A Perfomance Study of Parallel Cauchy Reed/Solomon Coding

Peter Sobe and Peter Schumann
Faculty of Informatics/Mathematics
University of Applied Sciences Dresden, Germany
Contact: sobe@htw-dresden.de

Abstract—**Cauchy-Reed/Solomon coding is applied to tolerate failures of memories and data storage devices in computer systems. In order to obtain a high data access bandwidth, the calculations for coding must be fast and it is required to utilize parallelism. For a software-based system, the most promising approach is data parallelism which can be easily implemented with OpenMP on a multicore or multiprocessor computer. A beneficial aspect is the clear mathematical nature of coding operations that supports functional parallelism as well. We report on a storage system application that generates the encoder and decoder as C-code automatically from a parametric description of the system and inserts OpenMP directives in the code automatically.**
We compare the performance in terms of achieved data throughput for data parallelism and for functional parallelism that is generated using OpenMP.

I. Introduction

Nowadays, a computer typically owns a number of processor cores that can be utilizes to accelerate computation-intense applications by exploiting parallelism. Such an application is data en- and decoding which is a necessary operation for fault-tolerant memory and storage systems. Every block of data that is stored is involved in calculations that produce a high computational load. This computations normally slow down data access, but this can be mitigated by processing the data for coding in parallel. Such coding algorithms allow data-parallel calculations on independent blocks of data. A closer look on the en- and decoding algorithms reveals that calculations can be separated from another and functional parallelism can be applied as well. The question is whether data parallelism, function parallelism or a combination of both is the best choice.

The contribution of the paper is a performance evaluation of a failure-tolerant coding application that exploits the potentials of data parallelism and of functional parallelism. These different approaches to parallelism are implemented using OpenMP [1], a multiprocessing library with compiler support. Due to the clear mathematical concept of coding, the algorithms can be generated automatically with specification of code parameters. In addition, the OpenMP parallelism extensions are included automatically.

The rest of the paper is organized as follows. The technical background and related work are described in Section II, particularly the Cauchy-Reed/Solomon code that is selected for the coding algorithm. The method of automatic code generation including to organize the workload in different functions and to insert OpenMP directives is described in Section III. In Section IV we report on the achieved acceleration by parallelism and draw conclusions for further optimizations.

II. Background and Related Work

In the first part of this section the coding algorithm is described, together with the structure of data that is distributed accross the system. A second part is dedicated to the OpenMP approach to parallel programming and runtime support. In a third part, a selection of related work is addressed.

A. The application: Cauchy Reed/Solomon coding

This work is based on the Reed/Solomon code [2], particularly on the XOR-based variant introduced in [3] that is denoted by Cauchy-Reed/Solomon code (CRS). CRS is a so called erasure-tolerating code that allows to recalculate parts of the original data that got lost (or got erased) as result of hardware failures or unreachability of data in networks. In practice, CRS can be applied as well for correcting failures. For this, it is combined with error detecting codes that validate blocks as correct or corrupted, and CRS replaces the corrupted blocks by recalculated content.

The CRS coding works as follows: For encoding, data must be split in k parts that have to be assigned independent storage devices (or memory modules). A number of m additional blocks are calculated from the k original blocks. The m blocks are redundancy and used solely for decoding to recalculate lost blocks. The redundant m blocks are placed on additional storage devices.

The CRS code shows several beneficial properties.

- It is a so called regular code that separates original data and redundant data. This property allows to read the data without execution of decoding calculations in failure-free situations. When data is written (or changed) the encoding calculation have to be executed always.
- The code is optimal w.r.t. the amount of redundant data and the number of failures that can be tolerated. With m redundant blocks, every situation with up to m lost blocks among the original and redundant ones can be tolerated.
- The code uses XOR operations that are available as machine instructions. In addition, XOR is included in the MMX and SSE instruction set extensions for parallel computations on wide registers (128 Byte) of x86 processors.

- A specific coding and decoding algorithm can be generated for every combination of k and m ($k > 0, m > 0$). CRS can be applied for every distribution factor (k) and every number of additional blocks (m), where the latter parameter scales number of failures that can be tolerated.

