Evaluating the Usability and Performance of

**STAPL**

Brandon Caffie, Adam Fidel, Harshvardhan, Timmie Smith,
Robert Metzger, Nancy M. Amato, Lawerence Rauchwerger

Abstract

The Standard Template Adaptive Parallel Library (**STAPL**), a parallel programming framework written in the C++ Programming Library, allows parallel programmers the opportunity to create productive program applications with simplicity and expressivity. This research takes a deeper look into **STAPL** by assessing the usability and performance of very recognizable and simple programs. Five problems were taken from a computational and mathematical website, implemented using **STAPL**, and was then analyzed based on common parallel programming testing mechanisms, including execution time and scalability. We show that **STAPL** provides good usability and performance, as an undergraduate student who was at first unfamiliar with **STAPL** was able to create the code for each program and produce fast and scalable results.

Introduction

Modern technologies demand faster computing power. Today’s computer systems are designed with multiprocessor and multi-core architectures that execute software applications concurrently, or in parallel [1]. In order to allow computing power to reach its potential, there is a push to run applications in parallel rather than sequential. There are challenges that released parallel libraries face when attempting to make this push. One challenge these libraries encounter is the continuous increase of problem complexity and size for parallel programs. As the amount of data manipulation to solve real-world problems grows, it will cause the difficulty for dynamic programs to be harder to solve, and thus constant refinement of parallel programs are necessary. Programmability, which is defined as the process of simplifying parallel program development, is another impediment to developed parallel libraries as it is insufficient [1]. In addition, the portable performance of parallel libraries are lacking as scalability and efficiency are usually poor.

Parasol Laboratory at Texas A&M University are in the process of creating a parallel library in the C++ Programming Language that is focused on solving these challenges. Called the Standard Template Adaptive Parallel Library (**STAPL**), this parallel programming framework encapsulates high-level abstraction from users on a consistent interface across various memory systems. **STAPL** features include hiding detailed parallelism from the user for ease of use, inheriting traditional STL algorithms to run in parallel for efficiency, and having the portability to transfer **STAPL** code to any computer device [1].
This research illustrates the simplicity of STAPL by evaluating the usability and performance of simpler computational problems. These problems are listed on a website named Project Euler, which is where programmers challenge themselves to produce functional, timely code [4]. We will attempt to show that STAPL is adaptive for novice parallel programmers to use while also demonstrating scalable performance for each Project Euler program created in parallel.

Background

Developers using STAPL will have access to pContainers, pViews, and pAlgorithms. pContainers, or parallel containers, are containers similar to that of sequential STL with one important feature of hosting a shared object view. This means that these containers are shared in global address space by using a global identifier (GID) to track each element in the container [2]. pViews are abstract data types to reference pContainers, and can be thought of as a iterator space over a container. By decoupling the container and running a variety of algorithms over a collection of elements, pViews provide operations to interact over data [3]. Derived from its STL equivalent, pAlgorithms are executed over specified views via work function (function object) to achieve desired parallelism.

Methods

The following Project Euler Problems were implemented in parallel:

- Problem 1: Finding the sum of all multiples of 3 or 5 below \( n \).
- Problem 2: Finding the sum of even Fibonacci numbers below \( n \).
- Problem 3: Finding the largest prime factor of number \( n \).
- Problem 5: Finding the smallest positive number that is evenly divisible by all numbers between 1 and \( n \).
- Problem 7: Finding the \( n \)th prime.

An algorithm was created for each Project Euler problem in order to understand and realize the parallel implementation. These distinctive algorithms were then used in the next step as a template for creating the code in parallel. After refactoring and refining the code, tests were run on each program to determine the runtime and scalability.
Create array container from 1 to \( n \);
\[
\begin{align*}
&\text{for element } \in \text{container do} \\
&\quad \text{if element is a Fibonacci AND even number then} \\
&\quad \quad \text{return element;}
\end{align*}
\]
\[
\begin{align*}
&\quad \text{else} \\
&\quad \quad \text{return 0;}
\end{align*}
\]
return sum of even Fibonacci numbers;

