Master Thesis

Parallel STL Prototype Implementation and Evaluation on Algorithmic Motifs Found in Scientific Computing

Author(s):
Sgier, Sandro

Publication Date:
2016

Permanent Link:
https://doi.org/10.3929/ethz-a-010594298

Rights / License:
In Copyright - Non-Commercial Use Permitted
MASTER THESIS

Parallel STL
Prototype Implementation and Evaluation on Algorithmic Motifs Found in Scientific Computing

Sandro Sgier
21 January 2016

Supervision:
Prof. Torsten Hoefler
Prof. Markus Püschel
Tobias Gysi

Scalable Parallel Computing Laboratory
Department of Computer Science
ETH Zurich
Abstract

Parallel computing is becoming a field of major importance due to the trend toward increasingly parallel hardware architectures. Since all notable implementations of the C++ standard library, including the implementation of the GNU Compiler Collection, are single-threaded, there exist several competing, non-standard parallel (multi-threaded) extensions for the programming language. In the near future, these shall be replaced by a parallel version of the Standard Template Library, called parallel STL. We investigate the performance and the suitability of the parallel STL for the algorithmic motifs that commonly appear in scientific applications. We base our implementation on an existing prototype of the parallel STL, which is targeted to the Windows operating system. In order to replace the Windows-specific threading code, we implemented a new, portable back-end. In four case studies, each based on a widely used scientific algorithm, we compare the performance of the parallel STL against sequential and manually parallelized variants. Our work demonstrates that the parallel STL has the potential to yield strong speedups, and competes well with manually parallelized code in terms of performance whenever an application can be expressed in algorithms provided in the STL.
Acknowledgement

A special thanks goes to Tobias Gysi for his time, his guidance, and his readiness to provide competent advice quickly and at any time.

I would also like to thank Prof. Torsten Hoefer and Prof. Markus Püschel for their supervision and helpful intermediate feedback, and Hussein N. Harake from the Swiss National Supercomputing Centre for providing me with access to Greina, a computing cluster under his administration.
# Contents

1 Introduction 8

2 Prototype 10
  2.1 Thread Pool 11
  2.2 Partitioning Framework 15
  2.3 Execution Policy Objects 18
  2.4 Composable Iterators 18
  2.5 Tests 19
  2.6 Measurements 20
    2.6.1 Input Content 20
    2.6.2 Nondeterminism 21
    2.6.3 Initial Cache State 23
    2.6.4 NUMA effects 23
    2.6.5 Processor Affinity 23
    2.6.6 Compiler Optimization 24
    2.6.7 Target Machine and Tools 24
    2.6.8 Experiment Design and Results 24
  2.7 Summary 26

3 Case Studies 29
  3.1 Floyd Warshall 30
    3.1.1 Implementation 30
    3.1.2 Execution Time Model 32
    3.1.3 Measurement Results 33
    3.1.4 Summary 36
  3.2 k Nearest Neighbors 36
    3.2.1 Implementation 38
    3.2.2 Execution Time Model 38
    3.2.3 Measurement Results 39
    3.2.4 Summary 42
  3.3 Barnes Hut 42
    3.3.1 Implementation 43
    3.3.2 Execution Time Model 44
    3.3.3 Measurement Results 46
    3.3.4 Summary 47
  3.4 Closest Pair 48
    3.4.1 Implementation 48
    3.4.2 Execution Time Model 49
List of Figures

2.1 Existing prototype design ........................................ 11
2.2 Working principle of thread pool .................................. 12
2.3 Principles of a composable iterator ................................. 19
2.4 Q-Q-plot ............................................................. 21
2.5 Q-Q-plots of various samples ....................................... 22
2.6 pSTL algorithms with largest speedup ............................... 25
2.7 pSTL algorithms with medium speedup .............................. 25
2.8 pSTL algorithms with smallest speedup ............................ 26
2.9 Core scaling ......................................................... 27
3.1 Delegation of matrix partitions to different threads ............... 31
3.2 Execution times of pSTL variant of Floyd Warshall ............... 34
3.3 Execution times of different variants of Floyd Warshall ......... 35
3.4 Core scaling of pSTL variant of Floyd Warshall .................. 37
3.5 Scatter plot and regression model for nth element ............... 40
3.6 Execution times of pSTL variant of k Nearest Neighbors ....... 40
3.7 Execution times of different variants of k Nearest Neighbors .... 41
3.8 Core scaling of pSTL variant of k Nearest Neighbors ............ 41
3.9 Execution times of pSTL variant of Barnes Hut .................. 46
3.10 Execution times of different variants of Barnes Hut ............. 47
3.11 Core scaling of pSTL variant of Barnes Hut ..................... 47
3.12 Closest Pair - Divide and Conquer .................................. 49
3.13 Execution times of different Closest Pair variants ............... 50
3.14 Core scaling of pSTL variant of Closest Pair ..................... 50
List of Tables

2.1  List of pSTL algorithms ............................................. 10
3.1  Parameter fits of operation count models ...................... 45
## List of Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Singleton pattern</td>
<td>13</td>
</tr>
<tr>
<td>2.2</td>
<td>Method for submitting jobs to the thread pool</td>
<td>14</td>
</tr>
<tr>
<td>2.3</td>
<td>Worker thread</td>
<td>15</td>
</tr>
<tr>
<td>2.4</td>
<td>Attempt to pop a job</td>
<td>15</td>
</tr>
<tr>
<td>2.5</td>
<td>Basic mechanism for partitioning</td>
<td>16</td>
</tr>
<tr>
<td>2.6</td>
<td>Utility function for <code>my_for_each</code></td>
<td>17</td>
</tr>
<tr>
<td>2.7</td>
<td>Simple parallel implementation of <code>for_each</code></td>
<td>17</td>
</tr>
<tr>
<td>2.8</td>
<td>Sorting with different execution policies</td>
<td>18</td>
</tr>
<tr>
<td>2.9</td>
<td>Function object used for <code>transform</code></td>
<td>20</td>
</tr>
<tr>
<td>3.1</td>
<td>Pseudo code of the Floyd Warshall algorithm</td>
<td>30</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Since CPU clock rates have generally been stagnating, there exists a trend toward increasingly parallel architectures. Exploiting the full potential of those requires writing multi-threaded (parallel) programs, which can be a challenging task. Abstraction facilitates the development of parallel software and disburdens the developer of having to deal with technical details, such as specifics about the hardware architecture. Providing such abstraction is a major purpose of software libraries. For the C++ programming language, there currently does not exist a standardized library that provides abstraction for parallelization. A parallel variant of the C++ Standard Template Library (STL) is a step toward solving this problem. The STL is a collection of data structures and algorithms, written in the C++ programming language. STL implementations, including the one contained in the GNU Compiler Collection (GCC), are single-threaded. This is lost potential, since many STL algorithms are parallelizable by nature. The parallel STL (pSTL) provides an interface to run STL algorithms in parallel. It is also an attempt to replace competing, non-standard parallel programming extensions for the C++ programming language, such as TBB\[16\], C++ AMP[20], or Thrust[23].

In this work, we investigate the suitability of the parallel STL for the algorithmic motifs that commonly appear in scientific applications, and compare their performance to manual parallelization, i.e. parallelization by directly using elements from the C++ threading library. We base our implementation on a prototype of the parallel STL[9] targeted to the Microsoft® Windows operating system. We updated the prototype with a portable back-end, using exclusively built-in C++ functionality, particularly the threading library, and so obtained a highly portable pSTL prototype. Using this prototype, we looked at the parallel STL from two perspectives. On one hand, we measured speedups of each pSTL algorithm individually. Additionally, for two of the algorithms, we looked at the scaling with respect to the number of threads (core scaling). On the other hand, we performed four case studies, which contain important algorithmic motifs in the context of scientific applications. Each of the four case studies is based on a small application that implements one of the following widely used algorithms: Floyd Warshall, k Nearest Neighbors, Barnes Hut and Closest Pair. For each of those algorithms, we wrote a sequential, a manually parallelized, and a pSTL variant. The latter uses pSTL algorithms as exclusively as possible. These variants are compared to each other, and to predictions of simple timing models.
The core scaling of the pSTL variants is evaluated as well. By interpreting the results of a variety of measurements, we are able to draw numerous conclusions about the parallel STL, and about our prototype in particular.

Chapter 2 walks through some ideas and patterns applied to the implementation of the prototype, and presents results of the measurements of individual STL algorithms. In chapter 3, we present the case studies and their results in detail. Chapter 4 discusses several related parallel C++ extensions, and chapter 5 concludes.
Chapter 2

Prototype

The existing prototype, on which our implementation is based, targets the Windows operating system and provides the 32 STL algorithms listed in Table 2.1. The specifications of those algorithms can be found in the C++ reference [8]. All 32 algorithms are also provided in our own, portable pSTL prototype.

The existing prototype [9] uses the Microsoft Thread Pool API [21]. A thread pool is an entity that is responsible for managing a number of threads and coordinating work among them. We will have a closer look at thread pools in section 2.1. In order to obtain a portable prototype, we implemented our own thread pool and adapted all the code in the existing prototype that interacts with the Microsoft Thread Pool API. Additionally, since the existing prototype is written in Microsoft Visual C++ [22], we made a couple of smaller adaptions in order to make it compliant with the C++ standard (and hence compilable with GCC, for instance). We learned that it is possible to end up with a lean and efficient prototype of the parallel STL that uses no third party library. By examining the existing prototype, we also learned that the implementations of all STL algorithms interact with an intermediate partitioning framework, which itself contains almost all of the Windows-specific code. We can thus leave those implementations mostly unchanged and focus on the partitioning framework when we do the code replacement. Figure 2.1 provides an intuitive grasp of the

<table>
<thead>
<tr>
<th>adjacent_find</th>
<th>is_partitioned</th>
<th>reverse</th>
</tr>
</thead>
<tbody>
<tr>
<td>all_of</td>
<td>is_sorted</td>
<td>scan</td>
</tr>
<tr>
<td>copy</td>
<td>lexicographical_compare</td>
<td>set_difference</td>
</tr>
<tr>
<td>copy_if</td>
<td>mismatch</td>
<td>set_intersection</td>
</tr>
<tr>
<td>count</td>
<td>move</td>
<td>set_union</td>
</tr>
<tr>
<td>equal</td>
<td>nth_element</td>
<td>sort</td>
</tr>
<tr>
<td>fill</td>
<td>partial_sort_copy</td>
<td>swap_ranges</td>
</tr>
<tr>
<td>find</td>
<td>partition</td>
<td>transform</td>
</tr>
<tr>
<td>for_each</td>
<td>reduce</td>
<td>uninitialized_copy</td>
</tr>
<tr>
<td>generate</td>
<td>remove</td>
<td>unique</td>
</tr>
<tr>
<td>includes</td>
<td>replace</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: List of pSTL algorithms.
design of the existing prototype. By means of this sketch, we can define the task more clearly: we want to replace the Microsoft Thread Pool with an own thread pool and rebuild the connection between the thread pool and the partitioning framework. The thread pool shall be lean but provide the same features as were used within the Microsoft Thread Pool. Additionally, we need to adapt some Visual Studio specific code to the C++ standard.

