Parallelizing with MPI in Java to find the ninth Dedekind Number

Pieter-Jan Hoedt

Abstract

This paper handles improvements to an existing program to calculate Dedekind numbers. The main idea is to parallellize the algorithm, using the message passing interface in order to allow the program to run on a high performance computing cluster. Another, less fundamental idea presented here, is to represent an antichain by means of a bit sequence instead of a classic collection object. This way some operations can be executed much more efficiently, resulting in quicker computation. Both of these main changes are being discussed in terms of improvements and drawbacks. Also a little discussion on using MPI[1] within *Java* in a HPC[2] environment will be included in this paper in order to clarify the methodology.

1 Introduction

In 1897, Richard Dedekind introduced a rapidly growing sequence of natural numbers, which are still known as the Dedekind numbers. The n^{th} Dedekind number counts the number of antichains in the powerset of a set with n elements. This is a rather easy definition[3] for a number that has no closed-form expression and thus can't be calculated in a straight-forward fashion. This gives rise to the so-called Dedekind's problem - finding the n^{th} Dedekind number - which has been solved for $n \leq 8$ [1].

The 8^{th} Dedekind number, $M(8)$, has been found in 1991 for the first time [2], but De Causmaecker and De Wannemacker improved these algorithms by means of the $\mathcal{P}$-coefficients [3]. This allows to calculate $M(n)$ from $\mathcal{A}_{n-2}$, the set of antichains in the powerset of a $(n-2)$-set. Together with recent developments in computer science, this led to a multithreaded algorithm (Algorithm 1 provides a rough description) that allows to calculate $M(8)$ in a bit over 2 hours [4]. To calculate $M(9)$ though, new tricks will be needed, because $|\mathcal{A}_6| = M(6) = 7,828,354$ is quite a small number compared to $|\mathcal{A}_7| = M(7) = 2,414,682,040,998$. It is this latter though that defines the number of antichains that need to be summed over to calculate the 9^{th} Dedekind number.

[1] Message Passing Interface
[2] High Performance Computing
[3] one of several definitions

Algorithm 1 algorithm of De Causmaecker to calculate $M(n)$ in big lines

serial part
find equivalence classes of $\mathcal{A}(n-2)$
for all equivalence classes of $\mathcal{A}(n-2)$ **do**
    calculate |[$\emptyset$, representative antichain]|
end for

parallel part
for all antichains in $\mathcal{A}(n-2)$ **do**
    calculate partial sum p_i
end for
result = $\sum_i p_i$ ▷ incrementally in **for all** loop

For this paper, the program of De Causmaecker - based on this algorithm - was taken as a starting point to work towards a *Java* program that's capable of finding $M(8)$ faster and bring $M(9)$ within reach. The first thing to come is the replacement of the original representation to the bit sequence represenation, as presented by Salaets [5] and the consequences of this change. Next there's a little discussion on using MPI in *Java* and the *Java*-specific pitfalls that should be avoided. This will allow to understand the results obtained by translating the originally multithreaded approach to a program using open-mpi[4] in order to allow the program to be run by the VSC[5]. At last, the improvements made will be put into perspective by discussing how far the ninth Dedekind number is still out of reach.

2 Representing Antichains

An antichain in a powerset of a set is a subset of that powerset such that for every 2 sets A and B in the antichain, $A \not\subset B$ and $B \not\subset A$. Because of the definition this paper starts from, a more efficient representation of these antichains does have an impact on the speed of the algorithm. This section discusses the slight improvements that can be obtained by using the representation presented by Salaets [5] and possible drawbacks from its implementation in *Java*.