The coding algorithm handles every block as split in ω fragments. From $k \times \omega$ fragments taken from the original data, $m \times \omega$ redundant fragments are calculated. CRS constrains the value of ω by the relation $2^{\omega} > k + m$. The original and redundant data fragments are distributed block-wise across the devices (memory modules, storage devices, computers). The data layout is depicted in Fig. 1 for the example of a $k = 3, m = 2, \omega = 3$ code. In this case, every block is split in three fragments that are differentiated for the encoding and decoding calculations.

The calculation can be seen as a linear equation system $A \cdot o = r$, with o as a vector of original data fragments and r as the redundant fragments. The coding matrix A consists of elements $\in \{0, 1\}$ and is constructed under mathematical constraints in order to allow a decoding using the inverse matrix. The elements of A are factors and normally would require a multiplication with the data fragments. Due to the factors 0 and 1, solely a selection takes place whether a fragment is included in the equation or not. Every equation is reduced to a sum of selected fragments that is expressed by bitwise XOR-ing the fragments. The XOR operation is expressed by the symbol $\oplus$. The equations for encoding are shown in (1) with fragments that are numbered by $u0, u1, \ldots, u8$. The two redundant blocks contain the fragments $u9, u10 \ldots, u14$.

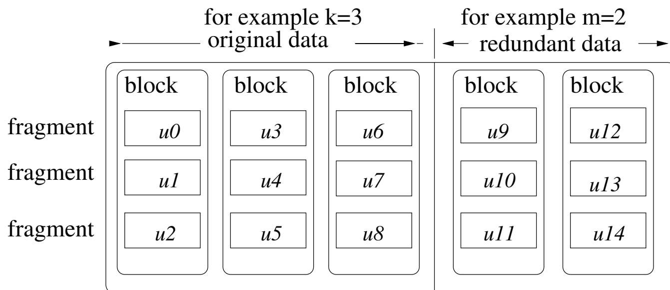

Fig. 1. Data layout for a system that distributes data across 3 devices and adds 2 redundant devices to store redundant data.

$$\begin{aligned} u9 &= u1 \oplus u2 \oplus u3 \oplus u4 \oplus u6 \ , \\ u10 &= u0 \oplus u1 \oplus u5 \oplus u7 \ , \\ u11 &= u0 \oplus u1 \oplus u2 \oplus u3 \oplus u8, \\ u12 &= u0 \oplus u1 \oplus u2 \oplus u5 \oplus u6 \oplus u8 \ , \\ u13 &= u0 \oplus u3 \oplus u5 \oplus u6 \oplus u7 \oplus u8 \ , \\ u14 &= u0 \oplus u1 \oplus u4 \oplus u7 \oplus u8 \end{aligned} \tag{1}$$

The equations can be optimized by elimination of common subexpressions. As a result, encoding can be expressed by equations that do not contain redundant calculations, but show data dependencies among them. The t-fragments are temporary and have to be calculated before being used in other equations. We denote this coding as an iterative style (see (2)), and the non-optimized variant as the direct style (see (1)) .

$$\begin{aligned} &t15 = u1 \oplus u2 \ , \quad t16 = u3 \oplus u6 \ , \\ &t17 = u0 \oplus u1 \ , \quad t18 = u5 \oplus u7 \ , \\ &t19 = u0 \oplus u8 \ , \quad t20 = t15 \oplus t19 \ , \\ \\ &u9 = u4 \oplus t15 \oplus t16 \ , \quad u10 = t17 \oplus t18 \ , \\ &u11 = u3 \oplus t20 \ , \quad u12 = u5 \oplus u6 \oplus t20 \ , \\ &u13 = t16 \oplus t18 \oplus t19 \ , \quad u14 = u4 \oplus u7 \oplus u8 \oplus t17 \end{aligned} \tag{2}$$

The principle of generating XOR-equations for en- and decoding is described in detail in [4]. A tool, called *cauchyrs* [5] is applied to prepare XOR equations from the code parameters. It creates XOR-based equations according to a given number of data storage resources (k) and redundancy storage resources (m). Additional parameters for the equations are the block length, as well as, whether the tool should generate equations in a direct style or in an iterative style. The equations are generated independently from the actual data storage operation (Write, read, update) in the storage or memory system.