Sections 2.1 - 2.4 discuss several of our design choices and implementation ideas. Section 2.5 briefly explains how we tested our code for correctness. Section 2.6 describes the setup and presents the results of a variety of measurements we made, and section 2.7 makes some intermediate conclusions. We start by describing the design and implementation of our thread pool in the following section.

2.1 Thread Pool

A thread pool is an implementation of the thread pool pattern\[38\], in which a number of threads are held in a container. Additionally, there is one or multiple queues that can hold jobs to execute. These jobs are delegated to the different threads. The thread pool controls the number of threads and assigns jobs to threads, with the goal of optimizing performance. There are two major advantages to this approach. First, once the thread pool is created and active, no threads are created or destroyed anymore. This reduces overhead. Second, the number of threads, which is system dependent, is transparent to the user, \textit{i.e.} they do not have to care about optimizing this aspect. An examination of the partitioning framework of the existing prototype reveals the properties our new thread pool should have:

1. **Global singleton access**: There must be one instance of the thread pool at maximum. We want our thread pool to exhaust all the available resources on the target machine. Having two instances simultaneously violates this target. Additionally, global access is comfortable, because we have to pass fewer dependencies.

2. **An interface for submitting jobs**: The general principle of the partitioning framework is to split the data into chunks and perform some routine on each of the chunks. This results in a number of separable, usually independent jobs, which are then submitted to the thread pool.
3. **An interface for waiting for submitted jobs**: Algorithms in the parallel STL must block until all work is done. That is why we need to have some way of making sure all jobs submitted to the thread pool are completed before the routine returns.

4. **An interface for helping the thread pool to complete its work**: Sometimes the master thread (which is not part of the thread pool) might be idle waiting for the thread pool to complete its work. It is therefore reasonable to provide the master thread with the means of taking a job from the thread pool and executing it on its own in order to speed up the procedure.

5. **Job stealing**: In our design the thread pool will have a local job queue assigned to each thread. Job stealing enables idle threads to “steal” work from other job queues, which makes the application faster.

6. **An interface for obtaining a unique ID corresponding to the current thread**: The implementations of some algorithms need this run-time information.

7. **An interface for obtaining the level of concurrency**: Knowing about the number of threads in the pool enables the partitioning framework to determine the optimal granularity of the jobs.

8. **Thread-safety**: Since multiple threads will be accessing the thread pool simultaneously, this requirement is obvious.
Williams 2012 provides some insightful ideas about the implementation of a thread pool satisfying the properties mentioned above. Our own thread pool is leaned on those ideas and we adopted some of his code. Figure 2.2 provides an overview of our thread pool design. To the right, we can see the collection of worker threads. One local queue is assigned to each of the threads. If a thread is idle, it checks whether there are jobs available in its own local queue. If yes, if pops a job from the front of the queue. If not, it tries to take a job from the back of the local queue of another thread. This mechanism is called “job stealing” or “work stealing”. When a user wants to submit a job, he calls the submission method, passing a callable and the ID of the local queue, to which the job is to be pushed, whereupon he receives a std::future tied to the job. The std::future is an object that can be used to wait for the job to be completed. It is described in more detail below.

We will now walk through some code snippets, which make up part of the implementations of the properties, and discuss them. We start by property 1: Global singleton access. The singleton pattern restricts the instantiation of a class to one object. This can be implemented by hiding the constructor and allowing other classes to access the single object only via a reference that can be obtained by calling a static method. Listing 2.1 shows our implementation of that method. By placing the constructor in the private section (line 8), we make sure that no other classes can construct an instance of thread pool. From the start of the program until another class calls get_instance(), no instances exist. When get_instance() is called for the first time, the static variable instance is still a null pointer and an instance is constructed. We call this lazy construction. Any subsequent calls of get_instance() will not create any new objects but return the pointer to the only existing instance. The mutex is there to ensure thread safety, even though it is unlikely that a user of thread pool will create a thread pool in an already multi-threaded application.

```cpp
class thread_pool {

    static thread_pool* instance;
    static std::mutex mtx;

    // ...

    thread_pool();

public:
    static thread_pool* get_instance() {
        std::lock_guard<std::mutex> lg(mtx);
        if (instance == nullptr) {
            instance = new thread_pool();
        }
        return instance;
    }

    // ...
};
```

Listing 2.1: Singleton pattern.
Property 2 (Interface for job submission) is slightly more complicated. In order to discuss our implementation, we need to explain two class templates first: \texttt{std::future} and \texttt{std::packaged_task}. A future provides a mechanism to access the result of a (potentially asynchronously executed) function. Conceptually, when we run a function in a background thread, we can tie a future to it. Later we can use this future to check if the function has already returned (at which point we say that the future becomes \texttt{ready}). We can also invoke a blocking call at the future, which does not return until it becomes ready. When the future is ready, we can use it to obtain the return value of the corresponding function. \texttt{std::future::get} returns the return value of the function the future is tied to. If the function has not returned, the call blocks. \texttt{std::future::wait} waits for the future to become ready but returns once a specified amount of time has passed. If zero is passed to this method, it simply checks if the future is ready. \texttt{std::packaged_task} is simply a way of creating a future and tying it to a function (or more general: a callable). We can pass a callable to the constructor of \texttt{std::packaged_task} and call \texttt{std::packaged_task::get} on that instance to obtain a future tied to the callable.

Now we know enough to discuss the job submission interface of our thread pool. Listing 2.2 shows the code. The function is templated on the type of the callable we pass. Even though it is generally a task of the thread pool, in our case the caller decides which thread to delegate the job to (via \texttt{thread_id}). The caller is usually the partitioning framework of the parallel STL and giving it the power to decide which job goes to which thread makes the application more deterministic and reduces overhead. The code is relatively simple: We wrap the callable \( f \) in a \texttt{std::packaged_task}, get a future tied to it, push the task to the correct local job queue and return the future. It is also worth mentioning here that a \texttt{std::packaged_task} cannot be copied, which is why we need to explicitly move it at line 8. The returned future can be used to wait or check if the callable has finished executing.

\begin{lstlisting}[language=C++]
template<typename F>
std::future<typename std::result_of<F()>::type> submit(F f, unsigned thread_id) {
    typedef typename std::result_of<F()>::type result_T;
    std::packaged_task<result_T()> job(f);
    std::future<result_T> result(job.get_future());
    local_job_queues[thread_id].get()->push(std::move(job));
    return result;
}
\end{lstlisting}

Listing 2.2: Method for submitting jobs to the thread pool.

Property 3 is implemented by returning the future after the job submission. An interface for helping the thread pool to complete its work (property 4) does not even need to be implemented separately. In order to understand why, let us have a look at Listing 2.3, which shows the main loop that runs on each thread. \texttt{initialize_thread} is an initialization function. There is no need to bother about its implementation here. \texttt{done} is simply an atomic flag that is
set when the thread pool is about to be destroyed. After the initialization, all the function does is repeatedly call \texttt{run\_pending\_task()}. The definition of \texttt{run\_pending\_task()} is shown in Listing 2.4. \texttt{thread\_pool\_job} is a copyable wrapper for \texttt{std::packaged\_task}. \texttt{pop\_local} and \texttt{pop\_other} try to pop a job from the thread-local queue or steal a job from another queue, respectively. If both fail (which means there is no undone work in any queue), the current thread yields. We realize that this is the same behavior as in property 4. Therefore, \texttt{run\_pending\_task\_task} can be used both by the worker threads and by the master thread from outside of the thread pool as a public method.

\begin{verbatim}
void worker_thread(unsigned my_index_) {
    initialize_thread(my_index_);
    while(!done) {
        run_pending_task();
    }
}
\end{verbatim}

Listing 2.3: Worker thread.[39].

\begin{verbatim}
void run_pending_task() {
    thread_pool_job job;
    if (pop_local(job) || pop_other(job)) {
        job();
    } else {
        std::this_thread::yield();
    }
}
\end{verbatim}

Listing 2.4: Attempt to pop a job.[39].

Property 5 (job stealing) is implemented in the previously mentioned method \texttt{pop\_other}. The method iterates across all the local queues, except for the one owned by the current thread, and tries to steal jobs from the back of them. As soon as the stealing succeeds, the method returns \texttt{true}. If it does not succeed at any queue, the return value will be \texttt{false}.

Properties 6 and 7 are trivial and property 8 (thread safety) can be implemented by making all the job queues thread safe, which is trivial as well.

### 2.2 Partitioning Framework

The purpose of the partitioning framework is to split data into chunks, create jobs and submit them to the thread pool. It is accessed directly by the parallel implementations of the STL algorithms. The typical inputs of a call to the partitioning framework are an iterator, a count, any sort of additional data the implementation needs, and a callable. Listing 2.5 shows the code of a basic but central method within the partitioning framework. Its name is not to be confused with \texttt{std::for\_each}. As we can see, the method is templated on the input types. The purpose of the method is to invoke a function (passed by the callable \texttt{Func}) on a range of elements in parallel. The argument \texttt{i\_First} is an iterator that points to the beginning of a container that holds the elements.
_Count specifies the number of elements on which we want to perform the operation, and _Data represents additional data that will be forwarded to _Func.

template< typename _FwdIt , typename _UserData , typename _Callback >
static _FwdIt _For_Each(_FwdIt _First , size_t _Count , _UserData _Data , const _Callback& _Func )
{
    auto thread_pool_ =
infrastructure::thread_pool::get_instance ();
    if(!_Count ) {
        return _First ;
    }

    size_t const hardware_threads = thread_pool_ ->
        get_hardware_concurrency();
    size_t const num_threads = hardware_threads != 0 ?
        hardware_threads:2;
    size_t const block_remainder = _Count % num_threads;
    size_t const block_size_prototype = _Count / num_threads + 1;
    size_t block_size = block_size_prototype;

    std::vector< std::future<void > > futs_ (num_threads -1); 
    for ( size_t i = 0; i < num_threads-1; ++i) {
        if (i == block_remainder )
            -- block_size ;

        _FwdIt block_end = _First ;
        std::advance(block_end , block_size); 
        futs_[i] = thread_pool_ -> submit([[ _First , block_size ,& _Data ,&
            _Func ]]() {
            _Func(_First , block_size , _Data );
        } ,
        i); 

        _First = block_end ;
    }

    block_size = block_size_prototype - 1;
    _Func(_First , block_size , _Data );

    for (size_t i = 0; i < (num_threads-1); ++i ) {
        while (futs_[i].wait_for(std::chrono::nanoseconds(0)) !=
            std::future_status::ready) {
            thread_pool_ -> run_pending_task();
        }
    }

    return _First ;
}

Listing 2.5: Basic mechanism for partitioning.

