Implementing Asynchronous Prefix Scan Algorithm in HPX Execution Model

Submitted by:
Chen Guo

Major Professor:
Prof. Hartmut Kaiser

Committee Members:
Prof. Evangelos Triantaphyllou
Prof. Feng Chen

Department of Computer Science, Louisiana State University
10/19/2014
Acknowledgements

I am pleased to acknowledge Prof. Hartmut Kaiser for his invaluable guidance during the course of this project work and for being incredible mentor for me both in the classroom and on the research field.

I also extend my sincere thanks to Prof. Evangelos Triantaphyllou and Prof. Feng Chen for serving the committee members of this project. Their continuous support throughout the project plays an important role that keeps me going.

Last but not least, I am grateful for my wife Yao Wei and my two-year-old son Aaron. They have always been the source of my motivation and the origin of my inspiration. I thank god every day for bringing them to my life!

October 2014

Chen Guo
Contents

ACKNOWLEDGEMENTS................................................................................................................. 2

CONTENTS........................................................................................................................................ 3

CHAPTER 1: INTRODUCTION............................................................................................................. 4

OVERVIEW........................................................................................................................................ 4
BACKGROUND AND MOTIVATION.................................................................................................... 4
OBJECTIVE......................................................................................................................................... 5
METHODOLOGY................................................................................................................................. 5

CHAPTER 2: TOOL DESCRIPTION..................................................................................................... 6

ENVIRONMENT AND DEVELOPING PLATFORM................................................................................ 6
C++14 STANDARD LIBRARY ............................................................................................................. 7
BOOST LIBRARY.............................................................................................................................. 8
HPX EXECUTION MODEL.................................................................................................................. 8

CHAPTER 3: ALGORITHM DESIGN AND IMPLEMENTATION............................................................ 9

SEQUENTIAL PREFIX SCAN............................................................................................................... 9
PARALLEL PREFIX SCAN................................................................................................................ 13
ASYNCHRONOUSLY PARALLEL PREFIX SCAN.............................................................................. 19

CHAPTER 4: PERFORMANCE ANALYSIS.......................................................................................... 23

PARALLELISM CAN BE S.L.O.W..................................................................................................... 23
STANDARD LIBRARY PERFORMANCE............................................................................................ 26
BOOST LIBRARY PERFORMANCE.................................................................................................... 30
HPX EXECUTION MODEL PERFORMANCE....................................................................................... 33

CHAPTER 5: APPLICATION AND CONCLUSIONS........................................................................... 36

APPLICATION OF PARALLEL PREFIX SCAN.................................................................................. 37
CONCLUSION................................................................................................................................. 39
FUTURE DIRECTION......................................................................................................................... 39

REFERENCES.................................................................................................................................... 40
Chapter 1: Introduction

Overview

This report describes and discusses the process of seeking a work-efficient design for the parallel prefix scan algorithm that would fit into the HPX (High Performance Parallel) execution model for C++ users, and the analytic results of the work done during the process. It is part of the HPX project that is ongoing within the Center of Computation and Technology and the STELLAR-GROUP, which aims to provide a general purpose C++ runtime system for parallel and distributed application development of any scale.

Background and Motivation

Algorithm designers with years of experience often know how to rely on a set of building blocks and on the tools needed to put the blocks together into an algorithm, and many of the parallel algorithms eventually fall back to sequential algorithms to look for building blocks and tools needed. Prefix scan algorithm, also commonly known as prefix sum, is a useful building block for many algorithms including searching, sorting, and building data structures. Being able to run prefix scan algorithm in parallel grants software programmers and algorithm designers more flexibility and scalability when it comes to processing a large amount of data in parallel or designing other parallel algorithms that relies heavily on prefix scan algorithm to improve their work-efficiency.

While the research on implementing parallel prefix scan algorithm are nothing new, the novelty of my project is to search and identify a design pattern that runs the algorithm not only
in parallel, but also asynchronously in order to fully utilize the computing advantage brought by multi-core CPU devices. Having determined the design pattern to use, my project takes the next step and implements the asynchronous prefix scan algorithm with several compatible designing tools given a variety of choices in hand, including C++ standard library, Boost library, and the HPX execution model, a general purpose C++ runtime system for parallel and distributed application development of any scale. Compared to conventional execution models, HPX allows me to design and implement the prefix scan algorithm not only in parallel, but also in an asynchronous fashion so that the risk of "data race" or "broken invariant" during the execution is significantly reduced, and multi-core CPU can reach its full potential to optimize the process time for large scale of data when using this algorithm.

Objective

The final goal of this project is twofold:

1. Search and identify a work-efficient way to design an asynchronous version of the prefix scan algorithm and prove its superiority on computing efficiency by comparing the execution time against those of the sequential algorithm and the parallel algorithm.

2. Attempt to implement the algorithm using several compatible programming tools, and eventually include it in the HPX Execution Model such that developers can use it as a cornerstone for designing and building systems in a time efficient manner, and algorithm designers can further apply it to designing other parallel versions of generic algorithms.

Methodology
To achieve the above goals, a series of methodology need to be followed:

1. Designing and implementing the sequential version of the prefix scan algorithm that can be applied to any container type that has an input and output iterator, which shall serve as the building block for parallel and asynchronous algorithm designs.

2. Based on the sequential version of algorithm, design and implement the asynchronously parallel prefix scan algorithm using additions from the Boost library, and compare its performance with the synchronous version.

3. To further improve the performance, extending the previous design by utilizing APIs provided by HPX execution model. To do that, the code from the previous step need to be modified and restructured to conform to the new API’s syntax and principles.

4. Comparing the performance differences in terms of execution time, and analyze the reasons for those differences based on algorithm designs.

5. Briefly specifying how the parallel prefix scan algorithm can be utilized to simplify and improve the design of other algorithms including quick sort and unique sort.