Create array container from 1 to \( n \);
\[
\begin{align*}
&\text{for element } \in \text{container do} \\
&\quad \text{if element is a factor of } n \text{ AND is prime then} \\
&\quad \quad \text{return element;}
\end{align*}
\]
\[
\begin{align*}
&\quad \text{else} \\
&\quad \quad \text{return 0;}
\end{align*}
\]
return max prime factor of \( n \);

(a) Finding the sum of even Fibonacci numbers

(b) Finding largest prime factor of \( n \)

Figure 1: Sample algorithms of Project Euler problems implemented in parallel

Implementing each program using \texttt{STAPL} was relatively straightforward, mostly due to the simple transition from STL to \texttt{STAPL}. The following \texttt{STAPL} algorithms, with their respective STL counterparts, were used during the coding process:

- \texttt{stapl::iota} → \texttt{std::iota}
- \texttt{stapl::replace_if} → \texttt{std::replace_if}
- \texttt{stapl::accumulate} → \texttt{std::accumulate}
- \texttt{stapl::max<element>} → \texttt{std::max<element>}
- \texttt{stapl::remove_if} → \texttt{std::remove_if}
- \texttt{stapl::plus<>()} → \texttt{std::plus<>()}

In addition to the STL-based algorithms, \texttt{STAPL} also provides its own set of algorithms for users to implement, such as \texttt{stapl::map_func} and \texttt{stapl::map_reduce}. These algorithms give the user the opportunity to run code faster and more efficiently. It is for this reason that many, if not all, the Project Euler problems implemented has more than one version to highlight the versatility that \texttt{STAPL} provides. The version of each program that has the fastest execution time was tested for further analysis.

Challenges

Some Project Euler problems were not properly documented to be included in the Results section. This section will describe the most significant challenges faced that ultimately used the most time during our research.

Load Imbalance

While running code for Project Euler Problem 3, which involves finding the largest prime factor of a number, we realized that the time to execute the code for large numbers over ten
thousand was taking a long time to execute. This was due to the different amounts of work required to find the prime factor for each element. The work required to find the number of prime factors of 6, for example, is a lot less than the work to find the prime factors of 100. After debugging the code line-by-line and finding the execution times for each, we found that our algorithm to find the prime factors of a given number was too slow to execute for large \( n \). With the initial data structure of our program using a \texttt{stapl::array<>} with a large number of elements, we have found that distribution of elements to different locations were uneven, and thus was imbalanced. Also called load imbalance in parallel programming, the result of improper distribution of elements causes the total execution of the program to be based on the location with the longest execution time. In terms of the program itself, the load imbalance comes from the increase of prime factors as the number of data increase. This issue also occurred in Project Euler problems 5, and 7, for they involve prime factors.

In an attempt to eliminate this issue, a partitioning strategy was used to evenly split the amount of data to different cores. Our first approach involves incorporating the following formula into code that will give us the most optimal partition amongst the data elements:

With inputs \( n = \) number of data and \( p = \) number of cores:

Output would be a set: \( s = \{p_0, p_1, \ldots, p_{p-1}\} \)

Where \( p_i = [a_i, b_i], 0 < a_i, b_i < n \)

such that:

\[
\max \left( \frac{\sum_{i=n}^b i}{\sum_{i=n}^b i} \right)
\]

Assuming the initial partitions have taken place to evenly distribute the number of elements per location, the formula above finds which location contains the greatest set of elements and switches the smallest element with that location with the its previous location. For a huge number of data elements, this would take a long time to execute, as this would happen one iteration at a time, and thus was inefficient in partitioning the data.

Our second approach to this was implementing the partition in a cyclic manner. Each element was put in a location that had the lowest sum in order to allow the same amount of data in each location. This ultimately dropped the time for large numbers of data, as it was dispersed in the most equal fashion.

![Figure 2: Example of 6 elements using cyclic partitioning](image)
Parallelism vs. Efficiency

For Project Euler Problem 5, which involves finding the smallest multiple that is evenly divisible by all numbers between 1 and n, the program was originally implemented using brute-force. The brute-force method uses the most obvious method to solving a problem; it is almost always is not the best method to solving a problem efficiently. In parallel, this method normally traverses through a data structure and applies a pAlgorithm (such as map_reduce) to the pView via work function. Some of the Project Euler problems, like Problem 1 for example, applies the brute force method well, primarily due to the execution of a simple work function. Problem 5 did not run well with brute force, especially for large numbers (N), as the time to execute would take days to fully execute.

