Parallel Algorithms in STAPL: Implementation and Evaluation†

Jeremy Vu, Mauro Bianco, and Nancy M. Amato
Parasol Lab, Dept. of Computer Science and Engineering
Texas A&M University, College Station, TX 77843
{cvu, bmm, amato}@cse.tamu.edu

The Standard Template Adaptive Parallel Library (STAPL) is a high-productivity parallel programming framework that extends C++ and the STL with unified support for shared and distributed memory parallelism. STAPL provides distributed data structures (pContainers) and parallel algorithms (pAlgorithms) and a generic methodology for extending them to provide customized functionality.

In this paper, we present the design and implementation of a STAPL pAlgorithm, p_unique_copy. We also present preliminary experimental results for p_unique_copy and another pAlgorithm, p_copy_if, for comparison. The results illustrate the scalability of the algorithms, as well as the sensitivity of their performance to the underlying data distribution. Finally, we present initial results based on the usage of STAPL anywhere_inserter, a construct for parallel insertions that allows for performance optimization when it is used in pAlgorithms.

†This research supported in part by NSF Grants EIA-0103742, ACR-0081510, ACR-0113971, CCR-0113974, ACI-0326350, CRI-0551685, CCF-0833199, CCF-0830753, by the DOE NNSA under the Predictive Science Academic Alliances Program by grant DE-FC52-08NA28616, by Chevron, IBM, Intel, HP, and by King Abdullah University of Science and Technology (KAUST) Award KUS-C1-016-04. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.


1 Introduction

Parallel programming is becoming mainstream due to the increased availability of multiprocessor and multicore architectures and the need to solve larger and more complex problems. To help programmers address the difficulties of parallel programming, we are developing the Standard Template Adaptive Parallel Library (STAPL) [1]. STAPL is a parallel C++ library with functionality similar to STL, the ANSI adopted C++ Standard Template Library [11]. The STL is a collection of basic algorithms, containers and iterators that can be used as high-level building blocks for sequential applications. Similar to the STL, STAPL provides a collection of parallel algorithms (pAlgorithms), parallel containers (pContainers), and views to abstract the data access in pContainers. These are building blocks for writing parallel programs. An important goal of STAPL is to provide a high productivity environment for the development of applications that can execute efficiently on a wide spectrum of parallel architectures.

In this work, we present the design, implementation, and analysis of a STAPL pAlgorithm, p_unique_copy. Like its STL counterpart, unique_copy, STAPL’s p_unique_copy copies the first element in each group of consecutive duplicate elements from an input to an output. The algorithm identifies duplicate elements based on a binary relation that the user provides to it; if no relation is provided, the algorithm will determine uniqueness based on the equality (‘=’) relation. Unlike its STL counterpart, p_unique_copy has multiple cases based on the properties of the binary relation. There are three different cases in total: 1) the relation is symmetric and transitive, 2) the relation is transitive but not symmetric, and 3) the relation is not transitive. Once the algorithm has identified the duplicate elements, it computes the output ranges for each processor, then copies the unique elements to the output in parallel. Two implementations of p_unique_copy are described. The differences lie in the copy phase; in the first implementation, each processor takes a shared-memory view of the output container. In the second implementation, each processor is made aware of only one subrange of the output.

We also present preliminary experimental results for p_unique_copy, along with results for another pAlgorithm, p_copy_if. These results were obtained using the STAPL pVector. First, we show results illustrating the scalability of p_unique_copy up to 64 processors. Second, we show results that give some indication of the pAlgorithm’s sensitivity to the amount of communication required.

Finally, we provide some experimental results for an optimization of the above pAlgorithms. In these optimizations, the STAPL anywhere_inserter is used when writing to the output. STAPL anywhere_inserter is a construct for parallel insertion that allows for some performance optimizations when it is used in pAlgorithms. These initial results using anywhere_inserter show moderate gains in performance compared to the unoptimized versions of the tested pAlgorithms.

2 Related Work

There are a number of parallel libraries and languages that have similar aims to STAPL [2, 3, 5, 6, 10, 13]. The PSTL (Parallel Standard Template Library) project [8, 9] explored the same underlying philosophy as STAPL of extending the C++ STL for parallel programming. PSTL provided distributed arrays and associative containers with support for specifying data distributions (e.g., ContainerRatio) and local and global iterators for data access. PSTL also provided parallelized STL algorithms along with parallel algorithms that had no STL counterpart. STAPL differs from PSTL by providing an integrated framework for all pContainers, which also allows users to customize the default behavior, such as specifying different data distributions. The PSTL is no longer an active project.

