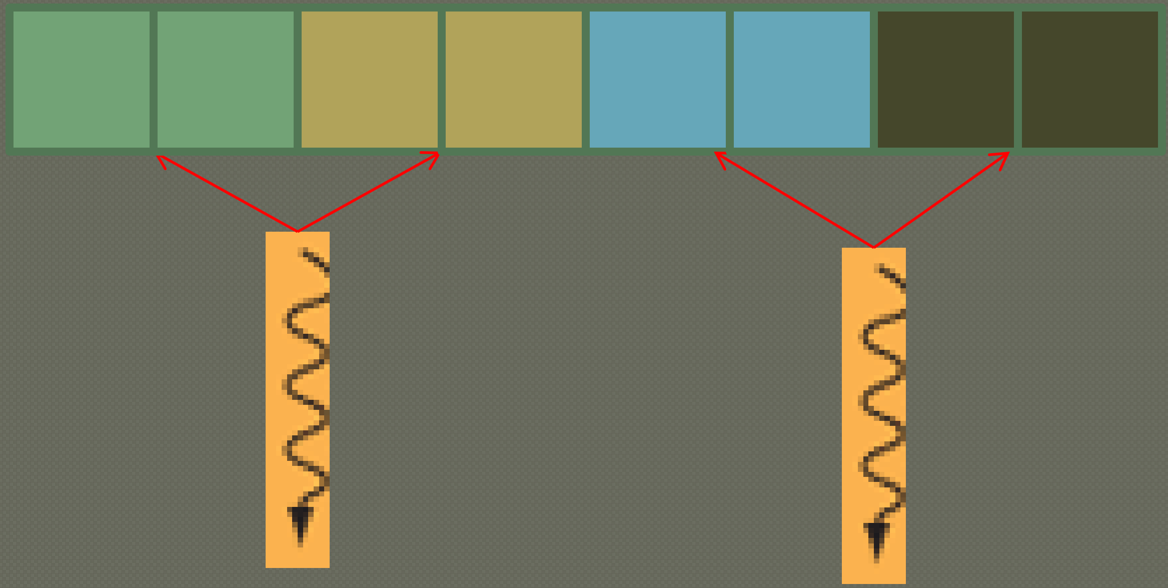
- My initial plan for doing this merging was to use a single block of threads on each device
- Initially each thread would be responsible for 2 list items, then 4, then 8, then 16 etc.
- Since each thread is responsible for more and more each iteration, the number of threads can also be decreased.

Merging(CUDA)



