# Sample Sort using the Standard Template Adaptive Parallel Library

Jessie Berlin, Gabriel Tanase, Mauro Bianco, Lawrence Rauchwerger, and Nancy M. Amato
Parasol Lab, Dept. of Computer Science, Texas A&M University, College Station, TX 77843
{jberlin, gabrielt, bmm, rwerger, amato}@cs.tamu.edu

**Abstract**

The Standard Template Adaptive Parallel Library (STAPL) is a parallel library designed to make developing software that takes advantage of parallelism on multi-processor machines easier. Written to be a superset of the Standard Template Library (STL) for C++, STAPL is intended to be straightforward to use, efficient, and portable. Since STAPL's core library consists of ISO Standard C++ components, it must provide the ability to sort.

We present a parametrized Parallel Sample Sort algorithm, developed with STAPL, that provides a choice in the method of sampling, the over-sampling ratio, and the over-partitioning ratio. Unlike many other Sample Sort algorithms, it covers the situation where the load on each processor is not balanced at the beginning of the algorithm without requiring an initial re-balancing step. In addition, we evaluate the general scalability and the conditions under which two versions of the parametrized Parallel Sample Sort algorithm perform well and poorly.

This research was supported in part by the Computing Research Association CRA-W Distributed Mentor Program

# 1 Introduction

Putting items in order is an important task in Computer Science. Hence, sorting is one of the most studied topics in field. Sorting has computational complexity $O(n \log n)$, and much research has been done to speed up sorting by using multiple processors. The emergence of multi-processor machines into the general market has given added importance to the pursuit of fast and efficient parallel sorting algorithms.

We analyze a version of the well known parallel sorting algorithm called Sample Sort. The general idea behind Sample Sort is to break down the work among multiple processors and have each processor sort its portion of the data.

There are three basic steps in the general Sample Sort Algorithm:

1. Find splitters, values that break up the data into buckets, by sampling the local data on each processor.
2. Use the sorted splitters to define buckets on the different processors and place the data in the appropriate buckets.
3. Sort each of the buckets.

We implemented the parametrized Parallel Sample Sort using STAPL (Standard Template Adaptive Parallel Library) [1], a superset of the STL (Standard Template Library) for C++ that is designed to make programming for parallel machines simpler.

In this paper, we discuss previous research on Sample Sort (Section 2), provide an overview of STAPL (Section 3), explain the details and implementation of the parametrized Parallel Sample Sort (Section 4), examine the experimental data (Section 5), review the conclusions drawn from the results (Section 6), and outline future work (Section 7).

# 2 Related Work

Finding the most efficient version of Sample Sort is an often studied problem. Though there has been little variation in the basic steps of the algorithm, many different approaches in the implementation of these steps have been developed in order to improve the running time and the load balance.

In order to achieve good load balance, Shi and Shaeffer [7] use *Regular Sampling* to pick their splitters. The local data on each processor is sorted and $p-1$ samples, where $p$ is the number of processors, are picked, evenly spaced, from each processor. Next, $p(p-1)$ samples are sorted. Finally, $p-1$ splitters are picked, evenly spaced, from the samples. Thus, the improvement in load balance is achieved by allowing the first step of selecting the splitters to be $O(n \log n/p)$, where $n$ is the number of data elements in the input. The parametrized Parallel Sample Sort Parallel Sample Sort also utilizes Regular Sampling, but does not sort the data on each processor before picking the splitters.

Blelloch et. al [2] partition each processor's $n/p$, elements into $s$ blocks of $n/sp$ elements and take one element, chosen randomly, from each block to form the samples. They also use Regular Sampling to pick the splitters from the samples. They prove that the probability is high that the ratio of the maximum bucket size to the average bucket size before the final sorting step is small. However, they also start completely from scratch, picking new samples, if any one of the buckets is too big.

Li and Sevcik's method, Parallel Sorting by Over Partitioning (PSOP) [4], involves picking $pk-1$ splitters, where $k$ is the over partitioning ratio. It creates $pk$ buckets in order to ensure that the maximum size of any bucket is small enough to result in good load balancing. The parametrized Parallel Sample Sort offers the user the option of over partitioning, but, unlike PSOP, does not sort the buckets on each processor in the order of decreasing size.

# 3 STAPL Overview

STAPL is designed to be a parallel superset of the Standard Template library (STL). It provides a set of C++ parallel container classes and parallel template algorithms that can be combined to provide different functionality. STAPL makes it easier for the programmer accustomed to C++ to program for distributed parallel machines, being C++ based.