The Multi-Core Standard Template Library (MCSTL) is another parallel library [12]. The design of the MCSTL is limited to shared-memory systems. The MCSTL’s goal is to parallelize many of the STL algorithms in a manner that only requires the user to recompile their C++ code with the MCSTL enabled. The OpenMP API is used to control multithreading. STAPL also provides parallelized STL algorithms; however, STAPL supports both distributed and shared memory systems. In addition to parallel algorithms, STAPL provides parallel containers and an abstraction for data access (views), while the MCSTL focuses solely on parallel algorithms.

Intel Threading Building Blocks (TBB) [7] provide thread-safe containers such as vectors, queues and hashmaps for shared memory architectures. In the same spirit as STAPL, TBB provides arrays and associative containers, and the interfaces provided resemble those of STL containers. TBB also provides generic parallel
algorithms, though they are not necessarily parallelizations of STL algorithms. Our work is distinguished from TBB in that we target both shared and distributed systems, and it is a design choice in STAPL that all containers should provide both STL compatible interfaces and additional interfaces optimized for parallelism. Being exclusively for shared memory, TBB does not provide support for data distribution.

Chapel is a new programming language developed by Cray that is focused on reducing the complexity of parallel programming [4]. The language proposes a formal approach for containers and data distributions, and provides default data distributions and specifies a methodology for integrating new ones. Also, although Chapel mentions associative domains, it does not appear to support multiple associative containers at this point. Finally, STAPL differs from Chapel and other parallel languages in that it is a library.

3 STAPL Overview

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. pContainers have a data distribution manager that provides the programmer with a shared object view that presents a uniform access interface regardless of the physical location of the data. Thread-safety is guaranteed by providing mechanisms that guarantee all operations leave the pContainer in a consistent state. Important aspects of all STAPL components are extendability and composability, e.g., the pContainers implemented within the framework allow users to extend and specialize them, and to operate on pContainers of pContainers.

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. 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 [14]) provide the interface to the underlying operating system, native communication library and hardware architecture. ARMI uses the remote method invocation (RMI) communication mechanism among computing processes 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). ARMI implements several optimizations for improving performance, such as the aggregation of RMI requests to the same destination to amortize latency. For more details on ARMI please see [14, 15].
4  \texttt{p\_unique\_copy}

\texttt{p\_unique\_copy} is a parallelization of its STL counterpart, \texttt{unique\_copy}. The sequential STL implementation is summarized in Algorithm 1. It is important to note that the algorithm uses the last identified unique element as the element on the left side of each comparison using the binary relation.

\begin{algorithm}
\caption{unique\_copy}
\begin{algorithmic}[1]
\State \textbf{Input:} A sequence of elements \(\{x_1, x_2, \ldots, x_n\}\) and a binary relation \(\oplus\)
\State \textbf{Output:} A sequence of elements consisting of only the first element of each group of consecutive duplicate elements.
\State 1: \texttt{last\_unique} \leftarrow x_1
\State 2: write \texttt{last\_unique} to the output
\State 3: \textbf{for} each element \(x_i, i = 2, \ldots, n\) \textbf{do}
\State 4: \textbf{if} the relation \texttt{last\_unique} \(\oplus\) \(x_i\) does not hold \textbf{then}
\State 5: \texttt{last\_unique} \leftarrow x_i
\State 6: write \texttt{last\_unique} to the output
\State 7: \textbf{end if}
\State 8: \textbf{end for}
\State 9: return
\end{algorithmic}
\end{algorithm}

Like \texttt{unique\_copy}, \texttt{p\_unique\_copy} also identifies unique elements and copies them to an output sequence. Duplicate elements are identified using a binary relation \(\oplus\) that is specified by the user. For two elements \(a\) and \(b\), if the relation \(a \oplus b\) holds true, then \(b\) is considered a duplicate element. \texttt{p\_unique\_copy} can be summarized as follows:

\begin{algorithm}
\caption{p\_unique\_copy}
\begin{algorithmic}[1]
\State \textbf{Input:} A sequence of elements \(\{x_1, x_2, \ldots, x_n\}\) and a binary relation \(\oplus\)
\State \textbf{Output:} A sequence of elements consisting of only the first element of each group of consecutive duplicate elements.
\State 1: Compute the duplicate elements using the binary relation \(\oplus\). Keep track of the duplicates by using a separate sequence of 0s and 1s, where 0 indicates a duplicate element and 1 indicates a unique element.
\State 2: Compute the subranges in the output that each processor will write to and copy the unique elements to the output.
\State 3: return
\end{algorithmic}
\end{algorithm}

We note that we implement two versions of \texttt{p\_unique\_copy} that differ slightly in the second phase. Next we will describe the implementation of \texttt{p\_unique\_copy}.

