The eighth edition of the Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG-2015) took place in Amsterdam, the Netherlands, on January 20th, 2015. The workshop was co-located with the HiPEAC 2015 conference. MULTIPROG aims to bring together researchers interested in programming models, runtimes, and computer architecture. The workshop is intended for quick publication of early results, work-in-progress, etc., and is not intended to prevent later publication of extended papers. This year we received a total of 13 submissions. The authors’ origin was mainly from Europe (29 authors), but we also had contributions from the U.S. (9 authors) and Asia (8 authors). Each submission was reviewed by three members from our Program Committee. The organizing committee selected seven papers based on the overall merit of each contribution. In the case of conflicts of interest, affected committee members did not participate in the discussion.

In addition to the seven papers, the workshop included three invited talks:

- Prof. Georgi Gaydadjiev from Maxeler Technologies gave MULTIPROG’s opening keynote: “Everything You Always Wanted to Know About Dataflow Engines Virtualization”

- Prof. Dimitrios Nikolopoulos from Queen’s University of Belfast gave an invited talk: “Evaluating Servers using Iso-Metrics: Power, QoS and Programmability Implications”

- Dr. Mauricio Alvarez from the Technical University of Berlin gave an invited talk: “Mapping Video Codecs to Heterogeneous Architectures”

We have assembled the seven accepted papers into these informal proceedings. We hope that you will enjoy the quality of the contributions. The 2015 edition of MULTIPROG was well attended and generated lively discussions among the participants. We hope these proceedings will encourage you to submit your research to the next edition of the workshop.
Organizing Committee:
- Ferad Zyulkyarov, Barcelona Supercomputing Center, Spain
- Oscar Palomar, Barcelona Supercomputing Center, Spain
- Vassilis Papaefstathiou, Chalmers University of Technology, Sweden
- Miquel Pericàs, Chalmers University of Technology, Sweden

Steering Committee:
- Eduard Ayguade, UPC/BSC, Spain
- Benedict R. Gaster, Qualcomm, USA
- Lee Howes, Qualcomm, USA
- Per Stenström, Chalmers University of Technology, Sweden
- Osman Unsal, Barcelona Supercomputing Center, Sweden

Program Committee:
- Mats Brorsson, KTH Royal Institute of Technology, Sweden
- Pascal Felber, University of Neuchatel, Switzerland
- Roberto Giorgi, University of Siena, Italy
- Daniel Goodman, Oracle Labs, UK
- Hakan Grahn, Blekinge Institute of Technology, Sweden
- Ali Jannesari, RWTH Aachen University, Germany
- Paul Kelly, Imperial College of London, UK
- Mikel Lujan, University of Manchester, UK
- Vladimir Marjanovic, High Performance Computing Center Stuttgart, Germany
- Tim Mattson, Intel, USA
- Simon McIntosh-Smith, University of Bristol, UK
- Avi Mendelson, Technion, Israel
- Dimitris Nikolopoulos, Queen’s University of Belfast, UK
- Andy Pimentel, University of Amsterdam, The Netherlands
- Oscar Plata, University of Malaga, Spain
- Yanos Sazeides, University of Cyprus, Cyprus
- Ruben Titos, Barcelona Supercomputing Center, Spain
- Dongping Zhang, AMD, USA

External Reviewers:
- Adria Armejach, Barcelona Supercomputing Center, Spain
- Madhavan Manivannan, Chalmers University of Technology, Sweden
- Anita Sobe, Université de Neuchâtel, Switzerland
- Nehir Sonmez, Barcelona Supercomputing Center, Spain
- Alejandro Villegas, University of Malaga, Spain
- Gulay Yalcin, Barcelona Supercomputing Center, Spain
- Foivos S. Zakkak, Foundation of Research and Technology Hellas, Greece
Index of selected Papers:

*Analysis of the overheads incurred due to speculation in a task based programming model.* Rahulkumar Gayatri, Rosa M. Badia and Eduard Ayguade

*Memory Link compression to speedup scientific workloads.* Chloe Alverti, Georgios Goumas, Konstantinos Nikas, Angelos Arelakis, Nectarios Koziris and Per Stenstrom

*On Parallel Evaluation of Matrix-Based Dynamic Programming Algorithms.* David Bednárek, Michal Brabec and Martin Kruliš

*Tangram: a High-level Language for Performance Portable Code Synthesis.* Li-Wen Chang, Abdul Dakkak, Christopher I. Rodrigues and Wen-Mei Hwu

*Task Parallelization as a Service: A Runtime System for Automatic Shared Task Distribution.* Luc Bläser


*Towards a scalable functional simulator for the Adapteva Epiphany architecture.* Ola Jeppsson and Sally A. McKee
Analysis of the overheads incurred due to speculation in a task based programming model.

Rahulkumar Gayatri§ †, Rosa M. Badia§, Eduard Ayguade § †
§ Barcelona Supercomputing Center † Universitat Politècnica de Catalunya,
♭ Artificial Intelligence Research Institute (IIIA), Spanish National Research Council (CSIC), Spain
{rgayatri, rosa.m.badia, eduard.ayguade}@bsc.es

Abstract. In order to efficiently utilize the ever increasing processing power of multi-cores, a programmer must extract as much parallelism as possible from a given application. However with every such attempt there is an associated overhead of its implementation. A parallelization technique is beneficial only if its respective overhead is less than the performance gains realized. In this paper we analyze the overhead of one such endeavor where, in SMPs, speculation is used to execute tasks ahead in time. Speculation is used to overcome the synchronization pragmas in SMPs which block the generation of work and lead to the underutilization of the available resources. TinySTM, a Software Transactional Memory library is used to maintain correctness in case of mis-speculation. In this paper, we analyze the affect of TinySTM on a set of SMP applications which employ speculation to improve the performance. We show that for the chosen set of benchmarks, no performance gains are achieved if the application spends more than 1% of its execution time in TinySTM.

1 Introduction

Currently eight-core chips are highly prevalent in the market and this trend is on a rise. To effectively use such hardware processing power an application should be as parallel as possible. But parallel programming is laborious due to the lack of standardization and portability issues. SMPs [1] is a task-based programming model which allows a programmer to write high-level parallel code for Symmetric Multiprocessors (SMPs). It consists of a source-to-source compiler and a runtime that comprises of a main-thread and a pool of worker-threads. The compiler processes the directives which are used to annotate parts of a sequential code as individual units of computation (tasks). The main-thread builds a task dependency graph (TDG) based on the data-flow between the tasks and independent nodes from this graph are scheduled to the worker-threads. The separation of the generation and the execution of a task makes it necessary to use the synchronization directives such as barrier and wait-on to maintain control between the main-thread and the worker-threads. The use of such directives, however block the main-thread until the synchronization-predicate has been evaluated.
which leads to underutilization of the available resources. Such inabilities to effectively use the available resources become more prominent as the number of cores on a chip keep increasing. To overcome this problem, the SMPSs framework provides a 

speculate

directive which avoids the synchronization pragma and generates tasks speculatively. The speculatively generated (speculative) tasks possess a transaction-like property, where their updates are committed or undone based on the success of the speculation.

To achieve the commit or abort type of behavior for the speculative tasks, a lightweight STM [5] library, TinySTM [4] is integrated into the SMPSs framework. SMPSs executes a speculative task as a transaction and commits its results only if the speculation has been successful, else the transaction is aborted. Speculation increases the parallelism that can be extracted from a given application. But its impact on the overall execution time of an application is also dependent on the overhead associated with its implementation. We measure the time spent in the TinySTM library to evaluate the overhead incurred due to speculation in SMPSs. For this we use Paraver [3], a performance analysis tool. We add paraver specific events to measure the time spent in the TinySTM library calls. We compare the execution times of the speculative and non-speculative versions of a set of benchmarks chosen from the domain of linear algebra and graph algorithms. We analyze the overhead of executing TinySTM library calls in the applications and their effects on the overall performance. We also present the behavior of the speculative versions of the benchmarks in the presence of weak and strong scaling. The analysis provides us with an upper-bound for the acceptable overhead with speculation. The contributions of this paper are:

– An empirical proof to show that the benefits of speculation improve with higher number of threads.
– An experimental demonstration that shows the negative effects of unoptimized number of calls to the TinySTM library.

The rest of the paper is organized as follows: in section 2 we discuss the SMPSs framework and the need for speculation in SMPSs and its implementation. We also explain the 

speculate

dpragma and its usage. In section 3 we present our evaluation on the various aspects that effect the efficiency of speculation. In section 5 we present our conclusions.

2 Speculation in SMPSs

The SMPSs framework provides compiler directives (pragmas) to annotate code blocks in a sequential program that can potentially be run in parallel. The annotated code blocks are treated as units of computation, tasks. The pragmas are processed by the SMPSs source-to-source compiler and the transformed code is passed to the SMPSs runtime. The SMPSs main thread builds a task-dependency graph (TDG) to honor the data-dependencies between the tasks. Independent
tasks from this graph are scheduled to the various execution resources. The separation of task generation and its execution makes it impossible for the main-thread to make any assumptions about its completion. Hence in-order to maintain control between the main-thread the framework provides synchronization pragmas such as:

```
#pragma css wait on(a)
#pragma css barrier
```

Listing 1.1: Synchronization pragmas in SMPSs

The first pragma in listing 1.1 halts the main thread until the last task that updates \( a \) has finished execution whereas the second one halts the main thread until all previously generated tasks have finished execution.

Synchronization pragmas limit the parallelism of a given application as they block the main thread from task generation. In a while-loop, if the loop-predicate is updated in a task inside the loop, a synchronization pragma is used at the end of the loop iteration.

```
while (residual > threshold)
{
    for (int i=0 ; i < x_blocks-1 ; i++)
        for (int j = 0 ; j < y_blocks ; j++)
            jacobi_task(i,j,parameters[i*np+j*block_y], residual);
    #pragma css wait on (residual)
}
```

Listing 1.2: while loop in the Jacobi application.

Listing 1.2 shows an example where a `wait` pragma in a `while-loop` does not allow the generation of tasks from more than one iteration at a time. The synchronization is necessary since the tasks that evaluate the predicates could potentially be executed on different threads. To overcome this drawback, a `speculate` directive has been introduced to the SMPSs framework and is used as follows:

```
#pragma css speculate values(a,b) wait(x)
```

Listing 1.3: Speculate Pragma

- `values` - tasks that update parameters \( a,b \) should be protected in case the speculation fails.
- `wait` - indicates the predicate that decides the commit or abort of the speculative tasks.

If the `speculate` pragma is inserted before a programming construct, then the tasks inside these constructs will be generated speculatively. In this context we define “invalid tasks” as tasks where the speculation fails and hence their updates should be undone, i.e., such tasks should be rolled back. Currently the `speculate` pragma can be used with the following programming constructs: `while-loop` and `if-condition`. Figure 1 shows the usage of the `speculate pragma`. The speculative
tasks apart from being control dependent on the predicate of the loop may also be data dependent on the earlier tasks. Hence the use of the *speculate* pragma may affect the TDG of the application in one of the following ways:

1) Tasks which are control-dependent on the earlier tasks.
2) Tasks which are data-dependent on the earlier tasks.

In case 1, simultaneous execution of speculative tasks with earlier (speculative and non-speculative) tasks is possible. In case 2, the SMPSs main thread adds dependencies between speculative tasks and earlier tasks. This only allows the overlap of speculative task generation with task execution. We evaluate and analyze the performance of the applications and the behavior of the SMPSs and TinySTM runtimes when the *speculate* pragma is used.

### 3 Performance Analysis.

The performance analysis was done on 4 different applications namely:

1) Lee-routing - generates control-dependent tasks when *speculate* directive is used.
2) Gauss-Seidel and Jacobi - iterative solvers which generate speculative tasks which are data-dependent on the earlier tasks.
3) Kmeans - data-clustering algorithm which also generates data-dependent speculative tasks.

The above mentioned applications are executed on an IBM dx360 M4 node. It contains 2x E5-2670 SandyBridge-EP 2.6GHz cache 20MB 8-cores. Thread affinity was controlled by assigning one thread to each core. The applications which generate data-dependent speculative tasks were executed with three different problem sizes which are shown in table 1.
We concentrate more on applications where the *speculate* pragma generates tasks that are data-dependent on earlier tasks, since they occur with more frequency in programming. The use of an external library effects the performance of SMPSs since it now requires to execute function calls outside of its framework. Hence we focus our investigation of STM-based speculation on the following points:

- We compare the performances of speculative and non-speculative versions of the applications.
- Effect of varying the task granularities on the performance of applications.
- Relative time spent in the TinySTM library compared to the total execution time.
- Overhead of invalid tasks.

We study the case of lee-routing application separately since this is the only application that adds *speculative* tasks which are independent of the tasks generated earlier. Figure 2 shows the performance comparison of the phase of Lee-routing where the tasks are blocked due to a synchronization pragma that enforces a control dependency.

![Fig. 2: Lee Routing.](image)

The parallelism extracted from the simultaneous execution of the speculative tasks and the tasks generated earlier is evident from the gain in the performance. In this phase, the non-speculative version does not improve the performance after 4 threads. Instead the performance worsens due to the extra resources that are unnecessarily used and the overhead of the SMPSs framework wasted.

<table>
<thead>
<tr>
<th>Problem-sizes</th>
<th>Gauss-Seidel</th>
<th>Kmeans</th>
<th>Jacobi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small</td>
<td>2048 unknowns</td>
<td>100 thousand</td>
<td>2048 unknowns</td>
</tr>
<tr>
<td>Medium</td>
<td>4096 unknowns</td>
<td>500 thousand</td>
<td>4096 unknowns</td>
</tr>
<tr>
<td>Large</td>
<td>8192 unknowns</td>
<td>1 million</td>
<td>8192 unknowns</td>
</tr>
</tbody>
</table>

Table 1: Problem sizes of the benchmarks chosen.
when there is no parallelism. The speculative version, however steadily improves in the performance timings until 8 threads. Even after that when scaled, the performance does not degrade but remains similar without much improvement either.

**Performance Comparisons**
Figures 3, 4 and 5 show that the speculative version of the Gauss-Seidel, Jacobi and Kmeans never perform better compared to their non-speculative counterparts. Irrespective of the number of threads used and the problem size, the regular SMPSs versions consistently performs better. The TinySTM library calls
that are used to maintain correctness is an overhead absent in the regular SMPSs application. The execution of transaction calls within every *speculative* task degrades the overall performance of the application. Figures 3, 4, and 5 show that although the speculative version scales similar to its non-speculative counterpart, it never benefits in the absolute performance. The increase in the number of tasks generated does not provide enough boost to the execution timings of the applications in order to overshadow the TinySTM overheads. However, with an increase in the number of threads the absolute difference in the performances between the speculative and non-speculative version reduces. With higher number of threads, more resources are available to avoid the parallelism extracted from the *speculate* clause. This is a positive result, since it proves that the idea is scalable. And if the overhead is reduced, performance gains can be achieved by the idea of speculation.

**Varying task granularities**

The task granularity, i.e., the size of the memory block that is updated by a single task is chosen as 512KB of data. The choice of task granularity plays an important factor in the performance of an application. Smaller task granularities will lead to an increase in the overhead of task creation and destruction, whereas larger tasks will not be able to make the best use of the concurrency available in the application. The results shown in the previous section used the task granularities that gave the best performance with the non-speculative version of the applications. We now show the effect of modifying task granularities on the application performances. The effects of higher task granularities on the TDG of an application and correspondingly on the calls made to the TinySTM library when the *speculate* clause is used are:

1) Decrease in the number of *speculative* tasks and consequently the number of transactions generated for a given problem size.
2) Increase in the size of the transactional copy performed by a single *speculative* task.

Figures 6, 7, and 8 show the changes in the performances of the Jacobi, Gauss-Seidel and Kmeans with different task granularities. The legend in the figures shows the task granularity. The performance of the non-speculative versions of the application with their optimal task granularities is shown for comparison. In all the three applications for all the three problem sizes, the best performance of the speculative version is achieved with the task-granularity of 2MB. This is the largest among the granularities chosen. Bigger task-granularities decrease the number of tasks generated but increase the amount of data processed within a single task. In case of *speculative* tasks, this leads to a decrease in the number of transactions created and destroyed but an increase in the data processed within a single load/store operation. This leads us to the conclusion that, with TinySTM the overhead of generating more number of smaller transactions is higher than creating less number of bigger transactions. The conclusion is valid since we compare the performance with the optimal task granularity of the non-speculative version of the applications. Although with larger transactions more time is spent
in rollbacks when the speculation fails, the granularities of the speculative tasks should be higher than the regular SMPSs tasks. This is in direct conflict with the purpose of speculation, i.e., to increase the parallelism from a given application, which in the case of SMPSs translates to increasing the number of tasks in the TDG for the worker-threads to choose from.

**Time spent in TinySTM**

The use of TinySTM to maintain correctness of a speculative task, hampers the overall performance of the application. Hence to inspect the effect of TinySTM, we analyze the time spent in different phases of a transaction during the execution of a speculative task. In this section, we present the relative time spent by the benchmarks in TinySTM while executing their speculative versions. Our
aim is to obtain an upper bound for the acceptable overhead associated with TinySTM based implementation of speculation. The minimum performance that can be achieved in speculation is the overlap of task generation with task execution. Hence finding an acceptable overhead for applications with cross-iteration dependencies can be safely generalized to all cases.

To calculate the amount of time spent in the TinySTM library during the execution of a speculative task, we trace the execution of the following TinySTM library calls:

- Start a transaction (stm_start).
- Load/store memory blocks into the transactional context (stm_load_bytes/stm_store_bytes).
- Abort the transaction in case of mis-speculation (stm_abort).
- Commit the transaction (stm_commit).