As a preliminary analysis, coding equations allow to assess the cost of coding by the number of XOR operations in relation to the number of original blocks that have to be written. The number of XOR operations directly follows from the equations that were generated by the cauchyrs tool. The more XOR operations, the more compute-intense the coding is. This normally would cause reduced data access rates with a negative influence on the write rate and with an influence on the read rate, in case of failures. The result show that the variation of the distribution factor k has only a little impact. However, increasing the failure tolerance by the number of redundant blocks m influences the costs noticeably as can be seen in Fig. 2. Seen roughly, encoding requires $m+1$ XOR operations per block for every single block that is written. For instance, when 1kByte is written and m is 2, 384 XOR operations have to be executed that combine two 8 Byte operands (1024 Byte/ 8 Byte $\times$ 3). In addition to the 384 instructions, 3072 Bytes must be moved through the processor for calculation. As a result of this high effort, only a few hundred MByte/s remain as data rate for sequential encoding, even when the storage system offers higher data access rates.

B. OpenMP for Multicore

OpenMP [1] is a multi processing library with compiler support that was first published in 1997 and is now available for all major compilers (C,C++, Fortran). OpenMP is supported for instance by the Gnu gcc compiler and MS Visual Studio. OpenMP leads to a multithreaded program execution and is suitable for shared memory multiprocessing platforms. The typical way to use OpenMP is to add a few compiler directives and library calls to the original sequential code. These so called pragma directives give hints to create thread pools in the way of fork-join-parallelism with a master thread and a number of worker threads. It allows to gradually add parallelism to a sequential program without changing the program marginally.

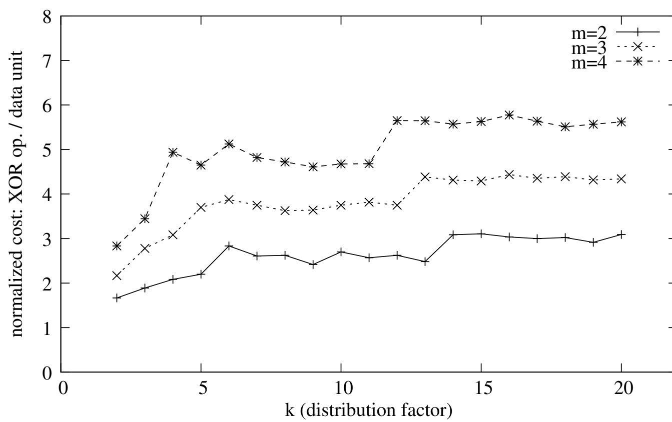

Fig. 2. Analysis of computational cost: number of XOR operations per data unit that is stored.

A common directive is *#pragma omp parallel* that marks a block to be executed by several threads. It creates a thread pool where every thread executes the program. The programmer distributes the workload among threads, particularly the work that is done within loops. A typical form is the combination of a parallel block and a for-loop (*#pragma omp parallel for*) that is illustrated in the C-program in listing 3. In a sequential program, the for-loops are the origin of data parallelism.

Another way to express parallelism are sections of different code blocks as illustrated in listing 4. These sections are used to express function parallelism, where threads execute different parts of the program code (i.e. functions).

```
#include <omp.h>
#define N_VALUES 10000

main()
{
  double a[N_VALUES], b[N_VALUES], d[N_VALUES];
  double c=3.14;
  int i, nt=8;

  omp_set_num_threads(nt); // set number of threads

#pragma omp parallel for
  for (i=0;i<N_VALUES;i++)
    d[i] = a[i]+c*b[i];
}
```

Fig. 3. *OpenMP* example of data parallelism

```
void calc (in , out ) {
# pragma omp parallel sections {
# pragma omp section {
    f1 ( in , out );
  }
# pragma omp section {
    f2 ( in , out );
  }
 .....
}}
}
```

Fig. 4. *OpenMP* example of parallel sections for function parallelism

Besides sections, explicit tasks (*#pragma omp task*) are another way to assign code blocks and functions to threads. This can be used to implement function parallelism as well.

OpenMP allows to add clauses in order to control, whether variables are shared among threads or have to be thread-private ones. Serialization and synchronization of threads is supported by additional directives, such as *#pragma omp critical* or *#pragma omp single*.