The code starts by fetching a pointer to the thread pool via its singleton interface (line 4). Lines 11-15 compute some quantities that we need in order get chunks that are as equally sized as possible. At line 17, we create a vector of futures on which we will later wait for jobs to complete. In the central for-loop, starting
at line 19, jobs that work on the chunks are created and submitted to the thread pool. For each job submission we retrieve a future and store it in the vector. The last chunk is not submitted but processed on the main thread (line 34). The for-loop starting at line 36 waits for all submitted jobs to complete. If some jobs are not finished yet, the main thread should not enter the idle state, which is why in that case it tries to help the thread pool complete its jobs. This is essential in a scenario where we have nested parallelism, which will be discussed in section 3.4.

By means of a simple example, we will now illustrate in how an STL algorithm implementation might make use of the partitioning framework. Let us say, we want to implement a parallel version of std::for_each, which we call my_for_each. It shall be designed for random access containers only.

We start by implementing a small utility called my_for_each_helper, as shown in Listing 2.6. All it does is providing a method which runs a callable on a specified range of elements. It works like the sequential std::for_each, except that we pass a count instead of an iterator to the end as a second argument.

```cpp
struct my_for_each_helper
{
    template < typename _InIt , typename _Fn >
    static void Loop(_InIt _First , size_t _Count , _Fn _UserFunc)
    {
        for (size_t _I = 0; _I < _Count ; ++_I)
            _UserFunc(_First[_I]);
    }
};
```

Listing 2.6: Utility function for my_for_each.

```cpp
template < class _InIt , class _Fn >
void my_for_each(_InIt _First , _InIt _Last , _Fn _Func)
{
    size_t _Count = std::distance(_First , _Last);
    _For_Each(_First , _Count , _Func , []( _InIt _Begin , size_t _Count , _Fn & _UserFunc ) {
        my_for_each_helper::Loop(_Begin , _Count , _UserFunc);
    });
}
```

Listing 2.7: Simple parallel implementation of for_each.

With this utility, our parallel for_each implementation is as simple as the method shown in Listing 2.7. One can mentally step through the code and verify, with the functionality of the thread pool and the partitioning framework in mind, that if we pass a random access container and a callable, the container will be split up in chunks, which in turn will then be processed by the thread pool.

The next section discusses execution policy objects, a means of controlling whether an algorithm is executed sequentially or in parallel.
2.3 Execution Policy Objects

In order to let the user decide about the execution policy, i.e. whether to run an algorithm sequentially or in parallel, we add a template argument to each STL algorithm, representing the execution policy. Additionally, a further function argument allows the user to pass a so called execution policy object. Depending on the type of this execution policy object, the compiler instantiates a sequential or a multi-threaded variant of that algorithm, using compile-time polymorphism. The example in Listing 2.8 illustrates how this works in practice, showing first how to call sort in the original STL, and then the pSTL call, including the execution policy object std::par.

```
1  std::sort(data.begin(), data.end()); // STL
2 3  std::sort(std::par, data.begin(), data.end()); // parallel STL
```

Listing 2.8: Sorting with different execution policies

The parallel STL provides three different types of execution policy objects:

1. `std::seq` orders a sequential execution, which is equivalent to using the original STL function.

2. `std::par` orders a multi-threaded execution.

3. `std::par_vec` orders a multi-threaded execution, where its code is written in a way that favors auto-vectorization by the compiler.

Additionally, there is a generic execution policy, called dynamic execution policy, which enables the decision of the appropriate policy to be made at run-time. We will not focus on the specifics of the dynamic execution policy. From now on, whenever we mention the parallel variant of an algorithm, we mean the pSTL call with the execution policy `std::par_vec`.

In the next section, we will discuss another type of utility: Composable iterators. Their purpose is to wrap multiple iterators into a single iterator.

2.4 Composable Iterators

Let us assume, we wanted to implement a parallel variant of `std::transform` in the same way as we did it for `std::for_each` in the previous subsection. Again only considering random access iterators, here is what the prototype looks like:

```
my_transform(InIt init_begin, InIt init_end, OutIt outit, Func f).
```

When we try to use the partitioning method `For_each` (Listing 2.5), we realize quickly that we are lacking a way to pass the output iterator. `For_each` takes only one iterator.

A solution to this problem is provided by a utility called composable iterator. Composable iterators are an application of the composite pattern. A composable iterator wraps multiple iterators and acts as a normal iterator from
the outside. If we increment it, all iterators that are contained inside are incremented. If we dereference it, we get the contained iterators in form of a `std::tuple`. Random access is possible, too, as long as the contained iterators implement random access, and returns a tuple as well. Figure 2.3 illustrates the principles of a composable iterator. On the left, we see simple iterators being aggregated into a composable iterator. Any operation (such as an increment) is forwarded to the contained iterators and changes their state, represented by a color. The right side visualizes the dereferencing procedure. With this tool at hand, let us return to `my_transform`. Instead of just passing the input iterator to the partitioner, like in `my_for_each`, we wrap the input iterator and the output iterator in a composable iterator, and then pass it. This leads to the desired behavior of `my_transform`.

2.5 Tests

We replaced the entire multi-threading back-end, rewrote the partitioning framework and made further adjustments. Thus, it cannot be taken for granted that the functionality of the pSTL prototype has not been affected. In order to restore and validate the proper functionality, we wrote several tests for each pSTL algorithm. The general procedure for writing a reliable set of tests for each algorithm involves the following steps:

1. Generate some reasonable inputs, possibly randomized. They should cover all possible use cases and a broad range of input sizes.
2. On each input, run the original implementation of the algorithm as well as its pSTL counterpart.
3. Check if the results are equal.

Applying these steps was straightforward. In the course of writing and executing the tests, a few bugs were discovered in the prototype, and fixed. The tests only check for correctness, not for performance, i.e. they do not check if the parallel variants actually lead to a speed-up.
2.6 Measurements

In order to use the parallel STL in real applications, it is important to understand how the individual algorithms compare to their sequential counterparts. In this section we study the results of a variety of measurements. Subsections 2.6.1 - 2.6.6 provide information and considerations about various aspects of the experiments, such as errors, tools and information about the target machine. Subsection 2.6.8 describes the experiment design and presents the results. Subsection 2.7 concludes.

2.6.1 Input Content

For each algorithm, we need a use case in order to conduct a measurement, i.e. we need to prepare a certain input. Here we are not referring to the size of the input (we will measure for different sizes), but to the actual contents. There are algorithms, where the actual content does not matter (e.g. copy). However, for certain algorithms this can make a difference of orders of magnitude. Consider find, which looks for the first occurrence of a certain value in a container: If we are searching for an element which occurs right at the beginning of the container, find terminates immediately. If, on the other hand, the element does not appear at all in the container, we have to iterate over the whole container, before terminating. The content of the input might also determine if an algorithm will be compute-bound or memory-bound. Consider for_each: If the callable contains a lot of floating point operations based on a single container element, for_each will be compute-bound. If the callable contains only very little code with maybe no floating point operations at all, we are memory-bound.

We cannot take into consideration every possible use case for every algorithm. Instead, we design a single, meaningful use case per algorithm and leave it up to the user of the parallel STL to investigate the performance in the context of a specific application. We describe our use cases for two examples: sort and transform. For sort, we used a randomly initialized vector of integers as input. For transform, we used the function object shown in Listing 2.9. It is a combination of a linear congruential random generator and an exponentiation. The only aim of this code is making sure the algorithm is compute-bound.

```cpp
auto user_func = [](const int & element) {
    const int a = 1664525;
    const int c = 1013904223;
    int new_element = element;
    for (unsigned int i = 0; i < 3; ++i)
        new_element = (std::exp(a * new_element + c) + 1) / std::exp(a *
                        new_element + c);
    return new_element;
};
```

Listing 2.9: Function object used for transform.
2.6.2 Nondeterminism

Naturally, execution times of programs are never deterministic. There will always be deviations from the expected execution time. In order to get as close as possible to the expected execution time, we execute multiple runs (usually 100 or 1000 for each measurement). Sometimes it is reasonable to assume that data points in a sample follow a normal distribution. However, when measuring computing performance, this is often not the case. According to [14], the reasons for this include scheduling, congestion and cache misses and typically lead to multi-modal and skewed distributions. Nonetheless, it is helpful to check for normality in our individual environment.

So called Q-Q-plots [27] (Quantile-Quantile-plots) provide a way to compare empirical distributions of samples to certain theoretical probability distributions. A Q-Q plot simply consists of plotting empirical quantiles of a sample against their theoretical counterpart in the distribution we like to compare to. In the case of a normal distribution, we plot the empirical quantiles against the theoretical quantiles of the normal distribution. If the points form a straight line, we have an indication that the sample points are normally distributed. Figure 2.4 shows the Q-Q-plot of a speed-up sample of size 1000 of the STL algorithm `count` with a container size of $10^5$. The blue dots represent one percentile each and the dashed red line indicates the linear trend. In this case the points almost form a perfect straight line and it is reasonable to assume normality. However, few algorithms and container sizes lead to samples with such straight Q-Q-plot lines. Figure 2.5 shows the Q-Q-plots of samples of four other algorithms: (a) `for_each` with container size $10^5$, (b) `move` with container size $10^4$, (c) `scan` with container size $10^5$ and (d) `partial_sort_copy` with container size $10^5$. Case (a) indicates a multi-modal distribution, case (b) a skew and case (c) fat tails. In case (d), outliers are apparent. In all four cases, assuming a normal distribution would result in false confidence intervals.

Since we cannot assume a theoretical probability distribution for the data points in our sample, it is impossible to compute standard errors or parametric confidence intervals. On the other hand, it is possible to compute nonparamet-
Figure 2.5: Q-Q-plots of various samples.
ric confidence intervals, which have an analogical interpretation\cite{14}. We use the statistically robust median to summarize the results and basic bootstrap to compute nonparametric confidence intervals of the median. The procedure of computing nonparametric confidence intervals using basic bootstrap is described in detail by Davison and Hinkley (1997)\cite{11}. The only assumption we make about the distribution of the sample is that the data points are independent. In all following charts, confidence intervals will be shown as red bars. In some (parts of) charts, there will not be any red bars, which means that the corresponding confidence intervals are very narrow and a visualization is not possible.

2.6.3 Initial Cache State

In order to get representative measurement results, we want the cache to be cold at the start of each run. This is the default scenario. However, when we execute multiple runs, there might be remnants of data in the cache from previous runs. Even in the first run, data from the container initialization might have persisted. This is why we flush the entire L2 cache before each run. This can be done by scanning through a different, contiguous container that takes more memory than the cache. We do this before starting the timer each time, because we do not want to include the flushing in our measurement.