2.1 The original Representation

In the original (*Java*) implementation of De Causmaecker the `AntiChain` class extends the `TreeSet` class and is used to represent an antichain. To be more correct, a set of sets is being represented, as there are no constraints that limit the representation to antichains. When used correctly, this lack of limitation doesn't cause any problems though. The `AntiChain` class provides several optimized methods to perform the operations that are needed for the algorithm as

[4] an open source implementation of MPI - `http://www.open-mpi.org/`
[5] flemish supercomputer centrum - `https://vscentrum.be`

described by De Causmaecker and De Wannemacker [3]. Next to these methods, there is also functionality to map each antichain to a unique `BigInteger` object. This `BigInteger` can be used to represent an antichain when no operations need to be performed.

Some other classes also make use of antichains. The most important one, considering the bit sequence representation, is the `AntiChainInterval` class. This class represents an interval of antichains, which is nothing more than a set of antichains. Although this class doesn't store the antichains it contains explicitely, it does turn out to be useful to access all antichains from an interval. This happens through iterators, where a special example is the `fastIterator`. The latter is used to iterate over all antichains - from $\emptyset$ to the $\{\{1, 2, \ldots, n\}\}$ - when calculating the n^{th} number (see Algorithm 1).

2.2 The new Representation

Salaets presented in his master thesis the idea to represent antichains by means of a bit sequence [5]. Compared to a classic `Collections` object, this should make memory usage more efficient and increase performance.

The representation uses the fact that every basic integer set is already represented by a bit sequence and thus can be interpreted as a number (e.g. $\{1, 3\}$ is represented by 101 which can be interpreted as the number 5 or $\{2\}$ is represented by 010 which is on its turn 2). To represent an antichain, which is a set of such basic integer sets, we can use these number-interpretations as indices for the element just like the integers are used as an index in the bit sequence (e.g.: $\{\{1, 3\}, \{2\}\}$ can be represented by 10010). This representation thus limits the number of bits by 2^n for a n-set and increases speed of standard set operations (e.g.: union, intersection, ...) as well as the encoding/decoding to/from `BigInteger` representation. For operations on a lattice of antichains (e.g.: join, meet, ...) though, this representation doesn't provide improving features. Because these latter are the more important ones in the algorithm, the performance gain is rather modest, but for the bigger numbers, these changes might make a difference.

Next to improving the speed of operations on antichains, the bit sequences allow to iterate over an interval by means of monotonic Boolean functions, which do have interesting properties in this representation. To go from one antichain to the next, it is possible to map an antichain to one antimonotonic function[6]. From this function, the next subset to be added can easily be found by taking that one with the smallest number interpretation that's not included yet. When this antimonotonic function is mapped back to an antichain, this is the next antichain.

2.3 Implementation

For the implementation of the bit sequence in *Java*, the `BitSet` class has been used from the `java.util` library. This is about the only way *Java* offers to

[6] A definition used by De Causmaecker in earlier publications [6]

represent bit sequences that can grow longer than a `long`, but it offers all operations needed as well as a way to iterate over all elements (basic sets) of an antichain.

The `AntiChain` class has thus been converted to a `SmallAntiChain` class that contains a `BitSet` and all operations of this new class have been altered accordingly. The `AntiChainInterval` class has gotten a new `fastIterator` that uses the iteration as described earlier and in more detail by Salaets [5].

2.4 Remarks

The advantages concerning memory usage and performance from using this new representation as stated by Salaets, should not be taken for granted. Programming with bit sequence in *OpenCL* is much more efficient than doing this in a higher programming language like *Java*. After all the `BitSet` class has a lot of extra fields, making this representation not as memory efficient as in *OpenCL*. Next to this language-issue, an antichain is quite sparse compared to an antimonotonic function - which provided the actual base for this representation. When looking at the memory usage[7], the `TreeSet` seems to need less space in general. Nevertheless the bit sequence representation seems to be compact and fast enough.

