Comparison of Radix Sort Performance Using Multiple Parallel Programming Models And Languages

Michael Capasso

capa1@umbc.edu

Alexander Kahan

akahan1@umbc.edu

Alexander Wooten

awooten1@umbc.edu

Department of Computer Science and Electrical Engineering

University of Maryland Baltimore County

12 May 2014

**Abstract**

As modern society continues to become more and more reliant on technology, the demand for faster, more powerful computers has increased tremendously, with no signs of slowing down any time soon. One of the most popular methods of meeting this demand has been an increased adoption of multi-processor architectures and parallel computing. What was once solely the domain of expensive supercomputers has now become an essential component of home computing, with each generation of personal computers utilizing processors containing an increasing number of cores. This steady advancement of processing power is undoubtedly a positive occurrence, but it does present some unique challenges for modern computer scientists and software developers. While some computational problems are embarrassingly parallel and easy to adapt to multi-processor architectures, others, such as sorting, present significantly more challenging conundrums. These issues are further complicated by the existence of multiple distinct parallel programming models across several languages. The purpose of this study is to examine the time performance of radix sorting using two common parallel models, message passing with Open MPI and shared memory with OpenMP, as well as two common programming languages, C and Python.

Keywords: Parallel computing, multi-core processors, parallel sorting, performance comparison, shared memory, message passing, MPI, OpenMP, C, Python

**1. Introduction**

In order to compare different parallel programming models and languages, we elected to use radix sort for multiple reasons. Sorting itself is an interesting problem. Because the act of sorting is dependent upon a single task being performed on a single set of data, it does not lend itself to an embarrassingly parallel dived-and-conquer or pipelined approach. This makes it a far more interesting problem to explore in our study than something simpler. Radix sort is uniquely suited to parallelization due to its non-comparative nature. Rather than directly comparing numerical values in the same manner as more traditional sorting algorithms such as merge sort or quicksort, radix sort works by dividing elements of a data set into buckets based on their least significant digits. After each pass, the elements are collected from their buckets and redistributed in order. This results in O(kn) worst case performance, where k is the maximum number of digits and n is the number of elements [1].

**2. Parallelization**

In order to parallelize this algorithm using message passing, the final buckets are divided up amongst the available processors and then merged. For example, if we have ten buckets and four processors, processor zero will hold final buckets 1, 5, and 9, processor one will hold final buckets 2, 6 and 10, processor two will hold final buckets 3 and 7, and processor three will hold final buckets 4 and 8. Additionally, each process will have its own ten local buckets. At the end of each iteration, they will send the contents of each local bucket to the processor with that final bucket. Processor zero will move the contents of its local buckets 1, 5 and 9 to its own final buckets for 1, 5, and 9. It will then send the contents of 2, 3, 4, 6, 7, 8, and 10 to each of the other processes. Bucket 2 goes to processor one, bucket 3 goes to processor two, etc. This has the potential to parallelize beyond 10 processors, since the data is still distributed among all p processors, even when p is greater than the number of buckets.

Parallelization using shared memory is done in a similar manner. In this case, the local buckets will be stored separately for each processor while the final buckets will be stored in shared memory. In both cases, the new worst case performance should theoretically be O(k(n/p)).

**3. Data**

In order to test our various programs, arrays of sequential numbers were randomized and sorted. For example, an array of size 1,000 would contain the numbers 0 through 999 in random order/ All parallel tests were performed using 8 processors on the Bluewave system.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Program** | **Trial 1** | **Trial 2** | **Trial 3** | **Trial 4** | **Trial 5** | **Arithmetic Mean** | **Standard Deviation** |
| Sequential | 0.084 | 0.053 | 0.059 | 0.068 | 0.058 | 0.064 | 0.01222 |
| C MPI | 1.165 | 1.158 | 1.162 | 1.150 | 1.152 | 1.157 | 0.00638749 |
| C OpenMP | 0.068 | 0.066 | 0.069 | 0.085 | 0.074 | 0.0724 | 0.007635 |
| Python MPI | 1.207 | 1.176 | 1.175 | 1.183 | 1.176 | 1.183 | 0.01358 |