2.6.4 NUMA effects

Another memory-related aspect we have to take into account are so called NUMA\cite{4} (non-uniform memory access) effects. The compute node, on which we carry out our experiments, is equipped with two sockets, each with its own processor and its own local memory. We call this design NUMA. The reason for preferring this approach over a single shared memory is that a memory can be accessed by only one processor at a time\cite{35}. This can starve other processors for data. Consequently, when we run an algorithm on a container in parallel, and it is clear which processor processes which part of the container, it is computationally more efficient to put the different parts of the container in the appropriate local memory, than to put the whole container in a single local memory. In NUMA architectures, data is always placed in the local memory of the processor, which accesses the data first. In our experiments, wherever it was possible, we wrote the code such that the container parts were put in the appropriate local memory, before measuring the execution time of a parallel algorithm.

2.6.5 Processor Affinity

In our scenario (having a thread pool), it is ideal for threads to not be migrated, as this causes unnecessary overhead. On GCC, the header `sched.h` provides methods to pin threads to certain processor. When threads are pinned to certain processors, we call it processor affinity\cite{28}. In our prototype, we wrote code for pinning threads to fixed processors. However, the thread pinning code did not work on the target machine.
2.6.6 Compiler Optimization

Compilers often eliminate code, on which the program output does not depend, in order to optimize the execution speed. For our experiment, this means that if we just run algorithms without relating them to the program output, the compiler might rearrange the code and the algorithms might not even actually run. This is why it is crucial to always conserve a part of the results of each run and relating it to the program output.

2.6.7 Target Machine and Tools

All measurements were carried out on a single compute node of Greina, a computing cluster at the Swiss National Supercomputing Centre in Lugano. The compute node has two sockets, each equipped with one Intel® Xeon® CPU E5-2680 v3 (Haswell microarchitecture), operating at 2.50GHz, providing a total hardware concurrency of 24. The operating system was Linux (kernel version 3.10.0). We used GCC with the optimization flag “-O3” to compile. Execution times were measured with the steady_clock from the header chrono.

2.6.8 Experiment Design and Results

Given the diversified nature of the parallel STL, there are endless possibilities of creating a measurement scenario. Ideally, the results would point out some of the advantages and disadvantages of the parallel STL. This is what we kept in mind when we designed the concrete experiments.

Henceforth, whenever we use the word speedup, we mean the ratio of the execution time of the optimized sequential variant to the execution time of the parallel variant. A very straightforward but insightful experiment is to fix the problem size and measure the speedups of all algorithms. This was done for three different container sizes: $10^4$, $10^5$ and $10^8$. Figures 2.6 - 2.8 visualize the results for all pSTL algorithms in a column chart. Consider the varying axis scaling. Each shade of blue represents a different container size. We can observe that there are extreme differences between the different algorithms. For large container sizes, some achieve an almost ideal speedup (e.g. for_each), whilst others do not even stand a chance against the sequential version (e.g. copy_if). Another remarkable point are the large differences in speedups for different container sizes. A possible explanation for this are certain constant overheads. Our thread pool manages 24 threads on the target machine, which might cause a significant synchronizing overhead. The three strongest algorithms in terms of speedup (for_each, transform and generate) are all compute-bound, which was arranged via a compute-bound function object. They are also the only compute-bound algorithms in the experiment. Clearly, for memory-bound algorithms, such strong speedups are harder to achieve, because the memory bandwidth is exhausted quickly with a growing number of threads.

Another interesting experiment is to fix an algorithm and the container size, and measure the speedup as a function of the number of threads (core scaling). This experiment was done for one compute-bound and one memory-bound algorithm. Figure 2.9 shows the resulting plots. As a representative of the compute-bound algorithms, we chose transform with container sizes $10^4$, $10^5$ and $10^8$ and a compute-bound callable (Figure 2.9(a)). For a large container, we
Figure 2.6: pSTL algorithms with largest speedup.

Figure 2.7: pSTL algorithms with medium speedup.
observe a near-ideal speedup. When the container gets smaller, overheads start to dominate. In the memory-bound realm, we chose `copy`. The results are shown in Figure 2.9 (b). Here, the connections between the measuring points do not indicate trends but are there to make the visual distinction between the curves easier. Besides the interesting dependency on the thread count, `copy` is also a nice example for measuring significant NUMA effects. We created three cases for the investigation of `copy`: One of each with container sizes $10^6$ and $10^8$, and one where the NUMA aware container initialization is switched off (container size: $10^8$). There are strong differences when compared to `transform`. One can immediately see that NUMA awareness has a crucial performance impact. Ignoring NUMA effects can truncate the speedup by almost one half. The absence of an ideal speedup for large containers is expected. Once the memory bandwidth is exhausted, adding extra threads will not improve the performance. An interesting effect is the very strong speedup with container sizes of $10^6$ when using less than three threads. The partly unexpected (non-monotonous) trends of the curves for the $10^6$ variant and the $10^8$ (without NUMA awareness) variant might be because of the failure of thread pinning.

2.7 Summary

Without any doubt, the parallel STL can lead to very strong speedups in some cases, especially when the containers are large (and this is usually where speedups are needed most). Additionally, the parallel STL completely encap-
Figure 2.9: Core scaling.
ulates the parallelization and all its building blocks (threads, futures, etc.). This means that an existing STL application can potentially be accelerated with almost no effort. Furthermore, the execution policy objects provide the user with a simple means control. He can decide whether to run a parallel or a sequential variant of an algorithm. The sequential variant might be preferable if there are potential data races, interference with further, manually programmed multi-threading, or even other applications running on the same machine.

There are also drawbacks in the parallel STL. Some are impossible to get rid of, others might be eliminated by further work. In the prototype, the algorithms usually have only a single, static implementation per execution policy, which makes their performance less competitive. It is clear that in some algorithms, the choice of the ideal implementation is dependent on the container size (or other factors). Therefore the parallel STL might be beaten by a customized, manual parallelization in some scenarios. A higher competitiveness requires a dynamic implementation of some algorithms. One example is \texttt{nth\_element}, of which the current implementation is less than ideal for very small (or large) \texttt{n}. If there were two implementations, one optimized for medium \texttt{n}, and one for very small (large) \texttt{n}, the algorithm could check \texttt{n} and dynamically decide on the most suitable implementation.

A further lack is that no NUMA aware allocators for STL containers are available. We have seen the potential impact of NUMA awareness after the measurements. Ideally, the parallel STL would provide NUMA aware allocators, which can be passed as arguments to the container templates.

The issues pointed out so far can potentially be eliminated by further work. However, there are cases that are restricted by nature. Some algorithms are just not parallel enough to make use of multiple threads. They are limited by their serial part (Amdahl’s law\cite{Amdahl}) or the inter-thread communication overhead. \texttt{remove} is a typical example.
Chapter 3

Case Studies

At this stage we assume a fully implemented and functional prototype of the parallel STL. The goal of this chapter is to investigate the suitability of the pSTL in the context of scientific applications with respect to performance and readability of the code. We selected four widely used algorithms (case studies), which can potentially test different limitations of the parallel STL:

1. Floyd Warshall
2. k Nearest Neighbors
3. Barnes Hut
4. Closest Pair (Divide and Conquer algorithm)

For each algorithm, we wrote different variants, using either only pSTL algorithms, manual parallelization or purely sequential code. Additionally we designed simple timing models in order to test the predictability and quality of the performance. In this chapter, we dedicate a section to each case study. Each section contains a description of the algorithm, a discussion regarding the implementations of the different variants, a simple execution time model, a visual presentation of measurement results, and a summary. All core scaling experiments were done with strong scaling. Subsections 2.6.1 - 2.6.6 provide detailed information about the measurement environment and methods, which are used for the case studies as well. As in chapter 2 the red bars represent 95%-confidence intervals, determined via basic bootstrap. Error bars not being shown at any point in any chart implies that they would be too narrow to be visible. Connected measuring points in a plot indicate a trend.

In order to have a consistent notation, let us define some additional quantities and identify them on our target machine (Intel® Xeon® CPU E5-2680 v3):

- \( B \): The DDR memory bandwidth for the full number of active threads. The bandwidth of our machine was measured with the tool `stream-scaling`\cite{26}: \( B = 70.275 \, \text{GB/s} \).

- \( C \): The capacity of the L2 cache. We have 24 cores, each of which is assigned 256 KB of L2 cache. This sums up to a total of \( C = 6 \, \text{MiB} \)\cite{10}.

- \( f \): The clock frequency of the target machine: 2.5 GHz, obtained via the shell command: `cat /proc/cpuinfo`.

29
• $M$: The hardware concurrency of the target machine: $M = 24$.

• $s_{\text{double}}$: The size of one double precision number on the target machine, determined via `sizeof`: $s_{\text{double}} = 8$ Byte.

## 3.1 Floyd Warshall

The Floyd Warshall algorithm computes the shortest path between any two vertices of a dense graph. Given a weighted graph with no negative cycles, let $N$ be the number of vertices and $A$ the $N \times N$ adjacency matrix representing the edge weights. The Floyd Warshall algorithm takes $A$ and two vertex indices $i$ and $j$ as inputs and returns the length of the shortest path between the two vertices. Listing 3.1 shows the pseudo code of the Floyd Warshall algorithm.

```
1 Input: A, i, j, N
2 D = A
3 for k from 0 to N-1
4   for i from 0 to N-1
5     for j from 0 to N-1
6       if D[i][j] > D[i][k] + D[k][j]
7         D[i][j] = D[i][k] + D[k][j]
8     endif
9  endfor
10 endfor
11 Output: D[i][j]
```

Listing 3.1: Pseudo code of the Floyd Warshall algorithm.

In a first stage, at line 3, the adjacency matrix is copied, because we do not want to invalidate it. The second stage consists of iterating over the whole matrix copy $N$ times, updating it each time. Floyd Warshall is usually the algorithm of choice for computing shortest path lengths in dense graphs. For sparse graphs, Dijkstra’s algorithm is more suitable.

The Boost Graph Library (BGL) contains an implementation of the Floyd Warshall algorithm which we use as a benchmark for our own implementations.

### 3.1.1 Implementation

We implemented three different variants: A naive sequential version, a manually parallelized version and a pSTL version. Additionally, we included the implementation from the Boost Graph Library. The sequential version is as simple as translating the pseudocode to C++. We used a one dimensional array to store the matrices (row major).

The two following sections describe the slightly more involved parallel implementations.

#### 3.1.1.1 Using the Parallel STL

For line 3, we use `copy` from the parallel STL. Unfortunately, there is no direct way to implement the remaining part of Floyd Warshall using the parallel STL.
We cannot simply iterate over a container, because inner kernel always access three elements at once. We cannot use composable iterators either, because the three accesses do not have the same incremental pattern. If we were to parallelize without the parallel STL, we would do it via the loop in the middle, the \( i \)-loop. We cannot parallelize the \( k \)-loop, because there are dependencies for each \( k > 0 \) on smaller \( k \). The \( j \)-loop is not ideal either, because our parallelization would be too fine grained and we would have a large number of jobs, which would increase the overhead portion. Additionally, it is not possible to change the order of the loop nesting of the \( k \)-loop, also due to data dependencies. In order to parallelize the \( i \)-loop (i.e. parallelize over the matrix rows) with the parallel STL, we need some sort of container holding elements referring to one row each. An ideal candidate for this endeavor is an index array. Once we have an array, which holds all \( i \)-values exactly once, our implementation becomes quite simple: we can use \( \text{for each} \) and tell it what to do for each matrix row. Figure 3.1 illustrates the concept. Each partition of the index array refers to a partition of the matrix, represented by a shade of blue.