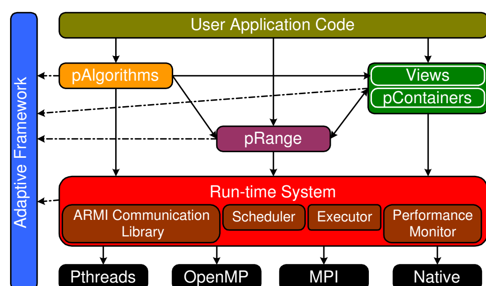



Figure 1: STAPL Infrastructure

STAPL consists of a set of components that include `pContainers`, `pAlgorithms`, `views`, `pRanges`, and a runtime system (see Figure 1).

`pContainers`, the distributed counterpart of STL containers, are thread-safe, concurrent objects, i.e., shared objects that provide parallel methods that can be invoked concurrently. While all `pContainers` provide *sequentially equivalent interfaces* that are compatible with the corresponding STL methods, individual `pContainers` may introduce additional methods to exploit the performance offered by parallelism and by the runtime system.

`pContainer` data can be accessed using `views` which can be seen as generalizations of STL iterators that represent sets of data elements and are not related to the data's physical location. `views` provide `iterators` to access single elements of `pContainers`. Generic parallel algorithms (`pAlgorithms`) are written in terms of `views`, similar to how STL algorithms are written in terms of iterators. A `view` can have subviews, views over smaller portions of the `pContainer`.

The `pRange` is the STAPL concept used to represent a parallel computation. Intuitively, a `pRange` is a task graph, where each task consists of a work function and a `view` representing the data on which the work function will be applied. The `pRange` provides support for specifying data dependencies between tasks that will be enforced during execution.

The runtime system (RTS) and its communication library ARMI (Adaptive Remote Method Invocation [6]) provide the interface to the underlying operating system, native communication library and hardware architecture. ARMI uses the remote method invocation (RMI) communication abstraction to hide the lower level implementations (e.g., MPI, OpenMP, etc.). A remote method invocation in STAPL can be blocking (`sync_rmi`) or non-blocking (`async_rmi`).

# 4 Parametrized Parallel Sample Sort

Before explaining how the parametrized Parallel Sample Sort works, it is important to define how a few terms are used:

**Threads** are processors. The number of threads is denoted by $t$. It is assumed that there is a one-to-one ratio between the number of subviews and the number of threads, and that the input is distributed approximately evenly across the threads, though it is possible for there to be a greater or lesser number of subviews than threads. Also, the parametrized Parallel Sample Sort can handle the case where the data is not evenly distributed across the threads, such as the case where one or more threads may have zero data elements.

**Buckets** are STAPL subviews. The number of buckets is denoted by $j$. The *over partitioning ratio* $k$ defines the integer number of buckets per thread, such that $j = tk$.

**Samples** are values drawn from the data set in order to provide a good representation of the distribution of the data. The *over sampling ratio* $s$ defines the desired number of samples to be selected from each thread, for a total number of samples $ts$. If a thread has less than $s$ local data elements to contribute as samples, it contributes all if its local data elements as samples.

**Splitters** are the $j - 1$ values drawn from the sorted samples that define the $j$ buckets such that every data element in bucket $i$ is less than splitter $i$, for every $i < j$.

The parametrized Parallel Sample Sort (Figure 2) consists of four main steps:

1. Sample the local data on each thread and sort the samples.

2. Select splitters from the samples.

```
1 sample_sort(N,T,S,K){
2  globally, create the p_array of samples(view_data, TS, p_min_max_element(view_data))
3  locally, select s samples from each subview's data using the sampling method
4  globally, sort the samples
5  globally, create the p_array of splitters(TK-1)
6  globally, select TK-1 splitters from the samples
7     step = (samples.size()- p_count(view_data, p_min_max_element(view_data)))/ TK
8     for(i = 0; i < TK-1; i++) {
9         samples_iterator+=step;
10         *splitters_iterator=*samples_iterator;
11         splitters_iterator++;
12     }
13 p_n_partition(view_data, view_splitters);
14 globally, redistribute the buckets (redistribute version only)
15     create a temporary pContainer aligned with the view over the original pContainer
16     p_copy(view_data, view_temporary);
17 locally, sort each bucket (of the view_temporary - redistribute version only)
18 p_copy(view_temporary, view_data); (redistribute version only)
19 }
```

Figure 2: Pseudo-code of the parametrized Parallel Sample Sort. N is the size of the input, T is the number of threads, S is the over sampling ratio, and K is the over partitioning ratio

3. Use the sorted splitters to define buckets on the different threads and place the data in the appropriate buckets.

4. Sort each of the buckets.

### 4.1 Sampling the Data

The parametrized Parallel Sample Sort allows the user to define the over sampling ratio $s$. If the user does not provide an over sampling ratio, a default of 128 is used.

The parametrized Parallel Sample Sort creates a pArray of samples of size $ts$. Each element in the pArray of samples is initialized to value of the maximum element found in the original data set.