Create range \( r \);
Initialize beginning element to 0;
Initialize iterations to 1;
while forever do
    for element \( \in \) range do
        if element is divisible by all numbers 1 to \( n \) then
            return element;
        add index of beginning element to the \( r \);
    increment iterations by 1;

Algorithm 1: Brute Force algorithm of Project Euler Problem 5

The amount of parallelism in this problem had to be limited in order to run the problem more efficiently and in less time. What we did to obtain our desired result included a plan to run the program over \( n \) elements to acquire the result, rather than checking every number endlessly until we find the first number that is divisible by every number from 1 to \( n \).

In our improved method, prime factorization was used for each number from 1 to \( n \), and the factorization results were stored in an array of maps that included the list of prime numbers and the number of occurrences in each. This data structure, along with using an array of counters, was used to find the result of the smallest multiple via multiplication. Each iteration involves finding the max between the number of occurrences of a prime number in the array of maps and the current integer stored in the counter array. Algorithm 2 exemplifies this process.
Create primes factors from 1 to \( n \);
Initialize array of counters with \( n \) elements;
Initialize ans to 1;

\[
\text{while } \text{number } \leq n \text{ do}
\begin{align*}
\text{for eachprimefactor } & \in \text{number} \text{ do} \\
\text{counter[number-1]} &= \text{find} \_\text{max}\{\text{counter[number-1]}, \text{prime factor occurrence}\};
\end{align*}
\]

\[
\text{while } \text{number } \leq n \text{ do}
\begin{align*}
\text{ans } &\ast= (\text{number} \_\text{counter[number-1]}) \\
\text{return ans;}
\end{align*}
\]

Algorithm 2: Enhanced algorithm of Project Euler Problem 5

Although some parallelism was deprived from this problem, a better algorithmic approach of this problem resulted in faster execution time and better performance.

**Experimental Results**

Determining the success of executing the Project Euler programs in **STAPL** will rely on runtime, strong-scalability and weak-scalability testing. The machine that will produce the results for this experiment is a 576 core system, locally named **CRAY**.

Although numerous programs were coded, the results for the first two programs will be shown, as proper documentation was not made in time for the other programs due to time constraints.

For each testing principle, a total of 32 iterations was executed. This was to eliminate biases and obtain precise, accurate results.

**Runtime**

Each program was tested with various amounts of data up to billion elements. The execution time of each amount of data was analyzed as the the amount of cores doubled. Tables 1 and 2 along with the visuals represented by Figure 3a and 3b respectively displays typical charts and graphs of the runtime of the Project Euler problems.
<table>
<thead>
<tr>
<th></th>
<th>99</th>
<th>999</th>
<th>9999</th>
<th>99999</th>
<th>999999</th>
<th>9999999</th>
<th>99999999</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.40998E-5</td>
<td>3.775E-5</td>
<td>7.44687E-5</td>
<td>4.11656E-4</td>
<td>3.80756E-3</td>
<td>0.0375361</td>
<td>0.374931</td>
</tr>
<tr>
<td>2</td>
<td>9.54687E-5</td>
<td>9.6E-5</td>
<td>1.14E-4</td>
<td>2.76719E-4</td>
<td>1.97178E-3</td>
<td>0.0188286</td>
<td>0.187169</td>
</tr>
<tr>
<td>4</td>
<td>1.31E-4</td>
<td>1.31219E-4</td>
<td>1.4E-4</td>
<td>2.22562E-4</td>
<td>1.06191E-3</td>
<td>9.52656E-3</td>
<td>0.0939813</td>
</tr>
<tr>
<td>8</td>
<td>1.86344E-4</td>
<td>1.78969E-4</td>
<td>1.85125E-4</td>
<td>2.36312E-4</td>
<td>7.07437E-4</td>
<td>5.41678E-3</td>
<td>0.0519532</td>
</tr>
<tr>
<td>16</td>
<td>2.20658E-4</td>
<td>2.51469E-4</td>
<td>2.41875E-4</td>
<td>2.48469E-4</td>
<td>4.97031E-4</td>
<td>2.85972E-3</td>
<td>0.0264027</td>
</tr>
<tr>
<td>32</td>
<td>3.24656E-4</td>
<td>3.26094E-4</td>
<td>3.08062E-4</td>
<td>3.14563E-4</td>
<td>4.94437E-4</td>
<td>1.79591E-3</td>
<td>0.0137933</td>
</tr>
<tr>
<td>64</td>
<td>4.15625E-4</td>
<td>3.71437E-4</td>
<td>3.9925E-4</td>
<td>4.33219E-4</td>
<td>4.69312E-4</td>
<td>1.93666E-3</td>
<td>0.044138E-3</td>
</tr>
<tr>
<td>128</td>
<td>5.5225E-4</td>
<td>4.97562E-4</td>
<td>5.16563E-4</td>
<td>4.47406E-4</td>
<td>5.815E-4</td>
<td>8.56031E-4</td>
<td>3.97394E-3</td>
</tr>
<tr>
<td>256</td>
<td>5.97125E-4</td>
<td>4.88062E-4</td>
<td>5.54156E-4</td>
<td>5.43281E-4</td>
<td>6.66906E-4</td>
<td>2.312E-3</td>
<td></td>
</tr>
<tr>
<td>512</td>
<td>6.54469E-4</td>
<td>5.78375E-4</td>
<td>5.72156E-4</td>
<td>5.73406E-4</td>
<td>6.90312E-4</td>
<td>1.49475E-3</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Execution Time (in seconds) of Project Euler Problem 1 for n elements with p processors.