At first glance there appears to be a race condition: Let us assume we are in any outermost iteration \( k \). Consider two threads \( A \) and \( B \), which execute concurrently, \( A \) processing the matrix row \( i_A = k \) and \( B \) processing a different row \( i_B \neq k \). Now consider any innermost loop index \( j \). The matrix element \( D[k][j] \), read by thread \( A \) at line 7, is conditionally written by thread \( A \) at line 8 at some point in time and we do not know which thread will reach \( j \) first or whether they reach it at the same time. However, it turns out that the condition at line 7 is never satisfied for \( i_A = k \), because \( D[k][j] \) is never strictly greater than \( D[k][k] + D[k][j] \). If \( D[k][k] \) were negative, we would have a negative cycle, which is not allowed. Hence, we can proceed without using locks or other arrangements.
3.1.1.2 Manual Parallelization

The higher level concept applied in our manually parallelized variant is the same: We parallelize over the rows in both the copy stage and the computation stage. What differs are the utilities we use for parallelization. We create a vector of threads (`std::thread`), where the vector size is equal to the hardware concurrency minus one. To each of the threads, we assign a partition of all the matrix rows, where the partitioning applies the same ideas as used in the partitioning framework. Likewise, we process the last partition on the current (master) thread. Unlike in the partitioning framework though, there is no need for futures and packaged tasks. The master thread can simply join all other threads once it is done processing its own partition.

3.1.2 Execution Time Model

In order to get some reference to relate our implementation, we designed a simple model that tries to roughly predict the execution time of the Floyd Warshall Algorithm. Most of our assumptions are deduced from the sequential pseudocode, which does not influence the result, if we neglect cache incoherence between processors (which happens rarely if $N$ is sufficiently large) and use the proper bandwidth corresponding to the full number of threads.

We assume that our L2 cache has a perfect associativity (in reality, it is 8-way associative). Furthermore, we assume that our matrix does not fit into the L2 cache, but at two rows of the matrix do fit. This is obviously not true for very small and very large matrices. However, we do not focus on very small matrices for two reasons: First, parallelizing is about accelerating slow code and very small matrices are processed quickly anyway. Second, for very small matrices, modeling becomes extremely complex, since different types of overhead start becoming significant (e.g. parallelization overhead, cache incoherence). On the other side, processing matrices large enough for two rows not fitting in the cache is far from feasible in a reasonable time. We would need to process a matrix with a row length of more than 131072 elements. Given the complexity of Floyd Warshall, $O(N^3)$, it is clear that this cannot be done within a few hours. One last assumption we make is a cold cache, when the algorithm is invoked.

Recall that the algorithm consists of two stages: Copying and actual computation. The copy stage is straightforward: We are memory-bound and have $N^2$ read accesses and $N^2$ write accesses of double precision numbers, all of which reach across the memory bus eventually. Hence, the model time spent in the copy stage, depending on the number of vertices $N$, is given by

$$t_{\text{copy}}(N) = \frac{2s_{\text{double}}}{B} N^2.$$

(3.1)

The computational part is a bit more involved. The inner kernel (lines 7-9) consists of one addition, even though we see a second conditional addition. In the actual implementation we store the sum in a register and reuse it if the condition is satisfied. We need to load 3 matrix elements and conditionally write one. A simple experiment shows that the condition is very rarely satisfied. For $N = 10^3$, which is the lowest $N$ in our range of consideration, and edge weights initialized under a uniform random distribution, we found that the condition is satisfied in only 1.137% of the cases, with a 95% confidence interval.
of [1.135%, 1.138%]. For larger $N$, this percentage is even lower. Therefore, we
eglect the write access in our considerations. The matrix elements we read are
$D[i][j]$, $D[i][k]$ and $D[k][j]$. Recall that two lines of the matrix fit in the
cache but the whole matrix does not, according to our assumptions. Since the
$j$-stride began at $j = 0$, we know for sure that $D[i][k]$ is in the cache. The
$k$-th row stays in the cache for the whole $k$-iteration once it is loaded, because
of the access to $D[k][j]$ at line 7. Since we have $N$ $k$-iterations, we count $N^2$
accesses for now. If $i$ is not equal to $k$, which is usually the case, $D[i][j]$ leads
to another main memory access. It cannot be in the cache from a previous
$k$-iteration, since the matrix does not fit in the cache. This leads to $N^3$ further
memory accesses. We have approximately $N^3$ floating point operations, which
means the compute stage is memory-bound as well. Our processing time for the
this stage is therefore given by

$$t_{\text{compute}}(N) = \frac{s_{\text{double}}}{B} (N^3 + N^2). \quad (3.2)$$

The total estimated execution time of the Floyd Warshall algorithm $t_{FW}$ is
equal to the sum of the execution times of the two stages. We arrive at

$$t_{FW}(N) = \frac{s_{\text{double}}}{B} \cdot (N^3 + 3N^2) \approx \frac{s_{\text{double}}}{B} \cdot N^3, \quad (3.3)$$

where the approximation is reasonable for all $N$ we consider.

In addition to our model, we introduce a function for the upper bound of
the execution time of the pSTL variant. Since most of our deductions are based
on cache issues, it is reasonable to compute the upper bound of the execution
time assuming a machine that has no cache at all. We have $N^2$ reads and $N^2$
writes in the copy stage and $3N^3$ reads in the compute stage (as in the model,
we neglect the writes). This leads to an upper bound of:

$$t_{FW,\text{upper}}(N) = \frac{s_{\text{double}}}{B} \cdot (3N^3 + 2N^2) \approx \frac{s_{\text{double}}}{B} \cdot 3N^3. \quad (3.4)$$

### 3.1.3 Measurement Results

We next present the results of a few experiments regarding the performance of
the parallel STL when applied to the Floyd Warshall algorithm. We measured
the execution times of all variants for a range of vertex set cardinalities under
the full hardware concurrency and the core scaling of the pSTL variant for two
different cardinalities.

Figure 3.2 shows the behavior of the execution time of our pSTL variant as
a function of the number of vertices. The chart is divided in two sub-ranges in
order to make the results for smaller vertex numbers better visible. In addition
to the measurement, the graphs of the execution time model and the upper
bound are also depicted. This yields a visual comparison between our theoretical
considerations and our measurement results. Subfigure 3.2 (a) indicates a close
fit for vertex numbers up to roughly 2300. At times the measured time is smaller
than the theoretical time, which is not a contradiction, because we assumed for
simplicity that only the elements of two matrix rows are in the cache. This is
not necessarily true in practice. For vertex numbers larger than 2300, shown
in Figure 3.2, the model starts getting too fast, with growing discrepancy.
The reason for this are again the oversimplified assumptions about the cache and its
Figure 3.2: Execution times of pSTL variant of Floyd Warshall.
behavior. Unsurprisingly, the upper bound is always satisfied. The only reason for it to not be satisfied would be overheads (e.g. parallelization) we did not account for, but they are too small and become significant only for very small numbers of vertices.

Besides relating the performance of the pSTL variant to theoretical consideration, it is also insightful to compare it against other variants. Figure 3.3 visualizes execution times of all our variants (pSTL, sequential, manual parallelization and Boost) for two different ranges of vertex set cardinalities. In Figure 3.3 (b), the Boost version was omitted, since it would introduce an unfavorable axis scaling. It suffices to know that it is much slower than the other variants and follows the trend indicated in Figure 3.3 (a). For very small vertex numbers, the sequential variants (including Boost) dominate. This is due to parallelization overhead. In the left chart it can be seen where the manually parallelized version crosses the two sequential ones. The pSTL version crosses much earlier, which cannot be seen in the chart due to our choice of the domain range. The Boost version is clearly dominated by our own sequential version. The reason for this might be the high level of abstraction in the Boost Graph

![Figure 3.3: Execution times of different variants of Floyd Warshall.](image-url)
Library. By far the strongest variant (except for very small vertex numbers) is the pSTL variant. It clearly dominates our manually parallelized version, because it accesses a thread pool internally. This avoids creating new threads in each k-iteration, which is done in the manually parallelized version and creates unnecessary overhead. If we accessed the thread pool in the manually parallelized version, it would be as fast as the pSTL variant, because then it would essentially do the same.

Besides the measuring the execution time as a function of the vertex set cardinality, it is also insightful to have a look at the core scaling, i.e. the execution time as a function of the number of threads. Subfigures 3.4 (a) and (b) show the core scaling of the pSTL variant for the vertex set cardinalities $N = 500$ and $N = 5000$. The indicated ideal speedup was computed with the sequential variant as a reference. For both problem sizes, our implementation is far from a perfect speedup, which is expected, since we are memory-bound. It was also expected that for the large problem size ($N = 5000$), it is a bit closer to the ideal speedup (in terms of ratios). This is because the parallelization overhead accounts for a smaller fraction of the execution time.

### 3.1.4 Summary

This case study shows a clear advantage of the parallel STL. With very little effort, the sequential variant of the Floyd Warshall algorithm could be parallelized and a speedup of almost 5 could be reached for large containers. Even the relatively naive manual parallelization, that creates threads in each k-iteration, requires more effort and knowledge. Yet, the pSTL variant beats all other versions in terms of performance. Although, for obvious reasons, a more involved manual parallelization could keep up with the pSTL variant. Besides outperforming other variants, the pSTL variant is also predictable to some extent, with respect to its execution time, as Figure 3.2 shows.

A disadvantage is the necessity of an index array and the non-apparent data dependency due to the use of `for_each`, which makes it harder for the compiler to analyse the code. Both also affect the code readability in a negative way. In our case it might be desirable to have STL calls with more complex access patterns, instead of only simple, point-wise iteration. Furthermore, it is a general problem that the STL only provides one-dimensional containers. There is an approach to tackle this problem, though. A data structure called Array View provides a multi-dimensional view on containers.

### 3.2 k Nearest Neighbors

The $k$ Nearest Neighbors algorithm is used in the field of machine learning. It is a nonparametric method for classification and regression[1]. In our case, we use it for classification.

