Parallel Algorithms in STAPL: Sorting and the Selection Problem*†

Anna Tikhonova, Gabriel Tanase, Olga Tkachyshyn
Nancy M. Amato and Lawrence Rauchwerger
{atikhono,gabrielt,olgat,amato,rwerger}@cs.tamu.edu

Technical Report TR05-005
Parasol Lab
Department of Computer Science
Texas A&M University
College Station, TX 77843-3112

August 08, 2005

Abstract

Parallel and distributed processing provide massive computational power demanded by modern computing. The development of properly designed parallel libraries is crucial for the advancement of parallel processing and moving parallel computing into the mainstream. The Standard Template Adaptive Parallel Library (STAPL) is a parallel library designed as a superset of the (sequential) ANSI C++ Standard Template Library (STL). STAPL provides routines that are easily interchangeable with their sequential counterparts and allows users to execute programs on uni- or multiprocessor systems that utilize shared or distributed memory, insulating less experienced users from managing parallelism and, at the same time, allowing more sophisticated users sufficient control to achieve higher performance gains [1].

Our research is focused on designing and implementing a set of parallel sorting and selection algorithms in STAPL. This paper will use the Selection Problem (Nth Element Algorithm) as an example of a parallel algorithm and discuss the way it is implemented in the STAPL environment. Our goal is to design the most appropriate algorithms for STAPL parallel library that will provide good efficiency without sacrificing generality and portability.

*This research supported in part by NSF Grants EIA-0103742, ACR-0081510, ACR-0113971, CCR-0113974, EIA-9810937, ACI-0326350, and by the DOE.

†Work performed at the Parasol Lab during research internship in Summer 2005 and supported by the CRA Distributed Mentor Project.

1 Introduction

Parallel and distributed processing provide massive computational power demanded by the coming generation of scientific and engineering applications. However, the complexity of parallel architectures generally makes the construction of parallel and distributed applications notoriously difficult. The development of user-friendly parallel libraries is crucial for the advancement of parallel processing and moving parallel computing into the mainstream. In particular, there is a growing need for properly designed parallel libraries that not only provide viable means for achieving scalable performance across a variety of applications and architectures, but also allow users with varying levels of experience manage the complexity of parallel and distributed programming. Simply said, parallel libraries should be easy to use for users with low experience, yet powerful enough to be useful to more experienced users. This may be achieved by allowing necessary flexibility as well as providing an avenue for future improvement.

Sorting is a fundamential operation that is used in a large range of applications spanning both the scientific and commercial realms. There are many sequential and parallel solutions that are interesting to study and to theoretically analyze. Most sorting algorithms are relatively easy to understand, which allows to make straightforward comparisons between the parallel and sequential versions.

One of the primitives commonly used for sorting is the Selection Problem. STL provides this operation as `std::nth_element`. The Nth Element algorithm partially orders a range of elements. It arranges elements such that the element located in the nth position is the same as it would be if the entire range of elements had been sorted. Additionally, none of the elements in the range `[nth, last)` is less than any of the elements in the range `[first, nth)`. There is no guarantee regarding the relative order within the sub-ranges `[first, nth)` and `[nth, last)`. One of the common uses for the Nth Element algorithm is finding the median within a large collection of elements in the case when the entire range does not need to be sorted.

Parallel Selection Problem is an interesting problem to study because it requires both communication and computation. If the input data to be sorted is too large to fit in a uniprocessor's cache, operating on this data sequentially can take entirely too long because of the memory access bottleneck. In these cases, doing the operation in parallel can yield a significant speedup over the sequential solution.

2 STAPL Overview

The Standard Template Adaptive Parallel Library (STAPL) is a parallel library designed as a superset of the (sequential) ANSI C++ Standard Template Library (STL). STAPL offers the parallel system programmer a shared object view of the data space. The objects are distributed across the memory hierarchy which can be shared and/or distributed address spaces. Internal STAPL mechanisms assure an automatic translation from one space to another, presenting to the less experienced user a unified data space. For more experienced users the local/remote distinction of accesses can be exposed and performance enhanced.

