International Conference on Computational Science, ICCS 2012

Data parallel skeletons in Java

Herbert Kuchen, Steffen Ernsting

Department of Information Systems
University of Muenster
Leonardo-Campus 3
48149 Muenster, Germany

Abstract

In the past years, multi-core processors and clusters of multi-core processors have emerged to be promising approaches to meet the growing demand for computing performance. They deliver scalable performance, certainly at the costs of tedious and complex parallel programming. Due to a lack of high-level abstractions, developers of parallel applications have to deal with low-level details such as coordinating threads or synchronizing processes. Thus, parallel programming still remains a difficult and error-prone task. In order to shield the programmer from these low-level details, algorithmic skeletons have been proposed. They encapsulate typical parallel programming patterns and have emerged to be an efficient and scalable approach to simplifying the development of parallel applications. In this paper, we present a Java binding of our skeleton library Muesli. We point out strengths and weaknesses of Java with respect to parallel and distributed computing. A matrix multiplication benchmark demonstrates that the Java Generics deliver poor performance, thus the Java implementation is unable to compete with the C++ implementation in terms of performance.

Keywords: algorithmic skeletons, parallel programming, distributed programming, programming environments, message passing, Java for HPC

1. Introduction

In the past decades, the strategy for raising the performance of microprocessors was to push a single core to continuously increased clock rates and to introduce complex and highly power consuming features like out-of-order execution and branch prediction. This led to a power problem because the power required by those microprocessors grew at a faster rate than the frequency and along with it the performance. In order to meet the growing demand of computing performance, chip designers nowadays have turned to multi-core processors. Raw performance increases now come from increasing the number of cores instead of pushing the frequency to its limits. On the one hand, this strategy results in high performance gains while, at the same time, keeping power consumption on a rather constant level. On the other hand, this strategy also leads to a significant growth in complexity of programming multi-core
processors. Today, parallel programmers have to deal with rather low-level concepts like multi-threading and message passing. Due to the lack of high-level abstractions, portable parallel programming can be a very complicated and error-prone task. Developers not only have to deal with to some extent completely different computing platforms and programming approaches, they additionally have to deal with message passing standards such as MPI [1] and multithreading concepts like OpenMP [2] or the Java concurrency API. In order to shield the programmer from low-level details and thus reducing complexity, algorithmic skeletons have been proposed [3]. They encapsulate typical parallel computation and communication patterns, thus hiding low-level details of parallelism from the programmer.

In this paper, we present the skeleton library JMuesli, a Java binding of the data parallel part of our skeleton library Muesli [4, 5]. In order to guarantee platform independence, JMuesli makes only use of pure Java features and does not make use of any native code through the Java Native Interface (JNI). Not least because C/C++ compiles to machine code, which can natively be executed, and Java is interpreted by a virtual machine, Java is commonly known for delivering worse performance than C/C++ does. But with the development of efficient techniques like just-in-time (JIT) compilation and faster (and concurrent) garbage collectors, this position is at least questionable. In this paper, we want to figure out where the strengths and weaknesses of Java are with respect to parallel and distributed computing in comparison to C++.

The remainder of this paper is structured as follows: related work is discussed in section 2, section 3 introduces the C++ skeleton library Muesli, pointing out its concepts and benefits. Section 4 describes in detail how the concepts of Muesli are implemented in Java. Additionally, with a small program for matrix multiplication, we demonstrate how to implement parallel applications using JMuesli. Section 5 provides experimental results containing measured runtimes and speedups of the matrix multiplication benchmark. Finally, section 6 concludes the paper and gives a brief outlook to future work.

2. Related Work

In [3] Cole establishes the algorithmic skeleton approach by introducing an initial set of algorithmic skeletons: fixed degree divide and conquer, iterative combination, cluster, and task queue. He discusses on general implementation details independent of whether the underlying host language is of functional or imperative nature and elaborates on the strengths and weaknesses of the skeletal approach.

Today, there are several approaches to skeleton programming with diverging programming paradigms. Most of which provide skeletons in terms of program libraries for imperative programming languages like C/C++ and Java on top of MPI or other distribution libraries. In [6] González-Vélez and Leyton provide a state of the art overview of algorithmic skeleton frameworks and libraries. They describe different approaches in detail and summarize recent research on the topic of skeletal programming.

