* 1. Please see hw3.erl.

|  |  |  |
| --- | --- | --- |
| N | Workers | Speedup |
| 1000000 | 4 | 2.996609443376608 |
| 1000000 | 8 | 4.734409445925075 |
| 1000000 | 16 | 7.392374482250656 |
| 1000000 | 32 | 10.771345347224072 |
| 1000000 | 64 | 12.576175837397772 |
| 1000000 | 128 | 19.71217067513172 |
| 1000000 | 256 | 19.863190334966582 |

|  |  |  |
| --- | --- | --- |
| N | Workers | Speedup |
| 10 | 256 | 1.0347889382786861 |
| 100 | 256 | 1.159302879059593 |
| 1000 | 256 | 1.6565513108980379 |
| 10000 | 256 | 3.564325897867064 |
| 100000 | 256 | 8.88353946559858 |
| 1000000 | 256 | 20.762971762238053 |

After some experimentation, 90% or greater speedup is achieved around N 2200 (2100 – 2300).

* 1. Please see hw3.erl.
  2. Note: the tested parallel version of sum\_inv\_twin\_primes includes steps for acquiring the twin primes and summing the inverses of these.

|  |  |  |
| --- | --- | --- |
| 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. These problems represent “embarrassingly parallel problems”, in that given sufficiently large values for N, we can throw processors at the problems to parallelize and see speedup as a result. The operations in primes/3 and sum\_inv\_twin\_primes/3 are commutable and associative, allowing us to exploit the benefits of additional parallel processors.

That said, sum\_inv\_twin\_primes/3 showed much less speedup from throwing workers at it and often showed little or no noticeable speedup as the number of workers increased (notably, the average times and differences between these runs varied from run-to-run and even by time of the day). This may highlight the idea that a higher proportion of the time-hogging operations from the primes function could be delegated to work in parallel than could be done in the sum\_inv\_twin\_primes function. That is, more of the time-draining operations from the sequential version could be delegated to run in parallel in the parallel version (i.e. exploiting workers to increase speedup) with the primes function than could be done with the sum\_inv\_twin\_primes function.

Notably, we see that the sequential algorithms remain close to the parallel algorithms (i.e. speedup is minimal) when N is small or there are few processors. With fewer processors, the parallel algorithm becomes more akin to the sequential one (i.e. the number of processors approach one, which is also the number of processors allocated to the sequential algorithm).

Similarly, if we don’t have enough N to do parallel operations in the processors, then the advantage of having additional processors is minimized and we end up with a workflow close to that of the sequential algorithm—the number of processors actually doing work approaches one. 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 increases).

Minimalized speedups for the parallel algorithms might also point to additional overhead requirements, such as communication between workers and the need for extra storage. This is especially true for the parallel version of sum\_inv\_twin\_primes, which, compared to its sequential counterpart, requires extra memory for additional data structures and lists, plus communication costs, and additional operations for appending lists, etc. The effect of this parallel overhead performance loss is less of a factor in the case of primes, which does not require additional operations or communication.

* 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.

For one, there may be differences in how the data to be sorted is added/accessed with respect to the cache. In the ascending case, close values may be stored together, so when we’re reading in to sort, many consecutive values will already be in order. The cache will bring in lots of data that it will need soon, hence minimizing cache misses and exploiting temporal locality (??). This is especially important with increased merge iterations, as a semblance of order mirroring storage will prevent deep searches for merged elements (and will make consecutive reading of values faster for the same reason—more values stored close together means fewer cache misses). In the random case, values stored next to each other in the array may not be similar, so merging may lead to more cache misses, more searching for values, and a final merged array in which consecutive values can point to memory addresses far away from one another.

We could test the above hypothesis by replacing the cache with a tiny one (say, a cache that can only hold one value). Then, we could no longer bring many neighboring values into the cache and the benefit of storing values near each other would be negated, hence we should see the effect of locality minimized.

Branch prediction and writing to storage could also play a factor here. Perhaps the machine tries to predict what the next operation will be (say, compare the next two values). If the data is already sorted, the algorithm may require less swaps, so the machine can predict with more accuracy (i.e. the next values should be next in line). If the data is unsorted, it may require more swaps, leading to more mispredictions.

* 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.

A big factor here might be the cost associated with traversing the list. In either case, traversal is a large cost, so minimizing traversals will affect comparative runtime and speedup. An ascending list requires minimal traversals during merging; the upcoming values will be near their places in final list and will require minimal traversal for swaps. For random lists, we may require more traversals to compare and swap, as the values may not be sorted and the algorithm will have to traverse to elements in the list in order to compare them.

Branch prediction could also play a role here, perhaps enhancing the effects of list traversal. Because traversing takes a long time, the machine might try to predict the next value to minimize this time. With the ascending list, it will predict the next value with more accuracy (less misprediction) than with a random list.

* 1. Bisection width =
  2. Transfer 1GByte/second

Bisection bandwidth = 1GB/sec. \* 10 = 10 GB/second = 10,000,000,000 bytes/second

|  |  |
| --- | --- |
| Source node *i* | Destination node *i* + |
| 0 |  |
| 1 |  |
| 2 |  |
| … | … |
|  |  |

Each message is 1 Kb = 1000 bytes

Message time =

When k = 4,

Time:

* 1. 2D toroidal mesh p = q2 nodes, bisection width is

In our case, p = 10k bisection width

* 1. Similar to (c) above, but bisection bandwidth = GB/second

Message time =  **seconds**

When k = 4,

Time: