StackedNeedles: An Evaluation of HPC Hardware and Software Utilizing Sorting Algorithms

Ethan Lazaro[1], Easton Ingram[2], Marcel Fallet[3], and Jacob Lambert[3]

[1]University of Connecticut
[2]Missouri University of Science and Technology
[3]Department of Defense

August 18, 2023

Abstract

Sorting algorithms are a computational building block of computer science. They are used frequently in processes that organize and analyze data. As humans continue to architect computer systems to handle the analysis of increasing volumes of data, a growing area of interest is optimization of hardware and software for systems that processes data quickly and sort it efficiently. This is of particular importance in the field of network analysis. One particular area of focus has been on comparing the efficiency of parallel sorting algorithms on hardware and software. This paper presents and discusses findings concerning the implementation of efficient parallel sorting algorithms across hardware and software using pthreads, MPI, Chapel, and SHMEM programming methodologies.

1 Introduction

In today's paradigm of high performance computing (HPC), the relentless pursuit of enhancing processing capabilities has paved the way for significant advancements in parallel computing. The ability to execute multiple tasks simultaneously has become a vital asset in performing computationally-intensive operations. As a result, parallel sorting algorithms have emerged as a method with applications in scientific computing, data analysis and artificial intelligence. The pursuit of achieving faster and more efficient sorting techniques has led to the exploration of both hardware and software-based parallel implementations [1].

Sorting algorithms are fundamental techniques used to arrange elements in a specific order within a data structure, such as arrays or lists. Different sorting algorithms exist, each with its strengths and limitations, characterized by their time and space complexity. Common sorting algorithms include Bubble Sort, Selection Sort, Merge Sort, Quick Sort, and Heap Sort. The choice of algorithm depends on factors like input size, data distribution, and performance requirements. Efficient sorting enables optimized data manipulation and enhances overall system performance. Figure 1 displays common sorting algorithms and their time and space complexities.

Sorting Algorithms	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Bubble Sort	Ω(N)	Θ(N^2)	O(N^2)	O(1)
Selection Sort	Ω(N^2)	Θ(N^2)	O(N^2)	O(1)
Insertion Sort	Ω(N)	Θ(N^2)	O(N^2)	O(1)
Quick Sort	Ω(N log N)	Θ(N log N)	O(N^2)	O(N)
Merge Sort	Ω(N log N)	Θ(N log N)	O(N log N)	O(N)
Heap Sort	Ω(N log N)	Θ(N log N)	O(N log N)	O(1)

Figure 1: Comparison of common sorting algorithms.

Hardware-based approaches leverage specialized parallel architectures like graphics processing units (GPUs) in contrast to conventional central processing units (CPUs). On the other hand, software-based comparisons concentrate on factors such as programming languages, libraries, algorithm optimizations, and more. In related work, efforts have been concentrated on applying parallel sorting algorithms to medium scale MIMD (Multiple Instruction, Multiple Data) architecture for parallel computers [4].

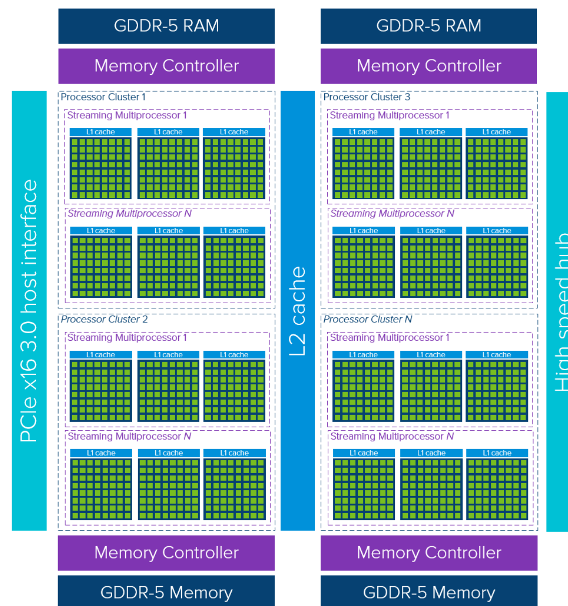

Figure 2: Topology of GPU architecture.