We use paraver [3] to perform this analysis and visually analyze the characteristics of the TinySTM library in SMPSs. Trace-events have been added to the SMPSs framework, which track the execution of the above mentioned TinySTM library calls. These events contain the information that represent the cumulative amount of time spent in the respective library calls. Paraver later analyzes these events to deduce the information in a readable visual format.

Figure 9 presents the relative time spent in the TinySTM library while executing speculative versions of the applications. The y-axis shows the relative time spent in TinySTM compared to the total execution time of the application. The legend in the figure represents the problem sizes of the application. The task-granularity chosen gives the best performance results for the speculative versions of the applications. An increase in the number of resources increases the number of speculative tasks scheduled in parallel (which also include invalid tasks). This will decrease the time taken to execute an application which will consequently increase the relative time spent in TinySTM. Hence longer histograms can mean one or both of the following:

- An increase in the number of invalid tasks due to increase in the number of threads and/or problem sizes of the application.
- Faster execution of the applications, which increases the relative time spent in TinySTM.

All three applications spend different amounts of time in the TinySTM library. The pattern of relative time is similar in Jacobi and Gauss-Seidel, but the amount of time is different. The difference in the iterative solvers allows Gauss-Seidel to converge faster but generates more tasks in every iteration of Jacobi. This implies that the time spent in TinySTM by Gauss-Seidel is influenced by its faster convergence whereas in the case of Jacobi, higher number of invalid tasks is more dominant. Since Jacobi spends more time in TinySTM, it implies that the presence of invalid tasks has more impact on the performance of the speculative versions of the applications. In the results presented the least overhead occurs when the Kmeans application is executed with smaller data size. Even here the
performance of speculative version is less than the regular SMPSs version. This shows that no performance benefits are gained when the overhead of TinySTM is more than 1% of the total execution time of the application.

An STM library tracks the memory locations that are accessed and updated inside a transaction. If a parallel transaction updates the same memory location then a conflict is detected and one of the transactions is aborted. The data analysis performed by the TinySTM library is unnecessary in the case of SMPSs due to the presence of a TDG for every application. This makes the use of TinySTM or any other STM library to implement speculation unnecessary since its major feature will be redundant. If a regular data-version based implementation is used instead of an STM based one, we obtain performance benefits from the idea of speculation as shown in figure 10. A detailed description about this implementation is presented in [10].

Invalid tasks
One of the major overheads in speculative task execution is the rollback of invalid tasks. Figure 11 shows the relative time spent by applications in rollbacks, which is shown in the y-axis of the figure. In the figure we observe that with increasing number of threads the relative time spent in aborts increases. Increasing the
number of threads is advantageous for speculative execution since it provides more resources to benefit from the increased parallelism. This implies that the applications will execute faster which will increase the relative time spent in rollback. Figure 11 also shows an increase in the problem size also increases the relative time in abort. Increase in the number of tasks generated in every iteration consequently increases the number of invalid tasks in case of mis-speculation. Hence the time spent in rollbacks increase with increasing problem size and the number of threads. The differences in the Gauss-Seidel and Jacobi algorithms can be observed in the abort times too. Jacobi spends more time in aborting invalid tasks compared to Gauss-Seidel. Compared to Jacobi and Gauss-Seidel, the relative time spent in aborting invalid tasks is higher with increasing threads in Kmeans. A comparison with the performance of Kmeans shows that it has least amount of absolute difference with its non-speculative counterpart. It faster execution has increased the relative time spent in aborting invalid tasks.

### 4 Related Work

Task-wait is a synchronization directive used from OpenMP 3.1 [11] which is similar to wait of the SMPSSs. It blocks the progress of the current task until all its children tasks have completed its execution. The idea of speculative loop execution is a widely researched topic, where different techniques have been proposed like [13], [14], [15]. But we believe that ours is the first work which implements and analyzes speculative task execution in a programming model and uses STM to maintain correctness.

### 5 Conclusion

To overcome the problems arising from the use of synchronization directives, SMPSS-framework provides a `speculate` pragma. When used, this pragma generates and executes tasks ahead in time. To maintain correctness, speculative tasks are executed as transactions using the TinySTM library calls. To gain performance benefits from this idea the overhead incurred due to TinySTM should be
less than the performance gains achieved by speculation. In this paper we compare the performances of speculative and non-speculative versions of a chosen set of benchmarks and the effect of TinySTM in SMPs. We conclude that in loops with loop-carried dependencies, no performance benefits can be achieved if the overhead of TinySTM is more than 1% of the total execution time.

References

1. Josep M. Perez Rosa M. Badia Jesus Labarta “Handling task dependencies under strided and aliased references” Proceeding ICS ’10 Proceedings of the 24th ACM International Conference on Supercomputing
3. “Parallel Program Visualization and Analysis Tool” http://www.bsc.es/media/1364.pdf
4. Pascal Felber, Christof Fetzer, Torvald Riegel “Dynamic Performance Tuning of Word-Based Software Transactional Memory” PPoPP’08
9. Dragojević, Aleksandar and Guerraoui, Rachid and Kapalka, Michal “Stretching Transactional Memory”

12
Memory link compression to speedup scientific workloads

Chloe Alverti1, Georgios Goumas2, Konstantinos Nikas2, Angelos Arelakis1, Nectarios Koziris2, and Per Stenström1

1 Chalmers University of Technology, Sweden
2 National Technical University of Athens, Greece

Abstract. Limited off-chip memory bandwidth poses a significant challenge for today’s multicore systems. Link compression provides an effective solution, however its efficacy varies due to the diversity of transferred data. This work focuses on scientific applications that typically transfer huge amounts of floating-point data, which are notoriously difficult to compress. We explore the potential of BFPC, a recently proposed software compression algorithm, but by modeling it for hardware compression and comparing it to three state-of-the-art schemes. We find that it can reduce floating-point data traffic by up to 40%, which for memory bound executions translates to significant performance improvement.

1 Introduction

Chip Multi-Processor systems (CMPs) have become the dominant design paradigm, utilizing the still increasing number of available transistors on-chip. However, delivering the peak performance of multiple cores to the application level is by no means a straightforward task. One of the main problems is the unsustainable off-chip memory bandwidth. Memory technology is not able to keep pace with the advancements in CPU technology and the increasing demand for memory access posed by multiple cores. Thus, memory bandwidth has emerged as a critical performance bottleneck.

This paper focuses on hardware link compression, an effective approach to save off-chip memory bandwidth by transferring data over the memory link in a compressed form. Research on the field has been rich and illuminating, highlighting important issues that need to be considered, such as the complexity in terms of compression/decompression latencies and hardware resources or their effectiveness (i.e. compression ratio) on diverse data types and application domains. Several novel lightweight compression algorithms that target a balance between compression efficiency and speed [1, 2, 3, 4] have been proposed. It has been shown that transferring compressed data over the link between on-chip cores and their last-level caches, and off-chip main memory (see Figure 1) has high potential especially for integer, multimedia and commercial workloads [1].

Performance improvement in state-of-the-art link compression schemes is sought mainly on single processor architectures. Alameldeen [4] has evaluated a
link compression scheme for CMP systems by using the significance-based Frequent Pattern Compression (FPC) algorithm as a complementary technique to the adaptive cache compression scheme [5]. Interestingly, most of the previous schemes focus on integer and commercial workloads, while they appear to have poor performance on scientific workloads and floating-point (FP) datasets.

The goal of this study is to evaluate the impact of link compression on the performance of scientific memory-bound applications running on modern CMP systems. We consider a simple compression scheme (similar to [1]), assuming uncompressed cache and main memory. Cache blocks are compressed before they are transferred over the link and decompressed after their reception, applying on the fly (de)compression. We initially evaluate three state-of-the-art algorithms proposed for cache [6] and link compression [2, 3]. Motivated by the poor performance of these schemes in floating-point datasets, we explore Martin Burtscher’s FPC [7] algorithm (BFPC), a software compression scheme for double precision floating-point data which we adjust for hardware compression. To the best of our knowledge, BFPC has not been applied in a hardware compression scheme before.

Our experimental evaluation leads to several interesting observations regarding the compressibility of the workloads and the sensitivity of each scheme to the (de)compression overheads. The main contributions of this paper are:

- A detailed study of link compression as a technique to increase the effective bandwidth and improve the performance of modern CMP systems, particularly for multi-threaded scientific workloads. Our results indicate that effective compression can lead to significant speedup, especially when memory bandwidth bound applications are considered.
- An initial evaluation of the BFPC algorithm adapted to our hardware link compression scheme. Our results indicate that BFPC reduces off-chip memory traffic up to 40% on hard-to-compress scientific floating-point datasets.

The rest of the paper is organized as follows: Section 2 provides information on our motivation and background on the link compression scheme and state-of-the-art approaches. Section 3 presents our link compression approach, Section 4 presents our experimental evaluation and Section 5 concludes the paper.

2 Motivation and background

2.1 Motivation

Despite their characterization as “compute-intensive” or “number-crunching”, large classes of scientific workloads are data-intensive and operate on large data sets, performing rather lightweight computations with limited data reuse. The performance of such applications is heavily dependent on the memory-link speed, i.e. on the capability of the memory subsystem to feed the fast processing cores with data. In that respect, memory-bound scientific workloads are ideal candidates for applying data compression. However, these workloads typically operate on floating-point data, which are very difficult to compress due to their binary representation in the IEEE 754 arithmetic format, which generates high entropy
to data representation. In this work we face the challenge of applying memory-link compression schemes on floating-point data in order to shed light on the opportunities of performance improvement inherent in this paradigm.

2.2 Compression at the memory link

We examine memory-link compression as a technique to deal with the bandwidth wall in modern CMP systems. We model a simple scheme, as the one proposed in [1]. The main objective is to reduce the off-chip memory traffic and to improve performance by transferring compressed data over the link. Each cache line is compressed before it is being transferred over the link and decompressed right afterwards (as shown in Fig. 1). Compression and decompression units are required on both sides of the link.

2.3 State-of-the-art compression methods

A major challenge in memory-link compression is that the block of data to be compressed is very small (typically a cache block) and therefore difficult to contain compressible redundancies. This challenge becomes even harder for floating point datasets that have very small inherent redundancy. This section presents three previously proposed hardware compression algorithms. The first two (bitwise compressors) aim at eliminating redundancy by exploiting nearby value locality [8] at the cache-line granularity [2, 6], while the third extracts frequent value locality using a small memory table [3] monitoring the whole off-chip traffic.

**BDI:** The Base-Delta-Immediate compression algorithm (BDI) is a delta encoding [1] scheme previously applied for cache [6] and memory link [9] compression. BDI is motivated by the fact that the words residing in the same cache line most likely belong to the same dynamic range. Thus, their relative difference is small and a cache block can be encoded as a single base value with an array of delta (Δ) values. BDI's latency overheads are estimated to be: a) 1 cycle for decompression and b) 6 cycles for compression [6]. As BDI exploits mainly the potentially small variance in the representation of integer values stored in the same cache line, it is not expected to compress FP data effectively.

**Diff3:** Diff3 is a variation of the Differential algorithm [2], proposed for link compression and energy minimization in embedded systems. It is a bitwise compressor that works based on the observation that several cache line words have in common some of the most significant bits (MSBs). Thus, it eliminates the MSBs of each word that are common with the first block’s word. The algorithm is designed for integer data, however it might compress FP data too, as the sign and
exponent fields (MSBs in IEEE-754 standard) of a cache block’s FP values tend to be more regular. Diff3 has low latency overheads, as it uses basic binary operations and executes compression and decompression in parallel. Here we assume a pessimistic overhead of 5 cycles for both compression and decompression.

**FVE:** The Frequent Value Encoder has been applied in different scenarios including cache [10] and link compression [3] and NoC architectures [11]. It is based on the observation that a small set of frequent values tends to occupy a large portion of the off-chip traffic [12] for many applications. Thus, it stores this set in a small table, constructing the frequent value (FV) dictionary. Then compression can occur by encoding these values using their corresponding indexes. In link compression, dynamically updated (de)compression FV dictionaries are kept in both sides of the link. The algorithm’s latency overheads depend on the size of the dictionary [3]. We assume a 512B dictionary and a pessimistic (de)compression overhead of 10 cycles (as in [13]).

### 3 A compression engine for floating-point data

As discussed in Section 2, state-of-the-art hardware compression schemes fail to compress FP datasets effectively. Therefore, we resort to a software compression algorithm designed explicitly for FP data. Specifically, we consider BFPC, a lossless, linear-time floating-point compression algorithm proposed by Burtscher et al. [7]. We chose BFPC for the following reasons: a) BFPC compresses scientific datasets exhibiting a compression performance competitive to that of known, complex FP compression algorithms (bzip2, gzip, e.a), while remaining simpler and faster (8 to 300 times) [7]. b) Its functionality and performance do not depend on prior knowledge of the specific data domain, contrary to other FP compression algorithms like geometric data [14], etc. c) It is a single-pass algorithm and thus (de)compression can occur on the fly. d) It interprets all doubles as 64-bit integers and uses mainly bitwise operations, simplifying a hardware implementation of the algorithm. BFPC has so far only been considered as a software algorithm. Next, we sketch the block diagram of BFPC.

#### 3.1 BFPC compression logic

**Compression:** BFPC compresses streams of IEEE 754 double-precision FP data by comparing each value with a predicted one. Two different predictors are used for better performance, implemented as hash tables that store predicted values: fcm [15] and dfcm [16]. To compress a value, a bitwise XOR operation is applied between the most accurate prediction and the original value. Accurate predictions lead to small value XOR results and, hence, compression is achieved by truncating their leading zero bytes.

**Prediction:** The prediction logic is based on the observation that specific sequences of FP values appear multiple times in a dataset. Consequently, keeping a dynamic record of value sequences can result in more accurate value prediction. The fcm hash key, which is the “pointer” to the fcm prediction table, represents
the sequence of the most recently encountered FP values. The corresponding hash table cell stores the value that followed this sequence the previous time it was encountered [15]. This stored value forms the prediction. The dfcm predictor has the same logic but works with deltas instead of absolute values.

Decompression: Decompression is the exact reverse procedure of compression, and the value’s lossless reconstruction is due to XOR’s property of being completely reversible. Decompression algorithm reconstructs from scratch the prediction tables and no metadata is needed for consistency.

3.2 BFPC in hardware link compression

BFPC can be used with small modifications in a memory-link compression scheme. A small adjustment is required for prediction. BFPC is designed to process big streams of double-precision floating-point data and it constructs prediction tables per stream. The algorithm’s efficiency is based on the gradual adaptation of the tables’ content to the input value-stream through constant update, taking advantage of the stream size. In link compression, as the block (stream) of to-be-compressed data is typically small (a single cache block), the construction (reinitialization) of the tables for every cache block would be inefficient. Hence, we model these tables as small caches, built in both sides of the link (as part of the (de)compression circuitry). Every word transferred over the link dynamically updates the tables and no extra signals are required to maintain the compression and decompression tables consistent (compression and decompression occur on the fly, as in FVE [3]). Consistency prerequisites are: a) the common initialization of the tables during the system start up and b) the “one-to-one” relationship between the compression and decompression circuitry (the data must be decompressed in the same order as they are compressed).

3.2.1 Encoding a word in steps

Fig. 2a depicts the required steps of encoding a cache block’s word (Wᵢ). Compression and decompression are inherently sequential due to the use of the prediction tables. However, they can be pipelined.

- **Step 1** reads the fcm and dfcm predicted values from the tables, indicated by the corresponding pointers (HashKeys). The content of the tables and pointers is updated in the same step, in parallel, to be set for the next prediction. The update of the tables consists of storing the word’s original value (TrueValue) to the positions pointed by the current HashKeys. These HashKeys form a type of “history register”, holding sequentially the bits of the previously encountered double values (words). Thus, updating them includes loading (shifting) in their least significant bits (LSBs) the current word’s TrueValue [15] or a delta value [16]. However, this value is previously right shifted to eliminate a large part of the highly irregular mantissa [15].
- **Step 2** applies a XOR operation to the TrueValue with the predicted values.
- **Step 3** selects the most accurate predicted value.
- **Step 4** counts the leading zero bytes of the XOR result (cnt) to be truncated.
Step 5 forms a 4-bit code (`code`), indicating the selected predictor and the `cnt` leading zero bytes. This `code` and the remaining bytes (the `residual`) of the XOR result form the compressed word. All `codes` are stored in the beginning of the compressed cache line and used as metadata in decompression.

---

**Fig. 2:** BFPC (de)compression block diagrams.

### 3.2.2 Hardware implementation and overhead estimation

The prediction tables are modeled as direct-mapped caches (DMC). The DMC size is an important design parameter, as it affects both compressibility and (de)compression latency. DMC is set to 512B in our experiments, as it gives high compression while keeping area and latency overheads low. `HashKeys` are implemented as registers.

Fig. 2a also demonstrates the potential hardware implementation of the previously described steps that form a pipeline. **Step 1**’s critical path is reading and updating the DMC. Although reading/updating a 512B DMC requires 0.4ns based on CACTI [17], we assume that it operates at the bus speed (i.e., 1ns for 1-GHz memory-link speed), thus the total delay of reading and updating DMC is 2ns or 2 CPU cycles\(^3\). Updating the table pointers (`HashKeys`), which is also part of Step 1, is done using parallel shifters and XOR units. **Step 2** is a simple XOR operation (1 cycle). In **Step 3**, the selection of the most accurate prediction is implemented with a comparator and mux (1 cycle). Leading Zero byte compression of the selected XOR result in **Step 4** is done by an LZC [18] in 1 cycle delay. Finally, generating and storing the 4-bit `code` along with the `residual` in the compressed cache line buffer (**Step 5**) are done using parallel shifters (1 cycle).