The parametrized Parallel Sample Sort also allows the user to select the *Sampling Method* (Figure 3), the method by which the samples are chosen from the local data on each thread. The options for the Sampling Method are as follows:

1. Even (default) - pick evenly spaced samples using a *step* of $l/s$, where $l$ is the number of elements on the thread.

2. Semi-Random - pick approximately evenly spaced samples using a *step* of a random number between 1 and $l/s$.

3. Random - pick the samples randomly from the data on the thread.

4. Block - pick the sequential first $s$ data elements on the thread.

In the case where a thread has fewer local data elements than $s$, it contributes all of its local data elements as samples.

Locally, each of the threads chooses $s$ samples using the chosen Sampling Method and places them in the corresponding part of the aligned pArray of samples. If a thread has less than $s$ elements, it places all its elements in the corresponding part of the aligned pArray of samples.

The samples are then sorted globally.

### 4.2 Selecting the Splitters

Before selecting the splitters, the parametrized Parallel Sample Sort counts the number of times the maximum element appears in the samples and subtracts that number from the size of the pArray of samples, to find the number of actual samples $a$.

This prevents the selection of the maximum value as a splitter. If the maximum value was chosen as a splitter, it would create an empty bucket since there would be no data elements with values greater than the splitter to place in the bucket.

The ideal is to pick splitters that separate the data into $j$ buckets of size $n/j$ in order to achieve load balancing. Load balancing means that every bucket is equal in size, so that no one bucket takes longer than any other bucket to be sorted.

The splitters are chosen using Regular Sampling [7] from the samples by stepping through the actual sorted samples using a step of size $a/j$, such that the splitters are values at the $a/j^{th}$, $2a/j^{th}$, ..., $(j-1)a/j^{th}$ positions in the pArray of samples.

### 4.3 Partitioning the Data

In order to create buckets defined by the splitters and place the data elements in the correct buckets, the parametrized Parallel Sample Sort uses `p_n_partition`.

`p_n_partition` is a parallel algorithm in STAPL that takes a view over the data and a view over the $j-1$ splitters and creates $j$ buckets, or to be more exact subviews, such that each element in bucket $i$ is less than splitter $i$, for all $i < j$. The elements in the $j^{\text{th}}$ bucket are greater than the $i^{\text{th}}$ splitter.

Because sending the elements to their correct buckets is done in `p_n_partition` the call to `p_n_partition` is the most expensive step of the parametrized Parallel Sample Sort, since it involves a large amount of communication among threads.

Due to the fact that the subviews that result from the call to `p_n_partition` may no longer be aligned with the underlying pContainer, some of the data elements in a few of the buckets will be remote, residing in different locations than the bucket. When these buckets are to be sorted, this proves to be problematic, since accessing remote data elements is expensive, requiring both read and write communication between threads.

Therefore, one version of the parametrized Parallel Sample Sort has a redistribution step following the call to `p_n_partition`. This involves creating a new pContainer aligned to the view that resulted from `p_n_partition`, copying the data from the view that resulted from `p_n_partition` into a view over that pContainer, and sorting the buckets, or subviews to be more exact, of the new pContainer.

The experimental results for tests run with second version of the parametrized Parallel Sample Sort, without redistribution, may also be found in Section 5.

### 4.4 Sorting the Buckets

Each thread sorts its buckets using a sequential sort.

For the version of the parametrized Parallel Sample Sort with redistribution, there is one final step of copying the view of the temporary pContainer back into the view over the original data elements.

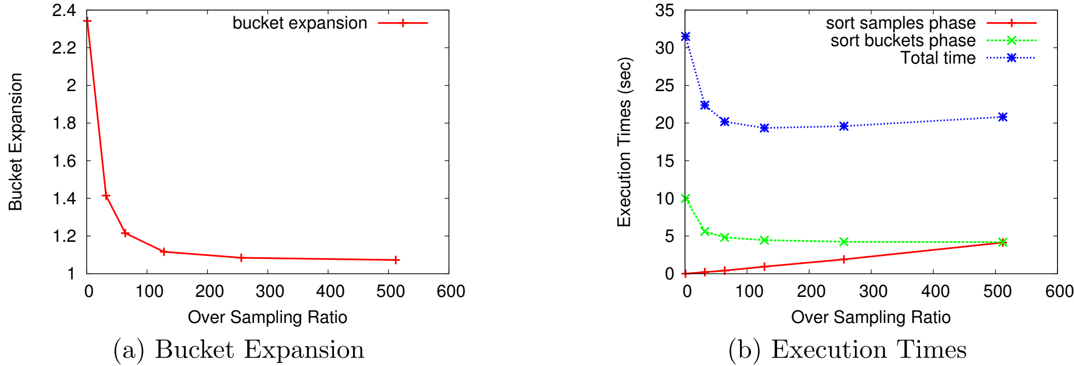
Due to the nature of views, there is no need for a final merge step, where the buckets are placed in order. Since the buckets are subviews, with distinct order within the view, and the buckets are defined based on sorted splitters, the buckets are already in order and the sort is complete.