One direct application of sorting algorithms is in the field of network analysis. In network analysis, sorting networks are abstract devices that consist of two items: comparators and wires. The wires run from left to right carrying values and the comparators connect two wires. When the values encounter a comparator, the comparator evaluates whether the value on the top wire is greater than the value on the lower wire. If the condition is met then the comparator swaps the values on the wires. Sorting networks can implemented on hardware or software. Sorting networks differ from other comparison sorts in the way that they are incapable of handling large inputs and their comparison sequence is set up in advance. This has led practitioners to focus on developing parallel sorting algorithms to increase the efficiency and scalability of these networks. Figure 3 shows a simple sorting network.

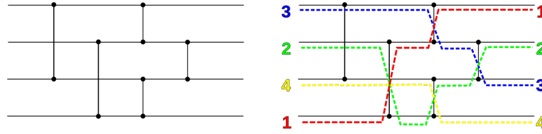

Figure 3: Simple sorting network.

The aim of this paper is to conduct a comparative analysis of parallel sorting algorithms implemented on both HPC hardware and HPC software stacks . By investigating and evaluating the performance, metrics and scalability of these approaches, this paper seeks to discuss their respective advantages, limitations, and potential applications.

2 Hardware Comparison

2.1 Hardware Libraries

2.1.1 POSIX Threads (Pthreads)

POSIX Threads is a shared memory library for multi-threading. A thread is a lightweight process that can be scheduled to be executed simultaneously and/or independently of a program. Furthermore, each thread has access to main memory with other threads. Using the Pthreads library is most effective in multiprocessor and multi-core systems where the process can be scheduled to run on another processor. The POSIX Threads programming model allows the programmer to control the creation, assignment, and termination of threads. Furthermore, the library also has synchronization mechanisms to organize threads for synchronized execution. The library is supported by C/C++.

2.2 Merge Sort Algorithm

Merge Sort is a classical divide and conquer algorithm that begins with a large unsorted array and recursively divides the array down until each sub array is trivially sorted (contains one element). Next, the algorithm compares adjacent elements/sublists and merges them back into one final sorted sub array. This algorithm can be made parallel at either the division phase or the merge phase of the algorithm. Using merge sort with parallel recursion uses threads to handle the subdivisions of the array and then join the threads back at the merge procedure. It is worth noting that this variation does not yield a tremendous speed up compared to the serial version as a bottleneck occurs at the merge procedure which is executed serially. One notable advantage of Merge Sort is its running time which is $O(n \log n)$. This is significant because other sorting techniques such as Bubble or Insertion Sort have a running time of $O(n^2)$ which can scale poorly and negatively impact the performance of a system. Another notable advantage of Merge Sort is its ability to perform on large datasets. This is important because scalability is a crucial metric for systems and sorting techniques need to be able to process large volumes of data. Conversely, a particular drawback to Merge Sort is that it uses an auxiliary array to perform its comparisons before merging back into the original array. Therefore there is a trade-off between the speed and space necessary to run the algorithm. Certain variations of Merge Sort try to reduce the space complexity while maintaining the speed [3].

```
   Input: A, p, r
   Output: sorted array A′
 1 if p ≥ r then
 2 |  return;
 3 end
 4 midpoint := (p + r)/2 ;                                  /* Compute the middle index.  */
 5 spawn P-Merge-Sort(A, p, midpoint) ;                /* Recursively sort in parallel.  */
 6 spawn P-Merge-Sort(A, midpoint + 1, r);
 7 sync ;                                  /* Wait for threads to complete execution.  */
 8 Merge(A, p, midpoint, r) ;                                /* Call Merge subroutine.  */
```

Figure 4: Parallel Merge Sort Psuedocode.

```
   Input: A, p, midpoint, r
 1 l-sub := midpoint − p + 1 ;     /* Compute lengths for left and right subarrays.  */
 2 r-sub := r − midpoint;
 3 LeftArr[l-sub], RightArr[r-sub];
 4 for i = 0, 1, ... n-1 in l-sub do
 5 |  LeftArr[i] = A[p + i];
 6 end
 7 for j = 0, 1, ... n-1 in r-sub do
 8 |  RightArr[j] = A[midpoint + 1 + j];
 9 end
10 i = 0, j = 0, k = p ;              /* Initialize pointers to maintain current index.  */
11 while i < l − sub and j < r − sub do
12 |  if LeftArr[i] ≤ RightArr[j] then
13 |  |  A[k] = LeftArr[i];
14 |  |  i++;
15 |  end
16 |  else
17 |  |  A[k] = RightArr[j];
18 |  |  j++;
19 |  end
20 |  k++;
21 end
22 while i < l − sub do
23 |  A[k] = LeftArr[i];
24 |  i++;
25 |  k++;
26 end
27 while j < r − sub do
28 |  A[k] = RightArr[j];
29 |  j++;
30 |  k++;
31 end
```

Figure 5: Merge Psuedocode.