---

Fig. 3 depicts a timing diagram of the compression pipeline. If the cache line size is 64 bytes (8 words), the overall compression delay can

\(^3\) We assume CPU frequency equal to 1GHz
Read the word’s True Value from the cache line (Wi)
Update the Hash Keys
Read the Predictions from the fcm and dfcm tables / Update the table with the True Value
Xor the True Value with the Predictions
Compare and Select the most accurate Prediction
Leading Zero byte counter of the selected Xor
Encode and output the compr. code and the residual bytes

Fig. 3: The overhead of the compression pipeline (in cycles)

be estimated to 14 cycles. However, in our experiment we assume a highly pessimistic overhead of 20 cycles.

Decompression: Decompression logic is very similar to compression. Fig. 2b depicts its basic steps. Note that the starting bit address of each word’s residual part in the compressed block depends on the size of the previous stored residuals. Thus, these addresses can be computed in parallel using a multi-stage carry lookahead adder network having as input the compression metadata tag. This stage is not depicted in Figure 2b. Even if decompression is slightly faster, we will assume for simplicity that it has the same overhead as compression.

4 Experimental evaluation

4.1 Experimental setup

We evaluate the four compression schemes (BDI, Diff3, FVE and BFPC) using Sniper [19], an execution-driven application level simulator. We run multithreaded versions (8 threads) of three memory-bandwidth bound scientific applications: i) the LU decomposition algorithm, ii) the Floyd-Warshall (FW) algorithm and iii) the Sparse Matrix-Vector Multiplication (SpMV) algorithm. We experiment with both FP and integer input datasets. As the efficiency of the applied algorithms greatly depends on the input data set, we collected actual FP from three sparse matrices taken from the University of Florida Sparse Matrix Collection [20] and integer data from weighted graphs holding information for USA roads [21].

Baseline parameters

<table>
<thead>
<tr>
<th>LU &amp; FW</th>
<th>SpMV</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 cores</td>
<td>OoO,</td>
</tr>
<tr>
<td>L1 cache size</td>
<td>4</td>
</tr>
<tr>
<td>L2 cache size</td>
<td>256</td>
</tr>
<tr>
<td>L2 block size/assoc/hit time</td>
<td>64B/8-way/8 cycles</td>
</tr>
<tr>
<td>shared cores (L2)</td>
<td>4</td>
</tr>
<tr>
<td>Memory controllers</td>
<td>1</td>
</tr>
<tr>
<td>Dram access penalty</td>
<td>100 cycles</td>
</tr>
</tbody>
</table>

Table 1: Baseline Configuration

Table 2: (De)compression overhead (cycles)

<table>
<thead>
<tr>
<th>comp. o.h</th>
<th>decomp o.h</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diff3</td>
<td>5</td>
</tr>
<tr>
<td>BDI</td>
<td>6</td>
</tr>
<tr>
<td>FVE</td>
<td>10</td>
</tr>
<tr>
<td>BFPC</td>
<td>20</td>
</tr>
</tbody>
</table>

Table 2: (De)compression overhead (cycles)

Table 1 presents the baseline simulation parameters and Table 2 summarizes the latency overheads of the compression algorithms. Due to simulation time constraints, we reduce the input data set of LU and FW and proportionally adapt the cache configuration. For SpMV we maintain a realistic CMP configuration. In all cases, the input dataset is at least 4 times larger than the LLC size. As we aim to evaluate the full potential of the studied schemes, for Diff3 and FVE we use a word size of 4 bytes in integer datasets and 8 bytes in FP datasets. On the other hand, the BDI algorithm examines itself different word granularities and selects the one that gives the best compression ratio.
We use 3 different configurations for the off-chip memory bandwidth: 2GB/s, 4GB/s and 8GB/s, as we aim to evaluate the efficiency of link compression in different traffic conditions. We assume a single memory controller, since this is essential to maintain BFPC and FVE encoding dictionaries consistent.

The efficiency of memory-link compression is evaluated using: i) the average main memory access latency (simulation statistic), ii) the data compressibility for each scheme: \( \text{Compression Ratio (CR)} = \frac{\text{Original Data Size (Total)}}{\text{Compressed Data Size (Total)}} \)

and iii) Speedup: \( \text{SpeedUp} = \frac{\text{Exec. Time without compression}}{\text{Exec. Time with compression}} \). We verify our evaluation by running each simulation multiple times and average the results excluding outliers.

4.2 Experimental results

4.2.1 Compressibility Figs 4a and 4b present the compression ratio (CR) for the simulated applications and various input datasets for each compression scheme. CR can be less than 1 due to compression metadata overhead.

Floating Point datasets. BFPC has the best performance across all applications and datasets. It reduces the off-chip memory traffic by 13% – 37.8%, whereas the other schemes mostly suffer from poor compressibility. BFPC’s lower efficiency for the SpMV kernel is attributed to its pure streaming nature, since BFPC’s performance is highly boosted by data reuse of any degree. The SpMV kernel usually runs for multiple iterations in a scientific application (data reuse exists). Due to simulation limitations though, we gathered statistics only from single iterations of the kernel. Comparing the two bitwise compression schemes, Diff3 seems to perform better most of the times. Diff3, contrary to BDI, eliminates the common MSBs (i.e. typically sign and exponent) of the cache block values that tend to be more regular.

Interestingly, we notice that the compressibility for most schemes (except BDI) is better in FW than LU, two kernels with very similar access patterns. As application datasets are modified during execution, the compressibility of
the original input dataset does not characterize the compressibility of the workload over time. Therefore, the way the dataset changes constitutes an important compressibility parameter. We attribute the better compressibility of the FW kernel to the fact that it performs additions, while LU performs divisions – an operation that changes values fiercely, introducing higher entropy to the data. This affects mainly the efficiency of the BFPC and Diff3 compression schemes, since they both exploit the regularity of the FP MSBs.

**Integer datasets.** In Fig. 4b we observe that the bitwise compressors are more efficient. This is expected, as these schemes eliminate the redundancy found in binary representation due to their small variance. Diff3 has slightly better CR than BDI (57.5% vs. 54% respectively, on average), possibly because it operates on a finer granularity than BDI (bit vs. byte). On the other hand, BFPC has interestingly the same compressibility (i.e. ~ 30%) as it had for floating-point workloads. This is important if overall consistency in compressibility is critical.

### 4.2.2 Effect on memory access latency

Here we focus on the effect of link compression on the average Memory Access Latency (MAL). In Sniper, MAL consists of the DRAM access time, the over link transfer time and the queuing delays due to the competition of the cores for the shared memory resource. Thus, a link compression scheme affects only the transfer time and the queuing delays.

Fig. 5 depicts the effect of ideal link compression on the average MAL, i.e. when zero latency for (de)compression is assumed. The first bar corresponds to the baseline system without compression. Interestingly, the transfer time is only slightly affected and compression mainly affects the queuing delays. As the bandwidth increases, compression’s effect is dramatically reduced. Hence, compression schemes are expected to be beneficial under high contention conditions.

Fig. 6 illustrates the impact of each link compression scheme (with and w/o accounting for (de)compression latency) on total MAL for different bandwidth configurations for various applications and input datasets. When the application is bandwidth bound (2GB/s), memory access time is significantly reduced for all compression schemes. Moreover, under intense bandwidth conditions, the impact of (de)compression overhead is small. This is attributed to the fact that when an application is completely memory bound the rate at which it will be able to complete requests is fixed (bandwidth bottleneck domination). Therefore, the addition of an extra waiting component (de-compression overhead) just leads the requests to spend less time in the queue but hardly changes the total number of requests that can be completed per second (and hence access latency). However, as available bandwidth increases, the benefit is eliminated, while in several cases the memory access time is penalized quite significantly.

### 4.2.3 Effect on performance

Fig. 7 demonstrates the impact of link compression on speedup for all applications and datasets. Each compression scheme
is evaluated for two bandwidth conditions (2GB/s and 8 GB/s) and by accounting for the (de)compression overhead (light-color bars vs. dark-color ones, respectively). Furthermore, we annotate the CR on the top of each application bars. When an application is memory-bandwidth bound (2GB/s) and the compression scheme is successful, the execution time is significantly reduced. In FP datasets the higher speedup is observed for BFPC and is between \(~1.18\) (CR=14\%) to \(1.41\) (CR=29\%). On the other hand, for the integer datasets, the highest speedup is provided by the bitwise compressors, i.e., \(1.64\) (CR=39\%).
When memory bandwidth is not a bottleneck (8GB/s bandwidth), the benefit of link compression on performance becomes negligible. In several cases, as expected, performance is actually penalized due to the introduced (de)compression time overheads. In particular, BFPC increases the execution time by 5%-10% for all applications and data sets and FVE by close to 5%. The overhead is higher for FW than LU because FW is characterized by smaller queuing delays. Therefore at 2GB/s bandwidth, the overhead of BFPC is nearly invisible for the LU kernel, whereas for the FW kernel it limits the performance by almost ∼10%. Regarding SpMV kernel though, the scheme’s negative impact is limited even for 8GB/s and benefit can still be noticed, since the algorithm remains memory bound (especially for large input matrices as Helm Matrix). The system could benefit by using a mechanism to monitor the demands in memory traffic while measuring the efficiency of the compression scheme and decide to turn on/off compression.

Finally, the effect of link compression on an application’s performance depends on the fraction of total execution time spent on memory accesses. Our kernels are memory bound at 2GB/s bandwidth, but not at the same degree. DRAM accesses constitute 90% of the total CPI for LU, while being much lower for FW (78%). This explains why slightly higher speedup is noticed in LU than FW, e.g when using BFPC with Helm input matrix, despite the similar compression ratio.

5 Conclusions

This work employs data compression on the off-chip link to boost the performance of scientific workloads, which manipulate large amounts of floating-point data that cannot be efficiently compressed by state-of-the-art compression schemes. We propose a hardware implementation of the FPC algorithm by Burtscher et al. (BFPC), that introduces small area overheads and has a compression ratio for FP data of 20% on average and up to 40%, while other schemes achieve less than 10% on average. This benefit off-chip traffic reduction can be translated, under bandwidth bound conditions, to performance improvement of up to 40% giving an important boost to scientific workloads.

For future work, we consider investigating an adaptive compression scheme, where compression is triggered by a monitoring mechanism only when bandwidth is constrained. We also plan to investigate the impact of Huffman coding on link compression, as a recent study has shown promising results for LLC compression [22].

Acknowledgments

Part of this work was carried out in National Technical University of Athens and was partially funded by project I-PARTS: “Integrating Parallel Run-Time Systems for Efficient Resource Allocation in Multicore Systems” (code 2504) of Action ARISTEIA, co-financed by the European Union (European Social Fund) and Hellenic national funds through the Operational Program “Education and Lifelong Learning” (NSRF 2007-2013).
On Parallel Evaluation of Matrix-Based Dynamic Programming Algorithms

David Bednářek, Michal Brabec, and Martin Kruliš

Parallel Architectures/Algorithms/Applications Research Group
Faculty of Mathematics and Physics, Charles University in Prague
Malostranské nám. 25, Prague, Czech Republic
{bednarek, brabec, krulis}@ksi.mff.cuni.cz

Abstract. Dynamic programming techniques are well established and employed by various practical algorithms, for instance the edit-distance algorithm. These algorithms usually operate in iteration-based fashion where new values are computed from values of the previous iterations, thus they cannot be processed by simple data-parallel approaches. In this paper, we investigate possibilities of employing multicore CPUs and Xeon Phi devices for efficient evaluation of a class of dynamic programming algorithms. We propose a parallel implementation of Wagner-Fischer algorithm for Levenshtein edit-distance that demonstrates utilization of task parallelism and SIMD parallelism which are native to both architectures. The proposed solution has been subjected to empirical evaluation and its results are also presented in this work.

Keywords: dynamic programming, edit distance, parallel, SIMD, MIC

1 Introduction

Dynamic programming is a well established method of algorithm design. It solves complex problems by breaking them down into simpler subproblems. It is especially useful when the subproblems overlap and identical subproblems are computed only once, hence redundant computations are avoided. On the other hand, algorithms based on dynamic programming are usually difficult to parallelize, since the subproblems are interdependent – i.e., one subproblem requires the results of previous subproblems.

In this work, we focus on dynamic programming algorithms which logically organize their subproblem results into regular matrix. Each result in the matrix is computed from a small subset of previous results, using a recurrent formula like

\[ x_{i,j} = f(i,j)(x_{i-1,j}, x_{i,j-1}, x_{i-1,j-1}) \]

Typical examples are the edit distance problem originally described by Levenshtein [5], the dynamic time warping algorithm [8], or the Smith-Waterman algorithm [11] for molecular sequence alignment.
The functions $f_{i,j}$ are often very simple, as it is in the case of the Wagner-Fischer dynamic programming algorithm \cite{12} for the Levenshtein distance,

$$f_{i,j}(p, q, r) = \min(p + 1, q + \delta^{v(p)}_{u(q)}, r + 1)$$

where the Kronecker $\delta$ compares the $i$-th and $j$-th positions in the input strings $u$ and $v$, respectively.

For our implementation and experiments, we have selected the Levenshtein distance as a representative of the examined class of problems. We will not employ any optimizations designed specifically for the Levenshtein distance (like the Myers’ algorithm \cite{9}), so our proposed solution is applicable for similar dynamic programming problems as well.

The dependencies between individual invocations of the recurrent formula $f$ significantly limit the parallelism available in the problem – for a $M \times N$ matrix, at most $\min(M, N)$ elements may be computed in parallel, using the diagonal approach illustrated in Figure 1. In addition, the computation of $f$ is typically too small unit of work to become a base of thread-level parallelism; therefore, the pure diagonal approach is not applicable in real hardware.

![Fig. 1. Dependencies in the matrix and the diagonals](image)

In this paper, we examine parallel implementation of this class of problems in two hardware environments – multicore CPUs and the new Xeon Phi devices which are based on the Intel Many Integrated Core (MIC) architecture \cite{4}. Both the multicore CPUs and Xeon Phi devices employ two levels of parallelism: The multicore nature of both devices allows concurrent execution of multiple independent threads and each core is capable of executing vector (SIMD) instructions.

We present an implementation of the Levenshtein distance problem which employs both thread-based and SIMD-based parallelism. We compare their performance in multicore and manycore architectures and we show that both levels of parallelism offer very significant speedup, allowing a manycore chip to execute about $6 \cdot 10^{10}$ invocations of the recurrent formula per second.

The paper is organized as follows. Section 2 overviews work related to efficient implementations of dynamic programming algorithms. In Section 3, we revise the fundamentals of multicore CPUs and Xeon Phi devices which influenced our implementation decisions. Section 4 briefly analyses the problem with respect to parallelism and cache hierarchy, while the proposed algorithm is described in
Section 5. The empirical evaluation is summarized in Section 6 and Section 7 concludes the paper.

2 Related Work

One of the first papers that addressed parallel evaluation of edit distance problem was presented by Mathies [7]. It was a theoretical work that described and algorithm for PRAM execution model. The proposed algorithm achieved $O(\log m \log n)$ time complexity using $mn$ processors, where $m$ and $n$ are the lengths of the compared strings.

Most practical parallel algorithms are based on an observation of Delgado [1], who studied the data dependencies in the dynamic programming matrix. Two possible ways of processing the matrix were defined in their work – unidirectional and bidirectional filling. Their idea allows limited concurrent processing of independent parts, but it needs to be modified for massively parallel environment.

A similar problem that uses dynamic programming is dynamic time warping (DTW) as defined by Müller [8]. Weste et al. [13] proposed a parallel approach that implemented the algorithm directly in CMOS chip. Even though this solution is rather specialized, it proposed some basic ideas that may be applied in highly parallel environments. More recent discussion on parallelization techniques for GPUs and FPGAs was presented by Sart et al. [10]. Their work focused mainly on a specific version of DTW algorithm, which reduces the dependencies of the dynamic programming matrix, thus allows more efficient parallelization.

Another example of a problem suitable for dynamic programming is the Smith-Waterman algorithm [11], which is used for protein sequences alignment. Farrar presented a SIMD implementation [3], which utilizes instructions of mainstream CPUs. It achieved a 2-8× speedup over other SIMD implementations. There are also many papers that address the parallelization of Smith-Waterman on GPUs. Perhaps the most recent work was presented by Liu et al. [6] and it combines many observations from the previous work on the subject.

Unlike these related papers, our work does not focus on a single dynamic programming problems. We are proposing a parallel approach, which may be used for various dynamic programming problems sharing the matrix shape of dependencies.

3 Multicore and Manycore Architectures

The pursuit for higher CPU frequency has reached an impasse and it has been abandoned for concurrent processing. Parallelism has easily found its way into mainstream processors and current architectures employ parallel processing on several levels. Furthermore, massively parallel platforms have emerged, such as GPGPUs or new Xeon Phi devices.

Current state-of-the-art CPUs employ parallelism on several levels. The most obvious one (and perhaps the most advertised one) is the explicit task parallelism
which is realized by integrating multiple CPU cores into a single CPU chip. It is further magnified by multithreading technologies such as hyperthreading (Intel) or dual-core module (AMD), which equip each physical core with two (or even more) logical frontends that share the internal resources of the core but appear as independent cores to the rest of the system.