2.5 Improvements

The improvements are not of the kind they'll allow to calculate the 9^{th} Dedekind number, but they are noteworthy nevertheless. In order to compare the original representation with the new one, each n^{th} number ($2 \leq n \leq 7$) was calculated 1000 times. The final results are the medians of all running times in order not to let outliers (startup, pagefaults, ...) influence the results. This has been done for two implementations. One with the `TreeSet` implementation and another using the bit sequence representation. The results of this little experiment can be found in table 1. It should be clear that the new representation causes an improvement in performance and therefore is worth keeping.

	$M(2)$	$M(3)$	$M(4)$	$M(5)$	$M(6)$	$M(7)$
original	1	1	1	3	17	1532
bit sequence	1	1	1	2	15	1140

Table 1: Running times in milliseconds on a 2.2 GHz Intel Core i7

3 MPI in Java

MPI is one of the most well-known and standardized message-passing systems around and has been integrated well in languages as *C*, *C++* or *Fortran*, but

[7]The serialization method has been used to approximate memory usage.

for *Java* the documentation and possibilities are limited. Therefore this section tries to clarify some issues that should be taken into account when using MPI in *Java*.

First of all, it is important to know that there are no libraries that provide the functionality of MPI within *Java*. The only way to get MPI working is to look for some library, providing the necessary MPI-operations. Several libraries are around for *C*, *Fortran*, ..., but the support for *Java* is limited to wrappers of these libraries. One of the first and best-known wrappers was provided by *mpiJava*, which simply made use of the Java Native Interface to invoke the MPI-operations (provided by *mpich*) in *C* [7]. The problem with **mpiJava** though, is that the last release dates back to 2003 and only supports functionality of MPI 1.1[8]. Next to this, HPC environments rarely support *mpiJava* and at the VSC, not even *mpich* is supported. Because the VSC only provided intel-mpi[9] and open-mpi, and only open mpi provided *Java*-bindings - based on *mpiJava*, open-mpi was used to implement the MPI-routines.

Because almost every implementation of MPI for *Java* consists of wrappers, MPI doesn't work any different for *Java* as compared to *C* for example and this results in some, probably unwanted effects caused by the way *Java* works. Programs using MPI are normally run with a command that looks like `mpirun -n X <program>` where the `-n` flag indicates the number of processes. This command would start the specified program `X` times distributed over the available processing units, which are processor cores by default. For *Java*, the open-mpi command is `mpirun -n X java <java-program>` and starts `X` JVMs[10] on every core. This results in the unwanted behaviour, because most cores share memory and every JVM needs a piece of this memory, limiting the memory available to each JVM.

To solve these inconveniences with the JVM, a *Java* program can use MPI in a combination with multithreading, which leads to a so-called hybrid MPI application. In order to do this efficiently, every multi-core node using one piece of memory can run one multithreaded program. This way, all cores can be used and memory isn't polluted with plural JVMs. To run a hybrid program, MPI provides some extra flags. The command to start exactly one hybrid *Java* program on each of `X` multi-core nodes could look like `mpirun -n X --map-by ppr:1:node --bind-to board java <hybrid-java-program>`.

4 Effects of Parallelization

Because of unawareness concerning the details described in section 3, two implementations arose throughout research. The first implementation is a naive mapping from the multithreaded version of the program to a MPI implementation, not worrying about memory usage etc. A second, hybrid implementation provides a more intelligent approach and tries as well to minimize the MPI over-

[8] in 2012, MPI 3.0 was released

[9] a commercial MPI implementation from Intel

[10] Java Virtual Machine

head. Both implementations will then be compared and their performance will be discussed.

4.1 A naive implementation

The first program is a quite literal translation of the multithreaded version. One process distributes the work, while the others actually work. This allows the work to be dynamically distributed over all available cores, because the distributor process assigns the next chunk of work to the first worker that's ready. This approach allows to use more cores and thus lets the algorithm work faster, but it does suffer from quite some message passing overhead. On top of that, this way of working leads to the memory leak described in the previous section and even requires the *Java* flag `-Xmx1g` to limit memory usage to only 1 gigabyte. If this flag isn't used, the first JVM will crash when trying to enlarge its heap. This memory limitation also leads to more garbage collection etc. It should be clear that this is not the most favorable approach in terms of memory usage.