## 5 Experimental Results

The parametrized Parallel Sample Sort allows for variation in three main parameters: the over sampling ratio, the sampling method, and the over partitioning ratio. We tested different over sampling ratios and different over partitioning ratios, as well as the general scalability.

### 5.1 Architecture Used

We evaluated the performance of the parametrized Parallel Sample Sort on a 640-processor IBM Cluster system. There are 40 p575 nodes, each containing 8 dual core IBM 1.9GHx Power5+ chips, and each has




(a) Bucket Expansion (b) Execution Times

Figure 4: Bucket expansion (a) and execution times (b) for an input of 32 million random integers with an over-partitioning ratio of 1 as the over-sampling ratio is varied from 1 to 512.

a shared memory of 32GB. The nodes are connected together by IBM High-Performance Switches (HPS). We used GNU GCC v4.1.1 and its corresponding STL implementation.

## 5.2 Over Sampling

When sampling the data, the goal is to get a good representation of the distribution of the data elements. That way, the splitters have a high chance of dividing the data into roughly equal sized buckets, resulting in no bucket taking much longer than any other bucket to sort in the last step of the parametrized Parallel Sample Sort.

The over sampling ratio determines how many data elements to pull from each thread as samples. In cases where the data values are widely distributed, such that there are not many duplicates of a value grouped together, it may not be necessary to take many samples from each thread to get values that represent the variation of values in the data set. In such a case, the over sampling ratio would not need to be very large.

In cases where there are many duplicates or the data elements are very close in value, it might be necessary to take more samples from each thread in order to get a good representation of the data. It would therefore be necessary to use a larger over sampling ratio.

It is also important to note the samples are sorted with a sequential sort before the splitters can be chosen. Higher over sampling ratios lead to longer times for the sorting samples step since number of samples to sort is equal to the over sampling ratio multiplied by the number of threads. This must be taken into consideration when using a higher over sampling ratio.

One way of measuring the effect of the oversampling ratio is to measure *bucket expansion*. Bucket expansion is ratio of the largest bucket to $n/j$ after `p_n_partition`. The ideal splitters would partition the data into $j$ buckets of size $n/j$, which would result in a bucket expansion of 1 and would mean that no one bucket is taking longer to sort than any other bucket. A high bucket expansion means that the splitters were poorly chosen, creating at least one disproportionately big bucket that takes longer to sort than the other buckets and results in a longer sorting time.

Figure 4 shows the bucket expansion (a) and the total execution times (b) for the parametrized Parallel Sample Sort run with different over sampling ratios from 1 to 512. For an input of 32 million random integers, generated with the same seed, a constant over partitioning ratio of 1, and 16 processors, the bucket expansion decreases as the over sampling ratio increases, starting to level off at 128.

The total time for the parametrized Parallel Sam-

ple Sort decreases as the the bucket expansion ratio is increased until in reaches 128, then increases as the bucket expansion rate is further increased. This is due to the increase in time for the sort samples phase as the over sampling ratio is increased. The sort buckets phase also begins to level off at an over sampling ratio of 128. Thus, 128 is the default for the over sampling ratio that we use the later tests.

## 5.3 Sampling Method

We tested the four Sampling Methods that the parametrized Parallel Sample Sort offers in order to determine if different Sampling Methods worked better or worse when sorting various data inputs. The four types of inputs we tested were Random 32 bit integers, integers generated using the `rand`() function, Duplicates 1, integers generated using `rand`() mod100000, Duplicates 2, integers generated using `rand`() mod100, and Sorted, where the data on each processor is initially sorted.

Tables 1 and 2 show the results for the total times for two different sets of runs, both sets of runs on 16 processors and with 16 million data elements, but with different seeds for the `rand`() function. Tables 3 and 4 show the results for the bucket expansions for the two sets of runs.

| | Even | Semi-Rand | Rand | Block |
|---|---|---|---|---|
| R | 21.4667 | 22.0424 | 20.9867 | 19.4635 |
| D1 | 21.695 | 21.9043 | 20.9742 | 20.0872 |
| D2 | 22.6771 | 23.1119 | 23.5013 | 24.1623 |
| S | 18.5255 | 88.0982 | 170.127 | 170.42 |

Table 1: Sampling Method: Total Time, seed = 1

| | Even | Semi-Rand | Rand | Block |
|---|---|---|---|---|
| R | 20.0655 | 20.4415 | 20.1627 | 21.4645 |
| D1 | 20.3001 | 20.0067 | 19.823 | 21.2529 |
| D2 | 20.9006 | 20.7069 | 20.916 | 21.5928 |
| S | 18.3416 | 86.9277 | 173.297 | 170.544 |