2.3 Environment

Comparisons between POSIX Thread implementations were completed on the Sherlock computing cluster. The Sherlock cluster features 32 compute nodes, and the first 16 nodes are equipped with 2 nVidia Tesla V100 GPUs. A summary of the architecture is shown in Fig. 6. Each node has 1 Terabyte of memory available. For the purpose of the study, experiments were launched on nodes 17-32 on the

Sherlock cluster. Figure 7 displays the topology of a single compute node for the Sherlock cluster. The data to be sorted are 64-bit unsigned integers. The pseudorandom generator used is a 64-bit Mersenne Twister written by Makoto Matsumoto and Takuji Nishimura. This is identical for all runs.

number of nodes	32 (not including login node)
node hostnames	node01, node02, ..., node32
RAM per node	512GB DDR4
Storage per node	1TB mounted to /local
CPUs per node	2x Intel Xeon E5-2650 v3 @ 2.30GHz
Cores per node	20 (10 per CPU)
Node interconnect	Infiniband FDR 56 Gbit/s
Node topology	All nodes connected to Infiniband switch
GPUs per node	2x NVIDIA Telsa V100 on node01–node16, 2x NVIDIA Telsa k40 node17–node32
GPU interconnect	PCIe x16 Gen 3.0 on nodes, Infiniband FDR between nodes

Figure 6: Sherlock cluster specifications.

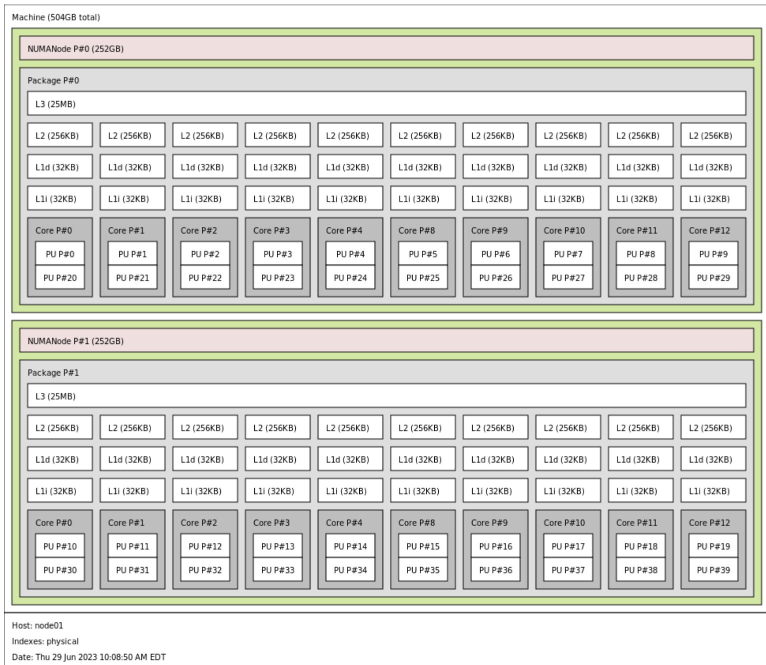

Figure 7: Topology of a compute node on Sherlock cluster.

2.4 Results and Discussion

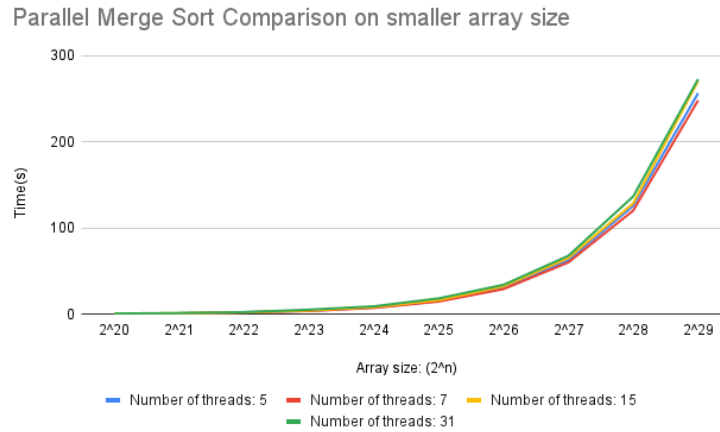

Figure 8: Pthread Parallel Merge Sort with small array sizes.