<table>
<thead>
<tr>
<th></th>
<th>900</th>
<th>9000</th>
<th>90000</th>
<th>900000</th>
<th>9000000</th>
<th>90000000</th>
<th>900000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7.11875E-5</td>
<td>3.99469E-4</td>
<td>3.67478E-3</td>
<td>0.036296</td>
<td>0.361639</td>
<td>3.61579</td>
<td>36.4298</td>
</tr>
<tr>
<td>2</td>
<td>1.20344E-4</td>
<td>3.59531E-4</td>
<td>2.74881E-3</td>
<td>0.0265592</td>
<td>0.264558</td>
<td>2.64559</td>
<td>26.4449</td>
</tr>
<tr>
<td>4</td>
<td>1.40906E-4</td>
<td>2.6025E-4</td>
<td>1.45278E-3</td>
<td>0.0133637</td>
<td>0.132366</td>
<td>1.32238</td>
<td>13.2225</td>
</tr>
<tr>
<td>8</td>
<td>2.01563E-4</td>
<td>2.67437E-4</td>
<td>8.91094E-4</td>
<td>7.2535E-3</td>
<td>0.0697762</td>
<td>0.696138</td>
<td>6.95948</td>
</tr>
<tr>
<td>16</td>
<td>2.29781E-4</td>
<td>2.79344E-4</td>
<td>6.04031E-4</td>
<td>3.87388E-3</td>
<td>0.035219</td>
<td>0.348481</td>
<td>3.48138</td>
</tr>
<tr>
<td>32</td>
<td>3.28938E-4</td>
<td>3.40969E-4</td>
<td>5.53625E-4</td>
<td>2.46E-3</td>
<td>0.0212311</td>
<td>0.210363</td>
<td>2.13763</td>
</tr>
<tr>
<td>64</td>
<td>3.95281E-4</td>
<td>4.25375E-4</td>
<td>5.07094E-4</td>
<td>1.43887E-3</td>
<td>9.25747E-3</td>
<td>0.0876315</td>
<td>0.870599</td>
</tr>
<tr>
<td>128</td>
<td>4.34219E-4</td>
<td>4.48125E-4</td>
<td>5.0625E-4</td>
<td>8.9775E-4</td>
<td>4.93769E-3</td>
<td>0.0440985</td>
<td>0.436699</td>
</tr>
<tr>
<td>256</td>
<td>5.84562E-4</td>
<td>5.78094E-4</td>
<td>6.10188E-4</td>
<td>7.72375E-4</td>
<td>3.04347E-3</td>
<td>0.0224767</td>
<td>0.218105</td>
</tr>
<tr>
<td>512</td>
<td>5.93406E-4</td>
<td>6.08313E-4</td>
<td>5.81812E-4</td>
<td>6.49438E-4</td>
<td>1.73441E-3</td>
<td>0.0114888</td>
<td>0.109439</td>
</tr>
</tbody>
</table>

Table 2: Execution Time (in seconds) of Project Euler Problem 2 for n elements with p processors.