4.2 A 'smarter' implementation

The second implementation tries to solve the memory leak by dividing the work statically. Instead of letting a lazy master process divide the work, every piece of work gets a number and each of the nodes knows which numbers of work it should execute. This has been done in such a way that work is divided equally and no message passing need to be done to divide the work. Every process then divides its jobs over its available cores by means of multithreading. The final results are then sent to one single process where everything is combined serially to the final result. This way a lot of overhead from MPI disappears and memory is used ways more efficient. Next to these advantages, this approach allows to do some work of the serial part of the algorithm (see Algorithm 1) in a multithreaded fashion. This becomes interesting, because in the end it is this part that turns out to be the time-consuming one.

4.3 A note on the serial part

As mentioned earlier, the algorithm consists of a serial part and a parallel part. The former part finds equivalence classes and calculates sizes of intervals which are both needed to allow efficient calculations in the parallel part. This necessary serial part leads to an issue though on a parallel HPC environment, because every node needs the results. Two ways to achieve this are (1) letting every node calculate this serial part for itself or (2) let one node calculate and broadcast it to the other nodes. The second way introduces extra broadcasting costs compared to the first, but this seems justified if the serial part needs a lot of processing time and the number of nodes doesn't get too large. Because time on the HPC environment of the VSC is limited, only a few test runs have been made, but they do seem to support the justifications. A first test indicated a

200ms advantage for (1) when calculating $M(7)$ on 5 nodes and even a 300ms advantage when calculated on 20 nodes, but calculating $M(8)$ on 20 nodes results in a 3s advantage for (2). Therefore the choice was made to let only one node calculate and broacast the results to the other nodes.

4.4 Comparing results

In order to compare both the naive and 'smarter' implementation, the respective programs have been executed several times to calculate $M(7)$ and $M(8)$, but each time the number of nodes provided to the program was altered. Because experiments on the VSC are rather expensive, each possible combination ran only once. This causes the following results to be indications rather than proving, but they can be used to compare the serial as well as the parallel part of the algorithm.

The serial part has two main tasks as already indicated by Algorithm 1: finding equivalence classes and calculating interval sizes. For the naive approach this needs to be done serially by one core because the processing units are cores. The 'smart' implementation on the other hand allows to calculate the serial part on a processing unit with mutltiple cores. This advantage has been used to calculate the interval sizes in a multithreaded fashion.

		equivalence classes	interval sizes
$M(7)$	naive	0.62	1.12
	'smart'	0.27	0.19
$M(8)$	naive	60.50	280.68
	'smart'	53.91	15.75

Table 2: Averages of running times of serial part of both implementations on the HPC environment from the VSC in seconds

Taking the average running time over all node-configurations from the experiment, table 2 gathers the results for this part. The impact of having multiple cores available becomes clear from these results. Whether multithreading is used or not, the 'smart' implementation always outperforms the naive one when it comes to the serial part.

For the parallel part, the running time as such isn't sufficient to decide which implementation is better. It is also important to know in how far the running times improve when more processing units are available. Table 3 shows the running times meassured from the moment the results from the serial part are broadcasted to the other nodes untill the end.

In this table something strange seems to be going on. For $M(7)$ the 'smart' implementation outperforms the naive one, but for $M(8)$ roles have changed. This can be explained by the overhead caused by the message passing as it is has been avoided in the 'smart' implementation. Whereas this overhead is practically the running time for $M(7)$, it becomes negligable when calculating $M(8)$.