According to our approach of providing algorithmic skeletons in terms of C++/Java class libraries, we will briefly discuss recent similar approaches to skeletal programming. However, with Eden [7] and Higher-Order Divide-and-Conquer (HDC) [8] there are notable skeleton approaches based on functional languages like Haskell. Another worth mentioning approach to skeletal programming is the Pisa Parallel Programming Language (P3L) [9], a structured high-level parallel language providing skeleton constructs to coordinate the parallel or sequential execution of C code.

The Edinburgh Skeleton Library (eSkel) [10] was one of the first libraries to implement algorithmic skeletons. As a structured parallel programming library, eSkel offers various skeletal programming constructs in terms of C functions on top of MPI. In its current version eSkel supports the skeletons pipeline, farm, butterfly (divide & conquer), deal (sort of farm), and haloswap and aims to adress the several issues raised by skeletal programming. However, a new version eSkel2 is currently developed in order to address more concepts for more flexibility.

SkeTo [11] is a C++ library on top of MPI. It offers skeletons for manipulating data structures such as lists, trees and matrices. It also provides an optimization mechanism called fusion transformation, which merges two successive
function calls into a single one. But, to the best of our knowledge, SkeTo does not provide explicit communication skeletons.

JaSkel [12] is a Java based skeleton library, which offers the skeletons farm, pipe, and heartbeat in terms of abstract classes. There are sequential, concurrent, and distributed variants of each skeleton. In order to switch variants, the programmer needs to change the signature of the implementing class. In JaSkel, distribution is based on AspectJ [13], an aspect-oriented Java extension.

Lithium [14] and its successor Muskel [15] are Java based skeleton libraries implemented on top of RMI. They provide task and data parallel skeletons by exploiting the so-called macro-data flow technique. To enhance load-balancing, Lithium has been extended to provide some RMI optimization mechanisms. Additionally, Muskel provides mechanisms for load balancing and also offers performance-oriented features, like for instance quality of service (QoS). Recent research addresses skeletal extensibility, autonomic components, and behavioral skeletons.

Calcium [16] and Skandium [17] are Java based skeleton libraries greatly inspired by Lithium and Muskel. Both task and data parallel skeletons are available and fully nestable. Calcium also provides file access and a performance tuning model to identify code that is responsible for weak performance. Whereas Calcium is enabled for distributed computing by making use of the Java grid middleware ProActive, Skandium only supports shared memory systems using Java threads for parallelization.

None of the above mentioned Java skeleton libraries make use of state of the art message passing libraries for distributed computing such as MPJE. Techniques like RMI rely on a client-server model and therefore may introduce scalability problems when it comes to hundreds or thousands of nodes.

3. The Muenster Skeleton Library

The Muenster Skeleton Library Muesli is a C++ template library offering various task and data parallel skeletons. For an efficient support of multi- and many-core computer architectures as well as clusters of both, Muesli makes use of OpenMP [2] and MPI [1]. Additionally, Muesli provides data parallel skeletons for GPU processing using CUDA [18]. Since there is not yet a quality CUDA binding for Java, we will subsequently consider only data parallel skeletons for multi-node, multi-core or many-core computers.

In Muesli, two classes of data parallel skeletons can be distinguished: computation skeletons and communication skeletons. Both computation skeletons such as map, zip, fold, and scan and their variants as well as communications skeletons such as permutePartition, broadcast, and gather (and their variants) are provided by Muesli as member functions of distributed data structures, i.e. classes that represent distributed data structures. At present, there are distributed data structures for arrays, matrices and sparse matrices offered by Muesli, namely:

```
template <typename T> class DistributedArray(...) // one-dimensional
template <typename T> class DistributedMatrix(...) // two-dimensional
template <typename T> class DistributedSparseMatrix(...) // two-dimensional
```

Listing 1: Distributed data structures offered by Muesli.