Another type of parallelism is embedded in every core on instruction level where independent subsequent instructions may overlap, since they are being processed by different circuits of the CPU core. In most cases, exploiting this type of parallelism is limited to code optimizations by a compiler.

Finally, the CPUs of the day also employ SIMD parallelism. It is implemented using specialized vector registers and instructions sets that operate on these registers, such as Intel SSE or AVX instructions. The vector registers are capable of comprising multiple values (e.g., in case of SSE, four 32-bit numbers) and the instructions perform their operations concurrently on all the values in the register. The SIMD instructions can be generated automatically by the compiler; however, in complicated algorithms, the utilization of SIMD instructions has to be designed by the programmer.

3.1 Xeon Phi Device

The Intel Many Integrated Core (MIC) architecture [4] takes the parallel processing one step further, both on the level of task parallelism and the SIMD parallelism. It is Intel’s answer to rapidly evolving GPUs, which are currently excelling in data parallel tasks. This architecture was employed in the Xeon Phi devices, which are parallel accelerators designed as extension cards to servers or PCs (in a similar way as GPUs). The device comprises a massively parallel chip and several gigabytes of on-board memory.

![Fig. 2. An illustrative schema Xeon Phi processor](image)

The Xeon Phi processor is based on older Pentium architecture, but is has been augmented in several ways. The most important differences from mainstream CPUs are the following:
- Simplified architecture allowed an order of magnitude increase in the number of integrated cores. The Xeon Phi has approximately 60 cores (exact figure depends on the model number).
- The cores employ in-order instruction execution which slightly limits the instruction parallelism. On the other hand, the chip implements four-way multithreading where each core is prefaced with four logical cores. Instructions from the four associated threads are interleaved on the physical core.
- The MIC architecture utilize vector instruction set for SIMD computing, where the registers are 512-bits wide – i.e., 4× wider than the SSE registers.
- The Xeon Phi chip has up to 8 dual-channel memory controllers, so it offers much higher memory throughput than common SMP or NUMA systems.

Let us summarize the observations made about CPUs and Xeon Phi devices and point out the most important implications for our code design:

The dynamic algorithm should be divided into concurrent tasks, so it can occupy tens (or better hundreds) computational cores. The vector instructions are very important, since SIMD execution units are present in both target architectures. Furthermore, algorithm design should be prepared for different sizes of SIMD registers.

4 Analysis

Each problem instance is characterized by a set $T$ of its elementary tasks. In our case, $T$ corresponds to a subset of the two-dimensional space $\mathbb{N} \times \mathbb{N}$ usually bounded by a rectangle. For Levenshtein distance, the dimensions of the rectangle corresponds to the sizes of the input strings. Each elementary task $T_{i,j}$ corresponds to computing $x_{i,j}$ using the function $f_{i,j}$.

A parallel algorithm may divide $T$ into a hierarchy of subtasks. A set $S \subseteq T$ is a correct subtask when, for every pair of dependent elementary tasks in $S$, all dependency paths between them are also contained in $S$.

Each subtask has its i- and j-projections: $P_i(S) = \{i \mid (i, j) \in S\}$, $P_j(S) = \{j \mid (i, j) \in S\}$. These projections help to estimate the size of subtask inputs and outputs:

The input $I(S)$ of a subtask consists of $x_{i,j}$ values outside $S$ which are input to some elementary tasks in $S$. Similarly, the output $O(S)$ consists of $x_{i,j}$ values inside $S$ which are read by some elementary tasks outside $S$. A simple geometric observation yields the following limits for their sizes:

$$|I(S)| \geq |P_i(S)| + |P_j(S)| + 1$$
$$|O(S)| \geq |P_i(S)| + |P_j(S)| - 1$$

If the projections $P_i(S)$ and $P_j(S)$ are contiguous (i.e., intervals in $\mathbb{N}$), then the constraints become equalities.

At the same time, the size of a subtask is trivially constrained by

$$|S| \leq |P_i(S)| \cdot |P_j(S)|$$
Consequently, the following relation holds:

\[ \rho(S) = \frac{|I(S)| + |O(S)|}{|S|} \geq 2 \cdot \frac{|P_i(S)| + |P_j(S)|}{|P_i(S)| \cdot |P_j(S)|} \]

The ratio \( \rho(S) \) denotes how many inputs and outputs must be transferred per a unit of work, provided the communication inside the subtask is done using a local memory. The ratio is important in determining cache efficiency - lower \( \rho(S) \) means that distributing data between subtasks requires less transfers.

This observation implies that the best cache efficiency will be achieved for square subtasks. For subtasks of the same shape, the \( \rho(S) \) ratio is proportional to \(|S|^{-1/2}\). It means that the shape is important for small subtasks which are likely to suffer from limited communication bandwidth. In large subtasks, the outer communication will likely become negligible compared to their inner complexity.

The projections also give constraints on parallel execution: Subtasks \( S_1, \ldots, S_n \) may run in parallel only if they are independent, which implies that their projections must be disjoint in both dimensions. Consequently, the size of projections is limited by

\[ \sum_{k=1}^{n} |P_i(S_k)| \leq |P_i(T)| \]

where \( T \) is the supertask divided into \( S_1, \ldots, S_n \) (analogically the j-projection). Since load balancing requires subtasks of similar sizes, \( |P_i(S_k)| \) is limited by \( |P_i(T)|/n \) and, consequently,

\[ |S_k| \leq \frac{|P_i(T)| \cdot |P_j(T)|}{n^2} \]

This constraint immediately explains the inherent difficulty in parallelization of our dynamic programming problems: The maximal available size of the subtasks is inversely proportional to the square of the required degree of parallelism.

Furthermore, the communication ratio \( \rho(S_k) \) is proportional to \(|S_k|^{-1/2}\) and thus to \( n \). Consequently, higher degree of parallelism induces higher communication costs per unit of work.

5 Algorithms

The analysis in the previous section offers a guidance on the construction of parallel algorithm for our problem: The complete task must be divided into a sufficient number of subtasks, using subtasks of approximately square shape. The subtasks will form a two-dimensional mesh, where only those residing on a diagonal may be executed in parallel.

Inside every subtask, a different form of parallelism is possible using vector instructions. However, the degree of vector parallelism is usually small (4 to 16 in our environment). At the same time, it is advantageous to maintain the lowest level of subtasks so small that they can handle internal communication using
CPU registers. To fit in the limited space, one of the subtask projections must be sufficiently small, typically equal to the size of a vector register.

In our algorithm, such subtasks are called *stripes*. Each stripe is $s$ elements wide and $s \cdot n$ elements long, where $s$ is the size of a vector register and $n$ is a carefully tuned constant. To allow vector-based parallelism, stripes are not rectangular but skewed as shown in Fig. 3.

![Fig. 3. Stripe](image)

Figure 3 also illustrates the meaning of *boundary arrays* used in our algorithm to replace the two-dimensional array $x[1 : N, 1 : M]$. The $w[1 : M]$ array is stretched along the (horizontal) $j$-dimension and each $w[j]$ stores the value of $x[i,j]$ for the greatest (lowermost) $i$ for which the task $T_{i,j}$ was already completed. Similarly, $y[i] = x[i,j]$ and $z[i] = x[i-1,j]$ for the greatest (rightmost) $j$ for which $T_{i,j}$ is completed. The figure shows the boundary arrays before the stripe execution while the primed versions $w', y', z'$ denote the meaning after the stripe execution. The $x[i,j]$ mark denotes the reference point used to specify the position of the stripe in Algorithm 1 and 2.

For smaller $n$, a stripe is too small to form a base for thread-level parallelization; selecting larger $n$ would violate the requirement for approximately square shape. Therefore, $n$ stripes are grouped (by sequential execution) into a larger subtask of size $s \cdot n$ times $s \cdot n$. Because of these dimensions, we call them *square subtasks*, although they are skewed to the shape of parallelogram.

The constant $n$ is tuned to achieve best performance: Selecting greater $n$ diminishes the overhead of parallel task creation and it also improves the communication ratio $\rho(S)$. On the other hand, larger subtasks lead to worse load balancing. In addition, a subtask whose internal communication can fit inside the L1 cache runs faster; therefore, the size $s \cdot n$ shall be sufficiently small, proportionally to the cache size. In our environment described in Sec. 6, selecting $s \cdot n = 256$ produces best or near-the-best results in both multicore and manycore cases.
The square subtasks form the base of parallel evaluation. A diagonally shaped set of square subtasks is selected for simultaneous execution; when they are completed, subsequent diagonal may be launched. Thus, the complete algorithm consists of a sequential loop over diagonals as shown in Algorithm 1. Due to the skewed shape of stripes, the algorithm must deal with incomplete stripes at the boundaries of the problem rectangle – the incomplete stripes are evaluated using scalar operations.

Algorithm 1: Iteration over stripes

Require: $N, M$ – problem size, $s$ – SIMD vector size, $n$ – stripe length
$u[1 : N], v[1 : M]$ – input sequences
$y[1 : N], z[1 : N], w[1 : M]$ – initial boundary values

Ensure: $y[1 : N], z[1 : N], w[1 : M]$ – final boundary values

for $p := 1$ to $\lceil(M + N)/(n \cdot s)\rceil$ do
  for $q := 1$ to $\lceil N/(n \cdot s) \rceil$ in parallel do
    for $r := 1$ to $\lceil N/s \rceil$ do
      $i := n \cdot s \cdot q + s \cdot r - (n + 1) \cdot s + 1$
      $j := n \cdot s \cdot p - 2 \cdot n \cdot s \cdot q - s \cdot r + n \cdot s$
      if stripe $(i, j)$ intersects with the rectangle $\{1 : N\} \times \{1 : M\}$ then
        if stripe $(i, j)$ is a subset of the rectangle $\{1 : N\} \times \{1 : M\}$ then
          evaluate stripe $(i, j)$ using SIMD
        else
          evaluate stripe $(i, j)$ using scalar operations
        end if
      end if
    end for
  end for
end for

Stripe evaluation consists of updating boundary values stored in the arrays $y[1 : N], z[1 : N], w[1 : M]$, at the indexes corresponding to the i- and j-projections of the stripe. Within a diagonal, the square subtasks have disjoint projections; therefore, their memory accesses do not overlap. Nevertheless, alignment to cache-line boundary must be observed to avoid false sharing – therefore, $n$ must be a multiple of cache-line size divided by the SIMD vector size.

Algorithm 2 performs SIMD evaluation of a stripe. It calls a vectorized version $\overline{f}$ of the elementary task function $f$, which evaluates $s$ elementary tasks at once. In addition, the algorithm requires several vector operations to manage the vector buffers. The operations are taken from the Intel® SSSE 3 instruction set and their semantics is shown in Table 1. The upper half of the table summarizes the operations required by the generic part of the algorithm, the lower half contains operations required to implement the specific Levenshtein distance problem as shown in the following definition:

$$\overline{f}(\varpi, \tau, \gamma, \pi, \nu) = \text{add} (\text{min}(\varpi, \text{add}(\tau, \text{cmpeq}(\pi, \nu)), \gamma)), \text{broadcast}(1))$$
Algorithm 2 Vectorized stripe algorithm

Require: \((i,j)\) – stripe position, \(s\) – vector size, \(n\) – stripe length in units of \(s\)

\[u[i : i + s - 1], v[j : j + (n + 1) \cdot s - 1]\] – input sequences

\[y[i : i + s - 1], z[j : j + (n + 1) \cdot s - 1]\] – status vectors

Ensure: \(y[i : i + s - 1], z[i : i + s - 1], w[j : j + (n + 1) \cdot s - 1]\) – updated status vectors

\[v' := reverse(v[j : j + s - 1])\]
\[u'[i : i + s - 1]; y' := y[i : i + s - 1]; z' := z[i : i + s - 1]\]

for \(k := 1\) to \(n\) do

\[v'' := reverse(v[j + k \cdot s : j + (k + 1) \cdot s - 1])\]
\[w'' := reverse(w[j + k \cdot s : j + (k + 1) \cdot s - 1])\]

for \(m := 1\) to \(s\) do

\[v' := alignr_1(v', v''); z'' := alignr_1(w', y')\]
\[y' := \text{shiftr}_1(v', w''); v' := alignr_1(y', w')\]
\[y' := y''; z' := z''\]

end for

\(w[j + (k - 1) \cdot s : j + k \cdot s - 1] := reverse(w')\)

end for

\(w[j + k \cdot s : j + (k + 1) \cdot s - 1] := reverse(y')\)

\(y[i : i + s - 1] := y'; z[i : i + s - 1] := z'\)

The algorithm uses vector variables \(u', v', v'', y', z', w', v''\) to cache a portion of the input strings \(u, v\) and boundary arrays \(y, z, w\). Some of these variables are shifted using alignr or shiftr instructions – thus, any misaligned reads or writes of the arrays are avoided.

The reverse operations are required to properly align the order of elements of arrays \(u, v, w\) with the arrays \(v, w\), because the \(f\) function must act on independent elements on an (anti-)diagonal. Although reverse may be implemented using a vector instruction, it may be eliminated from the algorithm by reversing the order of elements in the \(v, w\) arrays before invocation of the algorithm.

<table>
<thead>
<tr>
<th>(\text{reverse}(x_1, x_2, \ldots, x_n) = (x_n, x_{n-1}, \ldots, x_1))</th>
<th>(\text{alignr}<em>1(x_1, \ldots, x_n, y_1, \ldots, y_n) = (x_n, y_1, \ldots, y</em>{n-1}))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{shiftr}<em>1(x_1, \ldots, x_n) = (0, x_1, \ldots, x</em>{n-1}))</td>
<td>(\text{broadcast}(x) = (x, \ldots, x))</td>
</tr>
<tr>
<td>(\text{cmpeq}(x_1, \ldots, x_n, y_1, \ldots, y_n) = (-\delta_1, \ldots, -\delta_n))</td>
<td>(\text{min}(x_1, \ldots, x_n, y_1, \ldots, y_n) = (\min(x_1, y_1), \ldots, \min(x_n, y_n)))</td>
</tr>
<tr>
<td>(\text{add}(x_1, \ldots, x_n, y_1, \ldots, y_n) = (x_1 + y_1, \ldots, x_n + y_n))</td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Vector operations used in the SIMD algorithm
6 Experiments

The experiments were conducted on various string lengths, with 32-bit characters. We present only the results for strings of equal length, since they produce a square matrix that provides the greatest opportunity (and challenge) for parallel computation. Furthermore, we present results for a limited range of string lengths. Significantly shorter strings limit the parallelism, since the corresponding dynamic programming matrix does not have sufficient size. Larger datasets exhibit similar behavior in the terms of relative speedup as the presented ones.

The algorithms were implemented in C++11, using Intel SSE intrinsic functions for SIMD instructions and Intel Threading Building Blocks for task parallelism.

The experiments were conducted on a server equipped with four Intel Xeon E5-2630 CPUs (total of 24 cores) and an Intel Xeon Phi accelerator card with 61 cores, running Red Hat Enterprise Linux 7 and using Intel C++ Compilers for both CPU and Xeon Phi versions.

We measured wall time using standard C++11 clock interface whose precision was satisfactory given the run times were between seconds and thousands of seconds. All performed tests were repeated five times and the related measurements were within 1% range. We present the average value of these five measured values as the result of each test.

Figure 4 shows the execution time of various Xeon Phi and CPU implementations. The horizontal axis contains six different string sizes between 64000 and 2048000 elements. On the logarithmical vertical axis, the wall time is displayed divided by the product of string lengths. In other words, the vertical axis displays the time in nanoseconds per elementary task, where an elementary task consists of the computation of one 32-bit element in the matrix.
The left graph shows the results for scalar implementation while the right graph displays SIMD versions – four-fold vectorization (SSE) in the case of CPU and 16-element vectors (AVX-512) in the case of Intel Xeon Phi. In each graph, results for multicore CPU and Intel Xeon Phi are compared. In addition to the parallel version based on Intel TBB, single-threaded versions were also measured.

Each of the eight versions was independently manually tuned to achieve the best performance for strings of 100000 elements. Afterwards, the measurements for the six different string sizes were made with the same configuration, i.e., the graph shows scaling without any retuning.

For all except the parallel Xeon Phi versions, the graphs show very little dependence on the string size. Since the graphs display the time per elementary task, this fact demonstrates that, in the given range of sizes, the total time is almost precisely proportional to the product of string sizes. However, for parallel Xeon Phi, the time per elementary task significantly decreases until the string size reaches approximately one million. This shows that, for smaller strings, the computing power available in the 61 cores of Xeon Phi is not efficiently employed. Note that the implementation was tuned on data of similarly small size (100k) and the tuning included the selection of task size; thus, the inferior performance means that there is still not enough parallelism available in our algorithm for smaller data sizes. Nevertheless, there might be other factors that contributed to the performance degradation, in particular, the fact that the task-stealing strategy of Intel TBB is unaware of the complex cache structure of Xeon Phi.

Both the CPU and Xeon Phi versions show significant speed gain when vectorized (compare the two graphs in Fig. 4). The ratio is even greater than the degree of vectorization: The four-fold vectorization of the parallel CPU version brings about eight-fold speedup; similarly, the SIMD-based speedup in Xeon Phi is almost 18 in parallel version and almost 40 in single-threaded version, although the AVX-512 offers only 16 32-bit operations at once. This counterintuitive observation has an explanation: The algorithm is designed in cache-aware way, including the use of registers as L0-cache. Since the vector registers offer greater total space than the scalar registers, the vectorized version requires less bytes transferred between the registers and the L1-cache. At the same time, the use of vector reads/writes offers multiplied throughput; together, the total speed-up may be greater than the vector size.