Let $P \subset \mathbb{R}^3$ be a finite set of points and $f : P \to \{0,1\}$ a mapping that assigns a label to each point. Furthermore, let $k \in \mathbb{N}$ be a strictly positive integer, $g : \mathbb{R}^3 \to \{X \in 2^P : |X| = k\}$ a function that returns the $k$ nearest points in $P$ (with respect to the Euclidian distance and under the simplifying assumption that no two points have the same distance), and $h : \mathbb{R}^3 \to \{0,1\}$
Figure 3.4: Core scaling of pSTL variant of Floyd Warshall.
the function that maps \( s \in \mathbb{R}^3 \) to 1 if the majority of \( g(s) \) are assigned the label 1 and to 0 otherwise.

The \( k \) Nearest Neighbors algorithm performs the mapping \( h[6] \). In a nutshell, it takes a new point \( s \) as input, finds the \( k \) nearest points in \( P \) with respect to \( s \) and selects the label with the higher occurrence among those \( k \) points. The general implementation concept is simple: First we compute the distances between the new point and all points in \( P \) and store them in a container. This can be done in a straightforward way via a call to \texttt{transform}. The container can then be used to pick out \( k \) labels, corresponding to the points with the shortest distances, which usually involves the use of \texttt{nth_element}.

### 3.2.1 Implementation

Three different variants were written. One is a manually tuned version, and the two others use exclusively STL calls, with sequential and parallel execution policies, respectively. We first explain the latter two. We start by creating a vector of the size \( L \), where \( L \) is the cardinality of \( P \), our set of data points. The type of the vector is a pair of a floating point number (double precision) and an integer. The floating point number represents a squared distance and the integer a label (squared distances make expensive square root operations unnecessary). Then we call \texttt{transform}, where the inputs are the container holding the points in \( P \) (a vector of arrays of double precision numbers) and the container holding the labels (a vector of integers). The output is the newly created vector of pairs. We let \texttt{transform} compute the squared distance from each point in \( P \) to the new point and store it along with its label in the vector. Then we invoke \texttt{nth_element} with \( n = k \) on this vector, were the key is the squared distance entry of the pair. Finally, we call \texttt{count} on the first \( k \) elements of the newly arranged vector, where we count the occurrences of the label 1, and check if this count is larger than half of \( k \). We point out that in our case we hold a pre-allocated (under NUMA awareness) vector with pairs ready and pass it to the algorithm, instead of allocating it within the algorithm, because NUMA aware allocation is expensive.

The manually parallelized variant is somewhat more involved. The distance computation part is done in much the same way, except with direct access to the thread pool. Now we have again our vector of pairs after the computation of squared distances. But instead of implementing some parallel variant of \texttt{nth_element}, we split the vector in equally sized chunks and invoke, on separate threads, the sequential version of \texttt{nth_element} with \( n = k \) on each chunk. From each chunk, we can then pick out the \( k \) pairs with the smallest squared distances and aggregate them into a new vector of pairs. The new vector now contains \( k \times M \) pairs, where \( M \) is the number of threads (or, equivalently, chunks). On this vector, we call a again \texttt{nth_element} with \( n = k \). Finally, we can invoke \texttt{count} on the first \( k \) elements. We will discuss later, why this approach is faster than the STL approach with parallel execution policies.

### 3.2.2 Execution Time Model

We continue by setting up a model for the execution time of the pSTL variant of \( k \) Nearest Neighbors. We distinguish three stages: \texttt{transform}, \texttt{nth_element} and \texttt{count}. The \texttt{count} stage is very inexpensive compared to the other two,
which is why we neglect it. In the `transform` stage, we first need to figure out if we are memory-bound or compute-bound. In each iteration we make 5 additions (or equivalently subtractions) and 3 multiplications when computing a squared distance. Both additions and multiplications have a throughput of 96 operations per cycle on the target machine (corresponding to a throughput of 4 on each core, assuming vectorization). Hence, the additions are the bottleneck on the compute side. Our clock frequency is \(2.5 \text{ GHz}\). During execution, we therefore need roughly 21 ps per iteration. In each iteration, we also load 28 Bytes (3 double precision numbers for the position and an integer for the label) and write 12 Bytes (the squared distance in double precision and the integer label). These 40 Bytes are transferred between the cache and the main memory. With our given memory bandwidth of \(B = 70.275 \text{ GB/s}\) we arrive at 569 ps per iteration. Hence, we are memory-bound and our estimated time for the `transform` stage is given by:

\[
t_{\text{transform}}(L) = \frac{L \cdot 40 \text{ Byte}}{B}.
\]

The second stage is a call to `nth_element`, which on average has a time complexity linear in the container size\(^8\). Simple experiments show that, even when the container size is fixed, the execution time has a high variance and is highly dependent on the data. Consequently, the same holds for the k Nearest Neighbors algorithm as a whole. Therefore we explicitly mention here that our model will not predict the accurate execution time but its median and we will use relatively large samples when we conduct the performance measurements. Predicting the execution time of the `nth_element` routine itself theoretically is a very complex task, which we will avoid. Instead, we introduce a simple linear model and fit the parameters via regression. Our model is of the form

\[
t_{\text{nth element}}(L) = a_0 + a_1 L,
\]

where \(a_0\) and \(a_1\) are the parameters that require fitting. For the fitting, we sampled 100 times at each point of an equidistant grid from 0 to \(5 \cdot 10^6\) with a spacing of \(10^5\). Since we want the model to predict times for the pSTL variant, we passed the parallel execution policy. The type of the container was matched to the one in the k Nearest Neighbors implementation. For each element in the sample, we performed a linear regression to determine \(a_0\) and \(a_1\). Finally we took the medians for both, which resulted in the following values:

- \(a_0 = 958 \cdot 10^{-6}s\), with a 95%-confidence interval of \([-460 \cdot 10^{-6}s, 4773 \cdot 10^{-6}s]\),
- \(a_1 = 7.82 \cdot 10^{-9}s\), with a 95%-confidence interval of \([7.68 \cdot 10^{-9}s, 8.22 \cdot 10^{-9}s]\),

where the confidence intervals are again determined by using basic bootstrap. Figure 3.5 shows a scatter plot of the execution times versus the container sizes including our regression model (red dashed line). The extremely large variance is apparent. However, as the confidence intervals indicate, we can fit the parameters \(a_0\) and \(a_1\) quite accurately.

### 3.2.3 Measurement Results

Like in the Floyd Warshall case study, we measured the execution times of all variants for a range of data point numbers \(L\) and the core scaling of the pSTL
Figure 3.5: Scatter plot and regression model for \texttt{nth.element}.

variant with for two different values of $L$. Figure 3.6 shows the execution times of the pSTL variant for up to 20 million data points. The model prediction is shown along with some confidence space, indicated by dashed light blue lines, which represent the model under the 0.025 and 0.975 quantiles of the \texttt{nth.element} model parameters, respectively. We observe that the model is quite accurate, which is no surprise: If we look at the model of the transform stage, we can compute its execution time at any point within the depicted range and realize that it takes only a small fraction of the total time, which is to the largest part consumed by the \texttt{nth.element} stage, for which in turn we fitted a separate model. Nonetheless, the pSTL execution time is still significantly greater than the model time, which can be traced back to overheads.

Figure 3.7 shows the pSTL variant in comparison with the sequential and the manually parallelized ones. The dominance of the manually parallelized variant is striking, yet the explanation is simple: For \texttt{nth.element}, Figure 2.8 indicates a speedup close to one, after parallelization. We conclude that this stage of the pSTL variant is unlikely to be much faster than if it was sequential. The manually tuned variant, on the other hand, detects the hardware concurrency
(which is supposed to be transparent to the pSTL user), divides the container into chunks accordingly, and performs \texttt{nth\_element} on each chunk in a separate thread. Clearly, invoking \texttt{nth\_element} on 24 containers in separate threads is faster than invoking it on a single thread on a container 24 times the size. Of course, in the manually parallelized variant, there is an additional call of \texttt{nth\_element} afterwards, but that container is much smaller than the original one.

The fact that the \texttt{nth\_element} stage, which makes up a large part of the pSTL variant, achieves no significant speedup, affects the core scaling strongly. This can be seen in Figure 3.8. There is even a slight but significant increase in the execution time from 6 to 12 threads. The reason for this might be traced back to the problematic parallel implementation of \texttt{nth\_element}. The \texttt{transform} stage is the reason why there is some speedup at all.
3.2.4 Summary

This case study points out some of the shortcomings of the parallel STL in general, as well as of our prototype in particular.

By using an algorithm different from (but resembling) nth_element in our manual parallelization, we could beat the pSTL variant by a factor of almost 4. The parallel STL is lacking a suitable call, which indicates that the algorithm selection it provides is inadequate. Besides that, an optimized fusion of transform and nth_element would be useful. It would render an intermediate container unnecessary and compact the code. More generally speaking, this negative feature is stereotypical in programs that use the (parallel) STL. There is no optimization across multiple STL algorithms.

In addition, the parallel implementation of nth_element in our prototype is poor and there is no significant speedup over the sequential version. At least for some n, significant speedups are theoretically possible. For instance, consider n=1. We could use find, for which we found a relatively strong speedup, and then perform a simple swap. But in our prototype, there is only one implementation per algorithm, which is independent on the input. This is a general weakness.

3.3 Barnes Hut

Traditional N-body simulations have a quadratic time complexity (O(N^2)), because for each body, the force on it exerted by each other body needs to be computed. The Barnes Hut simulation is an approximation of the N-body simulation and has a time complexity of O(N log N) [5]. It achieves this complexity by dividing the space using an octree. When computing the force on a specific node, this enables the distinction between close other nodes, and distant other nodes. Distant other nodes are grouped into clusters represented by a single node. This strongly reduces the amount of computation needed. N-body simulations can be applied to many physical interactions. Here we restrict ourselves to gravitation. In our scenario, the bodies are initially placed in a cube under a uniform distribution. The initial velocities are all set to zero. If any particle hits a wall of the cube, it will bounce back elastically, i.e. the affected velocity component will be inverted. We also fix the number of time steps R = 10 in advance.

First, we fix some notation for this case study. Let N be the number of bodies. We introduce a time stepping scheme, starting at 0. Let Δt ∈ R be a spacing and let t_j, j ∈ {0, ..., R} form an equidistant grid on the time domain:

\[ t_j = jΔt. \] (3.7)

Furthermore, for any j ∈ {0, ..., R} and any i ∈ {0, ..., N}, let s_i^j, v_i^j, F_i^j ∈ R^3 and m_i ∈ R represent the approximated position, velocity, force and mass of the i-th body at time t_j, respectively.