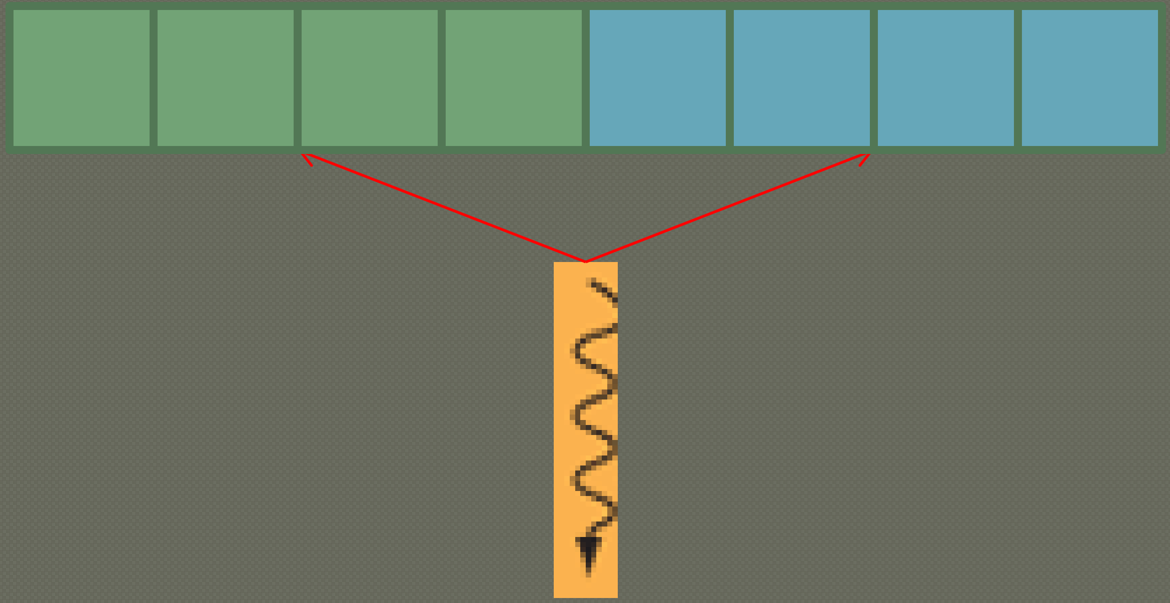
- Example

Merging(CUDA)



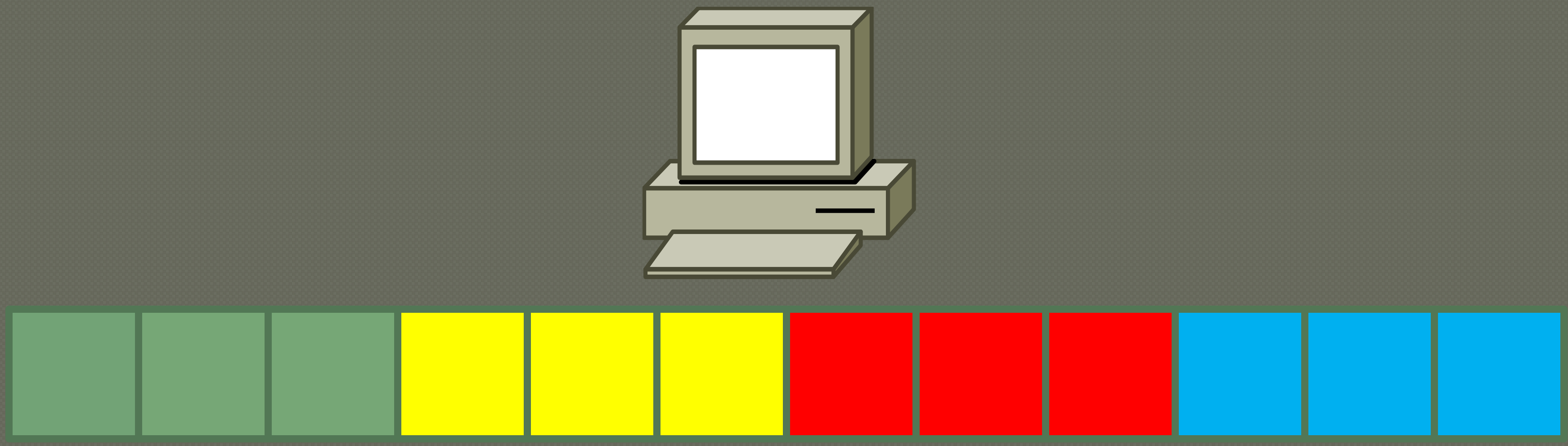
- Example

Merging(CUDA)