Whereas the class DistributedArray represents a one-dimensional (fixed) polymorphic distributed array, the classes DistributedMatrix and DistributedSparseMatrix represent two-dimensional (fixed) polymorphic distributed arrays (or matrices). The template parameter T denotes the type of the elements stored by the distributed data structure. By instantiating the parameter T, arbitrary element types can be generated. As the name suggests, distributed data structures are distributed among several MPI processes, each storing a (local) partition of the whole data structure. The distributed data structures deliver a transparent and flexible automated data distribution mechanism. While figure 1 (a) shows the programmer’s perspective of a matrix called global view, (b) shows the block-wise distribution of a matrix into sub-matrices (local partitions), which is referred to as local view. However, programmers are interested in having a global view of the data structures instead of dealing with local partitions. Therefore, the default view is the global view: every skeleton call affects the whole data structure and whenever programmers have to deal with
indices, most of these indices are global indices. The local view is just an internal representation of the whole data structure. But, since in some cases it may be useful for programmers to deal with local indices, there are variants of skeletons using local indices. Also, some communication skeletons require that the programmer be aware of this data distribution scheme. Additionally, there are getters and setters for global and local indices. When calling computation skeletons, distributed data structures are processed as a whole, e.g. the map skeleton applies a given user function to each element of a distributed data structure and replaces the element with its result. The fold skeleton combines all elements of a distributed data structure successively by means of a given associative binary user function. When skeletons involve communication between processes (fold, scan), this is handled within these skeletons and thus is fully transparent to the programmer. The programmer just needs to be aware of the data structure as a whole and not of its decomposition into local partitions when using computation skeletons. In order to avoid inefficiency, there is no other implicit communication, e.g. by accessing elements of remote partitions. For explicit communication, distributed data structures provide various communication skeletons, e.g. for permuting and broadcasting local partitions among processes or for gathering the whole data structure. For the proper use of communication skeletons, the programmer needs to be aware that distributed data structures are decomposed into local partitions, but not of the resulting communication details. Making use of these communication concepts significantly minimizes the risk of deadlocks, thus significantly reducing development time.

A key concept of Muesli is that the argument functions of skeletons can be partial applications rather than just C++ functions. Since a skeleton defines some parallel algorithmic structure, where the details are fixed by an appropriate argument function with a fixed number of arguments, this feature provides skeletons with more expressiveness: with an argument function being a partial application, the details of a skeleton can depend themselves on additional arguments, which are computed at runtime. The auxiliary function curry is used to transform ordinary C++ functions into partial applications. Within this process, a number of arguments are fixed to some values (arguments of the curry function), returning a function requiring less arguments than the original function.

4. Implementation

As already mentioned before, JMuesli is a pure Java implementation of the C++ skeleton library Muesli. As we want to achieve a comparison between Java and C++ in parallel and distributed computing (and skeletal programming), JMuesli entirely relies on the core Java API. We do not exploit operating system or platform specific features using JNI. This way, we also guarantee platform independence and universal portability.

4.1. Parallel and Distributed Computing in Java

Java is a platform independent programming language, which can be used on any machine or combination of different machines that support Java. Additionally, garbage collection and the implicit serialization mechanism help programmers writing reliable code a lot. With the Java concurrency API, Java offers a built-in support for multi-threading. Lots of easy-to-use features like thread pools and thread synchronization make a major contribution to simplifying parallel programming in Java. Last but not least, with the Java Development Kit (JDK) Java provides a huge standard library with lots of useful tools.

As a language designed with networking in mind, Java naturally provides techniques for distributed computing. On the one hand there are APIs for distributed objects and remote procedure calls like RMI and CORBA, on the other
hand there are rather low-level point-to-point communication techniques using the java.net or java.nio sockets, respectively. While the former is known to deliver poor performance, the latter, as said, is a rather low-level package, which involves complex socket programming. With MPJ Express (MPJE)[19] there is a quality implementation of Java bindings for the MPI standard. The system implements thread-safe communication in Java via message passing. At runtime, programmers may choose between pluggable transport devices either based on Java NIO, which is a pure Java implementation, or based on high performance interconnects like Myrinet, which are implemented using the Java Native Interface (JNI). For the benchmark, we used the Java NIO transport device.

4.2. Parallelization

In JMuesli, parallelism is achieved in two steps. For inter-node parallelization, the SPMD programming model delivered by MPI (MPJE) is exploited in a way that each process only stores a (local) partition of a distributed data structure. As pointed out in section 3, in Muesli, a distributed data structure is a simple template class, which provides computation and communication skeletons as well as various auxiliary methods. In order to maintain polymorphism when porting the distributed data structures to Java, we make use of Java Generics.

```java
public class DistributedArray<T> {...}
public class DistributedMatrix<T> {...}
public class DistributedSparseMatrix<T> {...}
```

Listing 2: Distributed data structures offered by JMuesli.