4.1 Computing the duplicates

There are three separate cases for this phase of \texttt{p\_unique\_copy}, each stemming from the properties of the binary relation being used to identify duplicate elements. Two specific properties are important to the algorithm, since their presence (or lack of) determines the amount of parallel work that is done during this phase. The three cases are as follows:

1. The relation’s properties include symmetry and transitivity.
2. The relation’s properties include transitivity but not symmetry.
3. The relation’s properties do not include transitivity.

The properties of the relation are inferred from user-defined type traits in C++, which are then used to decide which case the relation falls into. In the first two cases, we can employ parallelism to identify duplicate elements in the input. In these two cases, the identification algorithms are expressed at the highest degree of parallelism. In the third case, however, we cannot; because the relation is not transitive, the identification of
duplicate elements must be performed in a sequential manner. We note that symmetry without transitivity does not matter, so if the relation is not transitive, then this phase must be done sequentially. In the first two cases, where can employ parallelism, the identification algorithms are expressed at the highest degree of parallelism.

Some examples of relations that fall into the first case include the equality relation (\(a = b\)) and the “sibling” relation (‘\(a\) is a sibling of \(b\)’). Examples of relations that fall into the second case include the less-than (‘\(<\)’) and the less-than or equal-to (‘\(\leq\)’) relations. An example of a relation that falls into the third case is the relation \(a\) has a common factor greater than 1 with \(b\) between natural numbers greater than 1. Note that this relation is symmetric, but not transitive, so it still fits into the third case. Another example would be the “preys on” relation (‘\(a\) preys on \(b\)’), which is neither symmetric nor transitive.

We will now describe the different cases for the duplicate identification phase of \(p\text{unique}\text{copy}\). In each case the aim is to output a sequence of integers \(\{m_1, m_2, \ldots, m_n\}\) that consists of 0s and 1s. The use for this “marker” sequence is two-fold. First, each element \(m_i\) in the marker sequence indicates the uniqueness or duplicity of the corresponding element \(x_i\) in the input. If \(m_i = 1\), then \(x_i\) is unique; else if \(m_i = 0\), \(x_i\) is a duplicate element. The second use for this marker sequence is that it enables us to copy the unique elements to the output in a simple manner.

4.1.1 Case 1: The relation is symmetric and transitive

When the relation is both symmetric and transitive, we can employ a high level of parallelism to identify the duplicate elements in the input. Due to the properties of the binary relation, all comparisons can be performed simultaneously since none of the comparisons need to know what the “last unique element” is, as is done in STL’s implementation; that is, for every element \(x_i\) in the input, \(i = 1, \ldots, n - 1\), it is sufficient to use the comparison \(x_i \oplus x_{i+1}\) to determine the uniqueness or duplicity of \(x_{i+1}\). This is more concisely described in Lemma 1.

**Lemma 1** For any three elements \(x_{i-1}, x_i, x_{i+1}\), and a binary relation \(\oplus\)

\(x_{i-1} \oplus x_i, x_{i-1} \oplus x_{i+1} \Rightarrow x_i \oplus x_{i+1}\) if and only if \(\oplus\) is symmetric.