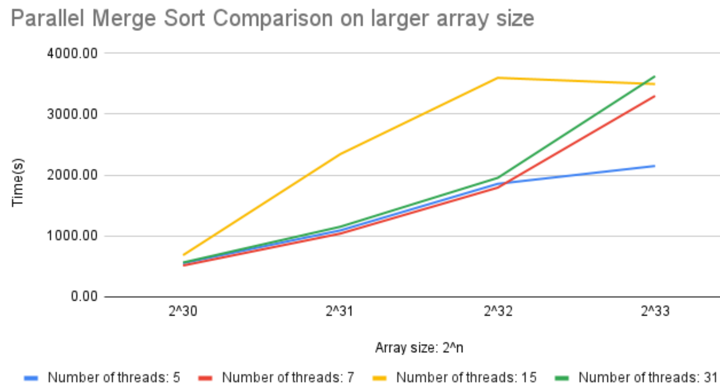

Figure 9: Pthread Parallel Merge Sort with large array sizes.

Size	Time(s)	Merge sublist size	Number of threads
2^30	1.564	2	8160
2^30	0.59537	3	4084
2^30	1.892	4	2039
2^30	0.59213	5	1020
2^30	0.58627	6	511

Figure 10: Pthread Parallel Merge Sort with merge subarray size mechanism.

Figure 8, displays the Pthread Parallel Merge Sort implementation with smaller array sizes. At smaller array sizes each number of threads sorted within a tenth of a second each other. The difference in execution time became more pronounced when the array size reached 2^{28} and 2^{29} of 64-bit integers. At

2^{29} elements, sorting the array with 5 threads executed in 256.52 seconds while sorting the array with 31 threads finished executing in 272.72. Figure 9, shows similar trends to figure with smaller array sizes. At array sizes of 2^{30}, 2^{31}, 2^{32}, 2^{33} using 15 threads completed execution in 679.72/ 2343.70/ 3597.89/ 3496.40 seconds respectively. At 31 threads a run time of 558.26/ 1147.92/ 1951.77/ 3625.25 seconds respectively was seen. A potential consideration for both is further optimizing the algorithm. To be more specific, the parallel merge sort algorithm implementation includes the built-in quick-sort primitive in the C language which might be the reason for the slow down. Figure 10 displays a table for a Pthread implementation that includes a mechanism to increase the sublist size during the merge procedure. This feature increases the performance of the algorithm by spawning less threads thereby increasing the efficiency of the program. For an array size of 2^{30} and a sublist size of 65536^2 the program spawned 8160 threads and completed execution in 1.564 seconds. In comparison a sublist size of 65536^3 produces 4084 threads and completes execution in 0.595 seconds. The correlation between the thread creation and the sublist size is based on the idea that less threads are needed when the sublist size are large. There is less work for the threads to do. Conversely, when the sublist size is small then there is more work for the threads to execute therefore more threads are spawned. A particular drawback of this feature is that a system can suffer from a performance struggle due to the thread creation overhead. Thread creation overhead is a system dependent issue where the process of allocating memory and resources become costly due to thread creation and consumption. Managing thread creation, assignment, and execution is tantamount to achieving an efficient parallel program.

3 Software Comparison

3.1 Programming Languages/Libraries Overview

3.1.1 Chapel

Chapel is a programming language that with the focus of making scalable parallel computing more productive. It aims to be as programmable(easy to write/read code) as Python with speeds comparable or faster than MPI. It is also open-source.