With assistance of a process identifier delivered by MPJE, each process can independently calculate its own partition, which is locally stored by a generic array. Since it is not allowed to straightforward create a generic array in Java, the Java Reflection API is used to overcome this limitation. Listing 3 illustrates generic array creation in Java.

```
// not allowed:
T[] a = new T[mySize];

// allowed:
import java.lang.reflect.Array;

public <T> T[] getGenericArray(Class type, int size) {
    return (T[]) Array.newInstance(type, size);
}

// Call:
T[] a = getGenericArray(Integer.class, mySize);
```

Listing 3: Generic array creation in Java.

For intra-node parallelization, Java thread pools are used to distribute the workload among a number of participating threads. The most important reason of using a thread pool over creating a new thread for each task is thread creation and destruction overhead. Usually skeletal programs contain multiple skeleton calls. Creating and destroying a number of threads for each skeleton call would lead to a significant overhead. In order to reduce this thread creation and destruction overhead to a minimum, a thread pool abstraction layer is introduced by the class ThreadPool. This class is implemented using the Singleton pattern to ensure that only a single thread pool instance is created for a single JMuesli instance. This thread pool instance is shared among all skeleton calls of a single program. However, there is no competition for resources, as skeleton calls are sequential. Only their execution is done in parallel.

Listing 4 demonstrates how the mapInPlace skeleton, which simply applies an argument function to each element of a distributed data structure and replaces the corresponding element with the result of that function call, is implemented using a thread pool. As previously mentioned, a skeleton defines some parallel algorithmic structure where the details are fixed by an appropriate argument function. Since Java is class-based and lacks first-class functions (methods), the mapInPlace skeleton expects a function object (or often called functor) containing a method that represents the argument function. FunctionObj1 indicates, that its argument function expects a single argument, i.e. the corresponding element of the distributed data structure. The generic type parameter T is defined by the corresponding distributed data structure class. It indicates the return and the argument type of the function object. However, implementation details of function objects are considered closely in section 4.3. In line 7 of listing 4, threads are started
by submitting tasks to the thread pool. Here, a task is an instance of an implementation of the `Runnable` interface provided with the JDK. Finally, threads are joined using a `CountDownLatch`. The `await()` method blocks until every thread has called the `countDown()` method.

```java
public void mapInPlace(FunctionObj1<T, T> fo1) {
    int start;
    CountDownLatch cdl = new CountDownLatch(numThreads);
    for (int i = 0; i < numThreads; i++) {
        start = i * nLocal / numThreads;
        threadpool.submit(new MapWorker(start, cdl, elementsPerThread[i], fo1));
    }
    try {
        cdl.await();
    } catch (InterruptedException e) {
        ...
    }
}
```

Listing 4: Parallelizing the `mapInPlace` skeleton with Java threads.

### 4.2.1. Combining MPJ Express and Java Threads

Whereas the `mapInPlace` skeleton introduces a very straightforward parallelization without any interprocess communication, in this section, we demonstrate how interaction of intra-node and inter-node parallelization is handled. Consider the `fold` skeleton, which successively combines all elements of a distributed data structure by means of a given associative binary user function. Listing 5 presents the scheme of the `fold` skeleton. At first, each process folds its local elements in line 6. Similarly to the `mapInPlace` skeleton presented in the previous section, Java threads are used to do this in parallel. Next, the local results have to be distributed among all processes in line 8. This is accomplished by the `allgather` communication routine. Note that for portability reasons, we want to keep the communication interface interchangeable and we therefore do not make use of collective communication routines provided by MPJE. Instead, we have implemented efficient collective communication routines by exclusively making use of simple send and receive methods as well as hypercube communication. However, we did not observe any major performance drawbacks. Finally, each process folds all local results to a single global result. Note that, again, Java threads are used for intra-node parallelization.

```java
public T fold(FunctionObj2<T, T, T> fo2) {
    T localResult;
    T[] globalResults;
    T result;
    localResult = localFold();
    allgather(localResult, globalResults)
    result = foldGlobalResults(globalResults);
    return result;
}
```

Listing 5: Scheme of the `fold` skeleton.