The duplicate identification algorithm for this case can thus be expressed as follows:

<table>
<thead>
<tr>
<th>Algorithm 3 Duplicate Identification, Case 1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> A sequence of elements ({x_1, x_2, \ldots, x_n}) and a binary relation (\oplus)</td>
</tr>
<tr>
<td><strong>Output:</strong> A sequence of integers ({m_1, m_2, \ldots, m_n})</td>
</tr>
<tr>
<td>1: (m_1 \leftarrow 1)</td>
</tr>
<tr>
<td>2: \textbf{for} each pair of elements (x_i, x_{i+1}), (i = 1, \ldots, n - 1) \textbf{parallel do}</td>
</tr>
<tr>
<td>3: \textbf{if} the relation (x_i \oplus x_{i+1}) holds true \textbf{then}</td>
</tr>
<tr>
<td>4: (m_{i+1} \leftarrow 0)</td>
</tr>
<tr>
<td>5: \textbf{else}</td>
</tr>
<tr>
<td>6: (m_{i+1} \leftarrow 1)</td>
</tr>
<tr>
<td>7: \textbf{end if}</td>
</tr>
<tr>
<td>8: \textbf{end for}</td>
</tr>
<tr>
<td>9: \textbf{return}</td>
</tr>
</tbody>
</table>

4.1.2 Case 2: The relation is transitive but not symmetric

In this case, a different approach is required. Owing to the properties of the binary relation, attempting to identify the duplicate elements in the same manner as the first case can produce incorrect results. In the first case, the assumption that we can use the immediate previous element \(x_i\) to determine an element \(x_{i+1}\)’s duplicity was a safe one; however, without both transitivity and symmetry, this assumption no longer holds true. For example, take the following sequence of integers and the ‘\(<\)’ relation:

\{60, 100, 70, 20, 50, 40, 30, 10\}

If these were passed as input to `p_unique_copy` and the duplicates were identified using the first case, the output would be:

\{60, 70, 20, 40, 30, 10\}

However, a cursory check should be enough to verify that that the output should actually be:

\{60, 20, 10\}

At first glance, the problem seems to become inherently sequential. In this case, however, we can use the parallel prefix algorithm to identify the proper element to use on the left side of each comparison.

For some input sequence \(\{a_1, a_2, \ldots, a_n\}\) and an operator \(\otimes\), the parallel prefix will output a sequence \(\{a_1, a_1 \otimes a_2, \ldots, a_1 \otimes a_2 \otimes \cdots \otimes a_n\}\).

By modeling the operator \(\otimes\) as a function of the binary relation \(\oplus\), the parallel prefix outputs a sequence of elements that we can use to identify the duplicate elements in the input. This function works as follows:

For any two elements \(a\) and \(b\), if \(a \oplus b\) holds true, then \(a \otimes b = a\). If \(a \oplus b\) does not hold true, then \(a \otimes b = b\). The resulting output sequence is essentially a sequence of “last known unique” elements. We then use this sequence to identify, in parallel, the duplicate elements. The identification algorithm for this case is given in Algorithm 4.

**Algorithm 4 Duplicate Identification, Case 2**

**Input:** A sequence of elements \(\{x_1, x_2, \ldots, x_n\}\) and a binary relation \(\oplus\)

**Output:** A sequence of integers \(\{m_1, m_2, \ldots, m_n\}\)

1: Perform the parallel prefix algorithm on the input sequence using the operator \(\otimes\) as described above.

We get a sequence of elements \(\{p_1, p_2, \ldots, p_n\}\).

2: \(m_1 \leftarrow 1\)

3: for each element \(x_i, i = 2, \ldots, n\), parallel do

4: if the relation \(p_{i-1} \oplus x_i\) holds true then

5: \(m_i \leftarrow 0\)

6: else

7: \(m_i \leftarrow 1\)

8: end if

9: end for

10: return

We present a brief example of how the algorithm works in this case. Take the previous example input, a sequence of integers \(X = \{60, 100, 70, 20, 50, 40, 30, 10\}\) and the binary relation ‘<’. As previously noted, the algorithm to identify duplicates from Case 1 will identify the elements inaccurately. So we begin by performing the parallel prefix algorithm as described above. The output of the parallel prefix will be:

\(P = \{60, 60, 60, 20, 20, 20, 20, 10\}\)

So we have the the prefix sequence \(P\), input sequence \(X\), and the marker sequence \(M\):

\(P = \{60, 60, 60, 20, 20, 20, 20, 10\}\)
\(X = \{60, 100, 70, 20, 50, 40, 30, 10\}\)
\(M = \{m_1, m_2, m_3, m_4, m_5, m_6, m_7, m_8\}\)

We are now ready to identify the duplicate elements in parallel. The individual comparisons in this case, along with the resulting marker assignments will be:

\(x_1 \Rightarrow m_1 = 1\)
\(p_1 < x_2 \Rightarrow 60 < 100 \Rightarrow m_2 = 0\)
\(p_2 < x_3 \Rightarrow 60 < 70 \Rightarrow m_3 = 0\)
\[ p_3 < x_4 \Rightarrow 60 < 20 \Rightarrow m_4 = 1 \]
\[ p_4 < x_5 \Rightarrow 20 < 50 \Rightarrow m_5 = 0 \]
\[ p_5 < x_6 \Rightarrow 20 < 40 \Rightarrow m_6 = 0 \]
\[ p_6 < x_7 \Rightarrow 20 < 30 \Rightarrow m_7 = 0 \]
\[ p_7 < x_8 \Rightarrow 20 < 10 \Rightarrow m_8 = 1 \]

The marker sequence now indicates which elements in the input are unique and which are duplicates. In this case, we can see that our unique elements are \{60, 20, 10\}, which is the correct output in this example. We now briefly describe the third case, where the binary relation is not transitive.

4.1.3 Case 3: The relation is not transitive

If the relation is not transitive, then dependencies exist between the comparisons during this phase that we can’t get around, and the identification of duplicate elements must be performed sequentially. Currently, we have a single processor perform this sequential identification of duplicate elements.

**Algorithm 5** Duplicate Identification, Case 3

*Input:* A sequence of elements \{x_1, x_2, \ldots, x_n\} and a binary relation \(\oplus\)

*Output:* A sequence of integers \{m_1, m_2, \ldots, m_n\}

1: last_unique = x_1
2: \[ m_1 \leftarrow 1 \]
3: for each element \(x_i, i = 2, \ldots, n\) do
4: if the relation last_unique \(\oplus x_i\) does not hold then
5: last_unique = x_i
6: \[ m_i \leftarrow 1 \]
7: else
8: \[ m_i \leftarrow 0 \]
9: end if
10: end for
11: return

Before we finish our description of **p.unique.copy**, we would like to note that we are planning to implement the algorithm for this case differently in future work. Instead of having a single processor perform all of the identification work, we would like to create a “linear chain of tasks”. This linear chain of tasks would perform the sequential identification while taking into account the distribution of the data. This implementation would also write unique elements to the output as it identifies them, thus saving computation time. First, the processor that has possession of the starting subrange of the input would perform the STL’s **unique** algorithm on its data set. After it reaches the end of its input subrange, it will communicate with the processor that has possession of the next subrange in the sequence. The former would inform the latter processor of both the last known unique element in the sequence and also of the current position in the output where data is being written to. This process would repeat until the end of the input is reached.

4.2 Compute the output subranges and copy the unique elements

After the completion of the first phase, we have a sequence that indicates which elements in the input are duplicates and which elements are unique. We now need to compute the subranges in the output that individual processors will work with. This is done so that the unique elements in the input can be written to a single, uninterrupted sequence in the output.

It is during this phase that the two implementations of **p.unique.copy** begin to differ. In the first implementation, the processors take a shared-memory view of the output container when writing data. In the second implementation, each processor is made aware of only its assigned subrange of the output container. We describe the two implementations in the following sections.
4.2.1 First implementation

We perform a parallel prefix on the marker sequence using the integer addition operator. The result of this parallel prefix is another sequence of integers, \( \{c_1, c_2, \ldots, c_n\} \). In this new “count” sequence, each integer \( c_i \) indicates the number of unique elements that exist previous to the corresponding element \( x_i \) in the input. We then divide the input into \( p \) subranges of equal size and assign each subrange to a different processor. Each processor then reads the element \( c_j \) in the count sequence that corresponds to the first element \( x_j \) in its subrange of the input to determine the position in the output sequence that it will begin writing data to.

After each processor has determined where it will begin writing data to in the output, the unique elements in the input are copied to the output. This is done in a simple manner. Each processor traverses its input subrange and the corresponding subrange of the marker sequence simultaneously. When the processor encounters a unique marker, the corresponding element in the input is written to the output. This implementation of the copy phase is summarized as follows:

Algorithm 6 Copy phase, first implementation

1: Perform parallel prefix on the marker sequence, get the count sequence in return.
2: Assign each processor a subrange of the input.
3: Each processor checks the count sequence to determine the position in the output that it will begin writing data at.
4: Each processor copies the unique elements from its input subrange to the output.