7 Conclusions

We have proposed a scalable approach for implementing a class of dynamic programming algorithms. It utilizes both task-parallelism and SIMD parallelism, which are strongly supported in current CPUs and Xeon Phi devices. The experimental evaluation suggests that the solution scales well if the input data are sufficiently large. Furthermore, the results confirm applicability of the many-core architecture in this class of problems, since a single Xeon Phi outperformed four six-core CPUs by the factor of three.
The results also revealed that the use of SIMD instructions is crucial for the performance of the Xeon Phi, offering speed-up factor greater than the intuitively expected 16 due to the effect of total register size.

Some questions are still left open; in particular, the contribution of the task-stealing strategy to the inferior performance for smaller strings at Intel Xeon Phi. This could be an opportunity to apply a NUMA-aware task scheduling strategy which we previously developed [2].

Acknowledgements

This paper was supported by Czech Science Foundation (GAČR), projects P103-14-14292P and P103-13-08195, and by the Charles University Grant Agency (GAUK) project 122214.

References

Tangram: a High-level Language for Performance Portable Code Synthesis

Li-Wen Chang, Abdul Dakkak, Christopher I. Rodrigues, and Wen-mei Hwu

University of Illinois, Urbana, IL USA

Abstract. We propose Tangram, a general-purpose high-level language that achieves high performance across architectures. In Tangram, a program is written by synthesizing elemental pieces of code snippets, called codelets. A codelet can have multiple semantic-preserving implementations to enable automated algorithm and implementation selection. An implementation of a codelet can be written with tunable knobs to allow architecture-specific parameterization. The Tangram compiler produces highly optimized code by choosing and composing architecture-friendly codelets, and then tuning the knobs for the target architecture.

We demonstrate that Tangram’s synthesized programs are comparable in performance to existing well-optimized codes on both CPUs and GPUs. The language is defined in a concise and maintainable way to improve debuggability and to enable progressive improvement. This strategy allows users to extend their applications and achieves higher performance on existing architectures and new architectures.

1 Introduction

Performance portability has become critical for the adoption of heterogeneous architectures. The current practice is largely a tight coupling of architectures and languages targeting those architectures. As the industry gains increasing momentum toward heterogeneity and parallelism, motivated by the desire to decrease power utilization while increasing performance, the cost of writing, maintaining and porting a code to multiple architectures becomes a substantial economic burden. Architecture-specific optimizations are not necessarily transferable to other architectures, forcing programmers to reinvest in performance optimization. A portable and high performance programming language is therefore desired by both the industry and the research community.

There have been several works to tackle the programming challenge for heterogeneous architectures. First, OpenCL provides an abstract computing model allowing programmers to write portable code. OpenCL is however biased to GPU style programs over other architectures. It has been shown that non-trivial transformations [6, 8–10, 16–19] are required to port OpenCL kernels to other architectures. Since OpenCL code often has embedded architecture-specific optimizations, automating the porting process becomes challenging.

Domain Specific Languages (DSLs) [4, 13, 14] take a different approach by specializing the language with certain types of computation patterns for a particular domain. A DSL compiler translates the high-level computation patterns
to a general language, and potentially can target different architectures with adaptive implementations. Their applicability however is largely confined to the intended domain.

General-purpose high-level languages using a library of algorithmic skeletons [2, 5, 7, 15] can also provide abstraction for computation and data, and rely on adaptive libraries to achieve performance portability. To fully support general-purpose applications, the coverage of skeletons to architecture becomes an important issue.

In this paper, we present a general-purpose, high-level programming language, Tangram, that offers performance portability while maintaining productivity of the programmer. Tangram achieves this goal by decoupling architecture-dependent optimizations from high-level abstractions of computational patterns. Unlike algorithmic skeletons [5, 7, 15], which contain rich computation and data-structure libraries targeting a variety of architectures, Tangram only provides data-structure-based libraries. Transformations occur in Tangram either with data-structure-based libraries or by explicitly specifying an optimization rewriting rule. The rules are more aggressive than conventional compiler transformations, since the programmer gives the compiler more information from the application domain.

This paper makes following contributions:

- We propose a tuning-friendly high-level language that achieves high performance and performance portability on both CPUs and GPUs.
- We demonstrate that programs written in the proposed language are easy to maintain, debug, and incrementally improve, thus increasing efficiency of the programmer.

The rest of the paper is organized as follows: Section 2 describes Tangram’s design. Section 3 presents features of Tangram. Section 4 shows selected examples and their performance evaluation. Section 5 discusses the current limitation and potential future work. Section 6 discusses related work, and Section 7 concludes.

2 Tangram Language

Performance portability is addressed in Tangram using a mixture of syntax to allow for performance tuned variables, abstract containers that simplify the analysis and development of algorithms, and loop manipulation primitives that enable transformations such as tiling, vectorization, and threading. Tangram admits multiple functionally equivalent implementations. This approach enlarges the design space and enables performance portable support for different hardware and runtime primitives. Type qualifiers and annotations allow programmers to override the choice of codelets and/or to specify the chosen transformation. This strategy allows programmers to bypass the tuning phase and provide customized algorithms for a specific algorithm or architecture.
Tangram: a High-level Language for Performance Portable Code Synthesis

The spectrum of the reduce codelets contains two atomic implementations using different architectural components and two compound codelets that perform different tiling transformations.

2.1 Codelet

Tangram adopts the codelet programming model [23] and extends it to facilitate performance portability, productivity, and maintainability. A codelet in Tangram is a coarse-grained code segment. Multiple codelets share the same name and function signature if they offer equivalent functionality for the current program. These codelets form a spectrum. Each codelet in a spectrum can be implemented either using different algorithms or the same algorithm but with different optimization techniques. Substituting one codelet for another in the same spectrum is considered a transformation of the program.

There are two kinds of codelets. Atomic codelets do not invoke other codelets in their implementations, while compound codelets do. Compound codelets may not be a scheduling quantum in execution, unlike the original codelet execution model. Leaving the decision of scheduling to either runtime or compiler allows the implementation to adapt across architectures. Multiple independent instances of an atomic codelet can form a kernel, which is similar to an OpenCL kernel. Furthermore, before scheduling is determined, a compound codelet can be decimated to multiple atomic codelets. The decimated codelets can then be fused. Fig. 1 shows an example of four codelets that implement the reduction pattern. The atomic codelet, shown in Fig. 1(a), performs the reduction sequen-
tially, while the compound codelets, shown in Fig. 1(b) and (d), partition the workload into multiple reductions and then performs another level of reduction.

To guarantee that each compound codelet can be transformed, each codelet spectrum contains at least one atomic codelet or a compound codelet invoking a spectrum which can terminate. Although a spectrum can be compiled to an unbounded set of possible codes due to recursion, in practice, the transformation step is pruned based on architectural limitations such as depth of the memory hierarchy and resource constraints.

2.2 Vectorized Codelet

A codelet can be also categorized based on its granularity. A sequential codelet executes sequentially. A vectorized codelet, on the other hand, is executed by a vector unit in semantic lock-step. The vectorized codelets are designed to describe a fine-grained communication pattern for vectorization, such as data shuffling across vector lanes. These codelets enlarge the design space by making use of the vector units prevalent in modern multi- and many-core architectures. Fig. 1(c) shows a vectorized codelet for the reduction where a tree-structure communication pattern, using the Kogge-Stone algorithm [11], is implemented.

Note that a vectorized codelet without communication across vector lanes is considered a group of sequential codelets with a particular scheduling, thus is regarded as redundant and is eliminated during compilation.

Tangram abstracts the architectural details, allowing vector length to be independent of the execution architecture constraints. Multiple instances, or parts of a vectorized codelet, can therefore be scheduled on an architectural vector unit.

2.3 Annotation and Qualifier

Sequential and vectorized codelets are distinguished by an annotation: vector. In a vectorized codelet, a _shared qualifier is used to label shared data across vector lanes, similar to OpenCL. Since codelets can be overloaded, a user may use a tag annotation to distinguish between them. The tag annotation is purely syntactical with no semantic meaning, but it may be used to differentiate between codelets during debugging. In Fig. 1(c), for example, the tag tag(kog) is used to document the fact that the codelet implements a Kogge-Stone reduction tree. The env annotation is used to label either architectural- or algorithmic-specific codelets, and provide strong hints to the compiler which can be used while synthesizing the kernel. Tuning-friendly qualifiers, such as _tunable and _fixed, can be used to label variables that are parameterized based on the granularity of the scheduling, the tiling factor, vector size, and cache line size of the target architecture.

2.4 Container

High-level languages provide containers to represent arrays. Containers allow programmers to reflect their intentions more clearly and avoid common bugs
found when using conventional linearized arrays. The container object also aids the compiler, since consecutive accesses along a high dimension can be inferred and proper optimizations applied.

Tangram codelets only accept scalars or containers as parameters. A wrapper function, which is not a codelet, is a user-defined function containing conventional linearized arrays and converters which map the linearized arrays to Tangram containers. In Tangram, conventional linearized arrays in a wrapper function are always assumed to not alias for consistency with sequential semantics in parallel execution.

2.5 Primitive

Tangram provides loop and data manipulating primitives which operate on containers. Table 1 lists the primitives and their definitions. Both \texttt{map} and \texttt{partition} are used to build compound codelets. The \texttt{map} primitive indicates independent workloads, which can be scheduled in parallel. The \texttt{partition} primitive indicates a data usage pattern which can be applied for better sharing or caching in parallel computing.

Unlike most library-based high-level languages, Tangram does not provide computational primitives, such as histogram or FFT. One main reason is that properties of a computational primitive might vary across applications or domains. A reduction, for example, in one domain may maintain both associativity and commutativity, while it may only maintain associativity in others. For a library designer, choosing these computational patterns and generality is not an easy task. Therefore, Tangram lets users build their own computational patterns using built-in loop and data manipulating primitives.

Another reason is that Tangram is not a pure library-based high-level language. Since high-level languages often rely on libraries which are commonly missing in emerging architectures, these languages tend to not be able to perform well and users are required to hand optimize until a new version of the underlying library is released or the language is updated. Therefore, Tangram focuses on supporting performance portability without relying on these traditional computational primitives. Further, Tangram potentially can become an internal library-building language for other library-based high-level languages.

3 Features

This section introduces the main features of Tangram, including transformation, performance tuning, performance portability, productivity, and maintainability.

3.1 Transformation

Compared to traditional compilers, which require recognition of loop structures through sophisticated analyses on flattened data structures, Tangram’s use of
Table 1. Primitive examples in Tangram

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>map($f,c$)</td>
<td>given a function (or codelet) $f$ and container $c$, the result is a container where $f$ is applied to each element in $c$</td>
</tr>
<tr>
<td>partition($c,n,start,end,inc$)</td>
<td>an input container $c$ is split into $n$ containers starting from a sequence $start$ and ending at a sequence $end$ with increment as a sequence $inc$, the result is a container of containers</td>
</tr>
<tr>
<td>zip($c1,c2$)</td>
<td>given two input containers of the same length, the output is a container of $(c1_i,c2_i)$ pairs</td>
</tr>
<tr>
<td>iter_product($c1,c2$)</td>
<td>given two input containers, the output is the Cartesian product of the containers</td>
</tr>
<tr>
<td>sequence($start,end,inc$)</td>
<td>outputs a container of integer sequence from $start$ to $end$ with increment $inc$</td>
</tr>
<tr>
<td>sequence($c$)</td>
<td>outputs an infinite integer sequence of the constant $c$</td>
</tr>
<tr>
<td>container_sequence($c,start,end,inc$)</td>
<td>given an integer container $c$ with indices from $start$ to $end$ with increment $inc$, the result is a sequence converted from the container.</td>
</tr>
</tbody>
</table>

Containers facilitate easy reasoning about transformations such as tiling, vectorization, privatization, blocking, coarsening, and thread-level parallelization [20]. The transformations are internally implemented within the map and partition primitives. Data layout transformation [22], locality-aware transformations, and scheduling can be achieved using strong locality hints implied by containers. Tiling exploiting architecture-specific memory, such as scratchpad or texture memory, can also be chosen during compilation.

Complicated transformations that are traditionally difficult in compiler frameworks can be described by programmers using substitute codelets. These transformations may include change of access pattern, application-specific transformations, or algorithm choices. Also, architecture- or environment-specific primitives can also be applied in substitute codelets with the env annotation. For example, a shuffle instructions in NVIDIA Kepler GPUs, SSE intrinsics in CPUs, or reduction primitives in MPI can be applied during synthesis in Tangram.

3.2 Performance Tuning and Portability

There are three tuning dimensions in Tangram. First, switching codelets can dramatically expand design spaces, compared to common high-level languages. For example, transforming a sequential to a vectorized codelet potentially can provide a finer-grained workload, which is infeasible in most refactoring tools. Furthermore, the env annotation enables tuning for specific environments or architectures while isolating the logic from these tuning parameters. Second, general loop transformations as described in the previous section can be tailored with proper granularities. Third, the tunable qualifier allows strong hints for parameterization from programmers, enabling use of adaptive-grained codelets. It also provides a smoother tuning curve than the coarsening techniques commonly used in OpenCL compilers [8, 9, 16, 17], since the coarsening techniques tend to
serialize independent workloads originally scheduled in parallel, while Tangram has more flexible adjustment to schedule workloads based on the _tunable qualifier.

A tuning-friendly programming language with rich design spaces in multiple architectures potentially can achieve high performance across different architectures. Therefore, performance portability in Tangram is achieved by retuning the program to another architecture. Specialized intrinsics or built-in primitives of a given new architecture can be easily introduced in Tangram by adding substitute codelets if necessary.

3.3 Productivity

Tangram, like most high-level languages, provides abstractions to reduce application complexity. Also, unlike OpenCL, Tangram benefits from codelet reuse within a kernel and even across kernels. Considering performance portability, OpenCL may require kernel rewriting to effectively adapt to different architectures.

3.4 Maintainability

Tangram is designed with user debugging in mind. Since optimizations are decoupled from logic, users express high-level intent rather than coding architectural details, which may introduce bugs and limit portability. Since codelets within a spectrum are supposed to be functionally equivalent, codelet substitution can be used for verification. Output difference from substitute codelets potentially would only arise due to bugs in the codelet implementation. Small variations, such as rounding errors, may happen among substitute codelets, and tolerance can be introduced during testing.

Building substitute codelets in Tangram can be considered as incremental improvement for a program. Since substitute codelets provide the same functionality, one codelet per spectrum is the only necessary part to generate correct results. Other languages may require programmers to rewrite the kernel to exploit a new architectural feature. Tangram only requires adding substitute codelets that materializes the feature. In this process, minimal code changes are required to adapt to new architectures.

Even when the applied algorithm varies in a given program, codelets might still be reusable across algorithms. Also, specialized intrinsics or primitives for an architecture can be introduced by adding new substitute codelets. These would propagate up to the compound codelets garnering benefits.

4 Example and Evaluation

Tangram is currently implemented in C++ using a directive-based approach, with the compiler implemented using Clang as a source-to-source compiler. Access pattern analyses in containers consider only stride-one accesses in all dimensions. A kernel can be synthesized by inlining multiple instances of one codelet
Fig. 2. Performance of 64M-integer Sum Reduction with a map primitive. Codes across different domains are evaluated to demonstrate the features of Tangram. Tangram currently works in a shared memory model, and in terms of output source codes, Tangram currently supports C with OpenMP for a multicores CPU, and CUDA for NVIDIA GPUs. All evaluations are performed on a NVIDIA Tesla C2050 GPU and a quad-core Intel Xeon E5520 2.2Hz CPU, with -O3 flags in gcc 4.4.5, nvcc 4.2, and icc 13.0.1.

4.1 Reduction

Reduction is a simple example to illustrate Tangram codelets and the transformations performed. In Reduction, four substitute codelets, shown in Fig. 1, are written to represent the spectrum. Fig. 2(a) and (b) show synthesized results against manually optimized Reduction. Tangram can achieve almost 100% of performance against the manually optimized program in both architectures. For GPUs, Tangram selects a four-level hierarchical reduction, which is an adjacent-tiled reduction, shown in Fig. 1(b), containing stride-tiled reductions, shown in Fig. 1(d), as both inner and outer reductions. In each stride-tiled reduction, a vectorized and a sequential reduction form outer and inner reductions, respectively. On the other hand, for CPUs, Tangram selects a two-level hierarchical reduction, which is an adjacent-tiled reduction containing sequential reductions as both inner and outer reductions. Particularly, the adjacent-tiled reduction is parallel-friendly for multi-processors or multi-cores, and the stride-tiled reduction is vectorization-friendly for GPUs due to its stride-one memory access.

We also evaluate the results by removing the stride-tiled compound codelet from the spectrum. Tangram is robust enough to find an alternative kernel and achieves up to 85% of performance against the manually optimized code on GPUs. This demonstrates robustness of both built-in general loop transformation and codelet switching.