STAPL supports the SPMD model of parallelism with essentially the same consistency model as OpenMP. The STAPL infrastructure consists of platform independent and platform dependent components. These are revealed to the programmer at an appropriate level of detail through a hierarchy of abstract interfaces. The platform independent components include the core parallel library, a view of a generic parallel/distributed machine, and an abstract interface to the communication library and run-time system [2].

2.1 STAPL Components

STL provides users with a collection of containers, algorithms and a mechanism to abstract data access - *iterators*. In a similar manner, STAPL provides users with parallel containers (`pContainers`), parallel algorithms (`pAlgorithms`) and an entirely new construct called `pRange` which allows random access to elements in a `pContainer`.

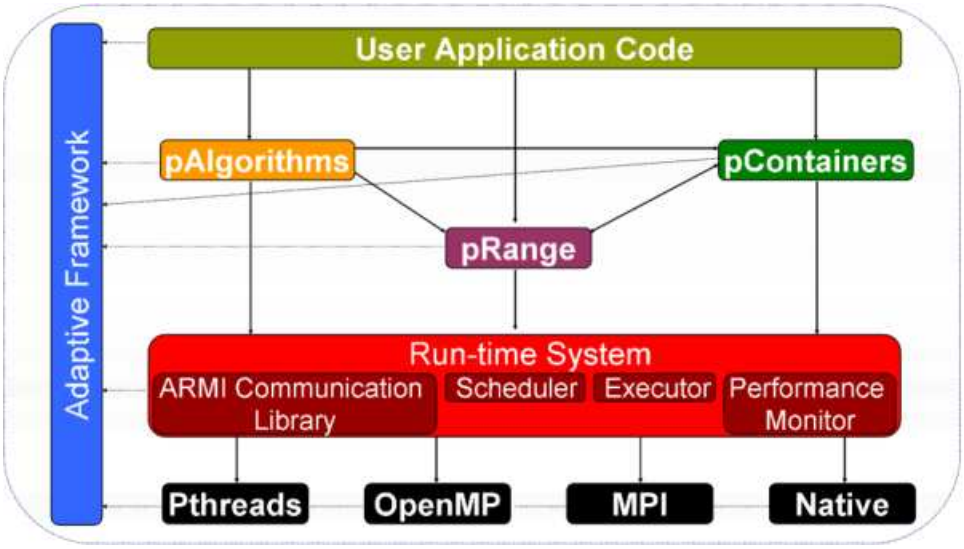


Figure 1: STAPL components.

pContainers: The core STAPL library consists of `pContainers` (distributed data structures with parallel methods) and `pAlgorithms` (parallel algorithms). The `pContainer` is the parallel equivalent of the STL container. Its data is distributed but the programmer is offered a shared-memory object view. The `pContainer` distribution can be user specified or computed automatically. Currently, STAPL has two `pContainers` that do not have STL equivalents: parallel matrix (`pMatrix`) and parallel graph (`pGraph`) [2].

pAlgorithms: `pAlgorithms` are parallel equivalents of algorithms. pAlgorithms are sets of parallel task objects which provide basic functionality, bound with the `pContainer` by `pRange`. Input for parallel tasks is specified by the `pRange`, (intermediate) results are stored in `pContainers`, and ARMI is used for communication between parallel tasks. STAPL provides parallel STL equivalents (copy, find, sort, etc.), as well as graph and matrix algorithms.

pRange: `pRange` provides a shared view of a distributed work space. `pRange` is divided into `subranges`. `Subranges` of the `pRange` are executable tasks. A task consists of a function object and a description of the data to which it is applied. `pRange` also supports parallel execution. It provides a clean expression of computation as parallel task graph and stores Data Dependence Graphs (DDGs) used in processing `subranges`.

ARMI Communication Library: STAPL's ARMI (Adaptive Remote Method Invocation) Communication Library uses the remote method invocation (RMI) communication abstraction that assures mutual exclusion at the destination but hides the lower level implementation (e.g., lower level protocols such as MPI, OpenMP, Pthreads, and mixed mode operation) [2].