Table 3.1: Results sorting 1,000 items

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Program** | **Trial 1** | **Trial 2** | **Trial 3** | **Trial 4** | **Trial 5** | **Arithmetic Mean** | **Standard Deviation** |
| Sequential | 0.263 | 0.274 | 0.305 | 0.275 | 0.341 | 0.2916 | 0.03171 |
| C MPI | 1.176 | 1.166 | 1.165 | 1.173 | 1.171 | 1.17 | 0.004658 |
| C OpenMP | 0.099 | 0.095 | 0.091 | 0.101 | 0.088 | 0.0948 | 0.005404 |
| Python MPI | 1.296 | 1.303 | 1.295 | 1.271 | 1.289 | 1.291 | 0.01213 |

Table 3.2: Results sorting 10,000 items

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Program** | **Trial 1** | **Trial 2** | **Trial 3** | **Trial 4** | **Trial 5** | **Arithmetic Mean** | **Standard Deviation** |
| Sequential | 2.928 | 3.012 | 2.811 | 2.836 | 3.045 | 2.926 | 0.1035 |
| C MPI | 1.264 | 1.264 | 1.266 | 1.272 | 1.263 | 1.266 | 0.003633 |
| C OpenMP | 0.290 | 0.200 | 0.230 | 0.235 | 0.232 | 0.2374 | 0.03262 |
| Python MPI | 6.233 | 5.802 | 8.531 | 8.892 | 4.994 | 6.89 | 1.726 |

Table 3.3: Results sorting 100,000 items

**4. Analysis: MPI vs. OpenMP**

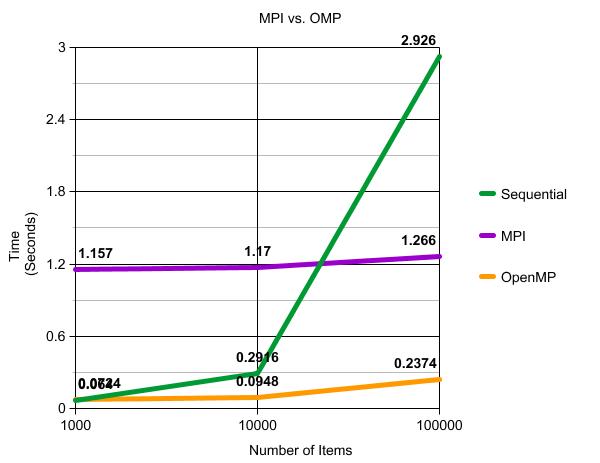


Figure 4.1: Average Runtimes of MPI vs. OpenMP

From Figure 4.1, we can see that the OpenMP program performed significantly better than both the MPI implementation and the sequential control program. Both the OpenMP and MPI programs demonstrate similar patterns, with a slight increase in run time between 1,000 and 10,000 items and a larger increase when the array size was upped to 100,000, as was expected due to the simple fact that 100,000 is significantly larger than 10,000. Analytically, it makes sense that shared memory would perform better than message passing. Given that both programs use very similar algorithms, the chief difference between the two is that MPI has to take time to send the contents of each processes' local buckets to the other processors and place the results in the corresponding final buckets. This introduces communication time as well as lag associated with blocking routines. In contrast, the OpenMP implementation uses shared memory, so no message passing is required. The separate processes simply complete their portion of the work then store the results in the shared memory location.

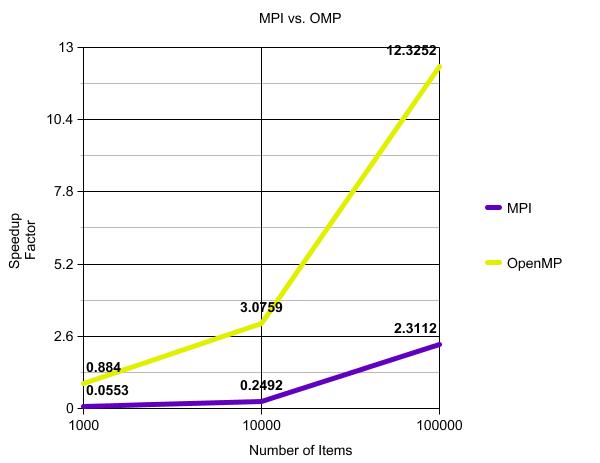


Figure 4.2: Speedup of MPI vs. OpenMP

Figure 4.2 compares the speedup factors of the OpenMP and MPI implementations. Speedup was calculated using the standard formula , where S(p) is the speedup factor given p processors, ts is the average runtime of the sequential program, and tp is the average runtime of the parallel implementation. When sorting 1,000 items, the shared memory implementation demonstrated slightly negative speedup. With 10,000 and 100,000 items, however, the speedup factor was significant. In contrast, the MPI program demonstrated negative speedup for both 1,000 and 10,000 items. It did speedup with 100,000 items, but by less of a factor than the OpenMP program did for 10,000 items.