Chapter 2: Tool Description

Environment and Developing Platform

This project is implemented and tested using a machine with Windows 8 64-bit operating system, and an I7 Intel CPU that has 8 Cores and 32GB RAM. This environment is intended to provide the project application with sufficient computation power for measuring its
performance under large scale of data. The whole application program is written in C++, and the IDE selected to compile and build the source code is Microsoft Visual Studio 2013. The reason C++ was chosen as the go-to language for implementing this program is because C++ is a low-level compiled language, which allows it to be compiled directly to a machine’s native code, making it one of the fastest languages in the world when optimized. Such language is more suitable for programs like device driver or very high performance programs that really need access to the hardware. Since the main objective of this program is to search and identify the design that brings better computing performance, it makes sense to choose a low-level compiled language like C++ to implement the algorithm over some other object-oriented programming (OOP) languages such as Java or C#, which relies on a just-in-time compiler (JIT-compiled) and involves quite a lot boxing and unboxing of object that takes extra CPU resources and time.

As for Microsoft Visual Studio 2013, it is chosen as the IDE due to its user friendly interface design and the full support of C++ standard library, which makes it very convenient and powerful platform to write, compile, build, and debug code.

**C++14 Standard Library**

C++14 is the informal name for the most recent revision of the C++ ISO/IEC standard, formally "International Standard ISO/IEC 14882:2014(E) Programming Language C++". C++14 is intended to be a small extension over C++11, featuring mainly bug fixes and small improvements, and is therefore the standard library upon which this program is built. C++14 standard library provides all the necessary tools needed to implement the *prefix scan*
algorithm, and it comes with a full support of multi-thread and asynchronous functions. That being said, C++14 standard library has its limit in terms of multi-thread optimization and asynchronous computing capability, which is why the following two additions are introduced into this application.

Boost C++ Library

As the name suggest, Boost C++ library is a set of open-source peer-reviewed portable C++ libraries that are intended to give C++ standard library a “boost” by increasing the task and data structure that C++ supports, including linear algebra, pseudorandom number generation, image processing, regular expressions, unit testing, and of course, multithreading. Many of Boost’s founders are on the C++ standards committee, which makes Boost library a trusty and safe addition for C++ standard library. One major advantage of Boost library is that most of its libraries are header based, consisting of inline functions and extensive use of templates, and as such do not need to be built in advance of their use. Compared to C++ standard library, Boost libraries offer a variety of features that significantly enhances the program’s asynchronous computing capability.

HPX Execution Model

As useful as Boost C++ library can be for this application, there is a trade-off in computing performance. Since the Boost C++ library has a heave rely on the use of templates to build its library, the runtime performance receives a penalty as it generally takes a much longer time for the compiler to locate and identify the correct template to use from the massive template pools, which creates barriers to high level of scalability. As a result, HPX (High Performance
Parallex execution model was introduced as the second additional tool to further improve this program’s scalability by providing APIs with more condensed template pool and restructured algorithm implementation. HPX aims to provide a portable and highly optimized programming model which smartly utilizes the available resources to achieve unprecedented levels of scalability. Since the HPX library strictly adheres to the C++11 Standards and leverages the Boost C++ libraries, it leads to a smooth transition for someone with C++ and Boost background to use.

Chapter 3: Algorithm Design and Implementation

Sequential Prefix Scan

Sequential prefix scan algorithm is a one of the most fundamental and widely used algorithms in object-oriented programming and algorithm design. Its basic form is composed of two sequences of numbers and a binary associative operator: one input sequence of numbers \{X_0, X_1, X_2, \ldots, X_n\} and an output sequence of numbers \{Y_0, Y_1, Y_2, \ldots, Y_n\} who shares the same size as the input sequence. In the most common scenario, the binary associate operator \( \oplus \) is replaced with an addition, which would lead to an algorithm also known as (aka) prefix sums in which each output number is the sum of all the prefix number in the input sequence.

\[
\begin{align*}
Y_0 & = X_0; \\
Y_1 & = X_1 + Y_0; \\
Y_2 & = X_2 + Y_1; \\
Y_3 & = X_3 + Y_2; \\
& \ldots
\end{align*}
\]
For instance, the prefix sums of the natural numbers are the triangular numbers:

<table>
<thead>
<tr>
<th>Input</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>......</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>10</td>
<td>15</td>
<td>21</td>
<td>......</td>
</tr>
</tbody>
</table>

To sum up this basic prefix sum algorithm, we get the formula: \( Y[i] = Y[i-1] + X[i] \). While this formula looks rather trivial, it forms the basis of the higher-order function of *scan* in functional programming and the parallel version.

Before the higher-order function get discussed, one thing needs to be noted is that *prefix scan* algorithm is made of two different versions: *inclusive prefix scan* (aka *inclusive scan*) and *exclusive prefix scan* (aka *exclusive scan*). The only difference between these two versions is whether each result in the output sequence includes the corresponding input operand in the partial operation. For instance, this previous example we see is an inclusive scan because each result includes the corresponding input operand in the partial sum:

<table>
<thead>
<tr>
<th>Input(X)</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>......</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output(Y)</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>10</td>
<td>15</td>
<td>21</td>
<td>......</td>
</tr>
</tbody>
</table>

In comparison, each result in the output sequence does not include the corresponding input operand under the exclusive scan algorithm, and the results would become:

<table>
<thead>
<tr>
<th>Input(X)</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>......</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output(Y)</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>10</td>
<td>15</td>
<td>......</td>
</tr>
</tbody>
</table>

It does not require much attention to notice that the initial output value in the exclusive scan example shown above is replaced with a 0, which does not provide user the necessary flexibility.
during the computation. Also like mentioned above, the higher-order function of scan requires more computing capability than addition operator. Therefore, extending the current algorithm to a more generalized version of *prefix scan* proves to be necessary.