4.2 DGEMM

Double-precision General Matrix Multiplication (DGEMM) represents a common pattern of computation in scientific computing. It is chosen to demonstrate robustness of built-in tiling and coarsening transformation using high-level abstraction with containers. Fig. 3(a) shows synthesized GPU results against DGEMM in cuBLAS library, hand-optimized and naive implementations in the Parboil benchmark suite [21]. Tangram achieves up to 93.6% of performance against the well-tuned cuBLAS code, and outperforms the manually optimized and naive versions in Parboil by factors of $1.6 \times$ and $8.3 \times$, respectively.
Fig. 3. Performance of DGEMM with two 2K-by-2K matrices

Fig. 3(b) shows the corresponding Tangram CPU results against (multi-threaded) DGEMM in MKL library, and the OpenMP implementation in Parboil. With an extra SSE-intrinsic codelet, Tangram achieves up to 86.4% of performance against the well-tuned MKL DGEMM and outperforms the OpenMP version by a factor of $15.9 \times$. Without the SSE-intrinsic codelet, Tangram shows 50.9% of performance against MKL and outperforms the naive OpenMP version by a factor of $9.3 \times$. This shows that incremental improvement by introducing SSE intrinsics in the codelets dramatically boost the performance, while codelets without SSE intrinsics still gain reasonable benefits from tiling and blocking in Tangram compiler, and auto-vectorization in icc.

4.3 SpMV

SpMV (Sparse Matrix-Vector multiplication) is one of the most important building blocks in sparse linear algebra. It is chosen to demonstrate how containers can simplify analyses. Two SpMV versions with CSR (Compressed Sparse Row) and ELL (ELLPACK) formats are implemented to demonstrate cache selection and data placement, instead of automatic SpMV format selection. Implementing two versions can evaluate code reuse in a software development process. The only codelet that is different (out of 7) between CSR and ELL SpMV is shown in Fig. 4. Both CSR and ELL SpMV codelets compute sparse dot products using the same codelet (\texttt{spvv\_dotproduct}). However, due to different matrix formats, CSR and ELL SpMV codelets have different data partitions for accessing two containers holding matrix entries and column indices. Most information of memory accesses in CSR SpMV is data dependent and cannot be analyzed by the compiler due to indirect accesses, but abstraction of CSR SpMV still indicates critical information in \texttt{sequence(1)}, which means adjacent memory accesses within an instance (a row in this case) of a map primitive. In ELL, abstraction provides a critical hint in \texttt{sequence(...,1,...)}, which means adjacent memory accesses among adjacent instances of the map primitive. Tangram uses this information to synthesize different SpMV kernels, though both share almost the same codelets.

Tangram examines the lambda function (in the first two lines) and discerns that all instances of function applications share the same container \texttt{in\_vec}. This implies that the data of \texttt{in\_vec} should be cached and shared across the computation to garner a shorter latency and a higher bandwidth. Data may have irregular or indirect accesses, which can be inferred by analyzing the used codelet (\texttt{spvv\_dotproduct}). Architectures may provide memory structures to support
this irregular accesses. In this case, texture memory in GPUs potentially can be applied.

Fig. 5(a) and (b) show generated Tangram results against CUSparse and MKL libraries. Tangram can achieve almost 100% of performance against the existing libraries.

5 Current Limitation and Future Work

As shown in Section 4.1 and 4.2, a spectrum of codelets might impact performance of a program. Building a complete and robust spectrum may require expert’s efforts or thorough understanding of applications. It might therefore be challenging to a novice to come up with a minimal set of codelets that cover the algorithm and result in speeds that compete with the optimal libraries.

Redundant codelets within a spectrum might reduce maintainability and increase complexity of autotuning. Redundant codelet removal could alleviate such overhead using techniques such as AST-level preprocessing.
The current autotuning process is built in Tangram. Future work would introduce a description language for autotuning, so this would allow Tangram to either interact with other autotuning frameworks or support new architectures easily.

6 Related Work

Tangram is inspired by multiple high-level languages [2, 5, 7, 15]. Triolet [15] for example provides a similar language design based on list comprehension, but it adopts Python, instead of C++, and does not allow fine-tuned modification of the implementation syntax as can be done using codelets.

DSLs [4, 12–14] potentially can achieve all features Tangram can provide in their own domains. However, they are not general to cover applications from other domains. Rewriting-based languages [2] or libraries [13] have similar abilities to Tangram. They heavily rely on rewriting rules to seek performance gains. Tangram keeps balanced between rewriting rules and traditional optimizations, including loop transformations, thread scheduling, granularity adaptation, and data placement, and then achieves both high performance and productivity.

Conventional OpenCL compilers [6, 8–10, 16–19] rely on code patterns and use sophisticated analyses and transformations to optimize for specific architectures. Tangram inherits a certain degree of those transformations but simplifies analyses using abstractions and containers. PEPPHER [3] maintains multiple implementations of kernels across architectures, while Tangram maintains both architecture-dependent and architecture-independent codelets to synthesize kernels across architectures.

7 Conclusion

We present Tangram, a general-purpose, high-level language to support future parallel programming on multi-/many-core architectures. It achieves high performance across multiple architectures and provides productivity and maintainability by separating architecture-specific optimizations from computational pattern design, and embracing tuning-friendly interface, abstract containers, and codelet substitution. Tangram’s synthesized programs are comparable in performance to existing well-optimized counterparts on both CPUs and GPUs.

8 Acknowledgements

This work is partly supported by the Starnet Center for Future Architecture Research (C-FAR), and the UIUC CUDA Center of Excellence.

References

Task Parallelization as a Service
A Runtime System for Automatic Shared Task Distribution

Luc Bläser
HSR University of Applied Sciences Rapperswil
IFS Institute for Software
lblaeser@hsr.ch

Abstract. The native parallelization support in predominant programming languages focuses on local cores, while mostly neglecting the huge parallelization potential of distributed execution. We have therefore developed a runtime system that automatically distributes ordinary shared memory parallel tasks into the cloud, to execute them on a large number of cores, e.g. on a cluster. The runtime mechanism takes care of transmitting the necessary task code and data to the service, as well as of propagating task results and side-effect changes back to the client memory. For programs with long-running tasks or a high amount of tasks, the system is able to soon compensate the network transmission overheads and thereafter scale with the number of tasks up to the available server-side cores.

1 Introduction

Today’s mainstream programming languages principally target only on local multi cores with their institutionalized parallelization features, as it fits well to their underlying shared memory model. However, moving from local to distributed parallelization, imposes the substantial extra burden for the developers to leave their conventional programming model and close the gap to distribution on their own, e.g. by redesigning the programs for specific distribution frameworks (e.g. service, remoting, or grid computing architectures) or by engaging a different more distribution-friendly programming paradigm (e.g. descriptive dataflow models).

It is therefore of no surprise that parallelization in daily programming practice largely concentrates on utilizing local processors, while mostly neglecting distributed parallelization - unless there is a strong performance urge justifying the efforts of a corresponding dedicated solution. Another strong distribution obstacle is the fact that developers and users normally do not have remote processor power at hands, although many clusters and distributed systems would have free capacities – but unfortunately without an easy-to-use “parallelization-as-a-service” interface.

There has been intensive research done in this area, principally going into three main directions: (1) new/different programming models inherently suited
for distribution, such as Actors/MPI [5, 1], or dataflow/query models [13, 3], (2) distributed task/thread frameworks, e.g. in the grid computing area [6, 10, 12], and (3), distributed shared memory systems [9, 2, 15, 4]. While the first direction takes the more radical approach of tackling the distribution impedance already at its roots, it usually compels programmers to apply the different paradigm on top of or aside their ordinary imperative shared memory programming language. The second direction, represented by the grid computing or distributed thread/task frameworks, typically leads to visible seams in the program design: it necessitates explicit task and data offloading, marking data serializable, wrapping data in specific sub-classes and so on. The third direction, distributed shared memory systems, transparently enables distribution of normal shared memory programs across machines. Although, it is usually heavy-weight for this purpose, operating on the whole program rather than selectively on the distributed parallel parts. A more detailed survey of related work is given in Section 5.

Our research goal is to significantly ease distributed parallelization by providing a new runtime system that allows scaling up parallel programs seamlessly on massive processor power in the cloud, i.e. with the same programming model as for local parallelization and without requiring any explicit code and data transmissions. For this purpose, we propose an enhanced thread pool mechanism that transparently integrates remote multi-processing: Although described like conventional local parallel tasks operating on shared memory, the runtime system automatically distributes the tasks to a web service into the cloud. Behind the service, the tasks are to be executed on a large number of cores before their results and their potential effected memory changes are finally sent back to the client runtime. The web service is of our design and can abstract an arbitrary parallel processing infrastructure behind the interface, e.g. a high-performance computer cluster. We have realized this system for the .NET framework, to demonstrate the concept by the example of a popular shared memory programming platform.

The remainder of this paper is structured as follows: Section 2 describes the programming model of distributed parallel tasks. Section 3 explains the design and implementation of the runtime system. Section 4 presents performance and scalability results. Section 5 compares distributed task parallelization to related work. Section 6 finally draws a conclusion of this work.

2 Programming Model

Task parallelization as a service encourages programmers to implement and start distributed parallel tasks that can be dispatched and executed on remote processors.

2.1 Distributed Tasks

In our system, which is based on .NET, distributed tasks can be programmed like conventional local thread pool tasks offered by the .NET task parallel library [8].
A distributed task is implemented as an ordinary .NET delegate\(^1\) or lambda\(^2\). In principle, working with distributed tasks remains analogous to using local parallel tasks, i.e. they can be instantiated, started, and joined. Certain restrictions apply for distributed tasks: Inner synchronization and calls to IO are for example forbidden. A detailed explanation of restrictions is provided in subsequent sections.

Figure 1 shows a code example for factorizing a set of numbers in parallel, each number being factorized as a separate distributed task. No extra compilation step is involved here; adding a reference to the library of our cloud task parallelization is sufficient for the runtime mechanism. The code sample looks very similar to the local task parallelization, as depicted in Figure 2. The URL and access authorization code need to be specified in advance for the remote task parallelization service, before a set of tasks can be started. Accessing task results blocks as long as the corresponding task is not terminated, where task faults are propagated as exceptions. We deliberately did not unify the local and distributed task class because we would like to encourage explicit combined starts of multiple distributed tasks for reducing network roundtrips, whereas the existing local task class promotes starting one-by-one.

For increased convenience, distributed tasks can also be applied in the form of data parallelism (Figure 3), by using parallel invocations or parallel loops. As for a parallel loop, each loop step starts a distributed task that executes the body and is joined again at the loop end.

Alike local tasks, distributed tasks are allowed to also perform side-effect changes on disjoint memory locations in shared memory, as also illustrated in Figure 3. The modifications in the array `outputs` of the example become

---

\(^1\) A .NET delegate is a reference to a method and an associated object.

\(^2\) A .NET lambda is an anonymous delegate in the form of an inline statement or expression with access (closure) to variables of its lexical scope.
```csharp
var taskList = new List<Task<long>>();
foreach (var number in inputs) {
    var task = Task.Factory.StartNew(
        () => Factorize(number)
    );
    taskList.Add(task);
}
foreach (var task in taskList) {
    Console.WriteLine(task.Result);
}
```

**Fig. 2.** Analogous solution with local parallel tasks.

```csharp
distribution.ParallelFor(0, inputs.Length, (i) => {
    outputs[i] = Factorize(inputs[i]);
});
```

**Fig. 3.** Distributed data parallelism.

automatically visible after task completion. For this purpose, the runtime system collects side-effect changes of tasks at the server side and propagates them back to the client-side memory. The system detects certain data races, as described in the next section.

### 2.2 Task Isolation

Distributed tasks are required to be independent of all other active tasks and threads, i.e. read-only accesses on shared variables and arbitrary accesses on non-shared variables are allowed. The granularity of variable accesses is per field or array element. Notably, this does not constitute a strong limitation because for local task code, synchronization in task execution is usually also avoided for highest possible performance. This applies for both synchronization primitives and memory model atomicity/visibility. The demanded task isolation eases the distribution significantly, since it excludes information flow between active distributed tasks, as well as, between active distributed tasks and the remaining program code. Our system detects certain violations of task isolation, namely when tasks employ synchronization, or when write-write conflicts happen due to data races. Read-write conflicts are not detected though: The reading task will not see the change of another concurrent task (snapshot isolation). In contrast, data races in local concurrency are not detected at all, while our system detects at least write-write conflicts.

### 2.3 Security Concerns

The runtime system prevents distributed tasks from directly or indirectly executing IO operations, system calls, reflection or unsafe/unmanaged .NET code. IO and system calls are not allowed because we do not delegate these calls back to the client, such that they would otherwise become effective on the remote machines. If reflection and unmanaged code would not be forbidden, programmers
could accidentally or intentionally inspect or modify arbitrary program state or corrupt memory safety at the remote side.

3 Runtime System

The system for distributed task parallelization consists of three components: a client runtime library, the cloud processing web service, and a server runtime library.

3.1 Processing Roundtrip

The processing of distributed tasks involves the following steps, as illustrated in Figure 4: (1) The potentially executed program code and accessible data of the invoked tasks are collected and serialized by the client side library at runtime. (2) The serialized code and data are shipped to our web service which represents the cloud processor resources. (3) The service distributes the tasks on server-side compute nodes, currently by launching a HPC cluster job consisting of a HPC task per input .NET task, i.e. by using the default task scheduling of the cluster. (4) The code and data are deserialized and instantiated by the server runtime library on each server compute node. (5) The remote tasks are executed on the compute nodes. (6) When terminated, the results and modified data of tasks are collected and serialized by the server runtime library. (7) The serialized data of task completion is sent back to the initiating client over the web service. (8) The updates are finally made effective in local memory of the client.

3.2 Task Serialization

When tasks are started for distribution, the client runtime component serializes the necessary code and data in two phases by way of reflection.

In a first phase, a conservative context-insensitive code analysis determines all reachable program code. Starting from the task delegate, the transitive closure of potentially directly or indirectly invoked methods is calculated. Additionally, it records all potentially used classes and accessed fields within the reachable methods. The code of each visited method is examined to only contain supported instructions and calls according to the security constraints (the system triggers a runtime exception if the code cannot be distributed). The set of reachable methods and their intermediate language code is eventually serialized.

In a second phase, all potentially accessed task data is collected. For this purpose, the system generates a partial heap snapshot, being the graph of objects that are reachable via references from the task delegate, by only considering the references occurring in potentially accessed fields according to the preceding code analysis. For the collected objects, only the state of accessible fields needs to be serialized. Besides the objects, the snapshot also includes static fields and constants that can be used by tasks. Because of the required task isolation, the runtime serialization delivers a consistent state without need of synchronization,
i.e. the system never blocks other running threads. Due to the conservative analysis, the snapshot may, however, include data that is not effectively accessed by the tasks and therefore also not required to be isolated: the state of this data may be inconsistent though but it is also never accessed by the distributed tasks.

### 3.3 Task Results

The server runtime component returns all necessary information of completed tasks, such as the task delegate result value, modifications on transmitted objects and static data (updates of fields and array elements), as well as, all reachable new objects that have been created by the remote task execution. The client in turn performs the in-place updates on arrival of the task completion information, i.e. modifications are applied to the corresponding objects and static fields of its input snapshot. We perform change detection by comparing the field and array element state before and after task execution. With this approach, the client runtime also detects certain data races, namely illegal write-write conflicts across distributed tasks. If remote task execution fails (e.g. due to a thrown exception), an aggregated exception is propagated to the client.

### 3.4 Service Design

Task code (program metadata and intermediate language code) and data (object graph and static fields) are encoded in an own binary format to reduce client-to-service traffic as much as possible. The service functionality basically comprises
two operations, one for starting a set of tasks and another for awaiting the termination of a set of tasks. To reduce network roundtrips, multiple tasks can be sent in one bunch, where the task instances can also share the same task code. To support secured network transmission, a HTTPS service binding can be used.

4 Experimental Results

Distributed task parallelization is intended for running computing-intensive tasks and/or a large amount of tasks, offering a high potential of parallelization.

4.1 Measurement Setup

For an experimental evaluation of our current system version, a set of synthetic parallel problems have been implemented on the basis of distributed tasks and eventually run in an environment with a MS HPC computer cluster behind the cloud service. The cluster comprises 32 nodes with 12 Intel Xeon cores, 2.6 GHz each (of which we were allowed to use 100 cores, 8 nodes with 12 cores plus 1 node with 4 cores for our experimental study). The client and web service each run on an Intel processor, 2 cores, 2.9 GHz machine, with 100Mbit/sec bandwidth and 1ms network delay between client, service and the cluster. All measurements have been performed by using compiler-optimized 64-bit .NET program assemblies. For all runtime results, the minimum of three repeated runtimes is considered, to reduce negative influences of temporary network speed fluctuations.

4.2 Performance Scalability

To study the performance scalability, we measure the runtime for a set of independent computation tasks. To start with a first scenario, we compute the factorization of a set of sample 64-bit numbers, each composed of two random prime factors around $2^{32}$. Naturally, the same random seeds and numbers are used to ensure reproducible measurements. Each number is factorized independently in a parallel task. The comparison involves three processing approaches: (1) with distributed tasks, (2) with local tasks, and (3) sequential execution. Figure 5 shows the runtime in seconds depending on the amount of input numbers, which is equal to the number of parallel tasks. As expected, the parallel speedup of distributed tasks scales linearly with the number of tasks – in this scenario, each task runs on a separate instance of the 100 available cores. Local parallelization only offers a speedup of 2 on the two core client machine. Of course, the speedup also depends on the number of free cores available in the cluster.

4.3 Cost Breakdown

The runtimes for distributed task parallelization involve different performance cost factors, which vary from problem to problem: (1) the effective task execution time on the server side, (2) the network transfer time from the client over
Fig. 5. Performance scaling by number of tasks.