4.2.2 Second implementation

Currently, the second implementation works as follows: first, the input is divided into \( p \) subranges of equal size. Then the total number of unique elements per subrange is computed. This is done by having each processor traverse its subrange of the marker sequence while counting the number of unique markers. The result is a “count” sequence \( \{c_1, c_2, \ldots, c_p\} \). This count sequence differs from the first implementation in that it first has \( p \) elements rather than \( n \) elements. Second, each element \( c_i \) in this count sequence is an integer that indicates how many total unique elements there are in a corresponding subrange of the input. If added together, the accumulated elements of this count sequence give the size of the final output of \( \text{p\_unique\_copy} \).

After the accumulation is done, one processor uses the count sequence to compute \( p \) separate subranges in the output, each corresponding to one of the \( p \) subranges of the input. Each output subrange, along with the appropriate input subrange and marker subrange, is assigned to a separate processor. Each processor then copies the unique elements in the input to the output in the same manner as for the first implementation. This implementation of the copy phase is summarized as follows:

Algorithm 7 Copy phase, second implementation

1: Assign each processor a subrange of the input.
2: Compute the number of unique elements in each subrange using the marker sequence.
3: One processor computes the subranges in the output corresponding to the input subranges and assigns the output subranges accordingly.
4: Each processor copies the unique elements from its input subrange to the output.

When all processors have finished, \( \text{p\_unique\_copy} \) returns the output sequence.

5 Analysis

We are also interested in the theoretical time complexity of \( \text{p\_unique\_copy} \). We provide an analysis of the \( \text{pAlgorithm} \)’s time complexity based on the different cases, as well as based on the differences between the two implementations of \( \text{p\_unique\_copy} \) at the copy phase. The overall running time can be expressed as
$O(T_{\text{Phase1}} + T_{\text{Phase2}} + T_{\text{Copy}})$, where $T_{\text{Phase1}}$ is the amount of time it takes to identify duplicate elements, $T_{\text{Phase2}}$ is the amount of time it takes to compute the output subranges that each processor will write to, and $T_{\text{Copy}}$ is the time it takes to copy the unique elements to the output.

First, let us assume that data is uniformly distributed across the machine. We begin by creating the marker sequence $M$. In STAPL, this initialization is performed in $O(n/p)$ time. After this is the identification phase. As explained in Section 4, there are three cases to this phase.

### 5.1 Identification

#### 5.1.1 Case 1

In the first case, all that occurs is the duplicate identification algorithm. Each individual task in the algorithm performs a single comparison and a single assignment. There are $n - 1$ tasks which can be divided among the number of processors $p$ available. The Case 1 duplicate identification algorithm can thus be done in $O(n/p)$ time.

#### 5.1.2 Case 2

In the second case, the prefix sequence $P$ is constructed and initialized - this again takes $O(n/p)$ time. Then the parallel prefix algorithm is performed; this is known to take $O(n/p + \log p)$ time. After the prefix completes, the Case 2 duplicate identification algorithm is performed. Like in Case 1, this algorithm creates $n - 1$ tasks, each of which performs a single comparison and a single assignment. These tasks can be divided among the available $p$ processors. The duplicate identification algorithm takes $O(n/p)$ time. Case 2, altogether, runs in $O(n/p + \log p)$ time.

#### 5.1.3 Case 3

Only the Case 3 duplicate identification algorithm is used in this case. As previously explained, this algorithm performs a linear traversal of the input sequence. A total of $n - 1$ comparisons are done and a total $n - 1$ assignments are made. In this case, the duplicate identification algorithm runs in $O(n)$ time.

### 5.2 Subrange computation and copy phase

After the duplicate elements are identified, the algorithm computes the output subranges and then copies the unique elements to the output. There are two implementations of this phase.

#### 5.2.1 First implementation

A parallel prefix is performed on the marker sequence, yielding the count sequence $C$. This takes $O(n/p + \log p)$ time. Each processor determines where it will begin writing data to in the output, then the actual copy is performed. This should run in $O(n/p)$ time. It should be noted, however, that if the output container does not provide random access then this phase can potentially take $O(n)$ time because each processor has to traverse the output until it reaches its starting position.