In the more generalized version, two new parameters are introduced: the binary associative operation $\oplus$ which takes two arguments of the same type, and an initial value. The binary associative operation $\oplus$ can be either addition, multiplies, or even Boolean operation such as maximum or minimum. The initial value is used to initialize the first result in the output sequence, and it works both inclusive and exclusive scan algorithm. For instance, if the initial value is set to 3, the previous examples shown above would each generate a different set of output sequence:

- **Inclusive sums:**

<table>
<thead>
<tr>
<th>Input(X)</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>......</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output(Y)</td>
<td>4</td>
<td>6</td>
<td>9</td>
<td>13</td>
<td>18</td>
<td>24</td>
<td>......</td>
</tr>
</tbody>
</table>

- **Exclusive sums:**

<table>
<thead>
<tr>
<th>Input(X)</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>......</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output(Y)</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>9</td>
<td>13</td>
<td>18</td>
<td>......</td>
</tr>
</tbody>
</table>

Considering these previously discussed features and the fact that this parallel *prefix scan* algorithm, when falling back to sequential execution, should simulate the flexibility of a generic algorithm in the C++ standard library, the final implementation should work with any container type that possesses an input and an output iterators, such as an array or a linked list. As a result, the sequential *prefix scan* algorithm was implemented as follows:
A few key points to be noted about this algorithm:

1. The input and output container types are deduced by templates to work with the most primitive iterator types: input and output iterators;
2. *Iterator_traits<InputIterator>::value_type* is used to pre-define the type $T$ the algorithm can work with, which is the initial value type. It specifies that $T$ by default has to be a type that can be pointed at and dereferenced by an iterator;

3. *BinaryOp*, which represents the binary operation type, is also by default a `std::plus<T>` operation that takes two arguments with $T$ type.

4. Initial value *init* and binary operation *op* are supplied with default argument values $T()$ and `BinaryOp()`, each of which represents an instance of its type. In the case where these two arguments are not specified at the point of invocation, the default argument values will be used.

5. Within the for loop, `(void)` casts the result of `++first` which prevents any possibly defined comma operators (i.e. `InputIterator::operator,()`) from being used.

While the sequential *prefix scan* algorithm is quite straight-forward to use and easy to comprehend, it is not quite a time-efficient way to produce output results when the input number sequence becomes really large. The cause for the slow-down is simple: each *output*[$N$] has to wait for the result from previous *output*[$N$-1] to compute before the algorithm can continue, and thus the computing complexity of the computation is $O(N)$. As multi-core CPU becomes increasingly popular among the industry, the need for designing parallel algorithms that would approach the efficiency of the sequential algorithm, while still taking advantage of the parallelism in the CPU is obvious.

**Parallel Prefix Scan**

The parallel *prefix scan* algorithm that is currently known and used by most programmers and algorithm designers is the one presented by Prof. Blellock in 1990 based on an algorithmic
pattern that often appears in parallel computing: *balanced trees*. This pattern aims to build a balanced binary tree on the input data and comprises two “sweep” phases – *up sweep* and *down sweep* – to produce the *prefix scan* results. As a result, a binary tree with $N$ input nodes has $d = \log_2(N)$ depths with $2^d$ nodes at each depth. In the *up sweep* phase, the algorithm traverses the balanced tree from terminal nodes to root node computing partial operation at internal nodes of the tree. This phase looks as follows for *parallel prefix sums* algorithm:

It can be seen from the *balanced tree* above that at each depth, the sum of two internal nodes were computed to generate one upper-level node until the sum of the root node is computed, which also represents the sum of all the input sequence numbers. It is also worth noting that each node is made of a special data structure consisting of three property values: *index* of the input sequence, *sum* after each depth, and *fromleft* value that shall be used during
the *down-sweep* phase computation. This special node structure and the *up sweep* phase algorithm can be implemented in C++ as follows:

```c++
//Tree node structure//
Class TreeNode
{
    Public:
        TreeNode(): sum(0), fromleft(0), index(0) {}
        int sum;
        int fromleft;
        int index;
}

//up sweep phase implementation//
void up_sweep_phase(std::vector<TreeNode>& nodes)
{
    size_t size = nodes.size();
    for (size_t depth = 0; depth <= std::log2(size) - 1; ++depth)
    {
        size_t partition_size = (size - 1) % pow(2, depth+1) + 1;
        int k = 0;
        while (k<size - 1)
        {
            nodes[k + pow(2, depth + 1) - 1].sum += nodes[k + pow(2, depth) - 1].sum;
            k += partition_size;
        }
    }
}
```

The *up sweep* algorithm first *for* loops through each depth. Within each depth, the partition size is calculated based on the size of the input sequence and the current depth. This partition size is then used in the nested *while* loop to locate the specific pair of internal nodes for the *sum* calculation in order to produce the nodes of the next depth. Aside from the obvious difference in algorithm design, the most notable difference between the sequential version of the *prefix scan* and this parallel version is that the sequential version uses `<template>` to
deduce the type of input sequence container it accepts as arguments, and any container type that possesses *input* and *output iterators* is a valid argument type, while the parallel version only accepts container types that possess *random access iterators* such as `std::vector<>*`. The reason for the *random access iterator* is because of this step:

```
{... nodes[k + pow(2, depth + 1) - 1].sum += nodes[k + pow(2, depth) - 1].sum; ...}
```

This particular calculation requires the container indices to be advanced to specific positions by a certain distance, which is beyond the capability of *input* and *output iterators* as they can only be incremented by one position during each iteration.

The *down sweep* phase algorithm uses the same concepts as the *up sweep* phase as it traverses back down the tree from the root node to the terminal nodes, using the partial sums from the *up sweep* phase to produce the output scan results. This phase looks as follows for the previous *prefix sum* example:
The *down sweep* phase begins by assigning 0 to the *fromleft* value of the root node, and on each step, each node at the current level passes its own *fromleft* value to its left child, and the sum of its *fromleft* value and the *sum* value of its left child to its right child until the *fromleft* value of all the terminal nodes were computed. To produce the final output sequence, it requires one more step to calculate the sum of the original input value for each node and its corresponding *fromleft* value after the *down sweep* phase. The *down sweep phase* can be implemented in C++ as follows:

```cpp
//down sweep phase implementation//
void down_sweep_phase(std::vector<TreeNode>& nodes)
{
    size_t size = nodes.size();
    for (size_t depth = std::log2(size) - 1; depth >= 0; ++depth)
    {
        vector<TreeNode>::size_type partition_size = (size - 1) % pow(2, depth + 1) + 1;
        int k = 0;
        while (k < size - 1)
        {
            nodes[k + pow(2, depth) - 1].fromleft = nodes[k + pow(2, depth + 1) - 1].fromleft;
            nodes[k + pow(2, depth + 1) - 1].fromleft += nodes[k + pow(2, depth) - 1].sum;
            k += partition_size;
        }
    }
}
```

Despite the fact that the parallel *prefix scan* algorithm described above has been well known for a long time and widely used to improve computing efficiency on modern parallel hardware such as a GPU, there exists several limits about this algorithm design that make it not a desirable choice for the HPX execution model. First, the restriction on the type of input container and iterators that the algorithm can accept makes it not very versatile candidate as all
the generic algorithms in C++ standard library are expected to have the capability of deducing their argument types through the use of `<template>`. While HPX is yet to become a part of the generic algorithm library, it aims to provide the same versatility.