Table 2: Sampling Method: Total Time, seed = 2

Tables 1 and 2 show that the best total running time for the Random 32 bit integers data for the first seed occurs with the blocked Sampling Method, but

| | Even | Semi-Rand | Rand | Block |
|---|---|---|---|---|
| R | 1.11623 | 1.1327 | 1.15027 | 1.14967 |
| D1 | 1.11623 | 1.12752 | 1.15152 | 1.14967 |
| D2 | 1.12056 | 1.28013 | 1.3853 | 1.43847 |
| S | 1.12493 | 8.75658 | 15.7534 | 15.999 |

Table 3: Sampling Method: Bucket Expansion, seed = 1

| | Even | Semi-Rand | Rand | Block |
|---|---|---|---|---|
| R | 1.20558 | 1.16455 | 1.16091 | 1.12941 |
| D1 | 1.20558 | 1.19115 | 1.19454 | 1.12941 |
| D2 | 1.27857 | 1.22625 | 1.27857 | 1.27857 |
| S | 1.12493 | 8.55824 | 15.7541 | 15.999 |

Table 4: Sampling Method: Bucket Expansion, seed = 2

for the second seed occurs with the even Sampling Method. For the Duplicates 1 data, the best time with the first seed also occurs with the blocked Sampling Method, but with the second seed occurs with the random Sampling Method. For the Duplicates 2 data, the best time with the first seed occurs with the even Sampling Method, but with the second seed occurs with the semi-random Sampling Method. For the Sorted data, the best times for both seeds occur with the even Sampling Method.

The same lack of a clear trend to indicate the best Sampling Method for a given data input is evident in the bucket expansions. For all four data inputs using the first seed, the lowest bucket expansions occur when the even Sampling Method is used. However, when the second seed is used, the Duplicates 1 and 2 inputs incur the lowest bucket expansions when the semi-random Sampling Method is used, but the Random 32 bit integers input has the lowest bucket expansion when the blocked Sampling Method is used and the Sorted input has the lowest bucket expansion when the even Sampling Method is used.

Due to the fact that huge jumps with the STAPL iterators are costly because the data they iterate over is not guaranteed to be stored sequentially in memory, the semi-random and random Sampling Methods take longer than the even and block Sampling Methods. The block Sampling Method is the fastest. However, we chose the default Sampling Method, and the one used in later tests, to be the even Sampling

Method, since the block Sampling Method resulted in worse total times for the Duplicates 2 and the Sorted inputs.

## 5.4 Over Partitioning

The theory behind over partitioning is twofold. First if the over partitioning ratio is increased and more buckets are created, each thread should be running k sequential sorts, where k is the over partitioning ratio, then the complexity of the sorting step for each thread should be $O((n/p)\log(n/pk))$ instead of $O((n/p)\log(n/p))$ in the ideal case where the size of the buckets are all equal.

However, the quicksort in the STL sort already does this type of partitioning. Thus, there should be little or no gain from a higher over partitioning ratio in this sense.

The more important reason to over partition is the fact that the subviews that result from the call to `p_n_partition` are most likely not aligned with the original pContainer. This means that some of the subviews have remote elements. If a subview has a large number of remote elements, the sorting step will take significantly longer. By over partitioning, the hope is that the smaller subviews will have fewer remote elements and a higher chance of aligning with the underlying pContainer.

### 5.4.1 Redistribution

Since redistribution eliminates the problem of having the remote element access during the bucket sorting phase, over partitioning should not result in significant speedup for the version of the parametrized Parallel Sample Sort.

Figure 5 shows the effect of increasing the over partitioning ratio for the version with redistribution. The times remain relatively the same for the partition phase, the sort buckets phase, and the total time, increasing slightly as the over partitioning ratio is increased. For the redistribution phase, higher over partitioning ratios lead to significantly longer times for 16 or more processors.

Therefore, for the parametrized Parallel Sample Sort with redistribution, the over partitioning ratio should be low, and the default value used for later tests is 1.

### 5.4.2 Without Redistribution

Figure 6 shows the effect of increasing the over partitioning ratio for the version of the parametrized Parallel Sample Sort without redistribution.

As the over partitioning ratio is increased from 1 to 16, the times decrease until 64 processors is reached. Over partitioning ratio 20 results in uniformly longer times from 1 to 32 processors.

However, there is a visible jump, especially in the times for over partitioning ratio 16, when it the parametrized Parallel Sample sort without redistribution reaches 64 processors. Running the same tests, in which we averaged three runs with exactly the same input of 1 million random elements generated with same seed, on 32 and 64 processors with two different seeds produced widely different times. For the second seed, 64 processors, and an over partitioning ratio of 16, the running time was 10.2483 seconds. For the third seed, 64 processors, and an over partitioning ratio of 16, the running time was 77.9435 seconds.