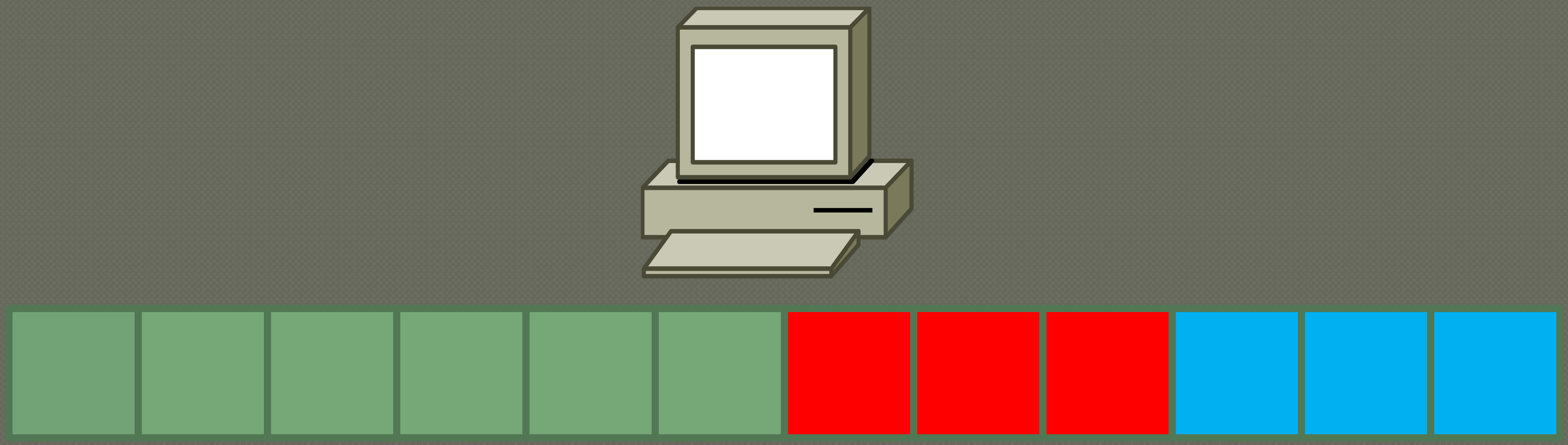
- Works in theory but CUDA has a limit of 512 threads per block
- NOTE: This is how I originally implemented the algorithm and this limit caused problems

Merging(MPI)



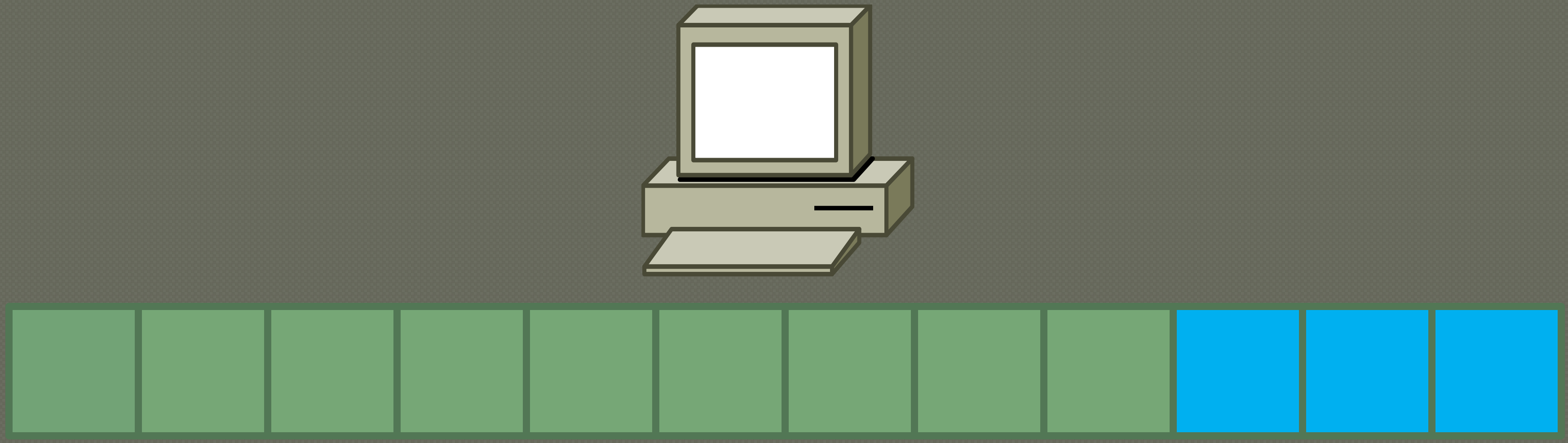
- At this point, the list on each Dell node will consist of 4 sorted lists after CUDA has done it's work.
- We just Merge those 4 lists using a sequential Merge function.

Merging(MPI)



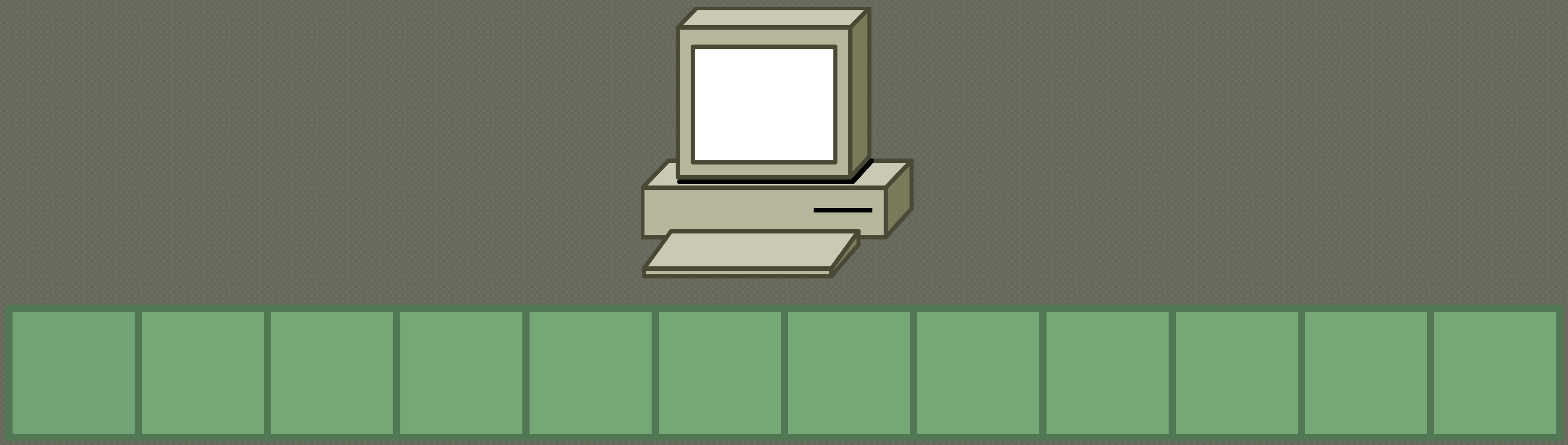
- 1st merge

Merging(MPI)