<table>
<thead>
<tr>
<th>Runtime costs (seconds)</th>
<th>Factorization (10 numbers)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node execution</td>
<td>18.4</td>
</tr>
<tr>
<td>Network transfer</td>
<td>1.9</td>
</tr>
<tr>
<td>Cluster dispatching</td>
<td>0.3</td>
</tr>
<tr>
<td>Task serialization</td>
<td>0.3</td>
</tr>
</tbody>
</table>

Table 1. Runtime costs breakdown.

the service to the cluster, (3) the cluster dispatching costs, and (4) the accumulated effective overheads of our runtime mechanism, that is task serialization, deserialization, and change/result propagation. Table 1 depicts the breakdown of runtime costs in seconds for the factorization of 10 numbers, i.e. for 10 tasks. In this scenario, task execution represents the most significant part: this is the time where tasks are executed in parallel on the cluster. Network transfer constitutes the second most substantial portion. The remaining cost factors, including our own runtime mechanism, are relatively small.

4.4 Performance Comparisons

For a more general performance comparison, we evaluate the runtimes for different problem cases: (1) Mandelbrot fractal computation for a specified image size, as a representative of a parallel problem with a higher data amount compared to the task computation time. (2) Knight tours computation on a chess board of a specified size, as a representative for relatively long-running task computations. (3) Primes scanner counting primes in a specified number range, as a representative of relatively short-running tasks. Table 2 shows the runtimes in seconds, rounded to two significant figures, for the specified instances of these problems. We again compare distributed task parallelization (100 cores), local task parallelization (2 cores) and sequential execution. Once more, significant performance improvements can be achieved with the runtime support of dis-
<table>
<thead>
<tr>
<th>Runtime (seconds)</th>
<th>Mandelbrot (10000 x 1000)</th>
<th>Primes Scanner (range $10^7$)</th>
<th>Knight Tours (6 x 6 board)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distributed</td>
<td>8.0</td>
<td>4.7</td>
<td>120</td>
</tr>
<tr>
<td>Local parallel</td>
<td>20</td>
<td>9.2</td>
<td>1100</td>
</tr>
<tr>
<td>Sequential</td>
<td>37</td>
<td>19</td>
<td>2200</td>
</tr>
</tbody>
</table>

Table 2. Performance comparison of parallel problems.

As expected, the examples confirm that the runtime system is able to reach a high parallel speedup by the large amount of remote cores. However, the gain of parallelization needs to compensate the involved overheads, which are primarily the network transmission time, depending on the size of task serialization, the data bandwidth and network delay. Distributed task parallelization is therefore generally beneficial if a large amount of tasks is executed, tasks are running sufficiently long, or tasks entail relatively low data transfer.

5 Related Work

5.1 Distributed Data Parallelism

Microsoft DryadLINQ [13], built on the distributed runtime engine Dryad [7], has close relation to our system: It permits automatic distributed processing of .NET LINQ [11] queries on clusters. While this model is oriented on descriptive programming of distributed dataflows in terms of queries, our system promotes more imperative task or data parallelization. Therefore, our system also allows distributed tasks to perform side-effects or changes that are propagated to the client, while the only backflow in DryadLINQ evaluations are the query results. Side effects of delegates inside the queries of DryadLINQ are ignored.

MapReduce [3] and in particular, also the Hadoop MapReduce implementation\(^3\), are popular dataflow programming models for high-scale distributed parallelization. The integration in a client program is, however, less seamless than in our model: Data is to be explicitly passed to the map and reduce functions from files or serializeable key-value sets. This is different to the shared memory illusion of our model, where the data of distributed tasks is automatically transmitted. MapReduce does also not directly incorporate a cloud approach where clients can easily offload their dataflows to a service. Though, such architecture can be designed around.

\(^3\) [http://hadoop.apache.org](http://hadoop.apache.org)
Other grid computing systems such as Pegasus[4] and Swift [14] also facilitate DAG-like task workflow distribution on cloud computing resources but again with explicitly programmed data and task transmission. CIEL [12] supports powerful parallel task workflows with dynamic task spawning implemented in a specific language (Skywriting). Tasks can trigger batch commands, or invoke Java/.NET code in a less transparent way than our system: by denoting the class name and passing arguments and results.

5.2 Distributed Task Parallelism

Existing distributed thread/task programming frameworks, such as JPPF[5], Hadoop, ProActive Parallel Suite[6], the already mentioned CIEL Skywriting [12], Alchemi [10], Manjrasoft Aneka .NET Tasks[7], and many others, make distribution significantly more visible than in our system: heap data from the client program is not automatically shared across distributed processes but must be passed as explicitly serializeable objects within task parameters and results, or has to be managed in specific grid heaps or distributed data collections. However, the focus of our system is on enabling mostly seamless and convenient task parallelization on remote processor resources.

5.3 Message Passing Models

The Actor model [5, 1] facilitates inherent distribution of active instances (actors) across machines, because actors only interact via explicit message communication and do not share memory. If applied within a conventional shared memory language (e.g. with MPI), this indispensably provokes a semantic gap, since programmers need to think in a different paradigm than the native language and have to stick to particular conventions. For example, actor communication must not be bypassed by ordinary references.

5.4 Distributed Shared Memory

Various systems have realized virtual shared memory on distributed computers, be it at the operating system level [9, 4] or at the runtime system of a programming language [2, 15]. While this can establish automatic distribution of an entire program, our system employs distribution only selectively for task parallelization. Moreover, we provide the distribution as a service for use by a possibly open group of clients.

---

6 Discussions and Conclusions

The presented runtime system enables seamless distributed task parallelization with the illusion of shared memory. While the programming model remains principally identical to working with local parallel tasks, the runtime engine automatically dispatches tasks over a service onto remote processor resources in the cloud. In contrast to other less seamless systems, this liberates developers from any distribution-specific programming artefacts, such as developing explicit remote code, realizing explicit communication, implementing any serialization, or wrapping/marking/attributing code or data for distribution-awareness.

Of course, distributed task parallelization is not appropriate for all classes of parallel problems. It is rather designed for computing-intensive tasks or a large amount of tasks, where it can achieve very high speedups. Thereby, the total task execution time has to be significantly larger than the network-dependent transmission time of task data between the client and the service.

We see a high potential if programmers can use “parallelization-as-a-service” in a way that is as simple and convenient as our task parallelization in the cloud. Certainly, there is room for various improvements that we would like to address in future: (1) The runtime system could be enhanced to support more features, especially nested task starts, task chaining, task canceling, as well as, remote monitoring and debugging. (2) It could be investigated on alleviating task isolation by permitting well-defined synchronizations across tasks. (3) It would be interesting to offer a public parallelization service where users can directly consume and perhaps also offer multi-processor power on demand.

Acknowledgment

I gratefully appreciate the feedback from Peter Sommerlad, Svend Knudsen and Thomas Corbat, helping to improve this paper. I also express my thanks to Henrik Nordborg from the Microsoft Innovation Center for Technical Computing for offering access on the Microsoft HPC Cluster at the HSR during the experimental evaluations.

References

Performance and Energy Consumption
Evaluation of the Client-Server Synchronization
Model for Concurrent Lists in Embedded Systems

Lazaros Papadopoulos¹, Ivan Wahulya² Paul Renaud-Goud², Philippas Tsigas², Dimitrios Soudris¹, and Brendan Barry³

¹ School of Electrical and Computer Engineering, National Technical University of Athens, Greece
{lpadop,dsoudris}@microlab.ntua.gr
² Computer Science and Engineering, Chalmers University of Technology, Gothenburg, Sweden
{ivanw,goud,philippas.tsigas}@chalmers.se
³ Movidius Ltd., Dublin, Ireland
brendan.barry@movidius.com

Abstract. Modern multicore embedded systems target devices like smartphones and wearable gadgets in which power efficiency is an important design constraint. A major issue in respect with the applications executed in such systems is the synchronization of the concurrent accesses in shared data structures. Embedded systems synchronization solutions are usually based on mutexes that provide poor scalability and, most important, they lead to high energy consumption. In this work, we propose an energy efficient client-server synchronization model for embedded system architectures, which is loosely related to similar solutions proposed in the High Performance Computing domain. Our results show that our algorithm achieves comparable performance against the mutex-based solutions for the queue and the ordered list data structures, while it leads to reduced power consumption by up to 18.6%.

Keywords: Multicore platforms, Concurrent data structures, Energy efficiency

1 Introduction

Mutexes are widely used in embedded systems for the synchronization of memory accesses on shared data. Their simple control mechanism allows the convenient design of lock-based concurrent data structures. However, they suffer from well-known problems like poor scalability and deadlocks. Additionally, in the embedded systems there exist the major issue of the low energy consumption design constraint. Mutexes designed as spinlocks lead to high energy dissipation due to polling. Therefore, as the number of integrated cores in the embedded systems
is constantly increasing, alternative means of synchronization should be evaluated. The traditional lock-based data structures based on mutexes will become inefficient, since they will suffer from poor scalability, high energy consumption and unfairness issues.

Atomic operations or similar low level synchronization primitives, which lead to lock-free data structure designs, have not been widely adopted yet to embedded systems. One reason is that the extra hardware required for controlling these mechanisms may hinder the adoption of such solutions for embedded systems with very low power requirements (e.g. chips utilized as coprocessors in high-end mobile devices).

In the last years, the High Performance Computing (HPC) and the embedded system domains tend to face similar challenges [1]. The main reason is the fact that embedded systems integrate more and more cores on a single chip. Therefore, the efficient synchronization for accessing shared data structures, which has been always a design challenge in the HPC domain, is now a challenge for embedded systems, as well. On the other hand, the power dissipation, which is a traditional design constraint in the embedded systems, is recently a major design challenge towards the exascale computing [2]. The mutex limitations have been extensively studied in the context of the HPC domain and several alternatives have been proposed. The ever-increasing number of integrated processors in the embedded systems imposes now similar challenges. Therefore, it is reasonable to adopt solutions for efficient synchronization of shared data that are proposed for the HPC domain and evaluate them in the embedded systems.

Towards this end, we try to face the problem of efficient synchronization of concurrent list data structures in embedded systems, inspired by solutions proposed in the HPC domain. We propose a client-server model synchronization for embedded systems, loosely related to the Remote Core Locking algorithm proposed in [3]. We apply the algorithm in two of the most widely used concurrent data structures, the queue and the ordered list. The embedded platform in which the algorithm is evaluated integrates the 8-core Myriad chip [4], which targets modern mobile devices. By taking advantage of the Myriad chip hardware specifications, we managed to achieve comparable performance against the corresponding lock-based list designs, while the average power consumption is much lower.

The rest of the paper is organized as follows: In Section 2 we describe the related work. Next, we provide a summary of the technical specifications of the Myriad chip. The details of the queue and ordered list implementations are described in Section 4. The experimental results are presented and analyzed in Section 5. Finally, in Section 6 we draw our conclusions.

2 Related Work

A first evaluation of the client-server model in concurrent queues is presented in [5]. It describes a set of concurrent queue implementations based on a client-server algorithm, which is applicable only to concurrent queues. In this work
we extend this approach in two ways: First, we improve the algorithm to be applicable not only to concurrent queues, but to any list data structure (e.g. stacks, ordered lists etc.). Second, we present the evaluation of the algorithm apart from the concurrent queues to ordered lists, as well. We intend to show that the client-server model is not limited to queues, but it can be extended to be applied to more concurrent data structures and provide performance and energy efficiency.

The client-server approach is loosely related to the Remote Core Locking technique that is proposed for HPC in [3]. The main idea is the utilization of a dedicated server that executes the critical sections of the application. A similar technique is the Flat Combining algorithm in which the role of the server is played by all threads in a periodical manner[6].

Several hardware based synchronization techniques have been proposed for embedded systems: For instance, the CLock approach tries to combine the advantages of both locks and Transactional Memory by detecting conflicts [7]. Synchronization-operation Buffer is a hardware block targeting the minimization of blocking operation [8]. However, in this work we try to achieve efficient low power and high performance synchronization of the accesses of shared data by exploiting the embedded system’s specifications, without the need of extra hardware.

Finally, although there are many works in the concurrent data structure area that propose lock-free data structures, which utilize low-level atomic primitives (e.g. compare-and-swap) [11], these are not yet widely supported in embedded systems.

3 Myriad Platform Description

The Myriad chip is an heterogeneous multiprocessor system, designed to act as a co-processor connected between a sensor system such as a set of cameras and the host application processor [4]. It is designed to perform the heavy processing on the data stream coming from a sensor system and feed the application processor with relevant metadata. The Myriad chip is utilized in the context of Project Tango, which aims at the design of a mobile device capable of creating a 3D model of the environment around it [10].

Myriad integrates a 32-bit SPARC V8 RISC processor core (LEON3) utilized for managing functions such as setting up process executions, controlling the data flow and interrupt handling. Computational processing is performed by the Streaming Hybrid Architecture Vector Engine (SHAVE) 128bit VLIW cores with an instruction set tailored for streaming multimedia applications. The Myriad SoC integrates 8 SHAVE processors as depicted in Fig.1.

In respect with the memory specifications, the platform contains an 1MB on-chip SRAM memory (named Connection Matrix – CMX) with 128KB directly linked to each SHAVE processor and provides local storage for data and instruction code. Therefore, the CMX memory can be seen as a group of 8 memory “slices”, with each slice being connected to each one of the 8 SHAVEs. The
CMX memory is accessible to all SHAVEs and to LEON processor, as well. Myriad avails also a 64MB off-chip memory and a small memory dedicated to LEON (LRAM).

3.1 Myriad Synchronization Primitives

Myriad platform avails 8 mutexes, which are implemented as spinlocks. This means that each SHAVE spins to the associated Test-and-Set register until it manages to acquire the mutex. Analysis of experiments on the Myriad platform shows that the mutex implementation is a fair lock with round-robin arbitration. It is important to state that no other kind of locking is available in Myriad, apart from the hardware-implemented spinlocks. There is no support for any atomic primitive or any other lock-free mechanism.

To achieve fast arbitration between the SHAVEs, Myriad chip avails a specific set of registers dedicated to this purpose. Each SHAVE has its own copy of these registers and its size is 4x64 bit words. They are accessed in a FIFO pattern, so each one of them is called "SHAVEs FIFO". Each SHAVE can push data to the FIFO of any other SHAVE, but can read data only from its own FIFO. A SHAVE writes to the tail of another FIFO and the owner of the FIFO can read from any entry. If a SHAVE attempts to write in a full FIFO, it stalls and enters a low power mode by shutting down several of its own power islands. It immediately recovers when the FIFO becomes non-full. Finally, LEON does not avail FIFOs. SHAVE FIFOs can be directly accessed by the programmer and can be utilized to achieve efficient synchronization between the SHAVEs. They provide an easy and fast way for exchanging data directly between the SHAVEs (up to 64 bits per message), without the need to use shared memory buffers.
4 Client-Server Model Description

In this section we describe the concurrent list implementations we evaluated on Myriad platform in the context of this work: a queue and an ordered list. Concurrent queues are extensively used in work-stealing algorithms for efficient task scheduling, while the concurrent ordered list is utilized in streaming aggregation implementations [9]. Both implementations were designed as linked lists accessed by all SHAVEs.

The client-server algorithm is based on the idea that a server arbitrates the access requests to the critical sections of the application. In our context, a dedicated SHAVE plays the role of the server, while the rest ones are the clients that perform operations on the shared data structure. Instead of allowing the clients to access the data structure directly, they notify the server about the type of operation they need to perform. The server completes the requested operation and responds by sending an acknowledgment to the client (along with operation-specific information, if necessary). The communication between the server and the clients is message-based.

To design a message-based communication in Myriad, there are two approaches we can follow: utilizing communication buffers allocated in CMX memory or take advantage of the SHAVE FIFOs. As shown in [5] a SHAVE spinning on a memory location, is inefficient in terms of energy consumption and performance. On the other hand, communication based on messages being exchanged through the SHAVE FIFOs is very low power, since the SHAVEs stall when the FIFOs are full. Additionally, access to SHAVE FIFOs is not expected to require more cycles than the accesses to the CMX memory.

Fig. 2 shows the memory allocation of the list in the CMX. SHAVE0 is the server, while the rest of the SHAVEs are clients. Each client allocates the list elements it constructed in its own CMX slice. The server stores in the list the addresses of the allocated elements, rather than the actual elements. The locality we achieve with this technique improves both the performance and the energy efficiency of the concurrent list, since both the client and server access only their local CMX memory slice.

The implementation of the algorithm is presented in Fig.3, along with an example. Client0 sends the address of the newly created element e3 to the server’s FIFO. After the insertion, the server responds with an acknowledgment (insert_fin). It is important to state that the client is not required to wait until it receives the acknowledgment. Instead, it can proceed with other computations, thus increasing the parallelism. Client1 requests an element from the concurrent list. Then, it stalls until it receives a response to its own FIFO from the server. The server extracts the element e0 and writes its address to Client1’s FIFO. It is important to state that there is no need for the clients to access the head or the tail of the list. Head and tail pointers are accessed only by the server and since they are allocated in the server’s memory slice, the list implementation is both performance and energy efficient.

Although the client-server model serializes all operations in the shared data structure, the communication based on SHAVE FIFOs is very energy efficient.
Fig. 2. Concurrent List memory allocation in Myriad

Fig. 3. Client-Server implementation
and provides adequate performance, as shown in the next section. We argue that it is a feasible solution for the synchronization of concurrent data structures that do not provide much parallelism (e.g. queues, stacks, ordered lists etc.). On the contrary, we expect it to be inefficient in terms of performance, compared with the lock-based solutions, for data structures in which increased parallelism can be achieved (e.g. hash tables). However, it can be a viable alternative to coarse-grained locking or to concurrent data structures that utilize a relatively small number of lock instances.