Second, this algorithm depends on two special data structures – *balanced tree* and *tree node* – to accomplish the implementation. The *balanced tree* data structure requires the size of input sequence to be \(X: X \in 2 \text{ to the power of } N, N>0\) for the algorithm to work properly, which is not preferable for applications that may generate random number of input data. Under scenarios where input size is random, the user has to find the biggest \(2 \text{ to the power of } N\) within the input size first by converting the input size to its binary number format via a function below:

```cpp
std::vector<bool> to_binary(int num)
{
    std::vector<bool> num_binary;
    while (true)
    {
        int remainder = num % 2;
        bool binary = remainder == 0 ? false : true;
        num /= 2;
        num_binary.push_back(binary);
        if (num == 0) break;
    }
    std::reverse(num_binary.begin(), num_binary.end());
    return num_binary;
}
```

By invoking this converting function, an integer number that represents the size of the input sequence shall be converted to its binary format and stored in a `std::vector<bool>` container. For instance, number 2000 is converted to \(\{1, 1, 1, 1, 1, 0, 1, 0, 0, 0\}\), in which the first 1 represents \(2^{10} = 1024\) and also the biggest \(2 \text{ to the power of } N\) within 2000. The parallel *prefix scan* algorithm shall work with the first 1024 input elements first before moving on to the
second biggest $2 \text{ to the power of } N$ within 2000, which is 512. This iteration continues so on and so forth until the size left is too small to justify a parallel algorithm and shall be processed sequentially. Not only does this lead to abundant separation and iteration among calculation, it is also quite tricky to justify the point where it should start processing input sequentially.

Last but not least, this parallel algorithm practically states that it processes two elements per thread, so the maximum size of the input sequence this algorithm can scan is determined by the machine’s hardware. For instance, the maximum array size this algorithm can scan on an NVIDIA 8 Series GPU is 1024 due to hardware limit. Given that the initial input size is 1024, 512 threads need to be generated at the first depth for the parallel computation, which could lead to a performance hit due to the issue of Overhead (the issue of overhead will be discussed in more detail in the next chapter). For HPX execution model whose main target is to offer scalability for large size of data, this harsh limit on maximum input size clearly does not fit into its scheme. After considering all three factors discussed above, it is obvious that this version of parallel prefix scan algorithm does not provide the necessary versatility and scalability that HPX envisioned for its algorithm design.

Asynchronously Parallel Prefix Scan

The final version of the parallel prefix scan algorithm used in HPX should be one that conforms to the C++14 standard proposal, provides the type deduction capability of a generic algorithm, and proves to be work-efficient when dealing with a large arbitrary number of data. Given these requirements, the best possible solution seems to be falling back to the sequential version of the algorithm created in the first section and first handling each partition of the input
sequence independently, and then compile all executions asynchronously to compute the final results using a key feature from the C++ standard library called `std::future<>`. A future is an object that can retrieve a value from some provider objects or asynchronous operations and store it in a memory place called shared state. Whenever the value in the shared state is ready to be retrieved, the future object would notify the creator of the asynchronous operation and allow it to use a variety of methods to query, wait for, or extract a value from the future object.

By creating a future instance via the AIP `std::async()` and specifying the execution policy to be `std::launch::async`, a new thread is created each time on top of the ongoing thread to execute the operation asynchronously. The actual implementation in C++ looks as follows:

```cpp
vector<future<int>> vec_f, vec_f2;
vector<vector<double>> vec_ret;
vector<double> input, intermediate_input, intermediate_output;

// ..., Complete input generation ..., //
int hardware_threads = thread::hardware_concurrency(); // return the number of cores
vector<double>::size_type partition_size = input.size() / hardware_threads; // get partitions
vec_ret.resize(hardware_threads); // pre-generate vector sizes
intermediate_input.resize(hardware_threads);

// First loop of future<> generation //
for (auto threadNum = 0; threadNum < hardware_threads - 1; ++threadNum) {
  future<int> f = async(launch::async, [&vec_ret, &input, &intermediate_input, threadNum, partition_size]() -> int{
    vector<double> sequential_ret;
    sequential_ret.reserve(partition_size);
    inclusive_scan(begin(input) + threadNum*partition_size, begin(input) + (threadNum + 1)*partition_size,
    back_inserter(sequential_ret));
    intermediate_input[threadNum] = sequential_ret.back();
    vec_ret[threadNum] = move(sequential_ret);
    return 0;
  });
  vec_f.push_back(move(f));
} // continued next page ..., //
```
//Main thread execution/
vector<double> sequential_main;
sequential_main.reserve(partition_size);
inclusive_scan(begin(input) + (hardware_threads - 1)*partition_size, end(input), back_inserter(sequential_main));
intermediate_input[hardware_threads - 1] = sequential_main.back();
vec_ret[hardware_threads - 1] = move(sequential_main);

//Wait for each thread to finish/
for (auto threadNum = 0; threadNum < hardware_threads - 1; ++threadNum)
    vec_f[threadNum].wait();

//Calculate intermediate results/
inclusive_scan(intermediate_input.begin(), intermediate_input.end(), back_inserter(intermediate_output));

//Second round of future<> generation/
for (auto threadNum = 0; threadNum < hardware_threads - 1; ++threadNum){
    future<int> f;
    if (threadNum != 0)
    {
        f = async(launch::async, [&vec_ret, threadNum, &intermediate_output](){
            for_each(begin(vec_ret[threadNum]), end(vec_ret[threadNum]), [&](double& d){
                d += intermediate_output[threadNum - 1];
            });
            return 0;
        });
        vec_f2.push_back(move(f));
    }
}

//Main thread execution/
for_each(sequential_main.begin(), sequential_main.end(), [&]{
    d += intermediate_output[hardware_threads - 2];
});

for (auto i = 0; i < vec_f2.size(); ++i) //wait for each threads to finish/
    vec_f2[i].wait();
Before the first loop, the code above first figures out how many cores the machine provides by calling `thread::hardware_concurrency`, which represents the maximum threads the code can generate to achieve “true parallelism” when running the program. During the first loop, an instance of `std::future<int>` is created for each iteration by calling `std::async()` with a `lambda` function in which each partition of the input sequence is calculated independently with an sequential version of `inclusive_scan()`, and then this `std::future<int>` instance is pushed into a vector in sequence. One thing to be noted is that instead of calling `vec_f.push_back(f)`, `move(f)` was sent as the argument. This is because `std::future` class does not provide a copy constructor and each `future` instance is unique, and thus cannot be copied but only moved by reference.