The example code from the sample sort implementation included in Fig. 11 highlights the simplicity of parallelization using Chapel. Since the parallelism is done by the forall statement, no explicit communication routines are called by the programmer.

```
// Sort each bucket
forall bucket in 0..num_buckets-1{
    const b_size = bucket_sizes[bucket];
    quickSort(temp[bucket, 0..#b_size], 0, b_size -1);
}

// concatenate buckets back into source
forall bucket in 0..num_buckets-1{
    var j: int = 0;
    for i in 0..bucket-1{
        j+=bucket_sizes[i];
    }

    for elem in temp[bucket,0..#bucket_sizes[bucket]]{
        arr[j] = elem;
        j+=1;
    }
}
```

Figure 11: Chapel parallel sample sort code snippet.

3.1.2 MPI

MPI, or Message Passing Interface is a widely used library in the HPC community. Many examples, benchmarks, and tutorials exist for MPI (as such, example code is omitted for brevity). MPI used point-to-point communication (sends and receives).

3.1.3 SHMEM

SHMEM is a shared memory library that is available for C or Fortran. It uses the PGAS (partitioned global address space) paradigm. SHMEM is reported to have several benefits over MPI, such as making one-sided communication calls, which reduces the time needed for data to transfer between processing elements. Shown in Fig. 12, the code snippet from the SHMEM sample sort implementation demonstrates this functionality.

```
// Get bucket_sizes and data partition from PE 0
shmem_broadcast64(bucket_sizes, bucket_sizes, num_buckets,
    0, 0, 0, np, pSync);
shmem_ulong_get(temp[id], temp[id], size, 0);

// Sort each bucket
quickSort(temp[id], 0, bucket_sizes[id]-1);

// concatenate buckets back into source
int64_t j=0;
for (int64_t i=0;i<id;i++){
    j+=bucket_sizes[i];
}
for (int64_t i=0;i<bucket_sizes[id];i++){
    arr3[j] = temp[id][i];
    j++;
}

shmem_barrier_all();
shmem_long_sum_to_all(arr, arr3, size,
    0, 0, np, pWrk, pSync);
```

Figure 12: SHMEM parallel sample sort code snippet.

3.2 Parallel Sample Sort Algorithm

A popular sorting algorithm that is easily made parallel is bucket sort. The data to be sorted is divided and placed in assigned buckets, and each bucket is sorted in parallel using a conventional sort. Following this, the contents of each bucket are put back into the array, giving a sorted sequence. However, bucket sort falls short when the unsorted data is not uniformly distributed. Differing bucket sizes may cause the load on each processing element to be unbalanced, resulting in slow speeds.

Sample sort remedies the downfalls of bucket sort by uniformly distributing the data to the buckets. Sample sort was introduced in 1970 by W. D. Frazer and A. C. McKellar [2]. The algorithm takes a few random samples from the data to form splitters with the goal of evenly distributing the data across all buckets. After this, the algorithm is similar to bucket sort where each bucket is sorted in parallel using quick sort, and each bucket segment is placed back into the array.

Sample sort typically includes an oversampling factor for tuning the distribution. This number is the amount of random numbers sampled for each bucket. In this paper, the oversampling factor was consistently set to a value of 100. A study conducted by varying this factor would be necessary before deployment.

Sample sort has the same time complexity as bucket sort, with a best case of O(n), and a worst case of O(n^2). Sample sort attempts to evenly distribute the data amongst the buckets and limit

worst case scenarios. The downfall of sample sort is that it is quite memory intensive, as the data is copied into the temporary buckets to be sorted.

3.3 Environment

Comparisons done between Chapel, MPI, and SHMEM were completed on kruskal nodes 1-16. Kruskal features 128 compute nodes where each node has 1 TB memory available as shown in Fig. 13. Figure 14 displays the topology of a single kruskal compute node. Each bucket was assigned to a single processing element/task.

number of nodes	128 (not including login nodes)
node hostnames	n1, n2, ..., n128
RAM per node	1TB DDR4
Storage per node	1TB NVMe mounted to /local
CPUs per node	2x AMD EPYC 7713 (Milan) @ 2.0GHz
Cores per node	128 (64 per CPU)
Node interconnect	Infiniband HDR 200 Gb/s
Node topology	Fat tree
Fast storage	100TB Lustre file system connected to compute nodes via Infiniband HDR 100 Gb/s. Mounted at /scratch/users/username
GPUs per node	None

Figure 13: Kruskal cluster specifications.

The data to be sorted are 64-bit unsigned integers. The pseudorandom generator used is a 64-bit Mersenne Twister written by Makoto Matsumoto and Takuji Nishimura. Unsorted data is identical for all runs.