2.2 Adaptive Algorithm Selection Framework

A fundamental challenge for parallel computing is writing portable programs that perform well on multiple platforms or for varying input sizes and types. This may be very difficult because performance if often sensitive to the system architecture, the runtime environment, and input data characteristics. This challenge exists for parallel and distributed systems as well, due to the wide variety of system architectures. In addition, the other classic parallel computing trade-offs such as parallel management overhead, communication, load balancing, and locality must be thoroughly considered in order to design a good parallel algorithm. For example, the probability distribution of the data set can have a significant impact on the performance of certain sorting algorithms because it can influence the way the load is balanced, the amount of communication

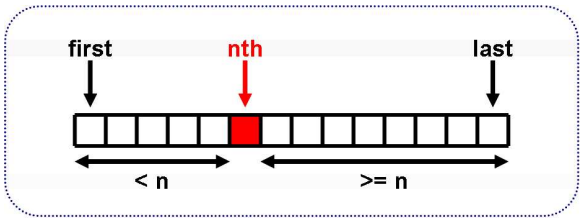


Figure 2: Nth Element algorithm (Selection Problem) specification.

that will be necessary, as well as the locality of the data.

STAPL addresses this problem by adaptively selecting the best parallel algorithm for the current input data and system from a set of functionally equivalent algorithmic options. Toward this goal, a general framework for adaptive algorithm selection was developed for use in STAPL. This STAPL framework uses machine learning techniques to analyze data collected by STAPL installation benchmarks and to determine tests that will select among algorithmic options at run-time. A prototype implementation of this framework was applied to two important parallel operations, sorting and matrix multiplication, on multiple platforms. The results show that the framework determines run-time tests that correctly select the best performing algorithm from among several competing algorithmic options in 86-100% of the cases studied, depending on the operation and the system [2].

3 Parallel Sorting and Selection Algorithms in STAPL

Fast sorting algorithms generally have the following three components:

1. a local computational phase;
2. an optional intermediate phase which calculates the destination of keys for the next phase;
3. a communication phase which moves keys across processors and often involves a transformation of the entire data set.

STAPL makes it easy to write efficient parallel algorithms that include these three generalized components in a straightforward manner. The next section will present an example of a parallel algorithm developed in the STAPL environment.

3.1 The Selection Problem (Nth Element Algorithm)

The Nth Element algorithm partially orders a range of elements. It takes the following arguments: a `pRange`, a `pContainer` and an iterator nth pointing to the nth element. The algorithm arranges elements in the range `[first, last)` such that the element pointed to by the iterator nth is the same as it would be if the entire range `[first, last)` had been sorted. Additionally, none of the elements in the range `[nth, last)` is less than any of the elements in the range `[first, nth)`. There is no guarantee regarding the relative order within the sub-ranges `[first, nth)` and `[nth, last)`.

Below we provide an STL interface of the Nth Element algorithm. Our goal is to stay consistent with the standard. Thus, the Nth Element algorithm implementations discussed in this paper follow the STL specification.

```
template <class pRange, class pContainer, typename Iterator>
void nth_element(pRange &pr, pContainer &pcont, Iterator nth);
```

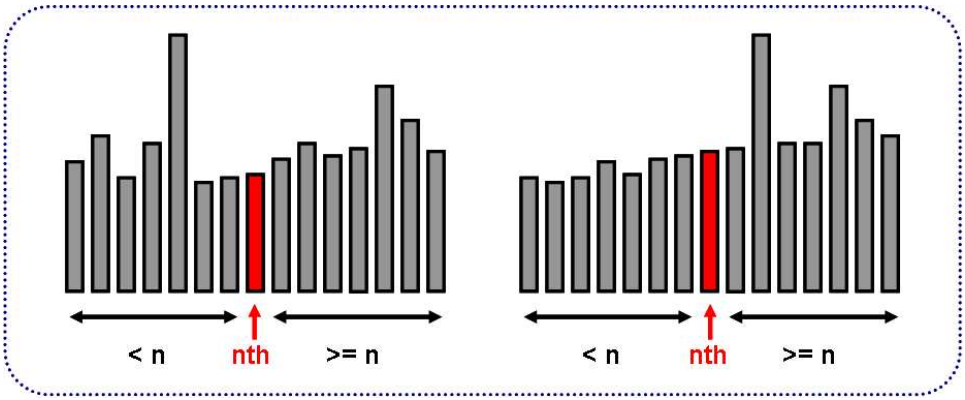


Figure 3: Data arrangement before and after the Nth Element.

```
template <class pRange, class pContainer, typename Iterator, class Compare>
void nth_element(pRange &pr, pContainer &pcont, Iterator nth,
                 Compare comp);
```