Another thing to be noted is that only \((hardware – 1)\) number of extra threads were created, and this is because the main thread that is currently running this program will also be used to handle part of the partition, which is what happened next in the code. After the execution from the main thread, the program wait for each thread generated during the loop to finish by calling `std::future::wait()`. The results from each thread and the main thread were then used as intermediate input to calculate the intermediate output for each thread \(N\), which were later used as input to calculate the final results for thread \((N+1)\). This process in practice works as follows:

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Main Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs: 1 2 3 4 5</td>
<td>6 7 8 9 10</td>
<td>11 12 13 14 15</td>
</tr>
<tr>
<td>After calling the sequential inclusive scan (plus) on each thread independently:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Outputs: 1 3 6 10 15</td>
<td>6 13 21 30 40</td>
<td>11 23 36 50 65</td>
</tr>
</tbody>
</table>

The sum of each thread is used as intermediate inputs \(\{15, 40, 65\}\) to calculate a intermediate output \(\{15, 55, 120\}\) by calling `inclusive_scan`. To get the final output, the intermediate output from thread \(N\) needs to be added to each element in thread \(N+1\):|

| Final Outputs: 1 3 6 10 15 | 21 28 36 45 55 | 66 78 91 105 120 |
The advantage of this asynchronous version of algorithm is that each thread, including the main thread, runs its own calculation asynchronously without affecting or waiting for each other. After each additional thread finishes its calculation, it sends a “ready” message to the shared state, and the main thread looks into the shared state for the “ready” message from all other threads before it proceeds. Theoretically, this design pattern cuts down the execution time based on the number of cores the machine offers during run time. For instance, if the original execution time with sequential algorithm for the input is $O$, and the machine has 8 cores, the asynchronous algorithm, in theory, should cut the execution time down to $O/8$ as all threads are running at the same time, which makes it a very work-efficient algorithm. Although in reality, the execution time for this asynchronous algorithm could be much longer than $O/8$ due to a variety of reasons, which is what will be discussed in the next chapter.

**Chapter 4: Performance Analysis**

Parallelism Can Be S.L.O.W

Having determined the design pattern to use in HPX for the *prefix scan* algorithm, the work left to be done is to make it work efficiently in HPX as parallelism can be S.L.O.W during actual execution. S.L.O.W stands for *Starvation, Latencies, Overhead, and Waiting for contention resolution*, and is accounted for the majority of the slow-down that a program experiences. *Starvation* occurs when there is insufficient concurrent work available for the hardware to maintain high utilization of all resources. *Latencies* are used to measure the time-distance delay
intrinsic to accessing remote resources and services. *Overhead* describes the extra work and execution time required for the management of parallel actions and resources on the critical execution path, which is not necessary in a sequential scenario. *Waiting for contention resolution* displays the delay due to the lack of availability of oversubscribed shared resources. Of these four possible reasons for performance slow-down, *Latencies* and *Waiting for contention resolution* occur mostly because of the hardware incapability instead of software design, and therefore are difficult to maneuver by program developers. On the other hand, *Starvation* and *Overhead* can be effectively controlled and reduced to a certain degree through better algorithm design, and thus become the main issues waiting to be addressed. The key to address *Starvation* and *Overhead* is through *fine-grained parallelism*.

*Starvation* happens when too many threads are generated at the same time while there is not enough workload to be processed, and in turn creates a significant amount of *Parallelism Overhead* that cannot be *justified* by program performance, that is, the program’s execution time is even longer under parallelism design than under sequential design due to *Overhead*. To void this situation, program developer has to design the parallel algorithm carefully to achieve a *fine-grained parallelism*. *Fine-grained parallelism* is achieved when “just right” number of extra threads were generated according to a specific workload so that each thread has its hands full processing data while the program benefits from the parallelism. The following example demonstrates the relationship between *Starvation* and *Overhead* and how it can be solved through *fine-grained parallelism*. 
Assuming workload to be process is 1 million integers:

<table>
<thead>
<tr>
<th>Main Thread</th>
<th>Thread 2</th>
<th>Thread 3</th>
<th>Thread 4</th>
<th>Thread 5</th>
</tr>
</thead>
</table>

Main Thread: one thread takes 300 milliseconds to process this workload, no Overhead time.

Main Thread + Thread 2: Four extra threads were generated, therefore each thread is assigned with 1/5 of the original workload, and the parallel execution time is cut down to 60 milliseconds, but each thread is actually starving for more work. Meanwhile, the time it takes to generate and manage a new thread is 80 milliseconds, and the Overhead time sums to be 320 milliseconds, which already surpasses the total execution time when running a single thread. This parallelism obviously cannot be justified and requires a better grained solution.

Main Thread + Thread 2: Only one extra thread was generated, therefore each thread is assigned ½ of the workload, and the parallel execution time is cut down to 150 milliseconds. Adding the one Overhead time from generating thread 2, the total execution time is 230 milliseconds, which is an improvement to the sequential execution. What about having two extra threads then? The parallel execution time will be cut down to 100 milliseconds, when adding the Overhead time for generated two more threads, the total execution time becomes 260 milliseconds, which is still better than sequential execution but worse than when only generating one additional thread. As a result, the fine-grained parallelism for this particular process with 1 million data set is 2 partition.

This example implies that with the time to generate new thread being fixed, the bigger the workload, the more beneficial it becomes to generate extra threads. For instance, if the
workload takes 3000 milliseconds to process, the 80 milliseconds of *Overhead* time becomes trivial and the program can benefit more from having more threads working together.