C. Related Work

For long time, research is directed to fast data en- and decoding for failure-tolerating systems, especially for compute-intense codes that flexible tolerate multiple failures. Around ten years ago, there was the need for specific hardware accelerated solutions, mostly implemented by FPGAs, e.g. [6]. Besides RAID controllers with a function-specific hardware, it is common to implement coding by software and using the GPU or multicore capabilities. The concept of multicore Reed/Solomon coding is described in [7]. An introduction of the translation concept to OpenMP and OpenCL is published in [8] that applies the concept of equations to multicore and to GPU-architectures. A purely GPU-directed work can be found in [9], [10].

Another way to implement software-based erasure-tolerant codes is to use flexible libraries, such as the jerasure library [11], a C/C++ library for matrix-based erasure-tolerant coding. At the time of publication (2007) this library contained sequential code. Meanwhile, several sources report on usage of this library in the context of GPU acceleration and multithreading.

A way between implementing the coding algorithms by oneself and the application of ready-to-use libraries is to develop the own functionality on base of code skeletons. These skeletons are pre-defined, reusable code components that can be applied to several algorithms and encapsulate parallelism. The most common example is Map-Reduce, but also other patterns of parallelism can be included in skeletons, such as a farm (master worker) or pipelined execution. An example for a skeleton programming library is [12] that supports parallel programming models like OpenMP, OpenCL and CUDA.

Even though it would be possible, in this work we do neither utilize a library nor build on skeletons. The parallel coding is implemented by pure C code that is automatically generated.

III. Equations, Program Code and OpenMP

In this section, we explain how the coding algorithm is transformed from an symbolic description (using equations) to C program functions, including the control statements for OpenMP parallelism.

Initially, data encoding and decoding is expressed by equations that have to be applied to data. For encoding, the equations do not change over time. Thus, encoding equations can be translated to C functions and included in the system before operations start.

Decoding equations change with the specific failure situation. Thus, these equations are generated on on demand and JIT compilation techniques have to be applied to produce a specifically optimized decoder for a failure situation. For small

system configurations, it is possible to calculate all sets of decoding equations in advance and to provide C functions for all failure cases that can be tolerated.

We follow the approach of decoupling the equation generation and the coding operations that are related to data. Fig. 5 shows the system components that are involved into en- and decoding. The cauchyrs tool generates equations, as well as C source code. The C functions are later used in the storage system to provide write and read operations with application data.

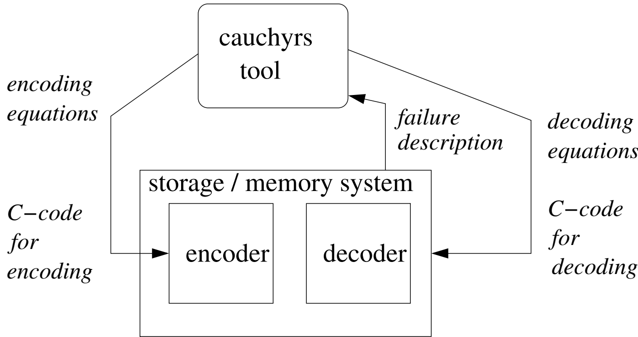

Fig. 5. Automatic C-code generation for the system that executes the coding algorithm.

From the internal representation of the coding equations, C functions can be generated that take an array of bytes as the first input parameter and an array of calculated redundancy bytes as the second parameter. Fig. 6 shows an encoding function, according to the previously given example with $k = 3$, $m = 2$ and $\omega = 3$ as parameters. The unit numbers can be recognized as the macro parameters of Of() and Rf(). The redundant units $u9, \ldots, u14$ are expressed by the array r, with macro parameters 0 to 5.