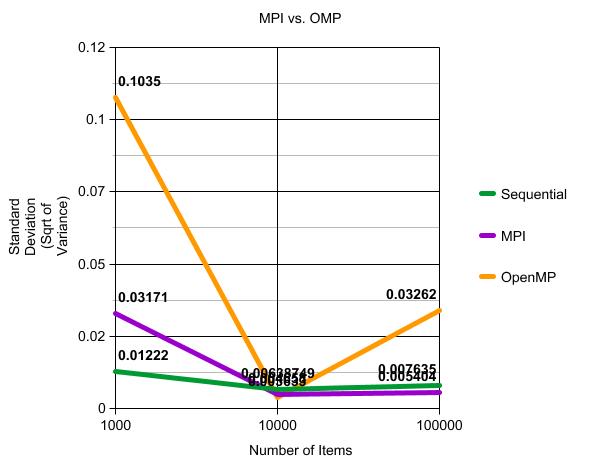


Figure 4.3: Standard deviations of MPI vs. OpenMP

While runtime is undoubtedly important, it is just as vital to explore performance consistency. For this reason, we elected to also explore the variance of our data. For ease of presentation, we have included the square root of the variances, also known as the standard deviations. All three programs displayed similar patterns, with the largest variance at 1,000 items, a decline in variance at 10,000 items, and a gradual increase while approaching 100,000 items. OpenMP had the greatest amount of variability of the three. That said, its highest standard deviation was 0.1035, which translates to a variance of only approximately 0.01 seconds. Given that this implementation performed significantly better than MPI, this level of variance is acceptable.

**5. Analysis: C vs. Python**

Before taking a look at our experimental results, it would be beneficial to give a little background on the C and Python programming languages. C is a statically-typed, general-purpose language developed in 1972 by Dennis Ritchie at Bell Labs [2]. It is closely associated with UNIX-based operation systems and is one of the most widely used programming languages in the world. Python, on the other hand, is a dynamically-typed general-purpose language developed in 1991 by Guido van Rossum at CWI in the Netherlands [3]. It is becoming increasingly popular due to its ease-of-use. For our experiment, both programs used MPI.

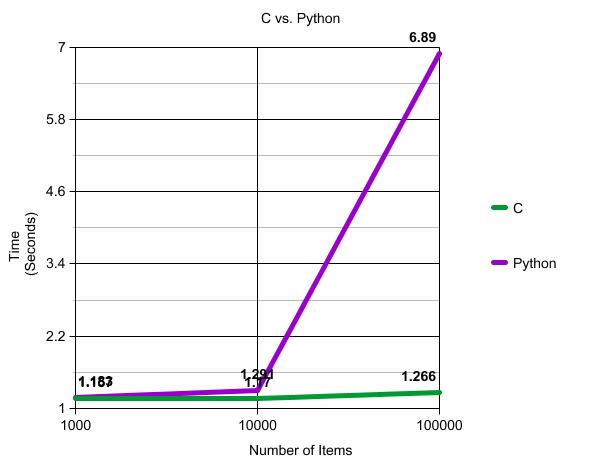


Figure 5.1: Average runtimes of C vs. Python

Figure 5.1 clearly shows that the C implementation performed dramatically better than the Python implementation, with the latter taking nearly seven seconds to sort the 100,000 item array. This is not only significantly slower than the C implementation, but it also takes much longer than a sequential radix sort.