**Standard Library Performance**

As demonstrated in the last section of Chapter 3, the *prefix scan* algorithm can be fully implemented asynchronously using some key features from the C++14 Standard Library such as `std::future<>` and `std::async()`. But how does it actually perform against the sequential version in runtime? A test was conducted using these two versions of *prefix scan* algorithm and the results are shown in the following chart, which demonstrates the performance differences between them given an increasing size of data set to work with:

![Prefix Scan Performance Comparison](chart.png)

It is quite clear that the asynchronous version of the *prefix scan* algorithm performs much better than the sequential version according to this test. During the test, the asynchronous
version partitioned the data sets based on the maximum number of CPU cores the testing machine provides, which is 8 cores in this case, and thus the original workload was divided into 8 pieces and handled asynchronously. It is also worth noting that as the size of the data set increases, the improvements of execution time resulting from the asynchronous algorithm become more significant.

<table>
<thead>
<tr>
<th>Size</th>
<th>Sequential Run Time</th>
<th>Async Run Time</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000000</td>
<td>281</td>
<td>188</td>
<td>66.9%</td>
</tr>
<tr>
<td>5000000</td>
<td>1469</td>
<td>919</td>
<td>62.6%</td>
</tr>
<tr>
<td>10000000</td>
<td>3032</td>
<td>1887</td>
<td>62.2%</td>
</tr>
<tr>
<td>50000000</td>
<td>14108</td>
<td>8453</td>
<td>59.9%</td>
</tr>
<tr>
<td>100000000</td>
<td>30064</td>
<td>18548</td>
<td>61.7%</td>
</tr>
<tr>
<td>500000000</td>
<td>139976</td>
<td>90474</td>
<td>64.6%</td>
</tr>
</tbody>
</table>

For instance, given an input size of 1,000,000, asynchronous version reduced the execution time to its 61.2%, and this number becomes 59.9% given an input size of 50,000,000. When the input size got doubled to 100,000,000, the reduction percentage began to rise to 61.7%. And yet when the input size was increased to 500,000,000, the improvement became even less significant and was back to 64.6%. One major assumption was proved during this test: while the asynchronous version generally performs better than the sequential version, the best result was produced when the workload was partitioned into a fine-grained parallelism. Since the partition size is set to be 8 in this test, when the workload size was as small as 1,000,000, the Overhead generated from additional threads compromised the advantage that parallelism provides. Obviously, this advantage from parallelism was maximized when the workload size was increased to 50,000,000, which means 8 appears to (in fact, it’s not) be the fine-grained partition size that best utilizes the CPU resources when the input size is 50,000,000 for this algorithm. So what happened after that? The algorithm performance started to go down when
the input size got further extended, which may imply that the current level of parallelism did not partition the input size into pieces that were small enough to be handled efficiently, and more threads need to be generated to further partition the workload.

To dig a little further into where the performance differences come from when using these two different versions of algorithm, a few snapshots on resource monitor were captured while running the program:

The first snapshot was captured from the resource monitor when the sequential version was running the program, and it can be seen clearly that not all 8 cores were fully utilized. Instead, the program task was migrated from one core to another and being executed sequentially through time. In comparison, the second snapshot shows that the asynchronous version has managed to keep all 8 cores busy from the get-go by utilizing all the available resources, which explains why it leads to an overall better performance.

Conceivably, one common way to justify the fine-grained parallelism for a particular input size is to test it through different sets of cores. Such a test can be done and represented in a chart as follows, in which the input data size is a fixed 100,000,000.
According to the test, the biggest performance boost occurs when the first thread got introduced and the algorithm started to run asynchronously with half of the workload, which cut down the execution time to less than half of the original time. As more threads starting to get introduced into the execution, the execution time continued to decrease and finally reached the bottom of the chart at the partition size of 4. After that, the execution time quickly started to rise up and became more stable toward the end even though the partition size gets increased drastically. Combined this finding with the results from the previous test, it can be concluded that even though the machine can provide as many as 8 cores' function (including virtual cores), 4 is the fine-grained partition size for this particular input size, which also happens to be the number of physical cores the machine possesses. When the partition size equals the number of physical cores of the testing machine, the program is practically executed in a “true parallelism”.

29
Boost Library Performance

One major advantage Boost library bring to the table is that it supports continuation of future via `boost::future<>::then()`. Future continuation allows one asynchronous operation, upon completion, to invoke a second operation and pass the first operation’s return value to it as input. The current C++ Standard does not allow one to register a continuation to a future. With `boost::future<>::then()`, instead of waiting for the first asynchronous operation to complete, a continuation is “attached” to the first operation, which runs in the same thread as the first operation. Future continuation helps to avoid blocking waits or wasting threads on pooling, which then greatly improves the responsiveness and scalability of a program.

During the first asynchronous implementation of the prefix scan algorithm, two For loops were used to generate the future instances and the thread pool while one For loop works between them that calls the `std::future.wait()`. This middle step was necessary by then because the second asynchronous operation has to wait for the first operation to complete and produce intermediate outputs before it can continue. However, this is a not very work-efficient design because by the end of the first asynchronous operation when `std::future.wait()` was called, all previously generated threads would have to join the main thread, and by the time the second operation started, the main thread had to re-generate new future instances and a new thread pool, and re-allocate memory for them, even though the new threads literally were just a “continuation” of the old threads. This issue can be solved by using `boost::future<>::then()` and the new implementation snippet looks like following:
A few changes were marked with comments on this snippet: first, the For loop in the middle were removed and its function were replaced by boost::future<>::then(); second, the calculation for intermediate outputs were also moved under the second asynchronous operation because this design does not guarantee that all the intermediate inputs were produced by the time the second operation began, therefore each thread has to at least make sure its previous threads have all finished. Aside from that, it can be noticed that the new future instances do not call future.wait() because it is defined for boost::future<>::then() that each continuation will not begin until the preceding one has completed, which practically chains the
future instances together in sequence. So how does the involvement of Boost library change the performance for this program then? The following data were collected from the latest test:

<table>
<thead>
<tr>
<th>Size</th>
<th>Sequential Time</th>
<th>Standard Async Time</th>
<th>Boost Async Time</th>
<th>Standard Ratio</th>
<th>Boost Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000000</td>
<td>281</td>
<td>188</td>
<td>172</td>
<td>66.9%</td>
<td>61.2%</td>
</tr>
<tr>
<td>5000000</td>
<td>1469</td>
<td>919</td>
<td>797</td>
<td>62.6%</td>
<td>54.3%</td>
</tr>
<tr>
<td>10000000</td>
<td>3032</td>
<td>1887</td>
<td>1750</td>
<td>62.2%</td>
<td>57.7%</td>
</tr>
<tr>
<td>50000000</td>
<td>14108</td>
<td>8453</td>
<td>7953</td>
<td>59.9%</td>
<td>56.4%</td>
</tr>
<tr>
<td>100000000</td>
<td>30064</td>
<td>18548</td>
<td>17314</td>
<td>61.7%</td>
<td>57.6%</td>
</tr>
<tr>
<td>500000000</td>
<td>139976</td>
<td>90474</td>
<td>84176</td>
<td>64.6%</td>
<td>60.1%</td>
</tr>
</tbody>
</table>

While the improvement from the Standard library to the Boost library wasn’t quite significant, it is no coincidence that after the introduction of future continuation, this algorithm generally received a performance boost compared to the previous test, and showed a much better compatibility toward both small and large scales of data set. That being said, what hinders a more significant improvement from this Boost version is that generating and managing threads
instances still require a fairly amount of *Overhead* measured by milliseconds. That is why this Boost version calls for a deeper renovation on its design pattern to let it fit into the HPX scheme, in which the *Overhead* of new threads is measured by microseconds.

**HPX Execution Model Performance**

The conversion from the Boost version to the HPX version involves the utilization of a couple of new APIs provided HPX including `hpx::static_partitioner<>::call()` and `hpx::main()`. In both previous versions, the partition size is either user-defined or determined by the maximum number of cores the machine can provide. Either way, users of this algorithm have to go through a series of sample tests in hopes of identifying the *fine-grained* partition size that produces the “close-to-best” computing performance. In addition, the current design pattern still requires the same operations to be called back and forth between the main thread and newly generated threads, which practically divides the algorithm into two independent sequences. Solving these two issues potentially can produce more consistent results and increase the cohesion within the algorithm, and in turn improve the overall performance of the algorithm across the board. `hpx::static_partitioner<>::call()` aims to provide a solution for both of these issues by not only chaining two asynchronous operations together, but also introducing a “smart” thread pooling mechanism that automatically detects what the best *fine-grained parallelism* should be based on the machine’s hardware capability and the available resources during runtime, and generates threads accordingly. Within the `hpx::static_partitioner<>::call()`, two lambda functions were invoked in sequence and chained together by wrapping the first lambda function’s returning value into a `hpx::future<>` instance.
and feeding it to the second _lambda function_ who unwraps the data from _future_ and continues with the calculation.

The second key API – _hpx::main()_ – is designed to replace the _main()_ function from the C++ standard library when executing APIs from the HPX library, and to grant the users a great deal of flexibility on specifying how many cores will actually be used during runtime. During the tests for C++ and Boost versions, one thing in common is that once the number of threads generated surpasses the maximum number of cores the machine provides, it is guaranteed that the operating system will attempt to run the program on all 8 cores simultaneously, regardless of physical core or virtual core. One of the previous test actually proved that the algorithm works more efficiently when it achieves “true parallelism”, that is, only the 4 physical cores are being used during runtime. Previously neither C++ nor Boost supports specifying the number of cores to use during runtime, and yet by replacing _main()_ with _hpx::main()_ and supplying the command line arguments, HPX execution model enables this feature. Combining this feature with another HPX’s previously mentioned advantage – extremely low _Overhead_ per thread, it is possible to drastically improve this program’s scalability by generating a massive thread pool while still allocating resources efficiently. Here is the implementation from HPX:
using namespace hpx::parallel;
detail::algorithm_result<parallel_execution_policy, double>::type result =
    util::partitioner<parallel_execution_policy, vector<double>::iterator, double, vector<double>>::call(par, begin(input), input.size(),
        [&init_value](vector<double>::iterator part_begin, size_t part_size)
        ->vector<double>
        {
            vector<double> ret;
            ret.reserve(part_size);
            util::loop_n(part_begin, part_size, [&init_value, &part_begin, &ret](vector<double>::iterator const& curr)
                {
                    init_value = init_value + *curr;
                    *back_inserter(ret) = init_value;
                });
            return ret;
        },
    hpx::util::unwrapped([](vector<vector<double>>&& results) ->double{
        vector<double> intermediate_input, intermediate_output;
        std::for_each(begin(results), end(results), [&intermediate_input](vector<double> const& v){
            intermediate_input.push_back(v.back());
        });
        inclusive_scan(begin(intermediate_input), end(intermediate_input), back_inserter(intermediate_output));
        std::reverse(begin(intermediate_output), end(intermediate_output));
        std::for_each(begin(results)+1, end(results), [&intermediate_output](vector<double>& v){
            std::for_each(begin(v), end(v), [&intermediate_output](double& d){
                d += intermediate_output.back();
            });
            intermediate_output.pop_back();
        });
        return results.back().back();
    });
}

HPX Runtime Resource Allocation

CPU 0 100% CPU 1 100% CPU 2 100% CPU 3 100% CPU 4 100% CPU 5 100% CPU 6 100%
### Final Project Report – Parallel Prefix Scan in HPX