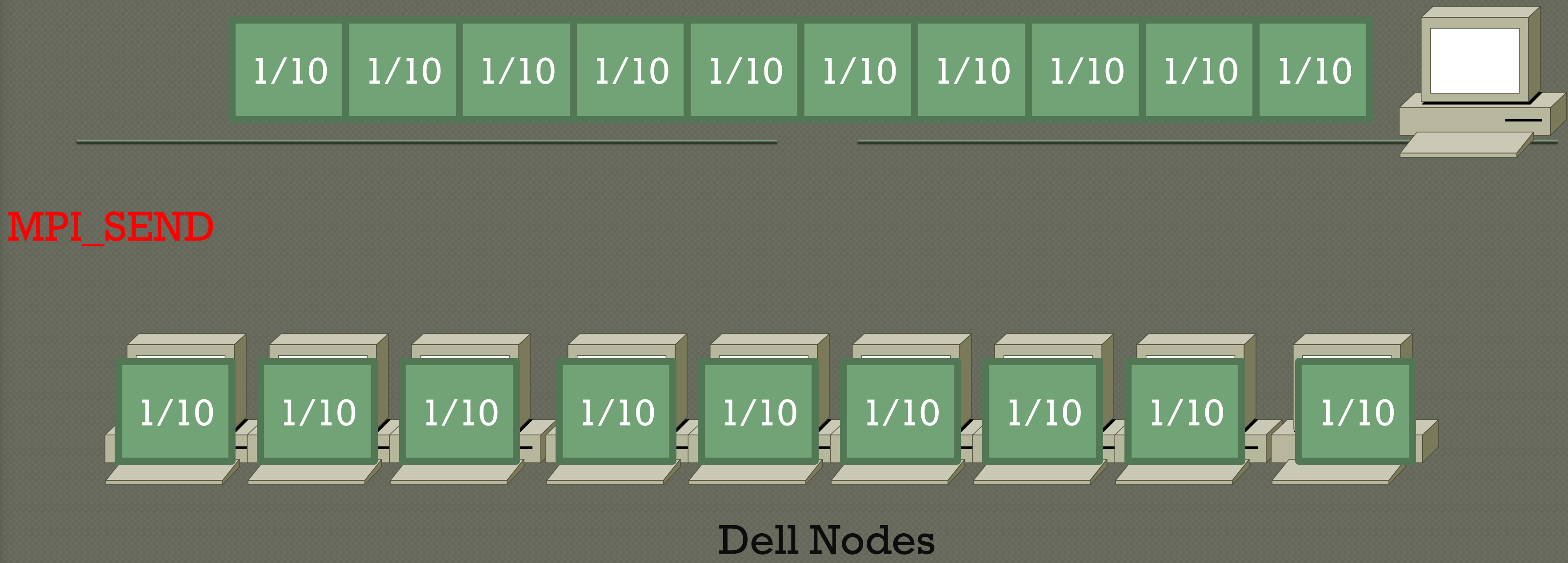
- 2nd merge

Merging(MPI)



- final merge

Final Merge(MPI Send/Rcv)



- Now we can send the data from each Dell Node to a single Dell Node which we will call the master node.
- As this node receives new pieces of merged data, it will just merge it with what it has already using the same previously mentioned sequential merge routine.
- This is a HUGE bottleneck in the execution time!!!

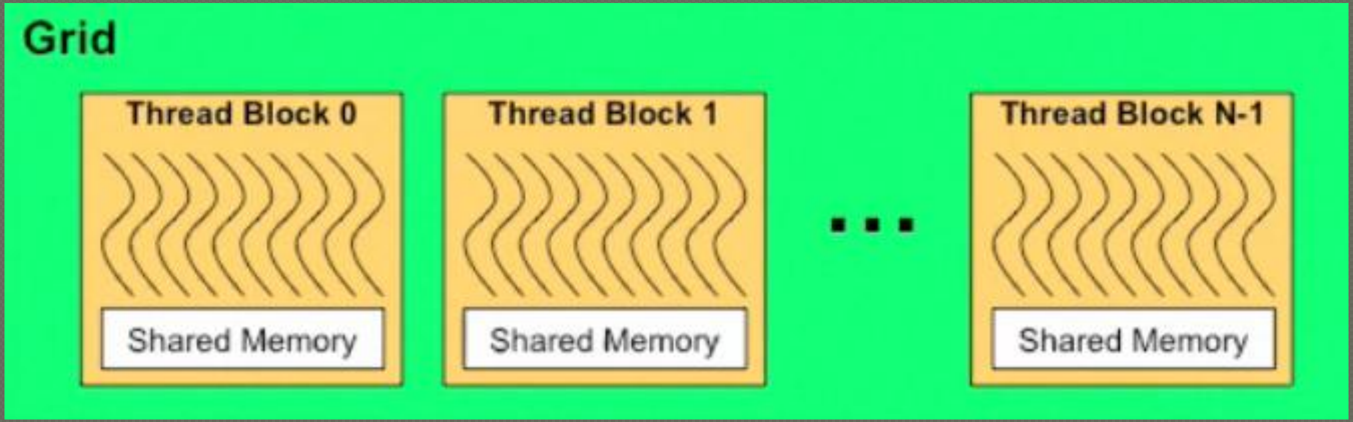
Problems

- I tested and implemented this algorithm using small, conveniently sized lists which broke down nicely.
- Larger datasets caused problems because of all the special cases in the overhead
  - Spent a lot of time tracking down special cases
  - Lots of "off by 1" type errors
- Fixing these bugs made it work perfectly for lists of fairly small sizes

Bigger problems

- The Tesla coprocessors on the Magic cluster only allow 512 threads per block.
- HUGE problem for my algorithm.
- My algorithm isn’t very useful if it can’t ever get to the point where it outperforms the sequential version

Solution



- If more than 512 threads needed then add another block
- Our Tesla devices allow for 65535 blocks to be created
- Using shared memories, should be able to extend the old algorithm to multiple blocks fairly easily