### 4.3. Higher-Order Functions and Partial Applications

Because skeletons take functions as arguments and are functions themselves, they can be called higher-order functions. But, since methods are not first-class in Java, it is not possible to pass them as arguments to other methods. Therefore, skeleton argument functions have to be implemented as classes, which contain a method that actually implements the desired argument function. With the function objects, we define a set of interfaces containing an abstract method `f()`, which has to be implemented by the programmer and can then be called by the skeletons.
Listing 6 exemplarily presents the abstract class FunctionObj1. It provides an interface for a skeleton argument function that expects a single argument. In case of additional arguments being required, JMuesli provides interfaces for skeleton argument functions that expect up to six arguments.

Listing 6 also illustrates how partial applications are implemented. A function object that shall be called with less arguments than expected, has to return another function object with the same functionality and the given arguments fixed. For that purpose, each function object offers a reference to the next higher function object (line 3). The hierarchy is defined by the number of expected arguments. Through this reference, a lower function object can forward a function call to the next higher function object. The method curry in lines 11 to 17 implements this procedure. It creates and returns an instance of the next lower function object, thereby fixing one argument. The returned (lower) function object offers a reference to the current (higher) function object in order to be able to forward calls of method f() (line 14). Note that the argument arg has to be declared final (read-only) so that the anonymous inner class may refer to it. For simplicity reasons, the curry method is only capable of fixing a single argument with a single invocation. This is not a drawback in flexibility, since the same result can be achieved by fixing one argument after another.

```java
class FunctionObject1<T, A1> extends FunctionObj {
    protected FunctionObj2 fo2;

    protected FunctionObj1(FunctionObj2 fo2) {
        this.fo2 = fo2;
    }

    public FunctionObj0<T> curry(final A1 arg) {
        return new FunctionObj0<T>(this) {
            public T f() {
                return (T)fo1.f(arg);
            }
        };
    }

    public abstract T f(A1 arg1);
}
```

Listing 6: Abstract class FunctionObj1 represents a higher order function in Java.

4.4. Serialization

Serialization is an important concept for distributed computing in general and for our skeleton library JMuesli in particular. Whenever a distributed data structure is parameterized with an arbitrary data type, elements have to be serialized before being transferred over network. Fortunately, for a programmer there is no real effort on serializing objects in Java. Java provides the so-called marker interface Serializable. Each class that implements this interface can be serialized. Same holds for every subclass of a class implementing this interface.

4.5. Case Study: Matrix Multiplication

Cannon’s algorithm [20] for matrix multiplication of two \( n \times n \)-matrices assumes the matrices to be split into \( p \) submatrices (local partitions) of size \( m \times m \), where \( p \) denotes the number of processes and \( m = n/\sqrt{\#\text{processes}} \). Initially the submatrices of \( A \) and \( B \) are shifted cyclically in horizontal and vertical direction, respectively. Submatrices of row \( i \) (column \( j \)) are shifted \( i \) (\( j \)) positions to the left (upwards). The core of the algorithm is the repeated local submatrix multiplication followed by a cyclic shift of the rows and columns of \( A \) and \( B \) by one position in horizontal (leftwards) and vertical direction (upwards), respectively. After \( \sqrt{\#\text{processes}} \) steps, each process has calculated a single submatrix of the resulting matrix. Listing 7 presents the implementation of Cannon’s algorithm using computation and communication skeletons offered by JMuesli.