Chen Guo  89-721-5779

<table>
<thead>
<tr>
<th>Size</th>
<th>Sequential Time</th>
<th>Boost Async Time</th>
<th>HPX Async Time</th>
<th>Boost Ratio</th>
<th>HPX Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000000</td>
<td>281</td>
<td>172</td>
<td>125</td>
<td>61.2%</td>
<td>44.5%</td>
</tr>
<tr>
<td>5000000</td>
<td>1469</td>
<td>797</td>
<td>610</td>
<td>54.3%</td>
<td>41.5%</td>
</tr>
<tr>
<td>10000000</td>
<td>3032</td>
<td>1750</td>
<td>1203</td>
<td>57.7%</td>
<td>39.7%</td>
</tr>
<tr>
<td>50000000</td>
<td>14108</td>
<td>7953</td>
<td>5984</td>
<td>56.4%</td>
<td>42.4%</td>
</tr>
<tr>
<td>100000000</td>
<td>30064</td>
<td>17314</td>
<td>12001</td>
<td>57.6%</td>
<td>39.9%</td>
</tr>
<tr>
<td>500000000</td>
<td>139976</td>
<td>84176</td>
<td>59472</td>
<td>60.1%</td>
<td>42.5%</td>
</tr>
</tbody>
</table>
The first snapshot shows how operating system allocates its resources while running the HPX version of algorithm, and it is quite obvious that only the first 4 physical cores are operating while the virtual cores are put to rest to save resources. It is worth noting that such snapshot was captured when the program was running asynchronously on 128 independent threads, which is a massive improvement on scalability compared to the Boost version. The scalability of this program was demonstrated in the next graph. Compared to the “Performance Trend” graph from the Boost version where the computing efficiency faded away quickly after the partition size reached 4 (optimal partition), the HPX version provides a rather consistent performance output even when the threads pool gets increased to 256. In fact, the optimal partition size for the HPX version arrived at 128 threads. This superiority on scalability can prove to be very user-friendly and convenient for some sequential algorithms that time-consuming and complicated to calculate. The last graph and the table present a visual and a statistical comparison between the Boost version and the HPX version in terms of execution time. The test results showed that the prefix scan algorithm was run almost 17% faster in the HPX execution model than in the Boost version, and cost only as low as 39.7% of the original sequential execution time. All these stats serve to showcase that the implementation of the asynchronous prefix scan algorithm in the HPX execution model is a big step-up.

Chapter 5: Application and Conclusions

Application of Parallel Prefix Scan
As mentioned above, parallel prefix scan algorithm is a simple but powerful parallel algorithm building block, and has been widely used in both hardware and software development such as NVIDIA’s parallel computing platform and programming model – CUDA, which enables dramatic increases in computing performance by harnessing the power of the GPU. With regard to parallel algorithm design, one common obstacle that many designers tend to face is that after parallelizing the input data sets and handling each section independently, how to put them back together in one piece and position them both correctly and efficiently. Such parallel algorithms include parallel filter and parallel unique, which will be the cases used to explain how parallel prefix scan algorithm can come to rescue.

Parallel filter is expected to work as follows: given an array of input data, produce an array output containing only those elements that test to be TRUE for predicate P. For example:

Let predicate P be x > 10 for each element x in array <17, 4, 6, 8, 11, 5, 13, 19, 0, 24>. The parallel filter algorithm thus should produce output array <17, 11, 13, 19, 24>. Finding the elements for the output is easy, but getting them in the right place seems somewhat tricky, especially if the algorithm runs asynchronously without knowing which thread finishes first. In this case, by introducing a second bit array and using parallel prefix scan to map the true elements, the positioning problem can be easily solved.

```
input  <17, 4, 6, 8, 11, 5, 13, 19, 0, 24>
bite   <1, 0, 0, 0, 1, 0, 1, 1, 0, 1>
```

Step 1 asks for each true element, the bit array inserts a value 1. Step 2 uses parallel prefix sum on the bit array and produces bit-sum <1, 1, 1, 2, 3, 4, 4, 5>. The last step is to parallel map to produce the final output: each element in the input array just needs to look up for its
corresponding value in the bit array and subtract it by 1 to locate its index position in the output array.

Another example, parallel unique works slightly different than the parallel filter but follows the same philosophy. Parallel unique is supposed to produce an output that only contains the unique elements from the input array. For example, for input array <1, 2, 3, 3, 4, 5, 6, 6, 7, 7, 8, 9, 10>, the output array should look like <1, 2, 3, 4, 5, 6, 7, 8, 9, 10>. The biggest obstacle stems from positioning the output element as well when the input array is being processed in parallel. Once again, a second bit array is introduced to solve the problem and map only the unique elements after sorting the input data sets:

Step 1: <1, 1, 2, 3, 3, 4, 5, 6, 6, 7, 7, 7, 8, 9, 10>
  <0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0>
Steps 1 generates the bit array by inserting 1 for element x[N] if x[N] equals x[N-1]. Steps 2 uses parallel prefix sum to produces the bit sum <0, 1, 1, 2, 2, 2, 3, 3, 4, 5, 5, 5, 5>. The last step is to parallel map to produce the final output: each element is in the input array just needs to look up for its corresponding value in the bit array and subtract its original position index by this value, and thus locates its new index position.

These two parallel algorithms only serve to showcase how parallel prefix scan can be applied in real world to simplify and expedite the calculation process of other algorithm. Some other more complex algorithm such as quick sort can also use parallel prefix scan to facilitate its performance time presented by big O notation, which will not be discussed any further at this point.

Conclusions
The main objectives of this project were accomplished as it successfully identified a suitable design pattern that enables the parallel \textit{prefix scan} algorithm to take full advantage of the computing power of a multi-core CPU and achieved asynchronization during runtime. Being able to run the algorithm asynchronously not only greatly improves its work efficiency, but also leaves programmers more flexibility in designing a “lock free” multi-thread application, and in turn avoids the risk of \textit{race condition} and \textit{deadlock}. On top of that, the final version of the implementation is a product that marries the virtues from several popular designing tools including C++ standard, Boost library, and the HPX execution model, and conforms to the design principles of generic algorithm. As a result, any potential user of this algorithm will find the structure of this algorithm familiar to their knowledge base and easy to start with. The tests conducted throughout the project also proved that the HPX execution model is a good fit for the \textit{prefix scan} algorithm and should have no problem including it as part of the HPX algorithm library.

\section*{Future Direction}

As stated above, one of the biggest advantages the HPX execution model brings to the table is its extremely low \textit{Overhead per thread} and scalability, and therefore it really excels when the testing machine can provide more cores to work with to enable a massive thread pooling, likewise a super computer. The machine used to run the tests for this project, while may stand out among regular computers, is nowhere near a competent computer that can unleash all the potentials of the HPX execution model. Any future effort that aims to explore the true
computing capability of this prefix scan algorithm should be spent on a computer that is powerful enough to accommodate HPX’s threading demand.

References