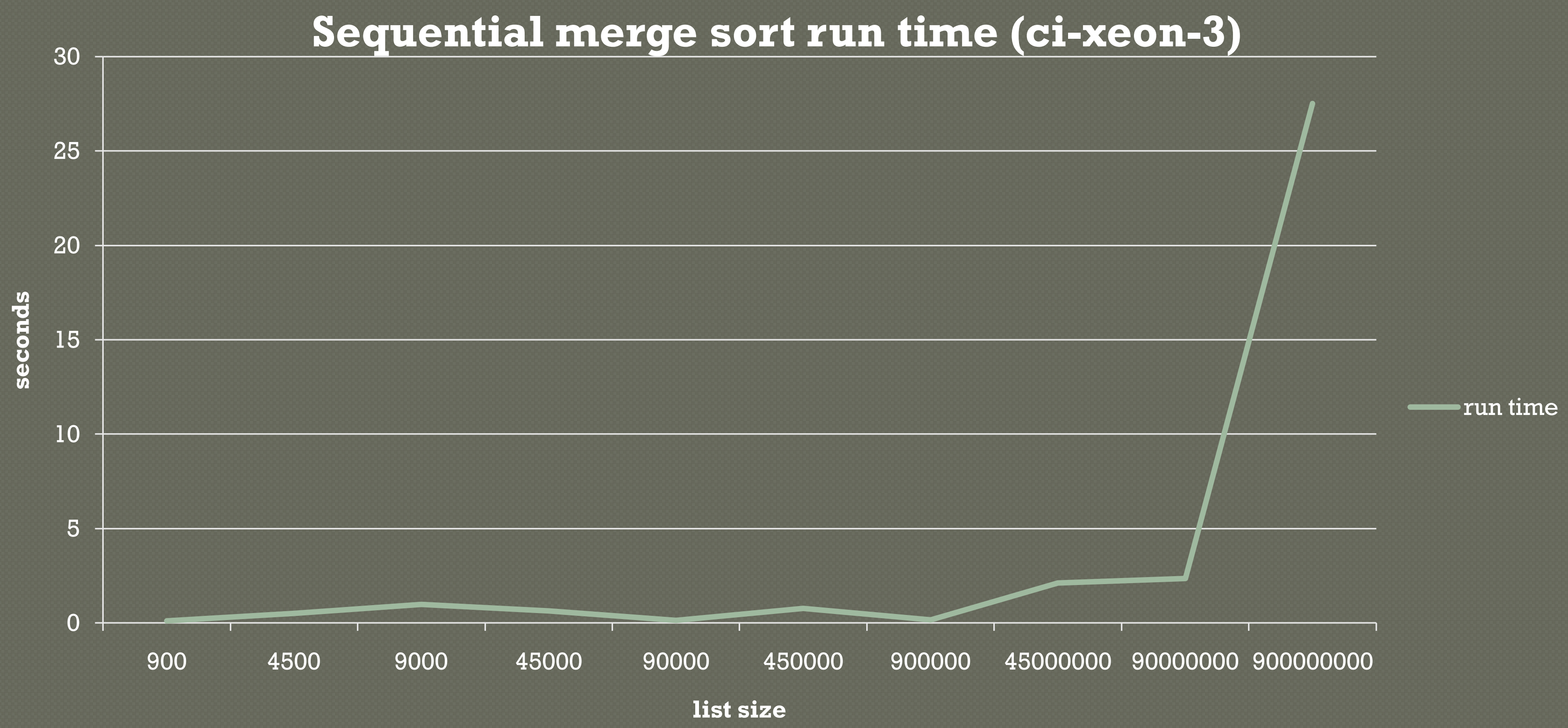
Solution results

- Was able to get all the math for breaking up threads amongst blocks etc.
- My algorithm now will run with lists that are very large…
  - But not correctly
- There is a problem somewhere in my CUDA kernel
  - Troubleshooting the kernel has proven difficult since we can't easily run the debugger (that I know of).

Analysis

- The algorithm is correct except for a small error somewhere
- Works partially for a limited data size
- All results are an approximation to what they would be if the code was 100% functional