This variation can be explained by the distribution of local and remote elements throughout the buckets. For the first seed, the over partitioning ratio of 16 resulted in 11,745 remote elements in one of the buckets. For the second seed, the highest number of remote elements in any of the buckets was 2,415 elements. The greater the amount of remote access during sorting, the greater the amount of read communication between the processors, and, therefore, the longer sorting time.

For almost all of the runs, until the times get close to 5 or fewer seconds, the dominant phase, as can be seen in the graphs, is the sorting buckets phase. Because this sorting phase is almost entirely dependent on the bucket with the most remote elements and the distribution of the remote elements can vary widely from one input distribution to another, the performance of the parametrized Sample Sort without redistribution is not stable or necessarily predictable.

It is also important to note that the 11,1745 elements in one bucket also means that, for the runs

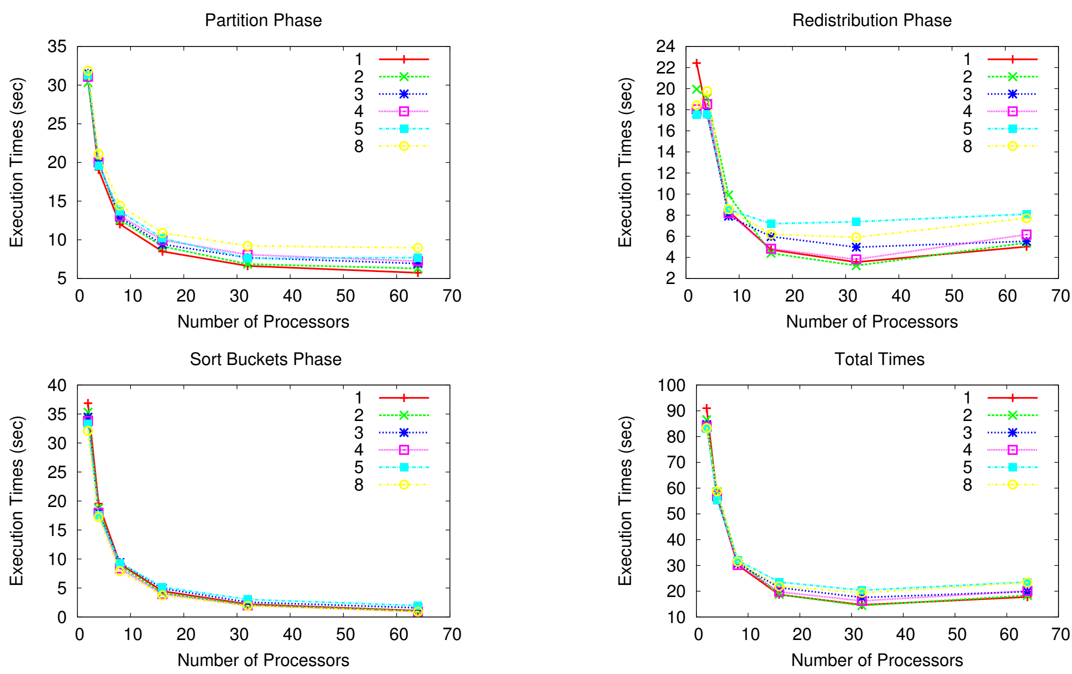



Figure 5: **With Redistribution:** Execution times for an input of 32 million random integers, with a constant seed, for over partitioning ratios ranging from 1 to 8 as the number of processors are increased from 2 to 64.