Typically, data that is written is longer than a single code word. Thus, fragments contain a number of Bytes (for instance 1024, or 2048), denoted by $blocklen$. A complete input data set covers $k \times \omega \times blocklen$ Bytes and is taken to produce $m \times k \times blocklen$ Bytes redundancy. The coding is repeated $blocklen$ times, which is expressed by a for-loop. This for-loop is the block that is distributed across several threads in the way of data parallelism. The inner block of the for-loop does not contain data dependencies to other iterations and therefore it can be easily handled by OpenMP. The directive *omp parallel for* precedes the for-statement. This instructs the compiler to introduce functionality that distributes the work across the threads in the thread pool. Special care must be taken for the local variables that store the values of the common subexpressions. These variables must be declared as thread-local and have to be assigned to every thread as private variables (see the *private* clause in Fig. 6)

The generated code for functional parallelism can be seen in Fig. 7 for a $k = 3$, $m = 2$, $\omega = 3$ system. The result of the computation is the same as of the data-parallel variant (see Fig. 6).

Depending on the number of cores (which is a parameter of the automatic code generation), a number of functions are created to cover different equations. The calculation of

```
#define UNIT_LEN 1024
#define Rf(a) a*UNIT_LEN+i
#define Of(a) a*UNIT_LEN+i

inline void encode(const char*, char*);

void encode(const char *n, char *r)
{
  int i;
  char t15,t16,t17,t18,t19,t20;

#pragma omp parallel for \
private (t15,t16,t17,t18,t19,t20)
  for (i = 0; i < UNIT_LEN; i++)
  {
    t15 = n[Of(1)]^ n[Of(2)];
    t16 = n[Of(3)]^ n[Of(6)];
    t17 = n[Of(0)]^ n[Of(1)];
    t18 = n[Of(5)]^ n[Of(7)];
    t19 = n[Of(0)]^ n[Of(8)];
    t20 = t15^ t19;

    r[Rf(0)] = n[Of(4)]^ t15^ t16;
    r[Rf(1)] = t17^ t18;
    r[Rf(2)] = n[Of(3)]^ t20;
    r[Rf(3)] = n[Of(5)]^ n[Of(6)]^ t20;
    r[Rf(4)] = t16^ t18^ t19;
    r[Rf(5)] = n[Of(4)]^ n[Of(7)]^ n[Of(8)]^ t17;
  }
}
```

Fig. 6. Automatically generated C-code that supports data parallelism

the equations within a function follows a direct coding style without data parallelism.

Equations are assigned to the individual coding functions (calc1, calc2, ...) as follows: First, equations are sorted according to the number of XOR operations. A number of calculation functions is created that are initially empty. Starting with the equation with the most XOR operations, the equations are assigned to the calculation functions, translated to C statements and placed in the function bodies. As long as there are empty coding functions, this is a round-robin assignment. When all calc-functions contain at least one equation, the next equation is assigned to the coding function with the least number of XOR operations.

In the example given in Fig. 7 it is not possible to generate an optimal balanced distribution. The 6 equations are distributed across 4 functions in the best way as possible, but with a slight imbalance. This effect disappears with larger system configurations that require more equations (due to higher values of m and/or ω).

IV. Performance Evaluation

The performance of the encoding operation was studied on two different systems, a 4-core AMD-Phenom II-X4, 3.2 GHz system and a 2× Opteron 6128, 2GHz (2×4 cores) system. The second system is slightly slower clocked, but offers the double number of processor cores.
We measured the data throughput in terms of the amount of original data (MByte) that was encoded during a fixed quantity of time (seconds). Fig. 8 shows data throughput under a varied number of threads on a 4-core system. All measurements show a clear advantage of the iterative encoder over the direct

```c
void calc0 (const char *n, char *r)
{
  for (int i = 0; i < UNIT_LEN; i++)
  {
     r[Rf(3)] = n[Of(0)]^ n[Of(1)]^ n[Of(2)]^
                n[Of(5)]^ n[Of(6)]^ n[Of(8)];
  }
}
void calc1 (const char *n, char *r)
{
  for (int i = 0; i < UNIT_LEN; i++)
  {
     r[Rf(4)] = n[Of(0)]^ n[Of(3)]^ n[Of(5)]^
                n[Of(6)]^ n[Of(7)]^ n[Of(8)];
  }
}
void calc2 (const char *n, char *r)
{
  for (int i = 0; i < UNIT_LEN; i++)
  {
     r[Rf(0)] = n[Of(1)]^ n[Of(2)]^ n[Of(3)]^
                n[Of(4)]^ n[Of(6)];
     r[Rf(5)] = n[Of(0)]^ n[Of(1)]^ n[Of(4)]^
                n[Of(7)]^ n[Of(8)];
  }
}
void calc3 (const char *n, char *r)
{
  for (int i = 0; i < UNIT_LEN; i++)
  {
     r[Rf(2)] = n[Of(0)]^ n[Of(1)]^ n[Of(2)]^
                n[Of(3)]^ n[Of(8)];
     r[Rf(1)] = n[Of(0)]^ n[Of(1)]^ n[Of(5)]^
                n[Of(7)];
  }
}

void calc(const char *n, char *r)
{
#pragma omp parallel sections
  {
     #pragma omp section
     {
       calc0(n,r);
     }
     #pragma omp section
     {
       calc1(n,r);
     }
     #pragma omp section
     {
       calc2(n,r);
     }
     #pragma omp section
     {
       calc3(n,r);
     }
   }
}
```