Analysis

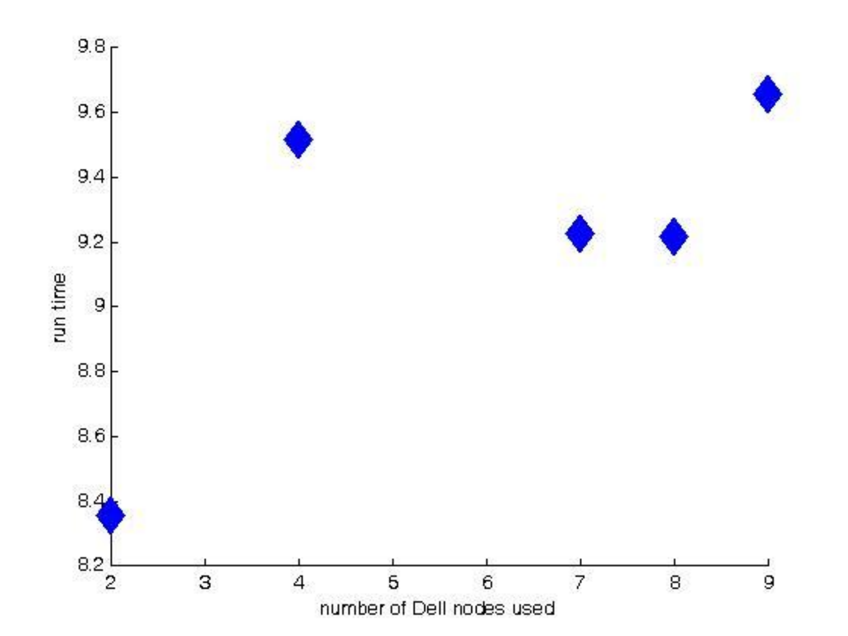


Running my parallel version using 900,000,000 inputs on 9 nodes took only 10.2 seconds (although its results were incorrect)

Analysis

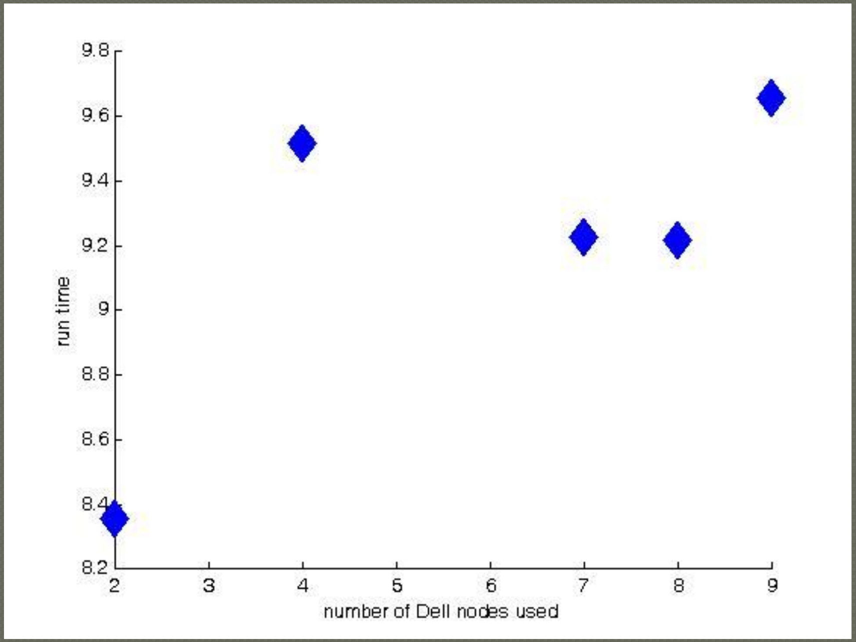
- This graph shows run time versus the number of Dell nodes which were used to sort a list of 900,000 elements.
- Each Dell node has 2 Intel Xeon CPUs running at 3.33GHz
- Each Tesla co-processor has 4 GPUs
- The effective number of processors used is:

*#of Dell Nodes*2*4*



Analysis

- Less Processors led to better performance!!!
- Why?
  - My list sizes are so small that the only element which really impacts performance is the parallelism overhead.



Analysis

- Communication setup eats up a lot of time
  - cudaGetDeviceCount()
    - Takes 3.7 seconds on average
  - MPI setup takes 1 second on average
- Communication itself takes up lot of time.
  - Sending large amounts of data to/from several nodes to/from a single node using MPI was the biggest bottleneck in the program.

Lessons

1. Don’t assume a new system will be able to handle a million threads without incident… i.e. read the specs closely.
2. When writing a program which is supposed to sort millions of numbers, test it as such.
3. Unrolling a recurrence relation requires a LOT of overhead. New respect for the elegance of recursion.