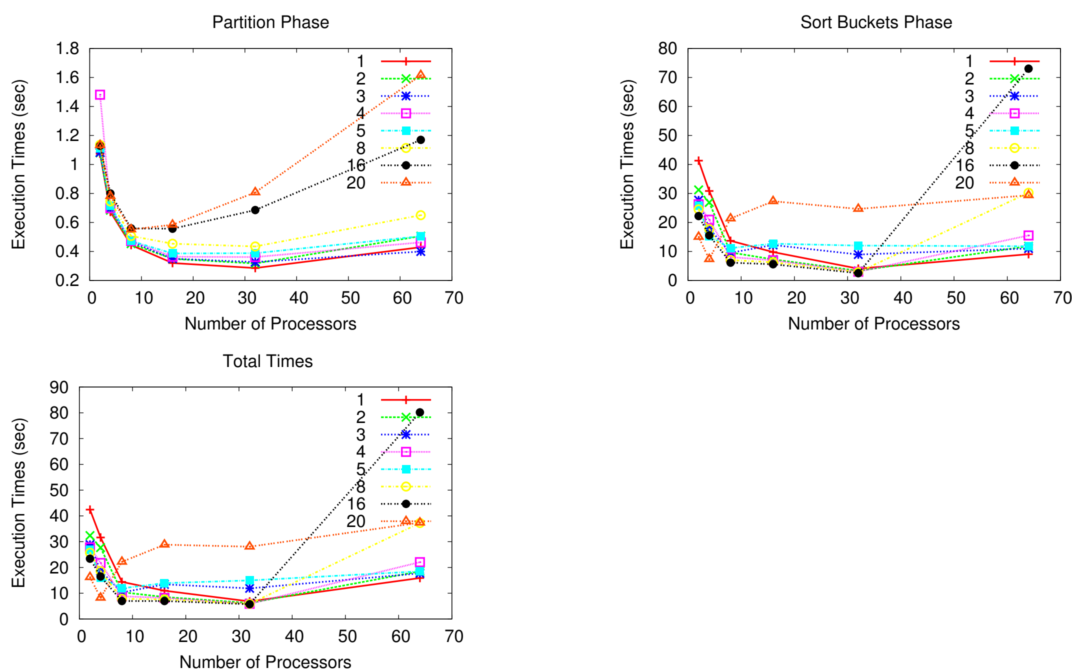



Figure 6: **Without Redistribution:** Execution times for an input of 32 million random integers, with a constant seed, for over partitioning ratios ranging from 1 to 64 as the number of processors are increased from 2 to 64.

with the first seed, a few of the splitters might have been chosen poorly. The ideal bucket size should have been $1000000/(64*16) \approx 977$. Upon inspection of the other bucket sizes, this was the case, with most of them varying between 600 and 1200. Out of the 1024 splitters that had to be chosen from the 8192 samples, one or two splitters might have been poorly chosen.

However, because over partitioning ratio 16 performed well up until 64 processors in the first run and performed well in one of the other runs discussed above for 64 processors, 16 is used as the default over partitioning ratio in the later tests.

### 5.5 Speedup

We measured the speedup for both of the versions of the parametrized Parallel Sample Sort, with redistribution and without redistribution.

#### 5.5.1 Redistribution

Figure 7 shows that the sort buckets phase of the parametrized Parallel Sample Sort with redistribution scales almost linearly, achieving super linear scaling for all but one of the input sizes. However, it also shows that the overall scalability of the algorithm is poor, since the maximum speedup achieved is 5 by the largest input size, 32000000 elements. This is due to the poor scalability of the partition and redistribution phases, which require a great deal of communication between threads.

#### 5.5.2 Without Redistribution

Figure 8 shows that the version without redistribution behaves erratically, with no clear trend of scalability. This is probably due to the problem of the distribution of the remote and local elements during the sort buckets phase, as described in Section 5.4.2.

Unless some method is found to decrease the cost of sorting remote data, use of the version of the parametrized Parallel Sample Sort without redistribution will result in unpredictable results.

## 6 Conclusion

In this paper we presented the parametrized Parallel Sample Sort, a parallel sorting algorithm written using STAPL. We described the design and implementation of two versions of the algorithm, with redistribution and without redistribution, and the parameters which can be varied. Our experimental results on an IBM P5 cluster show that the sort buckets phase of the parametrized Parallel Sample Sort with redistribution performs well with an over sampling ratio of 128 and an over partitioning ratio of 1, but that overall the parametrized Parallel Sample Sort with redistribution shows poor scalability. We also showed that the parametrized Parallel Sample Sort without redistribution performs poorly and erratically.

## 7 Future Work

Future work will focus on improving the partition and redistribution phases of the parametrized Parallel Sample Sort. If a method of reducing the cost of remote communication can be found, both versions of the parametrized Parallel Sample Sort will be re-evaluated. Also, iterator optimization may allow for some speedup in the sort buckets phase of the parametrized Parallel Sample Sort without redistribution.

In addition, the parametrized Parallel Sample Sort may be incorporated into an adaptive parallel sorting algorithm for the STAPL library. In such an adaptive parallel sorting algorithm, the parametrization could be used to optimize performance based on the data input.

Such a parallel sorting algorithm could then be called during both the sort samples phase and the sort buckets phase in place of the currently sequential sorts, which should result in a speedup for both phases in both versions of the parametrized Parallel Sample Sort.

## References

[1] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauch-

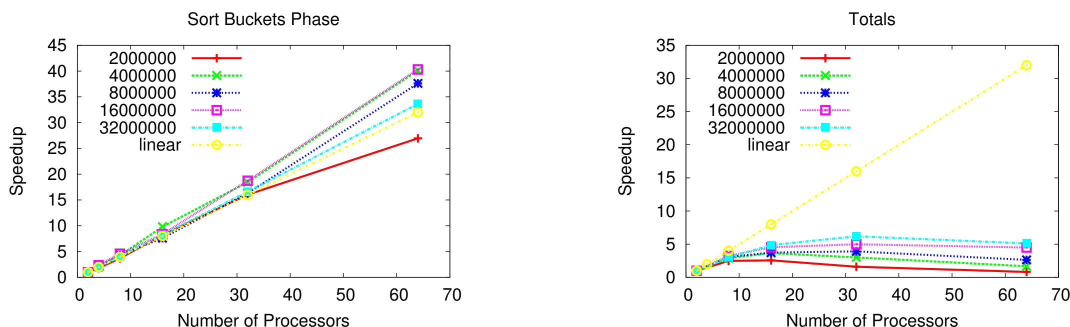



Figure 7: **With Redistribution:** Speedup for the sort buckets phase and the entire sort for random integer inputs, generated with a constant seed, of 2, 4, 8, 16, and 32 million elements as the number of processors is increased from 2 to 64.