# nodes		10	15	20	25	30	35	40
$M(7)$	naive	6.00	9.02	12.09	15.06	18.01	20.99	23.97
	'smart'	0.77	0.66	0.66	0.57	0.60	0.64	0.59
$M(8)$	naive	848.04	570.76	433.39	351.23	297.70	261.47	233.72
	'smart'	919.58	606.17	460.37	360.44	311.05	267.20	239.04

Table 3: Running times of multithreaded part of both implementations on the HPC environment from the VSC in seconds

Why the 'smart' approach turns out to be slower than the naive one with all its message passing overhead, can be linked to the statical job division. This omits the need for message passing, but the way the work has been divided doesn't seem good enough to challenge the naive implementation with its overhead. At last it should be clear from the table that the 'smart' implementation does provide a better speed-up. To calculate $M(8)$ with 40 nodes, both implementations perform equally well although there is a difference of more than a minute when working with only 10 nodes.

To give an idea of how fast the 8^{th} Dedekind number can be calculated now, table 4 presents the total running times from the experiment together with an extra calculation on 50 nodes.

	10	15	20	25	30	35	40	50
naive	19:55	15:12	12:58	11:34	10:36	9:59	9:34	08:35
'smart'	16:32	11:15	8:51	07:10	06:20	05:36	05:07	04:24

Table 4: Running times of both implementations for $M(8)$ on the HPC environment from the VSC in minutes:seconds

5 The ninth Dedekind Number

This paper described a successful integration of the bit sequence representation as presented by Salaets as well as a quite efficient implementation for running the algorithm in a HPC environment. Especially this last alteration allows to calculate $M(8)$ orders of magnitude faster than before, but several factors still slow down computation. The most important ones are the serial part, which still needs over one minute, and the non-optimal, static division of work.

This program might be ready to calculate the 9^{th} Dedekind number, but no attempt to test this has been undertaken. The main reasons therefore are the limited credits provided for calculation on the VSC and the time consumption of finding the equivalence classes. An attempt to calculate these on a 2.2 GHz Intel Core i7 processor needed to be interrupted after 3 hours. This is unfortunately too long for the few credits available for this project.

At last it is worth mentioning De Causmaecker introduced a new formula

during this research:

$$|\mathcal{A}_{n+3}| = \sum_{\alpha,\beta,\gamma \leq \rho \in \mathcal{A}_n} |[\bot, \alpha \wedge \beta \wedge \gamma]| \cdot |[\alpha \vee \beta, \rho]| \cdot |[\gamma \vee \beta, \rho]| \cdot |[\alpha \vee \gamma, \rho]|$$

. It provides a way to calculate $M(n)$ by means of $\mathcal{A}_{n-3}$. This could reduce the work significantly, but has a lot more variables in the sum, resulting in higher complexity. No valuable contribution has been made in this paper, but this is definitely a formula worth mentioning for future work on solving Dedekind's problem.

References

[1] Neil James Alexander Sloane et al. The on-line encyclopedia of integer sequences. `http://oeis.org/A000372`, 2002. Dedekind numbers.

[2] Doug Wiedemann. A computation of the eighth dedekind number. *Order*, 8(1):5–6, 1991.

[3] Patrick De Causmaecker and Stefan De Wannemacker. On the number of antichains of sets in a finite universe. *arXiv preprint arXiv:1407.4288*, 2014.

[4] Patrick De Causmaecker. Codes reports. `http://www.kuleuven-kulak.be/nl/onderzoek/Wetenschappen/Informatica/codesreports`, 2014. Runnable jar of original algorithm.

[5] Carl Salaets. Efficiënte berekening van dedekindgetallen. Master's thesis, KU Leuven, 2014.

[6] Patrick De Causmaecker and Stefan De Wannemacker. Partitioning in the space of antimonotonic functions. *arXiv preprint arXiv:1103.2877*, 2011.

[7] Mark Baker, Bryan Carpenter, Geoffrey Fox, Sung Hoon Ko, and Sang Lim. mpijava: An object-oriented java interface to mpi. In *Parallel and Distributed Processing*, pages 748–762. Springer, 1999.