At first, the initial shifting is performed by the communication skeletons rotateRows and rotateCols in lines 21 and 22. When called with a function object as argument, these skeletons calculate the number of positions each
submatrix has to be shifted by applying this function object to the row and column indices of the submatrices, respectively. According to the \textit{Negate} function object, a submatrix of row $i$ (column $j$) is shifted $i$ ($j$) positions to the left (upwards). When called with the argument $-1$ (lines 29, 30), submatrices of row $i$ (column $j$) are shifted one position to the left (upwards). The result matrix $C$ is instantiated in line 24. Next, the scalar product function object \textit{scproduct} (lines 1 to 9) is partially applied to $A$ and $B$. The getter methods \textit{getGlobalLocal} and \textit{getLocalGlobal} (line 5) treat the first argument as a global index and the second argument as a local index and vice versa. In order to fix the first two arguments ($A$ and $B$) of this function object, the auxiliary function \textit{curry} is used twice. The result is a function object that has two arguments fixed and expects another three arguments. When passed to the \textit{mapIndexInPlace} skeleton (line 28) this function object is applied to each element of the result matrix. In addition to the elements of the result matrix, it is provided with the row and column indices of these elements. This way, the local submatrix multiplications are performed.

```java
public DistributedMatrix<Integer> matmult(DistributedMatrix<Integer> A, DistributedMatrix<Integer> B) {
    Negate negate = new Negate();
    Scprod scproduct = new Scprod();
    A.rotateRows(negate);
    B.rotateCols(negate);
    DistributedMatrix<Integer> C = new DistributedMatrix<Integer>(...);
    FunctionObj3<Integer, Integer, Integer, Integer> fo3 = scproduct.curry(A).curry(B);
    for (int i = 0; i < A.getBlocksInRow(); i++) {
        C.mapIndexInPlace(fo3);
        A.rotateRows(-1);
        B.rotateCols(-1);
    }
    return C;
}
```

Listing 7: Implementation of Cannon’s algorithm using JMuesli.

5. Experimental Results

In order to figure out whether the Java implementation JMuesli is able to compete with the C++ implementation of our skeleton library Muesli, we have implemented the above-described matrix multiplication benchmark using data parallel computation and communication skeletons. The benchmarks has been conducted on PALMA, a multi-node, multi-core cluster computer at the University of Muenster. The compute nodes consist of two Intel Xeon Westmere X5650 processors with a total of 12 cores (6 cores each). The JVM used is the IcedTea6 JVM.
Table 1: Execution times (in seconds) for the matrix multiplication benchmark using JMuesli’s (Java) skeletons. Matrices are from size $n \times n$ with $n = 8192$, np and nt denote the number of processes and threads per process, respectively. Speedups (relative to a single core) given in parenthesis.

<table>
<thead>
<tr>
<th>nt/np</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>49134.10 (1.00)</td>
<td>10968.30 (4.48)</td>
<td>2117.75 (23.20)</td>
<td>488.84 (100.51)</td>
</tr>
<tr>
<td>4</td>
<td>14543.08 (3.38)</td>
<td>3505.44 (14.02)</td>
<td>654.27 (75.10)</td>
<td>193.57 (253.83)</td>
</tr>
<tr>
<td>8</td>
<td>7737.47 (6.35)</td>
<td>2016.84 (24.36)</td>
<td>402.29 (122.14)</td>
<td>125.88 (390.32)</td>
</tr>
<tr>
<td>12</td>
<td>5291.54 (9.29)</td>
<td>1410.17 (34.84)</td>
<td>332.42 (147.81)</td>
<td>101.57 (483.74)</td>
</tr>
</tbody>
</table>

Table 2: Execution times (in seconds) for the matrix multiplication benchmark using Muesli’s (C++) skeletons. Matrices are from size $n \times n$ with $n = 8192$, np and nt denote the number of processes and threads per process, respectively. Speedups (relative to a single core) given in parenthesis.

<table>
<thead>
<tr>
<th>nt/np</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8633.22 (1.00)</td>
<td>1975.12 (4.37)</td>
<td>432.44 (19.97)</td>
<td>76.87 (112.31)</td>
</tr>
<tr>
<td>4</td>
<td>2431.89 (3.55)</td>
<td>549.93 (15.70)</td>
<td>112.49 (76.75)</td>
<td>19.45 (443.87)</td>
</tr>
<tr>
<td>8</td>
<td>1189.37 (7.20)</td>
<td>272.87 (31.64)</td>
<td>55.23 (156.31)</td>
<td>10.28 (839.81)</td>
</tr>
<tr>
<td>12</td>
<td>861.05 (10.02)</td>
<td>182.35 (47.34)</td>
<td>36.74 (234.98)</td>
<td>7.41 (1165.08)</td>
</tr>
</tbody>
</table>

Tables 1 and 2 show the JMuesli (Java) and Muesli (C++) results, respectively, for the matrix multiplication benchmark. It was noticeable that the JMuesli implementation with the use of Java Generics faced some dramatic performance and scalability problems. The JMuesli runtimes exceeded the Muesli runtimes by a factor of around 6 to 14. The more processes used, the bigger the gap in performance between the two implementations. As a consequence thereof, the speedups achieved by the Muesli implementation exceeded those achieved by the JMuesli implementation by a factor of approximately 1.5 to 2. This is a huge advantage for the C++ implementation. But what is the reason for this poor performance of the Java implementation? In order to elaborate on this issue, we have implemented an int variant of the distributed matrix.