#### 5.2.2 Second implementation

First, the input into $p$ equal-sized subranges. Then the total number of unique elements per segment is computed. This takes $O(n/p)$ time. After this is done, one processor uses the results of these accumulations to compute $p$ separate subranges in the output, each corresponding to one of the $p$ subranges of the input. Each of the $p$ subranges can be computed in constant time, so this step takes $O(p)$ time. We note that this $O(p)$ time is for the current implementation, and that this can be improved to $O(\log p)$. Each output subrange, along with the appropriate input subrange and marker subrange, is assigned to a separate processor. Each processor then traverses its input and copies elements from the input to the output based on the contents of the marker sequence. This final step runs in $O(n/p)$ time.

Overall, there are six different cases for p\texttt{unique}}\_\texttt{copy}, counting the fact that there are two implementations of the p\texttt{Algorithm}. Assuming that the provided p\texttt{Containers} do provide random access, only the
third case will run in $O(n)$ time. The summarization of the theoretical complexity for each case, then, is as follows:

<table>
<thead>
<tr>
<th>Case</th>
<th>First implementation</th>
<th>Second implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$O(n/p + \log p)$</td>
<td>$O(n/p + \log p)$</td>
</tr>
<tr>
<td>2</td>
<td>$O(n/p + \log p)$</td>
<td>$O(n/p + \log p)$</td>
</tr>
<tr>
<td>3</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

6 Experimental Setup

Both implementations of `p_unique_copy` for cases 1 and 2 were tested. The relation we used for the Case 1 tests was the integer equality relation ("="). For Case 2, we used the less-than ("<") integer relation. We also present results of an evaluation of `p_copy_if` for comparison purposes. `p_copy_if` copies all elements in a specified range of data that satisfy a given unary relation $\oplus$ to an output container. `p_copy_if` is expected to run in $O(n/p + \log p)$ time, since it employs the parallel prefix to compute output subranges like `p_unique_copy` does.

All performance tests were run on an IBM Cluster 1600, designated "Hydra". Hydra has 52 p5-575 nodes. The p5-575 node is a high-performance, shared-memory multi-processor (SMP) running the 64-bit version of AIX 5L (5.3) as a single system image. Each node consists of 16 Power5+ processors running at 1.9GHz. Forty-nine of the nodes have 32 GB of DDR2 DRAM and 3 nodes have 64GB of DDR2 DRAM.

In all experiments, the STAPL `pVector` was used as both the input and the output `pContainer`. For `p_unique_copy`, the input was 10 million elements in size. For `p_copy_if`, the input was 100 million elements in size. We will next describe the parameters that were varied during the experiments.

Number of processors The number of processors was varied from 1 processor to 64 processors. We can show strong scaling by keeping the number of elements in the input container constant as the number of processors increases.

Distribution of data We are able to control how the data is distributed in the `pContainers`. For these initial experiments, however, we use the default distribution in all cases. In the default distribution, data is evenly distributed across the processors.

Percentage of elements copied For `p_copy_if` and both implementations of `p_unique_copy`, we can vary the percentage of elements that are actually copied to the output. We distribute the elements that are going to be copied evenly across the processors. Because both algorithms try to compact the data together in the output container, we can use this parameter along with control of the distribution of data to control the amount of remote write operations that are invoked during the copy phase of each `pAlgorithm`.

Anywhere_inserter We also run performance evaluations on test cases where the `pAlgorithms` use the STAPL `anywhere_inserter` to write data to the output. STAPL `anywhere_inserter` is a construct for parallel insertions, and it allows us to make some performance optimizations when a `pAlgorithm` uses it to write data to an output container. When `anywhere_inserter` is used, no guarantees are made about the order of the data in the output. The main optimization that this implies is that we can skip the subrange calculation phase and write unique elements to the output as we discover them.

7 Experimental Results

In this section, we present the results of performance evaluations for `p_unique_copy`. We would like to note that these results are preliminary since the STAPL framework was unable to provide certain optimizations at the time that we performed these tests.
7.1 \texttt{p\_unique\_copy}, first implementation

In Figure 2 we can see the scalability for \texttt{p\_unique\_copy}'s Case 1 performance, up to 64 processors, for 0\%, 50\%, and 100\% of elements copied. Note that 0\% means that only the default initial element in the input was unique. One unexpected result is the scalability of the \texttt{pAlgorithm} when 0\% of the elements are copied. It is not known at this time why the \texttt{pAlgorithm} scales less well for this test case in comparison to when 100\% of the elements are copied. We expect that the 0\% test case would scale at least as well as the 100\% test case, since the only difference between the two at run-time is the amount of data writes that are performed. One hypothesis is that since the 0\% test case runs faster on one processor than the 100\% test case (51.5 and 69.1 seconds, respectively), there is less room for improvement, proportionally, for the 0\% test case, and thus less scalability. This wouldn’t completely explain this phenomenon, however. Further investigation is required to understand what is happening.