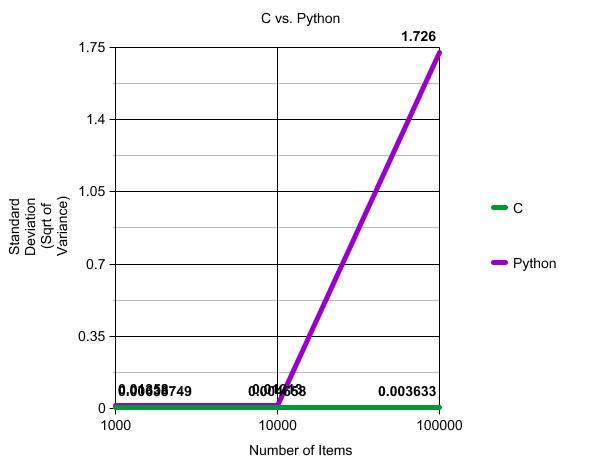


Figure 5.2: Standard deviations of C vs. Python

In addition to having slower runtimes, the Python implementation had a much greater variance in performance, with a standard deviation of 1.726 with a sample size of 100,000, which translates to approximately 3 seconds. Even with this large variation in run times, the very best run with Python was still slower than the C implementation. Given Python's poor and inconsistent performance, the C implementation is clearly the superior option.

We have several theories as to why Python did not perform as well as C. One possible explanation is that C and Python were designed with different goals in mind. C was designed with performance as a priority. Given its use as the basis of UNIX, it has repeatedly proven itself to be both powerful and reliable. Python, on the other hand, was designed to be easy to read and write, making it ideal for writing scripts and other non-performance intensive utilities. Another possibility to consider is the fact that we performed on a Linux machine. Linux is written and based around C. Python, on the other hand, is not native to Linux and requires its own unique interpreter that serves as a separate shell for the Python program.

After conducting our experiment, we sought our other research in order to corroborate our results. Our search led us to a similar experiment conducted by Patrick Braga for his blog, The Unix Geek [4]. While his test programs were not parallel and therefore not directly comparable to ours, he did produce interesting results. According to his research, Python is best suited for quick systems programs, where as C performs much better with programs that require large amounts of calculation or information processing.

**6. Conclusion**

Over all, our experiment was a success. We were able to produce results that clearly demonstrated which parallel programming models and languages are best suited for complex parallel algorithms such as sorting. Between message passing with MPI and shared memory with OpenMP, OpenMP was the clear winner. The OpenMP program ran significantly faster while still maintaining acceptable levels of variance. When comparing C and Python, C had noticeably better time performance and significantly less variance than Python. Our approach provided good data, but the scope of our study provided some limitations. Specifically, we only had time to study radix sort. In the future, it would be interesting to conduct similar experiments with various other parallel problems other than radix sort. As multi-core architectures become more and more common, it would be highly beneficial to establish a guide for new parallel programmers as to which approaches and languages are best for the various tasks and challenges they will face.

**Acknowledgments**

-Dr. Tyler Simon, who taught us literally everything we've ever known about parallel computing.

-UMBC, which provided the Bluewave supercomputer we used to test our programs.

- National Center for Educational Statistics, which has some great publicly available graphing utilities.

- http://www.researchomatic.com/ieee-citation-generator/, a free IEEE citation generator.

**References**

[1] Black, P , "Radix Sort". Retrieved May, 2013 Available: http://xlinux.nist.gov/dads/HTML/radixsort.html

[2] Mobbs, R , "A Brief History of C". Retrieved May, 2013 Available: http://www.le.ac.uk/users/rjm1/cotter/page\_06.htm

[3] van Rossum, G , "The History of Python". RetrievedMay , 2013 Available: http://python- history.blogspot.com/2009/01/

[4] Braga, P , "C vs Python: Speed". RetrievedMay , 2013 Available: http://theunixgeek.blogspot.com/2008/09/c-vs-python-speed.html

**Appendix: Our Code**

Source files publicly available at https://github.com/awooten1/ParallelRadixSort

1) mpi\_radix\_sort.c: MPI implementation of radix sort in C

2) omp\_radix\_sort.c: OpenMP implementation of radix sort in C

3) parallel\_radix\_sort.py: Driver program for the Python implementation of radix sort.

4) merge.py: Python file that provides merge function for parallel\_radix\_sort.py

5) radix\_sort.py: Python file that provides radix sort function for parallel\_radix\_sort.py