Figure 14: Topology of Node on Kruskal Cluster.

3.4 Results and Discussion

Figure 15 displays Chapel, SHMEM, and MPI parallel sample sort comparisons are presented using four buckets to partition the data. At a 2^{27} 64-bit integers, SHMEM took 4.888766 seconds to sort

the data, which outperforms both MPI, which took 7.798851 seconds, and Chapel, which took the longest time of 16.3981 s. At a size of 2^{30}, SHMEM completed in 38.914526 s and Chapel in 154.538 s, approximately four times slower.

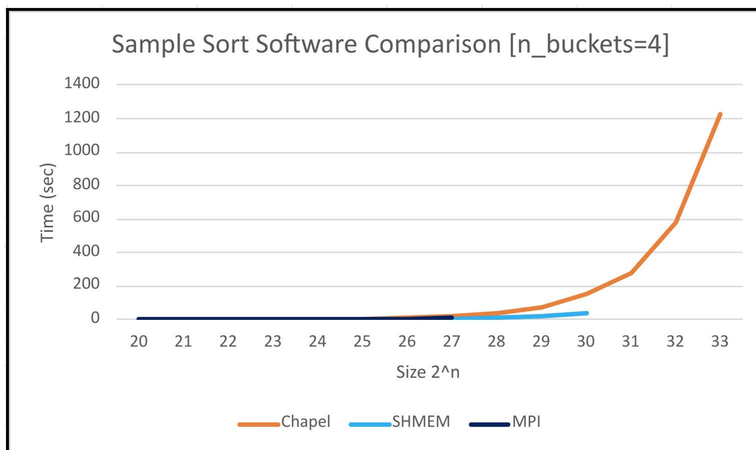

Figure 15: Chapel, SHMEM, MPI sample sort comparisons with n_buckets=4

Figure 16 shows the parallel sample sort software comparisons using eight buckets. Trends similar to the four bucket comparisons are seen.

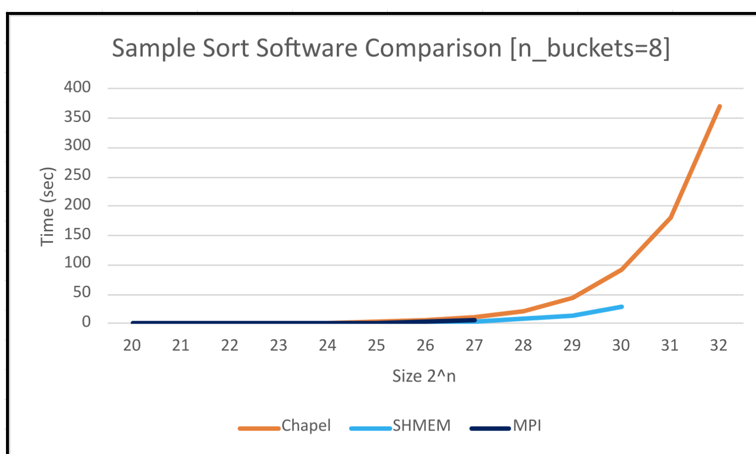

Figure 16: Chapel, SHMEM, MPI sample sort comparisons with n_buckets=8

Figure 17 shows that SHMEM(3.3143 s) was faster than both MPI(8.4029 s) and Chapel(7.3764 s) at a size of 2^{27} for 16 buckets. SHMEM was only two times quicker than Chapel. As the number of buckets increases, Chapel becomes more comparable with SHMEM. Chapel may have increased overhead with small sizes of data. Because of this, different comparison speeds might be seen with increased sizes.