Table 3 shows the JMuesli results when using an int implementation of the class DistributedMatrix instead of the generic version. In comparison with the generic variant, the int variant provided much better performance. Runtimes decreased by a factor of approximately 3 to 7 and speedups increased by a factor of approximately up to 2. In comparison with the Muesli results, runtimes and speedups of JMuesli and Muesli conformed to each other, being in the same order of magnitude. But there was still a clear advantage for the C++ implementation Muesli, amounting to a factor of two in runtime.

The comparison between the generic and the int implementation revealed a dramatic performance impact of Java Generics. Considering the implementation details of Java Generics closely, this performance impact is not surprising. Whereas C++ templates are translated to specialized representations for each generic type (code specialization), Java Generics are translated to a single representation for all generic types (code sharing). A technique called type erasure maps various instantiations of the generic type onto the single byte code representation by adding type checks and type conversions where required. Note that these type checks and conversions then happen at runtime, introducing a slight overhead whenever a generic declaration is invoked. Furthermore, this technique prevents generic types from being instantiated with primitive data types, which actually means that programmers are required to use wrapper classes. In total, all these drawbacks sum up to a huge overhead introduced by the Java Generics.

6. Conclusion and Future Work

In this paper, we have presented a Java implementation (JMuesli) of the data parallel part of our C++ skeleton library Muesli. It provides an efficient and scalable high-level approach to simplify portable parallel and distributed programming. Due to the exclusive use of pure Java, JMuesli is innately portable across a variety of platforms (that support Java). Parallel applications developed with JMuesli can even be run on heterogeneous grids that make use of
Table 3: Execution times (in seconds) for the matrix multiplication benchmark using an int implementation of JMuesli’s (Java) skeletons. Matrices are from size \( n \times n \) with \( n = 8192 \), np and nt denote the number of processes and threads per process, respectively. Speedups (relative to a single core) given in parenthesis.

<table>
<thead>
<tr>
<th>nt/np</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>14024.75 (1.00)</td>
<td>2999.07 (4.68)</td>
<td>560.92 (25.00)</td>
<td>93.24 (150.41)</td>
</tr>
<tr>
<td>4</td>
<td>3537.14 (3.96)</td>
<td>815.19 (17.20)</td>
<td>141.50 (99.11)</td>
<td>30.91 (453.71)</td>
</tr>
<tr>
<td>8</td>
<td>1770.30 (7.92)</td>
<td>386.07 (36.33)</td>
<td>73.74 (190.19)</td>
<td>20.62 (680.13)</td>
</tr>
<tr>
<td>12</td>
<td>1198.43 (11.70)</td>
<td>260.17 (53.91)</td>
<td>60.54 (231.66)</td>
<td>15.83 (885.93)</td>
</tr>
</tbody>
</table>

different operating systems and hardware architectures. Further, the large JDK, built-in exception handling, garbage collection, and other features provided by the Java platform make parallel and distributed programming with JMuesli very comfortable. Besides all these advantages, the use of pure Java also carries some disadvantages: experimental results showed that the Java Generics are not able to compete with C++ templates in the matter of performance and scalability. In contrast to C++ templates, Generics can not be parameterized with primitive data types. The restriction to wrapper classes and the implementation of Java Generics introduce a huge overhead, which clearly reflected in the results. Even when restricting to primitive data types, the Java implementation is not able to compete (in terms of performance) with the C++ implementation. There is still a clear advantage in performance and scalability for the C++ implementation. Besides providing more data parallel skeletons to raise the level of expressivity, future work could be directed to also implement in Java the task parallel skeletons provided by Muesli. But for any further development of JMuesli, it is essential to find a concept to overcome the performance issues raised by the Java Generics.

References