(a) Project Euler Problem 1.  (b) Project Euler Problem 2.

Figure 3: Runtime results for Project Euler problems
**Strong-Scalability**

During strong-scalability testing, we analyzed whether or not a given program can be computed faster as more cores are used to execute it. Data sizes were another important factor in testing strong-scalability, to understand how scalability is affected as the number of elements increases.

Figure 4 displays examples of scalability for increasing values of \( n \). Scalability was achieved by using the following equation:

\[
Scalability = \frac{time\ of\ P_1}{time\ of\ P_n}
\]

where \( n \) is the number of processors used to execute the program.

A linear slope of negative scalability shows that time increases as the number of processors increase. On the contrary, a linear slope of positive scalability means that time decreases as the number of processor increases; this is the ideal shape we are looking for in analyzing strong-scalability.

![Figure 4](image)

Figure 4: Strong-Scalability Results of Project Euler Problem 2 on various amounts of data

**Weak-Scalability**

The importance of weak-scalability testing is to validate execution time consistency as the data increases by a factor of the number of cores used. Every core will ideally consists of the same number of data elements.
As a simple example to understand weak-scalability in detail, let’s say program A will be running computation off of one core, having one million elements. Increasing the amount of cores by a factor of 32, for example, will present a total of 32 million data elements running in program B, with each core still holding one million elements. By the end of execution, the times from program A and program B should be very similar. The ideal slope of a weak-scalability graph should run in constant time.

Figure 5 shows the weak-scalability results of the Project Euler problems.

![Figure 5: Strong-Scalability Results of Project Euler Problem 2 on various amounts of data](image)

Discussion

As the results show, the usability and performance of Project Euler Problems 1 and 2 gives us a fair assumption that STAPL can be used by developers on any level and still present parallel capabilities that will be needed to solve problems of relatively any difficulty. Runtime results shown in Figures 3a and 3b ultimately show that STAPL provides fair performance for small data sets but even better performance for larger ones. Scalability results similarly favors larger amounts of data, as Figures 4 and 5 show that programs are more scalable as the data size increases.

What explains the significant increase in Figures 5 of Weak-Scalability is added communication from core to core and from node to node. Briefly, a node is a computer containing multiple cores; in the CRAY system each compute node holds either 16 or 32 cores. As more cores are added, data will then be distributed to different cores and/ or nodes. That data has to travel to their location and must also know where the other data elements are in the process. It is this that results in an increase of time for smaller number of cores. Beyond that the network has an idea of where the elements are, which explains the constant slope taking place afterwards.

In terms of usability, writing STAPL required little effort, primarily because the transition from STL to STAPL is very straightforward. The results from this study are from an undergraduate student who had little experience in the C++ programming language and no experience in parallel programming. Throughout the period of our research the person quickly gained knowledge of STAPL based on his previous experience with C++ STL and was able to implement these problems without having to fully understand the underlying
parallelization that occurs.

Conclusion

This paper successfully shows that STAPL is very simple to use and also produces the usability and performance that novice developers need when creating their first few programs in parallel. By measuring the runtime and scalability of everyday programs in parallel, we have shown the simplicity and expressivity of a parallel programming framework.

In the future, it would be interesting to see other implementations of the same programs that were created during this research, and determine whether or not those implementations are faster. Although STAPL is used to solve real-world problems, it facilitates the creation of parallel programs so that developers on any level can easily create parallel programs for their domain.

Acknowledgements

This research supported in part by NSF Grants CRI- 0551685, CCF-0833199, CCF-0830753, IIS-096053, IIS- 0317266 by the DOE NNSA under the Predictive Science Academic Alliances Program by grant DE-FC52- 08NA28616, by Chevron, IBM, Intel, and HP. 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.

References