Fig. 7. Automatically generated C-code with function parallelism

encoder. The variants of data parallelism scale with the number of threads used for encoding until the number of processor cores is reached. Unfortunately, the function-parallel encoder did not produce a higher throughput and is ranked on the level of direct data-parallel encoding. This can be explained by the same number of XOR operations as the direct encoder.

Surprisingly, measurements on smaller system configurations showed an up to three times better throughput of the function-parallel encoder in comparison to the best data-parallel variant. This effect motivated a deeper analysis of the generated assembler code through an inspection using objdump. We found that the gcc compiler generated SSE instructions for XOR operations on consecutive Bytes for the function-parallel encoder version, but only for small configurations (e.g. $k = 5$, $m = 2$) The compiler creates single Byte accesses for longer blocks and for higher values of k and m in the function-parallel version. The reason for this disadvantageous choice are runtime checks that have to be added for SSE acceleration that check that data in the arrays n and r do not overlap. We assume a compiler strategy that stops SSE vectorization when these runtime checks become too costly. A small hint to the compiler by declaring the pointers n and r as *__restrict__* allowed to use SSE vectorization for longer blocks and bigger system configurations.

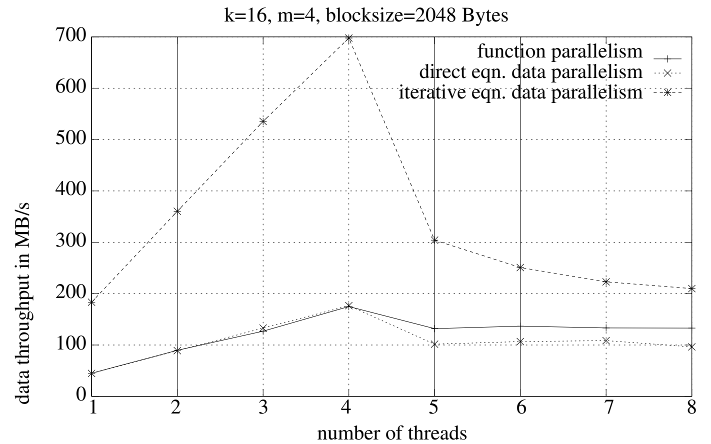

Fig. 8. Data throughput on a 4-core system

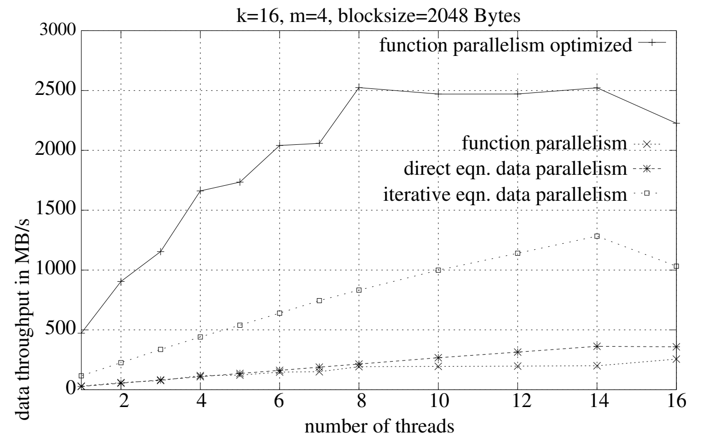

Fig. 9. Data throughput on a 8-core system, including the optimized version for SSE vectorization

After optimizing the function-parallel encoding function by restricted pointer parameters, the measurements were repeated on a 8-core system (see Fig. 9). Besides the two data-parallel coding functions that scale well up to 8 cores, the function-parallel coding variant showed a beneficial performance over the data-parallel versions.

