* 1. Please see hw3.erl.

|  |  |
| --- | --- |
| Workers | Speedup |
| 4 | 3.734103082557235 |
| 8 | 4.574294255837006 |
| 16 | 7.858342887484388 |
| 32 | 10.089374091526631 |
| 64 | 12.21394941134951 |
| 128 | 16.203408337775418 |
| 256 | 15.398868592893184 |

|  |  |  |
| --- | --- | --- |
| N | Workers | Speedup |
| 10 | 128 | 1.0170732235864777 |
| 100 | 128 | 1.2181658037351168 |
| 1000 | 128 | 1.5927807561467273 |
| 10000 | 128 | 3.4162611685176465 |
| 100000 | 128 | 8.155197640273236 |
| 1000000 | 128 | 17.413101223674897 |

After some experimentation, it took N 2200 to consistently achieve 90% or greater speedup.

* 1. Please see hw3.erl.

|  |  |  |
| --- | --- | --- |
| N | Workers | Speedup |
| 1000000 | 4 | 1.1790680837338547 |
| 1000000 | 8 | 1.3936672853144412 |
| 1000000 | 16 | 1.467924369866029 |
| 1000000 | 32 | 1.6798512101179834 |
| 1000000 | 64 | 1.9141003016763372 |
| 1000000 | 128 | 2.4228657928070176 |
| 1000000 | 256 | 2.346975805316466 |

* 1. “Embarrassingly parallel problem” (primes, at least), in that we can throw processors at it (to a degree) and see speedup. The operations are commutable (?) and a-word (?), allow us to really exploit the addition of processors.

However, we also see that the sequential version is pretty close up through N = XXX, and it takes until about N = 2200 to see a consistent 90% speedup. Moreover, the sequential and parallel versions were very close with low N values, suggesting a split at that point is largely irrelevant. If we don’t have enough N to do parallel operations in the processors, then the advantage of having additional processors is lost and we end up with a workflow close to the sequential algorithm. On the flip side, pumping up N allows the parallel algorithm to take larger advantage of its processors, as the size of the data to be operated on in parallel grows (and hence the factor separating the parallel algorithm from the sequential one grows).

Performance loss (with respect to the readings?)

* 1. Please see hw3.c.
     1. random array n=1000000 n\_trial=10 t\_avg=1.620e-01
     2. ascending array n=1000000 n\_trial=10 t\_avg=7.680e-02
     3. random list n=1000000 n\_trial=10 t\_avg=2.712e-01
     4. ascending list n=1000000 n\_trial=10 t\_avg=1.320e-02
  2. Sorting an array of random elements takes longer than an array of ascending elements by about a factor of 2 (2.109375 in this case). The steps of merge sort likely contribute to this, as an ascending array, generally, will require less swaps than a random array to achieve sorted-ness. Factors of parallelism also likely play roles here.

Accessing the data to sort in cache:

ascending = stored together, so bringing in lots of data you will need soon (what kind of locality?)—minimizing cache misses

random = values in the array are very different, but stored near each other, so when reading in data during sorting, we may require values stored far from each other—meaning more cache misses (not exploiting cache size)—could test this with larger ascending data (reduce the effectiveness of the locality available in the ascending trial)

Branch prediction: maybe tries to predict what the data is going to be, but it gets changed more often (more mispredictions) when we have more swaps = in random

How does the storage of the values come in here? Writing to memory a factor?

* 1. Sorting list of random elements takes longer than sorting one of ascending elements. In this case, sorting an ascending list is faster by a factor of 20.545, which is larger than the corresponding factor for arrays.

Big factor here might be the cost associated with traversing the list. In either case, this is a large cost, so minimizing traversals will affect runtime—ascending list is minimal traversals (go from one to the next), where the random list will have more traversal (have to go the values when we’re comparing them).

Branch prediction, potentially, again, maybe more so in terms of traversing the list (double down!). If traversing takes a long time, it might try to predict the next value. With the ascending list, it will predict the next value it needs more accurately—less mispredictions—than with a random list.

How does the storage of the values come in here? Writing to memory a factor?

* 1. , where c = the total number of clusters