Also shown is a scalability plot for \texttt{p\_copy\_if}, for comparison. In general, \texttt{p\_unique\_copy} does not scale as well as \texttt{p\_copy\_if}, though neither \texttt{pAlgorithm} scales as well as might be hoped. We would expect to see good scalability, based on their theoretical complexities.

There are a number of possibilities for this behavior. Initial investigation indicates that some serialization is occurring within \texttt{STAPL} during both cases 1 and 2 of the \texttt{pAlgorithm}. This would at least partially explain why the \texttt{pAlgorithm} doesn’t scale better given its theoretical complexity of $O(n/p + \log p)$ in the cases that were tested.

Another possibility is that this is due to the \texttt{STAPL} framework itself. There are a number of optimizations that have not yet been made, and we would like to see in the future the effects of these optimizations on \texttt{p\_unique\_copy}'s and other \texttt{pAlgorithm}'s performance.

Here in Figure 3 we see the scalability of \texttt{p\_unique\_copy}'s performance when the relation falls under
Case 2, also for when 0%, 50%, and 100% of the input is unique. These test cases scale similarly to their counterparts in Figure 2.

Of more interest for both Case 1 and 2 is the test case where 50% of the elements are unique. Slowdown occurs as the number of processors is increased from 32 to 64. This is because the algorithm compacts the unique elements together in the output, causing remote data copies to occur since the unique elements are distributed evenly across the processors. The appearance of this slowdown is somewhat unexpected for such a small number of processors, but is perhaps also due to some of the topics that have been discussed thus far.

![Figure 4: Plots of execution time and scalability for p_unique_copy and anywhere_inserter](image)

In Figure 4 we show the execution time and scalability of the first implementation of p_unique_copy, both with and without the usage of anywhere_inserter. The pAlgorithm using anywhere_inserter executes faster than without it, but we note that there is no decrease in execution time from 1 to 2 processors in the anywhere_inserter case. This seems to be due to the same serialization issue noted above, though further investigation is necessary to be sure. Another point of interest is the slowdown from 32 to 64 processors. We believe that this is due to some communication that is occurring through the usage of anywhere_inserter. This is because of communication that is invoked by anywhere_inserter. We should be able to reduce this in the future.

### 7.2 p_unique_copy, second implementation

In general, each test case of the second implementation of p_unique_copy is very similar to its analogous test case in the first implementation. There is, however, a notable difference.

![Figure 5: Scalability plots for the second implementation of p_unique_copy](image)

A close look at the scalability plots in Figure 5 will show that this implementation didn’t scale quite as well as the first implementation. This may be due to the manner in which the subrange computation was...
implemented. Recall that a single processor was used to read the count view and to compute the output subranges. Although the complexity of this method didn’t affect the overall theoretical complexity, the sequential manner in which this was implemented may be slow enough to affect the scalability.

8 Conclusion

In this paper, we’ve presented the design, implementation, and analysis for two implementations of a STAPL pAlgorithm. p_unique_copy is a parallelization of the STL unique_copy algorithm. It is a non-trivial parallelization; the pAlgorithm performs a different amount of parallel work based on its usage.

We have also presented a set of initial experimental results for p_unique_copy, along with a small comparison to another pAlgorithm, p_copy_if. Although the results do not fully support its theoretical analysis, we believe that, with further improvement and optimizations, p_unique_copy will be an efficient, scalable pAlgorithm that can be used in a variety of situations.

Such improvements include fine-tuning some of the subalgorithms, such as how the subrange computation is performed in the second implementation of the algorithm. We also would like to implement the “linear chain of tasks” mentioned in Section 4.1.3 for Case 3 of p_unique_copy. In addition, we would like to reduce the amount of computation necessary in Case 3 by performing the copy while the duplicate elements are being identified.

After making these improvements to the pAlgorithm, and with more optimizations to the STAPL framework, we would like to perform a more in-depth characterization and analysis of p_unique_copy, as well as other pAlgorithms. There are many things to investigate with respect to performance, including other data distributions, usage of other pContainers, and anywhere_inserter.

References