The idea of Barnes Hut is simple. Groups of bodies that are close to each other are represented by their center of mass, when interacting with a distant body. Let us assume, we have an index set P, corresponding to bodies forming a group at time t_j. Then the center of mass (represented by the mass m_P and
the position \( s_P \) of that group at that time is given by:

\[
m_P = \sum_{i \in P} m_i, \quad s_P = \frac{1}{m_P} \sum_{i \in P} m_i s_i^j.
\] (3.8)

The groups of bodies are assigned by means of an octree: The entire space is divided into octants. If an octant contains more than one body, it is recursively divided into subordinated octants. When computing the force on a certain body exerted by the other bodies in a certain octant, one first computes the quotient \( s/d \), where \( s \) is the width of the octant and \( d \) is the distance between the body and the center of mass of the bodies contained in the octant. Let \( \theta \) be a non-negative constant (usually equal to 0.5), fixed in advance. If \( s/d < \theta \), then we say the octant is sufficiently far away and we treat it as a single body given by the center of mass of its contained particles. Otherwise, we recursively perform the same routine to each of the subordinated octants. In the trivial case where an octant contains only a single body, we simply compute the force exerted by that body.

### 3.3.1 Implementation

Each time step consists of three stages:

1. Tree construction.
2. Force computation.
3. Update of positions and velocities.

The parallelization of the tree construction is cumbersome and, as we will see, this stage consumes only a small fraction of the total execution time, which is why we only implemented a sequential variant of it. On the other hand, the force computation and the update are ideal candidates for parallelization.

Our tree consists of nodes, representing an octant each. We will use the terms node and octant interchangeably. A node stores the center of mass of the contained particles and references to its subordinated nodes. For the tree construction, we create a root node, representing the whole space. The construction starts at the root node and inserting a new body at any node is done via the following recursive scheme:

1. If the node does not contain a body, put the new body in it.
2. If the node is an internal node, update the center of mass of the corresponding octant and recursively place the body in the appropriate subordinated node.
3. If the node is a leaf and therefore already contains exactly one body, update the center of mass, subdivide the node into eight subordinated nodes and recursively place both the old and the new body in the appropriate subordinated nodes.

The computation of the force exerted on a body \( b \) follows a similar scheme, again starting at the root node:
1. If the node is a leaf, calculate the force exerted by the contained body on $b$ (if it is $b$ itself, we do nothing) and add it to the net force of $b$.

2. Otherwise, compute the quotient $s/d$. If $s/d < \theta$, treat the node as a single body and compute the force exerted by the center of mass and add it to the net force of $b$.

3. Otherwise, recursively perform the routine on each subordinated node.

In all our variants, the forces are stored in a separate container. We can comfortably invoke \texttt{transform} to compute and store the forces.

After computing the forces on all bodies, represented by their index $i$, we proceed with the third stage, updating the states of the bodies according to the following explicit scheme:

1. $s_{i}^{j+1} = s_{i}^{j} + \Delta t \mathbf{v}_{i}^{j} + \frac{1}{2} \Delta t^2 \frac{\mathbf{F}_{i}^{j}}{m_{i}}$,

2. $\mathbf{v}_{i}^{j+1} = \mathbf{v}_{i}^{j} + \Delta t \mathbf{F}_{i}^{j}$.

Additionally, for each body, we check if it has crossed the boundary and take appropriate measures if need be. In order to save space, we do not use separate containers for each time step, which why we have to call \texttt{for each} and cannot make use of \texttt{transform} here.

Since in stage 2 and 3, we have to iterate across multiple input containers, we wrap them into a composable iterator for the variants that use the STL.

### 3.3.2 Execution Time Model

Let us consider two hypothetical cases. In the first case, $N$ is small enough that the whole octree fits into a L2 cache. Then we need to load very little data from the main memory. We are compute-bound. In the second case imagine that the octree is very large, such that almost every node access leads to a memory transfer. In this case, the memory interface cannot keep up with the computation and we are memory-bound. From those two cases we conclude that it will not be possible to set up a reasonable homogeneous execution time model of the pSTL variant. Instead, we design a model composed of two parts: For smaller $N$, we design a compute-bound model, and for larger $N$ a memory-bound model.

Let us first introduce the compute-bound model. Our implementation includes additions (subtractions are also counted as additions), multiplications, divisions and square roots of double precision floating point numbers. The number of those operations is nondeterministic. However, the variance is low and we can fit a simple model. For all of those numbers $c$, we introduce a super-linear model of the form

$$c(N) = c_0 N \log N.$$  

(3.9)

There is a slight flaw in this model: the third stage, where positions and velocities are updated, has only linear complexity. However, because of this very fact, it contributes only a small fraction to the total numbers, so this is expected to have a minor impact. Indeed, we have measured the contribution of the third stage to the total number of each operation, and even for the smallest $N$ of our consideration ($N = 50 \cdot 10^5$), the contribution to additions was
<table>
<thead>
<tr>
<th>Operation</th>
<th>Median</th>
<th>0.025-Quantile</th>
<th>0.975-Quantile</th>
<th>Rel. error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>7.690 · 10⁴</td>
<td>7.449 · 10⁴</td>
<td>7.885 · 10⁴</td>
<td>3.716%</td>
</tr>
<tr>
<td>Multiplication</td>
<td>5.715 · 10⁴</td>
<td>5.592 · 10⁴</td>
<td>5.872 · 10⁴</td>
<td>3.153%</td>
</tr>
<tr>
<td>Division</td>
<td>724.935</td>
<td>709.561</td>
<td>744.496</td>
<td>3.098%</td>
</tr>
<tr>
<td>Square root</td>
<td>707.558</td>
<td>692.115</td>
<td>727.224</td>
<td>3.199%</td>
</tr>
</tbody>
</table>

Table 3.1: Parameter fits of operation count models.

0.127%, to multiplications 0.220%, and to divisions 0.432%. Square roots do not occur at this stage. For all operation numbers, we fit the parameter $c_0$ by sampling $c(N) / (N \log N)$ on an equidistant grid from 0 to $10^6$ with a spacing of $50 \cdot 10^3$ and taking the median. Confidence intervals are obtained, as usual, via basic bootstrap. Table 3.1 shows the results, along with the average relative prediction errors with respect to the measuring points in the sample. Since additions and multiplications, which have the same throughput on Haswell, run on separate units on the processor, and there are more additions than multiplications, we can ignore the multiplications. With vectorization enabled, the throughput of the addition unit is 4 operations per cycle per core, and the average throughputs of divisions and square roots are both 4/13 operations per cycle per core, according to the Intel® Optimization Reference Manual[15]. Let $c_{add}(N)$, $c_{div}(N)$ and $c_{sqrt}(N)$ be the model prediction total operation counts of additions, divisions and square roots, respectively, as a function of the number of bodies $N$. Then the compute-bound part of our model is given by:

$$t_{cb}(N) = \left( \frac{1}{4} c_{add}(N) + \frac{13}{4} (c_{div}(N) + c_{sqrt}(N)) \right) \frac{1}{24f}, \quad (3.10)$$

where $f$ is the clock frequency of the target machine.

For the memory-bound part of our model, we have large $N$, and the memory required for the octree is much larger than the cache. Consequently, most of the node accesses will lead to cache misses and a memory transfer. Now we have the same problem as in the compute-bound part: we do not know the exact number of node accesses and therefore have to fit a model, too. It will be of the from of Equation 3.9 as well. After evaluating, we arrived at a median of 815.563 for the parameter fit, with a 95% confidence interval of [797.969, 837.970] and an average relative error of 3.157%. The memory space needed for one node is $s_{node} = 104B$ (determined via sizeof). Let $c_{access}(N)$ denote the model-predicted number of node accesses, as a function of $N$. Then the execution time predicted by the memory-bound part of our model is given by:

$$t_{mb}(N) = \frac{s_{node}}{B} c_{access}(N). \quad (3.11)$$

As we will see, both model parts predict an execution time much lower than was measured. It is likely that even for the lowest $N$ in our measurement range, we are already memory-bound, so the compute-bound model part might not be applicable to our measurement range. And the memory-bound model assumes the maximum streaming bandwidth, including pre-fetching and full use of all loaded cache lines. Clearly, the latter two are not present in our scenario, because we have a random access pattern and to not iterate uniformly
across the container. This led us to introduce an improved model, based on the throughput of tree nodes loaded from the memory under a random access pattern. We wrote a separate program to evaluate this throughput, defined as number of nodes loaded per time unit. It randomly loads Barnes Hut tree nodes from a large container. With this program, we measured a throughput of $f_{RA} = 221 \cdot 10^6 \text{s}^{-1}$ with a 95% confidence interval of $[218 \cdot 10^6 \text{s}^{-1}, 222 \cdot 10^6 \text{s}^{-1}]$.

The following formula describes our improved model of the execution time, denoted by $t_{\text{improved}}$, as a function of $N$:

$$ t_{\text{improved}}(N) = \frac{c_{\text{access}}(N)}{f_{RA}}. $$  \hspace{1cm} (3.12)

### 3.3.3 Measurement Results

We measured the execution time versus the number of bodies $N$ for all variants on an equidistant grid on the interval $[50 \cdot 10^3, 10^6]$, with a spacing of $50 \cdot 10^3$. Figure 3.9 shows the execution times of our pSTL variant in comparison to the model-predicted times. As we can see, the original models predict poorly. However, the improved model, which takes random access into account, is quite accurate in the observed range. For lower $N$, where the implementation is performs better than the model predicts, some locality is probably in play. The comparison of the different variants is shown in Figure 3.10. The parallelization via parallel STL achieves a remarkable speedup. It is still far from the ideal speedup, which is fine, since we are memory-bound. There is also a serial part in the algorithm (the tree construction), but it takes only a small fraction of the time. For $N = 50 \cdot 10^3$ and $N = 10^6$, the measured contribution to the total number of node accesses was 0.897% and 0.723%, respectively.

Furthermore, there is no significant difference with respect to performance between pSTL and manual parallelization, which is the reason why the curve corresponding to the manual parallelization is barely visible. It is hidden behind the pSTL curve.
Figure 3.10: Execution times of different variants of Barnes Hut.

Figure 3.11: Core scaling of pSTL variant of Barnes Hut.

The core scaling of the pSTL is favorable. For $N = 10^6$, it is shown in Figure 3.11. The parallel STL does a good job accelerating the application.

3.3.4 Summary

This case study shows again the potentially great speedup one can achieve by simply exchanging execution policies for an STL call. The pSTL variant is as good as the manually parallelized variant. Additionally, the pSTL variant gets close to the full utilization of the peak random access throughput between main memory and CPU, as the comparison to the improved model shows. The core scaling is adequate as well.

Furthermore, the possibility to use a simple transform at the most expensive stage (force computation) creates opportunities for compiler analysis due to the apparent data dependency.

By looking at the code, though, one will quickly see a clear shortcoming of
the parallel STL. Because some calls require three or more input iterators, which the STL does not support, we are forced to make use of composable iterators. This heavily affects the readability of the program. The wrapping code and the dereferencing via tuples clutter up the code. Enabling the use of variadic templates in order to pass an arbitrary number of input iterators would be a desirable feature and relatively straightforward to implement.