At the current status, the performance analysis revealed that without optimizations the function-parallel does not perform better than the data parallel variant. In the best case, one can achieve a near optimal distribution of the computational load that is comparable to the best possible load distribution of data-parallel coding. Regardless, it is beneficial to provide variants for function-parallel coding when there is a chance for compiler-based optimizations. We could show that gcc generates faster code from the function-parallel variant, compared to the data-parallel variant.

Further investigations are directed to use iterative equations for the functional parallelism as well and to distribute the equations in a way that common subexpressions within the functions are taken into account for the distribution of equations. When there are more cores than equations, a combination of function parallelism and data parallelism should offer additional room for performance improvements.

V. Summary

Coding for failure-tolerant storage can be significantly accelerated by multithreading on multicore computer systems. We demonstrated the automated program code generation that includes the placement of OpenMP directives for data-parallel processing. Function-parallel processing is possible as well, but requires a slightly more code-modifying technique to assign operations to threads. With the current optimizations, the function-parallel variant reaches the best performance of 2.5 GByte/s on 8 cores.

Data en- and decoding is an example of applications that base on relatively complex mathematical principles but show a simple and regular code structure. In such a case, automatic generation of program code including the control statements for parallel execution is a feasible technique, compared to flexible and high-optimized code libraries.

References

[1] "OpenMP Application Program Interface, Version 3.1," 2011, The OpenMP Architecture Review Board. [Online]. Available: http://www.openmp.org/mp-documents/OpenMP3.1.pdf

[2] I. S. Reed and G. Solomon, "Polynomial Codes Over Certain Finite Fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, p. 300, 1960.

[3] J. Bloemer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman, "An XOR-Based Erasure-Resilient Coding Scheme," International Computer Science Institute, Technical Report TR-95-048, Aug. 1995.

[4] P. Sobe and K. Peter, "Flexible Parameterization of XOR based Codes for Distributed Storage," in *2008 Seventh IEEE International Symposium on Network Computing and Applications*, Cambridge, MA, USA, Jul. 2008, pp. 101–110. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4579645

[5] P. Sobe, "cauchyrs Documentation," University of Luebeck, Institute of Computer Engineering, Tech. Rep., Sep. 2009.

[6] A. Wiebalck, P. Breuer, V. Lindenstruth, and T. Steinbeck, "Fault-Tolerant Distributed Mass Storage for LHC Computing," in *CCGrid 2003*, 2003.

[7] P. Sobe, "Parallel Reed/Solomon Coding on Multicore Processors," in *International Workshop on Storage Network Architecture and Parallel I/Os*. IEEE Computer Society, 2010, pp. 71–80.

[8] P. Sobe, "Parallel Coding for Storage Systems - An OpenMP and OpenCL-capable Framework," in *PASA- Workshop, GI Proceedings on ARCS (Workshops)*, 2012.

[9] M. L. Curry, A. Skejellum, H. L. Ward, and R. Brightwell, "Acellerating Reed-Solomon Coding in RAID Systems with GPUs," in *Proceedings of the 22nd IEEE Int. Parallel and Distributed Processing Symposium*. IEEE Computer Society, 2008.

[10] M. L. Curry, H. L. Ward, A. Skjellum, and R. Brightwell, "A lightweight, gpu-based software raid system," *2012 41st International Conference on Parallel Processing*, vol. 0, pp. 565–572, 2010.

[11] J. S. Plank, "Jerasure: A Library in C/C++ Fasciliating Erasure Coding to Storage Applications," University of Tennessee, Tech. Rep. CS-07-603, September 2007.

[12] U. Dastgeer, J. Enmyren, and C. Kessler, "Auto-tuning SkePU: A Multi-Backend Skeleton Programming Framework for Multi GPU Systems," in *Proceedings of the 4th International Workshop on Multicore Software Engineering*. ACM, 2011.