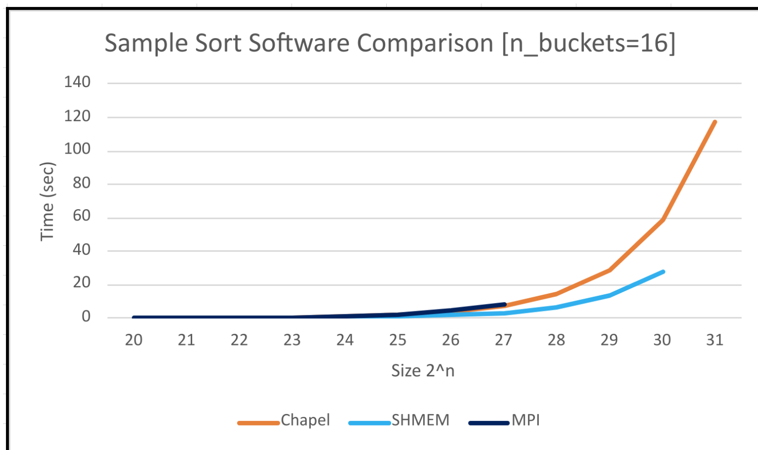

Figure 17: Chapel, SHMEM, MPI sample sort comparisons with n_buckets=16

At large numbers of 64-bit integer sorting, a significant speedup is seen by increasing the number of tasks/PEs as expected. For example, Chapel sample sort with 4/ 8/ 16 buckets and 2^{30} integers took 154.538/ 91.1531/ 59.1732 seconds respectively.

As shown in Figures 15, 16, 17, much higher scalability was achieved using Chapel over SHMEM and MPI. Data sizes greater than 2^{27} were not achieved with MPI due to size limits on communication routines, but Chapel was able to sort up to 2^{33} 64-bit integers on a single locale (much larger sizes can be run across multiple locales at the cost of extended runtime). It is imperative that the languages/libraries have the capability to sort larger data sizes. Despite the higher runtime, Chapel is easily scalable.

4 Conclusion

So far, a comparative analysis of hardware and software configurations is presented and an investigation of which architectural configurations would yield higher performance. This work implemented two types of parallel sorting algorithms, merge and sample sort, and compared it across hardware and software.

Comparisons between pthread implementations are presented by evaluating the efficiency and scalability of a parallel merge sort implementation. When the number of threads were increased, the performance suffered due to thread management and execution overhead. When the array size was smaller than 2^{30} the program executed faster with seven threads than with 15 threads. In comparison when the number of threads was 31, the program executed slightly faster than when the number of threads was 15. This comparison demonstrates that threading becomes more impactful when the number of tasks becomes very large.

Additionally, a Pthread implementation is presented with a feature that spawns threads based on the subarray size during the Merge procedure. This feature demonstrates the importance of understanding the cost thread creation. Reducing the number of threads needed to complete a task contributes to reducing the thread overhead while increasing the efficiency of the program.

Comparisons between Chapel, SHMEM, and MPI are presented by evaluating the performance of a parallel sample sort implementation for each. When the number of buckets for partitioning and sorting the data was four and eight, MPI was slightly faster than SHMEM and approximately four times as fast as Chapel. When the number of buckets was increased to 16, SHMEM was slightly quicker than MPI and roughly three times as fast as Chapel. Further investigations comparing Chapel, SHMEM, and MPI with various sorting algorithms are needed to establish a definitive performance metric.

It was shown that scalability is a crucial property when selecting software for sorting algorithms. Chapel is easily scalable, especially when compared to SHMEM and MPI. Chapel is the preferred choice for sorting larger sizes of data.

Additionally, the simplicity of parallelization available with Chapel is a desirable feature. Comparing the SHMEM code snippet to the Chapel code snippet shown above highlights the potential increase in productivity with Chapel.

Future research into comparing POSIX Threads implementations with other thread libraries such as OpenMP on CPUs and GPUs will contribute to developing a deeper understanding of executing multi-threading programs on CPUs and GPUs. Another area of interest could be comparing sorting implementations with the Rust programming language with C. This future study can lead to conclusions about how each language handles multithreading efficiency and scalability. Another potential research direction is investigating communication and cache optimizations with applications in developing novel sorting algorithms.

References

[1] Alkharabsheh, K., Alturani, I., Alturani, A., and Zanoon, D. Review on sorting algorithms a comparative study. *International Journal of Computer Science and Security (IJCSS) 7* (01 2013).

[2] Frazer, W. D., and McKellar, A. C. Samplesort: A sampling approach to minimal storage tree sorting. *J. ACM 17*, 3 (jul 1970), 496–507.

[3] Knuth, D. *The Art of Computer Programming Volume 3: Searching and Sorting*. Addison-Wesley Professional 1998, 1973.

[4] Tridgell, A., Brent, R. P., and McKay, B. D. Parallel integer sorting. Tech. rep., 1995.