### 3.4 Closest Pair

The closest pair problem is a problem in the field of computational geometry. Given a set of points in a vector space with a metric, it returns the pair of points with the shortest distance between them\[31\]. We restrict ourselves to a 2-dimensional space and to returning the distance instead of the actual pair. A naive way of doing this is to simply compute the distance between the points of all pairs and then identify the minimum. This approach is called **brute-force** and has a time complexity of $O(N^2)$, where $N$ denotes the number of points. Another algorithm, called **divide and conquer**, is of complexity $O(N \log N)$. It is the algorithm we implemented for this case study. The basic idea is to split the space in half and recursively execute the same algorithm on each half-space. This implies an opportunity to parallelize. However, there is still a serial part in the end, where we have to check a subset of the pairs of points which reach across the border where we split the space. We implemented this algorithm using STL calls. The purpose of this smaller case study is to show that the parallel STL supports nested parallelism, which arises when we run a parallel function recursively.

#### 3.4.1 Implementation

The recursive implementation involves the following steps:

1. Sort the points with respect to their first coordinate.

2. If the number of points is less or equal than 3, use the brute-force algorithm and return.

3. Call the algorithm recursively on the first and the second half of the sorted container, except that the sorting does not need to be done for all subsequent recursions. This yields two minimal distances $d_{left}$ and $d_{right}$.

4. Considering pairs of points close to and reaching across the border between the half-spaces, find the minimum $d_{mid}$ among their corresponding distances. There is a more sophisticated way to do this than simply considering all pairs of points within a certain range of the border. It involves sorting with respect to the second coordinate and we will not focus on the details here.

5. Return the minimum of $\{d_{left}, d_{mid}, d_{right}\}$.

Figure 3.12 illustrates the principle. We implemented a sequential, a manually parallelized, and a pSTL variant. The sequential variant is straightforward. The manually parallelized variant and the pSTL variant parallelize step 3 using the
thread pool directly or with a parallel `for_each`, respectively. In both cases it helps to switch to the sequential variant once a certain recursion depth is reached, e.g. when it is certain that all threads are busy. Otherwise there are unnecessary overheads and memory use. The pSTL variant switches at a fixed recursion depth of 8 (which means there will be total maximum of 256 jobs in all local job queues). The manually parallelized variant detects the hardware concurrency $M$ and calls the parallel recursion if and only if there are less than total of $2M$ jobs in all local job queues.

The correctness of the programs can easily be verified by comparing to the brute-force algorithm.

### 3.4.2 Execution Time Model

Our goal in this case study is to show that the parallel STL supports nested parallelism, for which an execution time model is of limited use. Besides that, the recursivity makes it difficult to introduce a reliable model. Therefore, we omit this part in this case study and continue with our measurement results.

### 3.4.3 Measurement Results

We have to point out that our measurements of the Closest Pair case study are not reproducible and sometimes difficult to interpret. Like for the core scaling of `copy` (subsection 2.6.8), we suspect that the reason for this is the failure of the thread pinning code.

We measured the execution times of our variants versus the number of points, ranging from $50 \cdot 10^3$ to $10^6$. Figure 3.13 shows the results. We observe that the pSTL variant performs very closely to the manually tuned variant. The tiny differences at some points might be due to the fact that the manually tuned variant adapts the recursion depth, at which it switches to the sequential variant, to the hardware concurrency. With respect to the sequential variant, we reach a speedup of roughly 3. The core scaling of the pSTL variant for $10^6$ points is shown in Figure 3.14. Clearly, we are far from an ideal speedup. The reason for this is at least partially of theoretical nature. The algorithm has a serial part and Amdahl’s law[2] implicates that there is a maximum achievable speedup lower than the ideal speedup.
3.4.4 Summary

While the parallel STL might not be particularly useful in this example, we have shown that nested parallelism is supported and we have found no apparent performance deterioration caused by nested parallelism via parallel STL, when compared to manual nested parallelism.
Chapter 4

Related Work

There is a number of parallel extensions for the C++ programming language. In particular, we take a look at the projects STAPL, C++ Amp, Thrust, and TBB, as well as papers written in their context.

4.1 STAPL

The **Standard Templates Adaptive Parallel Library** (STAPL) was introduced by Rauchwerger et al 1998 ²⁵. Its focus is on irregular, non-numeric programs and it is supposed to represent a superset of the STL, consistent with STL functions with the same names. Rauchwerger et al believe that nested parallelism is an important feature and make it a requirement for the STAPL. The parallel STL provides nested parallelism as well, as one of our case studies (section 3.4) shows. STAPL is in accordance with the STL design philosophy of separating containers from generic algorithms, and adopts the concept of iterators. Unlike the parallel STL though, STAPL uses its own types for containers and iterators. But it provides means for translating STL containers to STAPL containers. An et al 2001 ³ show that the performance deterioration caused by this necessary translation is minor. The parallel STL is able to work with common STL containers. However, performance might be significantly degraded, if the container memory is not allocated under NUMA ³⁴ (Non-Uniform Memory Access) awareness. This effect is particularly strong within memory-bound algorithms. Another key feature of the STAPL is that it is adaptive. It selects the best algorithm, the number of processors, scheduling, and the data layout. Furthermore, it supports distributed memory.

4.2 C++ AMP

C++ Accelerated Massive Parallelism is a programming model that provides means of accelerating C++ code by making use of data-parallel hardware, such as Graphics Processing Units (GPUs) ³⁰. It is implemented by Microsoft® based on DirectX 11 and its intent is to make it easier to program GPUs. At the same time it provides means of deeper control for developers with more expertise in the field ³⁰. C++ AMP provides, among others, routines and data structures
that resemble the ones in the parallel STL, such as the parallel_for_each function, which is the counterpart of the pSTL call std::for_each. C++ AMP is an open specification, and there are implementations outside of Microsoft® as well, such as Shevlin Park\cite{17}. Shevlin Park is based on Clang/LLVM and OpenCL.

4.3 Thrust

Thrust is the GPU equivalent of the parallel STL. It provides STL-like templated functions and data structures\cite{23}. Due to the nature of graphics processing units, its potential speedups are much higher than the ones of the parallel STL. However it is based on CUDA and therefore much less portable than our prototype of the parallel STL.

4.4 TBB

Threading Building Blocks is a C++ library designed by Intel® for writing programs that take advantage of multi-core architectures\cite{16}. Similar to the extensions discussed above, it provides a collection of algorithms and data structures for parallel computing. Like our pSTL prototype, it implements work stealing to balance the workload among CPU cores. A further similarity to the parallel STL (and to Thrust as well) is its heavy use of templates.
Chapter 5

Conclusion

Various experiments throughout this project made it possible for us to identify the strengths and weaknesses of the parallel STL in general, and of our prototype in particular. We divide this concluding chapter into three parts. Section 5.1 is related to the parallel STL in general and section 5.2 deals with the shortcomings of our prototype. Both summarize the findings we acquired while performing the experiments. Section 5.3 reveals the limitations of this work.

5.1 General Evaluation of the Parallel STL

Our measurements confirm that the parallel STL has the potential to strongly accelerate programs and reach speedups close to the theoretical maximum. Furthermore, all case studies except $k$ Nearest Neighbors indicate that the parallel STL can compete with manual parallelizations, if the application can be decomposed into STL calls. If an application consists of STL algorithms, then the parallelization can be done with almost no effort or expertise. It is as simple as adding, or exchanging execution policies to the STL calls. Technical details of the parallelization and the hardware abstraction are encapsulated and hidden from the programmer, which makes the code simpler and easier to read. The Closest Pair case study confirms that the parallel STL also supports nested parallelism, which is desirable, since recursion is an important paradigm in software design. In addition, STL algorithms, where the dependency of data is apparent, such as transform, facilitate compiler analysis and optimization. Another advantage of the parallel STL is the high level of abstraction, when compared to other programming models, such as OpenMP.

Then again, clear disadvantages were identified, too. One major drawback is that across separate algorithms, no optimization is possible. The $k$ Nearest Neighbor case study is an example, where an optimized fusion of transform and nth_element would be useful. The case study also indicates that the selection of the algorithms the (parallel) STL provides is not sufficient in some scenarios. The Barnes Hut case study points our the desirability of being able to neatly handle an arbitrary number of input iterators. The parallel STL does not provide this, and the only way to bypass the problem is to wrap multiple iterators into a composable iterator, which is cumbersome and complicates the code. Additionally, the Floyd Warshall case study points out that more com-
plex access patterns than simple strides through containers would be of use. Stencils, which we did not treat in this work, are another example. In this case, for computing a single data point, the data accesses to other points have fixed offsets, and it would be favorable if an abstraction of this pattern was provided, which the compiler understands. Multi-dimensional views of containers would provide a more intuitive way of programming, too. The parallel STL provides only 1-dimensional views on containers. Among the advantages of the parallel STL, we mentioned the apparent data dependency in calls like transform. However, in many cases, for_each is the only way of parallelizing code. This is hardly different from manual parallelization and suboptimal with respect to the visibility of data dependencies. The case studies Floyd Warshall and Closest Pair are representative examples of this problem.

We conclude this general evaluation of the parallel STL by saying that, while its drawbacks are significant and clearly apparent, most of them could be at least partly eliminated by further design choices. Arbitrary numbers of input iterators via variadic templates, and more complex access patterns and multi-dimensional views for containers are easy to design and implement. Furthermore, there are no obvious objections why the parallel STL should not provide a richer selection of algorithms.

5.2 Shortcomings of the Prototype

Our prototype is clearly not ideal with respect to performance. In the experiments in chapter 2, a significant number of pSTL algorithm showed very poor speedups, ranging from being only slightly faster than the sequential variant, to performing even worse by factors. While this can be put down partly to the theoretical nature of some algorithms (serial parts, communication, etc.), it is obvious that some implementations are suboptimal, e.g. implementations, where the parallel variant is slower than the sequential one.

A problematic design choice is that there is only a single static implementation per algorithm and execution policy. Clearly, for some algorithms, the choice of the optimal implementation depends on the input, particularly on the problem size. Also, the use of advanced techniques, such as parallel scan[18], is not at hand.

During the investigation of the algorithm copy, we have shown the potential influence of NUMA awareness on performance. Our prototype does not provide a clean way to initialize containers under NUMA awareness. One way to do this would be via allocator types, which can be passed as arguments to STL container templates.

5.3 Limitations of this Work

Our prototype is designed to run on single compute nodes only, and the backend of our prototype uses exclusively portable code from the C++ threading library. We did not write code targeted to massively parallel systems or graphics processing units. There are no reasons why implementations of the parallel STL should not be targeted to specific architectures and it would be of interest which of our findings apply to those as well.
Additionally, our case studies cover only a small subset of the pSTL algorithm selection, and the individual investigation of algorithms only a small subset of all possible use cases. In order to learn more about the suitability of the parallel STL in the context of scientific applications, more extensive research is necessary.
Bibliography


Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Parallel STL

Author(s) (in block letters):

For papers written by groups the names of all authors are required.

Name(s):
Sjier

First name(s):
Sandro

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date

Lucerne, 21st January 2016

Signature(s)

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.