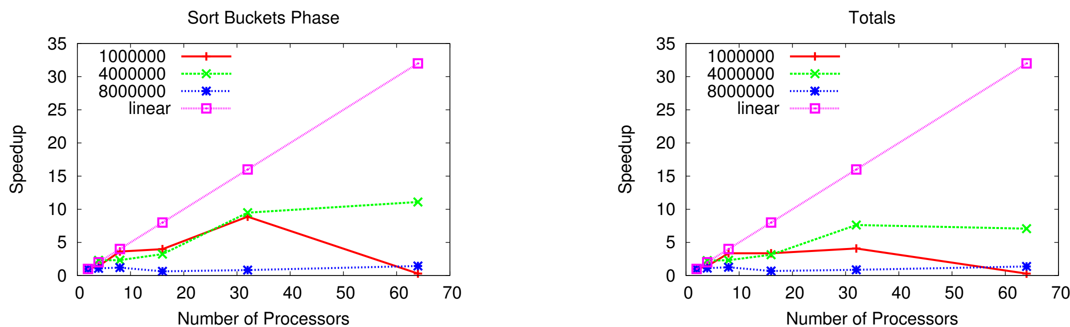



Figure 8: **Without Redistribution:** Speedup for the sort buckets phase and the entire sort for random integer inputs, generated with a constant seed, of 1, 4, and 8 million elements as the number of processors is increased from 2 to 64.

werger. STAPL: A standard template adaptive parallel C++ library. In *Proc. of the International Workshop on Advanced Compiler Technology for High Performance and Embedded Processors (IWACT)*, Bucharest, Romania, Jul 2001.

[2] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. An experimental analysis of parallel sorting algorithms. *Theory of Computing Systems*, 31(2):135–167, / 1998.

[3] D. R. Helman, D. A. Bader, and J. JáJá. A randomized parallel sorting algorithm with an experimental study. *Journal of Parallel and Distributed Computing*, 52(1):1–23, 1998.

[4] H. Li and K. C. Sevcik. Parallel sorting by over partitioning. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 46–56, 1994.

[5] D. R. Musser and A. Saini. *The STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1995.

[6] S. Saunders and L. Rauchwerger. ARMI: An adaptive, platform independent communication

library. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, San Diego, CA, June 2003.

[7] H. Shi and J. Schaeffer. Parallel Sorting by Regular Sampling. *Journal of Parallel and Distributed Computing*, 14(4):361–372, 1992.

[8] N. Thomas, S. Saunders, T. Smith, G. Tanase, and L. Rauchwerger. Armi: A high level communication library for stapl. *Parallel Processing Letters*, 16(2):261–280, Jun 2006.

```
1 sample_sort(N,T,S,K){
2  ....
3  locally, select up to s samples from each subview
5  if (choice == "even"){
6    step = l-1/S;
7    for(i=0; i<S && data_iterator!=local_end; i++){
8        data_iterator+=step;
9        *samples_iterator = *data_iterator;
10       samples_iterator++;
11     }
12  }
13 if (choice =="semi-random"){
14   for(i=0; i<S && data_iterator!=local_end; i++){
15       random_step=rand()% l-1/S;
16       data_iterator+=random_step;
17       *samples_iterator=*data_iterator;
18     }
19  }
20 else if (choice =="random"){
21   for(i=0, i<S && data_iterator!=local_end; i++){
22     do{
23         step = rand()% l-1/S;
24         data_iterator=data_begin;
25         data_iterator+=step;
26         insert data_iterator into a set
27       }while(data_iterator is already in the set);
28      *samples_iterator=*data_iterator;
29      samples_iterator++;
30     }
31  }
32 else if (choice =="block"){
33   for(i=0; i<S && data_iterator!=local_end; i++){
34       *samples_iterator=*data_iterator;
35       samples_iterator++;
36       data_iterator++;
37     }
38  }
39  ....
40 }//end sample sort
```

Figure 3: Pseudo-code of the parametrized Parallel Sample Sort sampling step. N is the size of the input, T is the number of threads, S is the over sampling ratio, K is the over partitioning ratio, and l is the number of elements on the thread.