There are two versions of the Nth Element algorithm. The two versions differ in how they define whether one element is less than another. The first version compares objects using operator `<`, and the second compares objects using a user-defined function object (for example, `Compare comp`). The post-condition for the first version of Nth Element is as follows. There exists no `Iterator i` in the range `[first, nth)` such that `*nth < *i`, and there exists no `Iterator j` in the range `[nth + 1, last)` such that `*j < *nth`. The post-condition for the second version of nth element is as follows. There exists no `Iterator i` in the range `[first, nth)` such that `comp(*nth, *i)` is true, and there exists no `Iterator j` in the range `[nth + 1, last)` such that `comp(*j, *nth)` is true.

The Nth Element algorithm may be used to find the median within a large collection of elements in the case when the entire range of elements does not need to be sorted. Of course, we could let a sorting algorithm (say, `std::sort`) do all the work. When the sequence is ordered, the median sample would be conveniently located in the middle of our data structure. The complexity of `std::sort` is $O(nlogn)$, while the complexity of the `std::nth_element` is estimated to be $O(n)$. This $logn$ factor gets significant with larger input sizes fairly quickly and should not be overlooked.

3.2 Sequential "Bucket" Implementation of the Nth Element Algorithm

The algorithm picks m - 1 splitter elements that partition keys into m buckets. Then, we could assign each element to the appropriate bucket. For example, in Figure 4, we have two splitters, namely 8 and 17. We also have three buckets with the following ranges: `bucket 0` contains all elements strictly less than 8, `bucket 1` contains all elements equal or greater than 8 but strictly less than 17, and `bucket 2` contains all elements equal or greater than 17. Consider an element with the numeric value of 14; 14 is equal or greater than 8 and strictly less than 17, so we copy 14 into `bucket 1`.

By traversing the sizes of the buckets, we find the bucket containing the nth element. Once we know which bucket contains the nth element, we can determine the position of the nth element by either calling `std::nth_element` (trivially, this insures the correct placement of the nth element) or by recursively calling our algorithm on the bucket containing the nth element. The terminating condition for the recursive call is a function of the size of the bucket containing the nth element. The final step of the algorithm is to sequentially copy elements from the buckets back into the original container.

The efficiency of this implementation depends on how well we divide the input, and this in turn depends on how well we choose the splitters. The complexity of this algorithm is dominated by the cost of traversing the sequence of elements, copying elements into appropriate buckets and back into the original container -

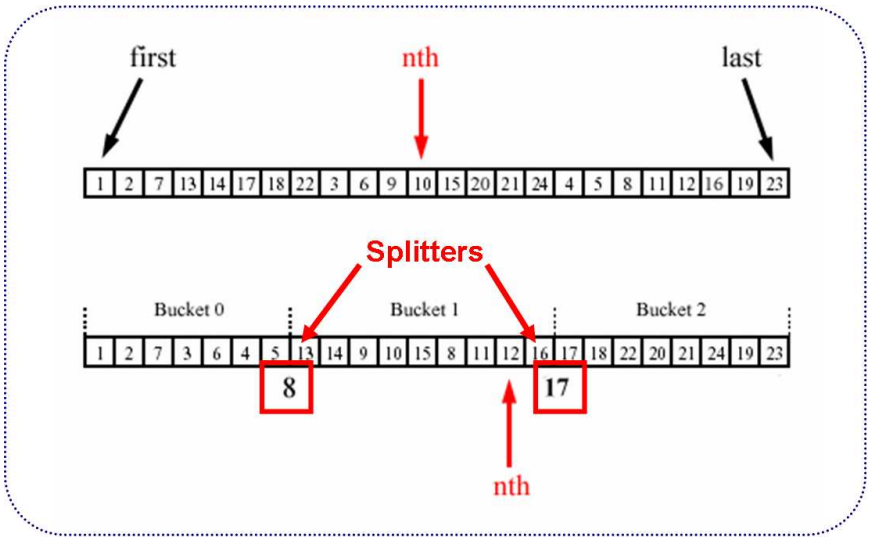


Figure 4: Distributing elements into buckets (sequential algorithm).

all of which are linear operations. The estimated complexity of this algorithm is $O(n)$. The steps of this implementation are summarized in Algorithm 1.

Algorithm 1 nth_element(*Iterator first, Iterator nth, Iterator last*) {

1: Determine bucket ranges.
2: Assign elements into appropriate buckets.
3: Find bucket `B` containing the nth element.
4: Recursively call the algorithm on `B` (or call `std::nth_element` on `B`).
5: Final step: Sequentially copy elements from buckets back into the original container.
6: }

3.3 Parallelizing the Nth Element Algorithm

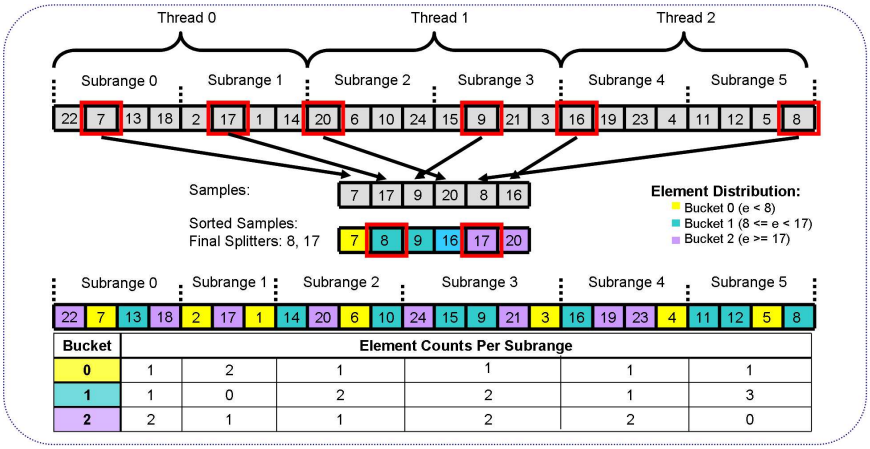
For the parallel implementation, we assume p to be the number of processors and each processor starts with n/p keys. The algorithm picks p - 1 splitter elements that partition keys into p buckets. Then, we let each processor act as a bucket and let our splitters define the bucket ranges. The next step is to distribute keys into the buckets. Communication between processes is very expensive in parallel computing. And, copying elements from one processor to another is an unnecessary operation for this step of the algorithm. It suffices to simply keep track of bucket sizes and increment the size counter for an appropriate bucket when an element within the particular bucket range is encountered.

By traversing the sizes of the buckets, we find the bucket containing the nth element. Once we know which bucket contains the nth element, we can determine the position of the nth element by either sorting (in parallel) the elements in the bucket (trivially, this insures the correct placement of the nth element) or by recursively calling our parallel algorithm on the bucket containing the nth element. The terminating condition for the recursive call is a function of the size of bucket containing the nth element.

The steps of the parallel algorithm are summarized in Algorithm 2.

Algorithm 2 p_nth_element(*pRange &pr, pContainer &pcont, Iterator nth*) {

1: Locally, select s random elements, called *samples*.
2: Globally, sort all selected samples.
3: Select m - 1 elements, called *splitters*.
4: Splitters determine the ranges of m virtual "buckets".
5: Total the number of elements in each "bucket".
6: Traverse totals to find bucket `B` containing the nth element.
7: Recursively call `p_nth_element(B.pRange(), B, nth)`.
8: Final step: copy elements back into the original container in parallel.
9: }



Bucket	Element Counts Per Subrange					
0	1	2	1	1	1	1
1	1	0	2	2	1	3
2	2	1	1	2	2	0

Figure 5: Sampling technique, distributing elements into buckets.

3.3.1 Distributing Elements into Virtual Buckets

This section discusses the process of distributing elements into virtual buckets in more detail. Figure 5 supplements this discussion. We include this section in order to demonstrate the way parallel algorithms are written in the STAPL environment.

The elements in our sequence are distributed among `subranges`. The number of `subranges` and the element distribution among `subranges`, as well as number of "virtual buckets" are not relevant to the parallel implementation. Samples are collected from all `subranges` and are sorted globally. Final splitters are selected globally and each `subrange` has a complete list of splitters. Using the list of splitters, each `subrange` computes the local counts of elements that fall within the range of each bucket. The total bucket counts are calculated by summing up the bucket counts collected from each `subrange`.

For example, in Figure 5 we show a pArray of size 24 distributed among 3 threads with 2 `subranges` per thread. The algorithm selects one sample per `subrange`. The samples are sorted globally and the final splitters are picked. We choose two splitters, namely 8 and 17, which define the ranges of 3 buckets. Then, `subranges` compute the local counts of elements that fall within the range of each bucket. As an example, let's compute the local bucket counts for `subrange 0`. `Subrange 0` contains 2 elements that fall within the range of `bucket 0`, 0 elements that fall within the range of `bucket 1`, and 1 element that falls within the range of `bucket 2`. Once all `subranges` finish computing local bucket element counts, the total bucket counts may be computed. As an example, let's compute the total size of `bucket 0`. `Subrange 0` contains 1 element that falls within the range of `bucket 0`, `subrange 1`: 2 elements; `subrange 2`: 1 element; `subrange 3`: 1 element; `subrange 4`: 1 element; and `subrange 5`: 1 element. Thus, the total bucket size is 7.

Example STAPL code for this step of the algorithm is provided below:

```
template<typename Boundary, class pContainer>
class distribute_elements_wf : public
work_function_base<Boundary> {
  pContainer *splitters;
  nSplitters = splitters->size();
  vector<int> bucket_counts(nSplitters);

  distribute_elements_wf(pContainer &sp) : splitters(&sp) {}

  void operator() (Boundary &subrange_data) {
    typename Boundary::iterator_type first1 = subrange_data.begin();
    while (first1 != subrange_data.end()) {
      int dest;
      pContainer::value_type val = *first1;
      if (nSplitters > 1) { //If at least two splitters
        pContainer::value_type *d =
          std::upper_bound(&splitters[0], &splitters[nSplitters], val);
        dest = (int)(d-(&splitters[0]));
      } else if (nSplitters == 1) { //one splitter
        if(val < splitters[0])
          dest = 0;
        else
          dest = 1;
      } else {
        dest = 0; //No splitter, send to self
      }
    }
    bucket_counts[dest]++;
    first1++; // Increment counter for the appropriate bucket
  }
};
```

3.4 Complexity Analysis

The average complexity of `std::nth_element` is $O(n)$, i.e. linear, while that of `std::sort` is $O(nlogn)$. As was mentioned above, the complexity of the sequential "bucket" implementation is $O(n)$. Ideally, the complexity of the parallel Nth Element algorithm would be $O(n/p)$, but in our case it is $O(n/p + (n/p^2)log(n/p^2))$, assuming that sorting a bucket of size n/p takes $O((n/p^2)log(n/p^2))$. This complexity is still significantly lower than the complexity of sorting the entire range, which is $O((n/p)log(n/p))$, making the overhead of parallelizing this algorithm acceptably low.

4 Conclusion

We have presented STAPL as a framework for writing efficient parallel algorithms and have provided implementation details for the Selection Problem developed within the STAPL environment. We have shown that writing parallel algorithms in STAPL is straightforward.

For future work, there are several improvements we plan to incorporate into the algorithm. Improvements in algorithm performance can be obtained by taking better advantage of data locality and/or better load balancing.

References

[1] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. STAPL: An adaptive, generic parallel programming library for C++. In *Proc. of the 14th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Cumberland Falls, Kentucky, August 2001.

[2] N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. Amato, and L. Rauchwerger. A framework for adaptive algorithm selection in STAPL. In *Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog. (PPOPP)*, Chicago, Illinois, Jun 2005.