In comparison with the client-server algorithm presented in [5], we notice that this model can be applied not only to queues but to any list data structure, queue, stack or ordered, since the server is the one that handles the element ordering. The main idea of the algorithm is that it focuses on transferring computational overhead from the clients to the server. Thus, while the server inserts or removes an element to/from the shared data structure, the client can perform other computations and thus the parallelism increases. It is important to state that by presenting this model we do not aim to provide a mutex-replacement solution, since the parallelization achieved by the mutex utilization cannot be achieved with the client-server model. Instead, we propose an alternative solution that in many cases provides similar performance results and low power consumption by avoiding the polling on mutex registers. Therefore, we argue that it should be evaluated along with other concurrent data structure designs.

5 Experimental Results

The list implementations we evaluated in this work are the queue and the ordered list. They were evaluated using a synthetic benchmark, which is composed by a fixed workload of 20,000 random operations and it is equally divided between the running SHAVEs. In other words, in an experiment with 4 SHAVEs each one completes 5,000 operations, while in an experiment with 8 SHAVEs, each one completes 2,500 operations. For the client-server implementations, we run the experiments using up to 6 clients.

The execution time was measured using the Myriad chip’s time counter registers. Power consumption was measured using a shunt resistor connected at the 5V power supply cable. Using a voltmeter attached to the resistor’s terminals we calculated the current feeding the board and therefore the power consumed by the Myriad platform. We run each experiment for 10 times and we present the average values.

All the results shown in the following figures are normalized: For the queue they are normalized to the implementation using 2 locks (one in the queue header and one in the tail). The sorted list’s results are normalized to the implementation with a single lock.

5.1 Performance Evaluation

Fig. 4 shows the experimental results for the concurrent queue. We notice that the client-server implementation outperforms the lock-based in the experiments
up to 4 SHAVEs. However, in the 6-SHAVEs experiment the queue with 2 locks (mtx-2 locks) requires 11% less cycles in comparison with the client-server. On the other hand, the mtx-1 lock implementation (i.e. queue with single lock) scales poorly, leading to increased execution time as the congestion increases. Also, it is important to state that when only 2 SHAVEs are utilized the client-server requires less than 40% cycles to execute the benchmark in comparison with the mtx-2 locks.

Fig. 4. Normalized execution time for the Queue implementation

Fig. 5. Normalized execution time for the Ordered List implementation
We compared the lock-based ordered list’s single lock and client-server implementations. The results are presented in Fig. 5. In the 4-SHAVEs experiment the client-server implementation outperforms the lock-based by 19%. In the 6-SHAVEs the lock-based performs slightly better, by 3.4%. However, we notice that even in high contention, the client-server approach provides comparable performance with the lock-based one for the concurrent ordered list.

5.2 Energy and Power Consumption Evaluation

![Graph showing energy consumption comparison](image)

Fig. 6. Normalized energy consumption for the Queue implementation

In this subsection we present the experimental results for the energy consumption and the average power dissipation for each implementation. The energy and the average power results for the concurrent queue are shown in Fig. 6 and in Fig. 7 respectively. Energy consumption curves follow the execution time of each experiment (i.e. the higher the execution time, the higher the total energy consumption). However, in respect with the average power consumption the client-server implementation requires an average of 3.4% less power in comparison with the mtx-2 locks implementation. We also notice that the mtx-1 lock implementation is more efficient than the mtx-2 locks. We assume that this is related with the architectural characteristics of the Myriad chip.

The energy consumption and the average power experimental results for the ordered list are presented in Fig. 8 and in Fig. 9 respectively. As in the concurrent queue, we notice that the energy consumption is dominated mainly by the execution time of each experiment. However, in the 6-SHAVEs experiment the client-server achieves 14.5% lower energy consumption in comparison with the lock-based ordered list. In respect with the power consumption, when 4 SHAVEs are utilized, the power consumed by the client-server is almost equal with the
Fig. 7. Normalized average power for the Queue implementation

Fig. 8. Normalized average power for the Ordered List implementation
However, the client-server model achieves 18.6% lower average power consumption in the 6-SHAVEs experiment. The reason is that as the contention increases, more SHAVEs are simultaneously in stall mode (instead of spinning as happens in the lock-based implementations). Therefore, the average power consumption is smaller in comparison with the lock-based. The same can be noticed for the queue implementation in Fig.7.

To summarize our observations, we notice that in respect with the execution time, as the contention increases by utilizing more SHAVEs, the lock-based implementation performs slightly better in comparison with the client-server, as shown in Fig.4 and in Fig.5. This is to be expected, since the lock-based implementation provides more parallelism and Myriad locks, as stated earlier, provide fairness. Regarding the power consumption, the increase in contention when utilizing the client-server model leads to more SHAVEs to be simultaneously in stall mode when FIFO the buffers are full. Therefore, we experience lower average power consumption, as depicted in Fig.7 and in Fig.9 for the 6-SHAVEs experiments in comparison with the lock-based.

The efficient memory allocation of the list, where each SHAVE accesses only its own memory slice and the fact the messages are exchanged through the SHAVE FIFOs lead to both low power and high performance list data structure design based on the client-server model. More specifically, we achieve comparable performance against the lock-based list implementations by utilizing the SHAVE FIFOs and by careful allocating the data structure in the CMX memory. Additionally, we achieve low power by forcing SHAVEs to access only their local memory slice and taking advantage of the stalling on SHAVE FIFOs, instead of spinning on locks.

The client-server technique could be used in applications which make use of a limited number of concurrent data structures (which is almost always the case). Alternatively, a global server could serve an arbitrary number of concurrent data
structures, in case the server does not become a performance bottleneck. Another interesting alternative of the proposed approach would be a SHAVE to be both a server and a client simultaneously. We argue that in case of low congestion this could be an efficient alternative. However, in case of high congestion, the increased workload of the SHAVE that would be both a client and a server, would make it a performance bottleneck for the whole application.

6 Conclusion and Future Work

This work is a step towards the evaluation of the client-server model in more complex and demanding data structures. Inspired by an idea from the HPC domain, we implemented it in the embedded systems and we took advantage of the hardware specifications that the Myriad chip provides. This work shows that the client-server model can be a feasible solution for low power embedded systems that execute applications which utilize concurrent list data structures. In the future, we intend to apply the client-server model to more complex concurrent data structures (e.g. skip-lists and trees) and implement it in various modern multicore embedded platforms.

References

Towards a scalable functional simulator for the Adapteva Epiphany architecture

Ola Jeppsson and Sally A. McKee
Chalmers University of Technology, Gothenburg, SWEDEN
olaj@student.chalmers.se, mckee@chalmers.se

Abstract. For many decades, Moore’s law allowed processor architects to consistently deliver higher-performing designs by increasing clock speeds and increasing Instruction-Level Parallelism. This approach also led to ever increasing power dissipation. The trend for the past decade has thus been to place multiple (simpler) cores on chip to enable coarser-grain parallelism and greater throughput, possibly at the expense of single-application performance.

Adapteva’s Epiphany architecture carries this approach to an extreme: it implements simple RISC cores with 32K of explicitly managed memory, stripping away caches and speculative hardware and replacing a shared bus with a simple mesh. The result is a scalable, low-power architecture: the 64-core Epiphany-IV is estimated to deliver 70 (single precision) GFLOPS per watt. Adapting software to fully exploit this impressive design remains an open problem, though. The only available simulator prior to the work presented here models a single Epiphany core. We build on that to develop a scalable, parallel functional chip simulator. This tool is a work-in-progress: the next step will be to add timing models and DMA interfaces to faithfully simulate user applications at scale.

1 Introduction

For many decades, shrinking feature sizes have enabled more and more transistors on chip, which has allowed processor architects to consistently deliver higher-performing designs by raising clock speeds and adding more hardware to increase Instruction-Level Parallelism. Unfortunately, techniques like speculative and out-of-order execution not only increase performance — they also increase power consumption, which, in turn, increases heat dissipation.

For Intel, the industry leader for desktop computer microprocessors, the single-core era culminated in 2005. The company cancelled its Tejas and Jayhawk projects, which analysts attributed to heat problems [4]. Power and thermal considerations have thus become first-class design parameters. Instead of focusing on single-core performance, chip manufacturers have turned to multi-core designs to deliver greater parallel performance. Many (including Intel and Adapteva) predict 1000-core chips by the year 2020.

Before we began this project, there existed only a single-core Epiphany simulator: simulating an Epiphany chip with all cores running concurrently was not
possible. Without full-chip simulation, hardware and software design decisions must be made based on analysis and experience or prototyping. Having a scalable simulator makes it possible to explore richer hardware design spaces and makes it possible to develop and optimize scalable applications (even before the chips for which they are designed become available).\footnote{The Adapteva community has over 5,000 registered members, and over 10,000 evaluation boards have been sold. Our simulator thus has the potential for high impact.}

\section{Adapteva Epiphany}

First, we briefly introduce the Epiphany architecture. Most information needed to implement the simulator can be found in the reference manual \cite{2}. Other information has come from the community forums and, in a few cases, testing things on hardware.

The Epiphany architecture is a many-core processor design consisting of “tiny” RISC cores connected with a 2D mesh on-chip network. Both the cores and the NoC are designed for scalability and energy efficiency, and therefore many things one would expect in modern multicore processors have been stripped away. For instance, there is no inter-core bus, no cache (and thus no coherence protocol), and no speculation (not even branch prediction). The result is an energy-efficient architecture that can achieve up to 70 GFLOPS/Watt \cite{2} and scale to thousands of cores.

\subsection{RISC Cores}

Each core is a simple RISC with dual-issue pipelines, one single-precision FPU, and two integer ALUs. The ISA includes about 40 instructions (depending on how you count). A core can execute two floating-point operations (multiply-accumulate) and one load per clock cycle. The register file has 64 general-purpose registers, and all registers are memory mapped. Each core has two event timers, and the interrupt controller supports nested interrupts. DMA units per core support two parallel transactions.
2.2 Network-on-Chip

Figure 1 shows the 2D mesh layout of the NoC. The Epiphany implements separate networks for different types of traffic: one for reads (the rMesh), one for on-chip writes (the cMesh), and one for off-chip writes (the xMesh). We collectively refer to these as the eMesh. Messages are first routed east-west, then north-south. Four eLink routers connect the cores on their north, south, east, and west edges. Assuming a clock frequency of 1GHz, total off-chip bandwidth is 6.4 GB second, and total on-chip network bandwidth is 64 GB per second at every core router.

2.3 Memory Model

The architecture has a shared, globally addressable 32-bit (4GB) memory address space. An address is logically divided into the coreID (upper 12 bits) and offset (lower 20 bits). Thus, every core can have up to 1MB of globally addressable memory. The upper six bits of the coreID determine the core’s row; the lower six determine its column. An address with coreID zero aliases to the local core’s memory region. The architecture thus supports up to 4095 cores.

The architecture supports TESTSET (for synchronization), LOAD, and STORE memory operations. The upper 64KB of each core’s memory region consists of memory-mapped registers. The configurations on the market (Epiphany-III and Epiphany-IV) also map a portion of the address space to 32MB of external RAM.

All local memory accesses have strong memory ordering, i.e., they take effect in the same order as issued. Memory accesses routed through any of the three NoCs have a weaker memory ordering. The router arbitration and dispatch rules are deterministic, but the programmer is not allowed to make assumptions regarding synchronization, since there is no way to know the global “system state”. Section 4.2 of the reference manual [2] lists the only guarantees on which the programmer can depend:
1. All local memory accesses have a strong memory ordering, i.e., they take effect in the same order as they were issued;
2. All memory requests that enter the NoC obey the following:
   (a) \textit{LOAD} operations complete before the returned data is used by a subsequent instruction,
   (b) \textit{LOAD} operations using data previously written use the updated values, and
   (c) \textit{STORE} operations eventually propagate to their ultimate destination.

3 The GNU Debugger

The simulator uses the common simulator framework for \texttt{gdb} (the GNU debugger), which is widely used and serves as the defacto standard debugger in the open-source community. Written by Richard Stallman in the 1980s, it was maintained by Cygnus Solutions throughout the 1990s until they merged with Red Hat in 1999. During this time \texttt{gdb} gained most of its target support, and many \texttt{gdb}-based simulators were written. Like the Adapteva core simulator on which we base our work, most of these model embedded systems.

\texttt{gdb} is divided into three main subsystems: user interface, target control interface, and symbol handling [6]. Simulators are most concerned with the user interface and target control interface. Compiling \texttt{gdb} with the Epiphany simulator target creates two binaries, \texttt{epiphany-elf-gdb} and \texttt{epiphany-elf-run}. \texttt{epiphany-elf-gdb} is linked with the target simulator and presents the standard \texttt{gdb} user interface, \texttt{epiphany-elf-run} is a stand-alone tool that connects to the simulator target and runs a binary provided as a command-line argument.

3.1 Simulator Framework

The GNU toolchain (\texttt{binutils}, \texttt{gcc}, \texttt{gdb}) has been ported to many architectures, and since writing a simulator in the process makes it easier to test generated
gdb has acquired several simulator targets. The process generally follows these steps:

- define the CPU components (register file, program counter, pipeline), instruction set binary format, and instruction semantics in a CPU definition file;
- write architecture-specific device models;
- write needed support code for a main-loop generator script; and
- write simulator interface code.

The CPU definition file is written in an embedded Scheme-based domain specific language. That definition is fed through CGEN [3] (CPU tools GENerator) to create C files for instruction decoding and execution within the simulator framework. Since code for the simulator interface and main loop tends to be similar across architectures, an existing simulator target can often be used as a base. For example, parts of the Epiphany implementation originate from an Mitsubishi M32R port.

4 Implementation

We next discuss details of our simulator implementation. Figure 3 shows how we extend the single-core simulator to model a many-core system. Our design is process-based: for every core in the simulated system we launch an
epiphany-elf-run process. When the Epiphany simulator target is initialized, it also initializes the mesh simulator, which connects to a shared memory file. The mesh network simulator uses POSIX shared memory to connect relevant portions of each core simulator via a unified address space, and all memory requests are routed through this mesh simulator. The register file resides in the cpu_state structure. Since the mesh simulator needs to access remote CPU state for some operations, we also store that state in the shared address space.

4.1 Single-Core Simulator Integration

The core simulator upon which we build was written by Embecosm Ltd [1], on behalf of Adapteva. Most instruction semantics had already been defined, but due to the design of the gdb simulator framework, the simulator lacked support for self-modifying code in the modeled system. Due to the small local memories (32KB) in the Epiphany cores, executing code must be able to load instructions dynamically (like software overlays). Enabling self-modifying code required that we modify software mechanisms intended to speed simulation. For instance, a semantics cache maintains decoded target machine instructions, and in the original simulator code, writes to addresses in the semantics cache would update memory but not invalidate the instructions in the cache.

We map the entire simulated 32-bit address space to a “shim” device that forwards all memory requests to the eMesh network simulator. Recall that the CPU state of all cores resides in the shared address space, where the eMesh simulator can access it easily.

Algorithm 1 shows pseudo code for the simulator main loop. The highlighted lines are from the original single-core main loop. Lines 1-6 and 9 are inserted by the main-loop generator script. The instruction set has an IDLE function that puts the core in a low-power state and disables the program sequencer. We implement something similar in software: in line 8 we check whether the core is active, and if not, we sleep until we receive a wakeup event.

In line 13 we check whether another core has a pending write request to a special core register. Writes to special core registers are serialized on the target core because they might alter internal core state. In line 10 and 18 we handle out-of-band events. Such events might affect program flow and are triggered by writes to special core registers, e.g., by interrupts or reset signals.

In line 19 we ensure that only instructions inside the core’s local memory region can ever reside in the semantics cache. Without this constraint we would need to do an invalidate call to all cores’ semantics caches on all writes. In line 21 we check whether the external write flag is set, and if so, we flush the entire semantics cache. This flag is always set on a remote core when there is a write to that core’s memory.

4.2 eMesh Simulator

As shown in Figure 3, the eMesh simulator creates a shared address space accessible to all simulated cores. This is accomplished via the POSIX shared memory
Algorithm 1: Main Loop (simplified for illustration)

Highlighted lines are the original main loop

while True do
  sc ← scache.lookup(PC);
  if sc = ∅ then
    insn ← fetch_from_memory(PC);
    sc ← decode(insn);
    scache.insert(PC, sc);
    old_PC ← PC;
  if core is in active state then
    PC ← execute(sc);
    PC ← handle_out_of_band_events(PC);
  else
    wait_for_wakeup_event();
  if ext_scr_write_slot.reg ≠ -1 then
    reg_write(ext_scr_write_slot.reg, ext_scr_write_slot.value);
    ext_scr_write_slot.reg ← -1;
    signal_scr_write_slot_empty();
    PC ← handle_out_of_band_events(PC);
  if old_PC ∉ local memory region then
    scache.invalidate(old_PC);
  if external_mem_write_flag then
    scache.flush();
    external_mem_write_flag ← False;
API. We use POSIX threads (pthreads) for inter-process communication.

The eMesh simulator provides an API for memory transactions (*LOAD*, *
STORE*, *TESTSET*) and functions to connect and disconnect to the shared ad-
dress space. We also provide a client API so that other components can access
the Epiphany address space (e.g., to model the external host or to instrument a
simulated application).

Every memory request must be translated. The translator maps an Epiphany
address to its corresponding location in the simulator address space. It also deter-
mines to which type of memory (core SRAM, external DRAM, memory-mapped
registers [general-purpose or special-core], or invalid) the address corresponds.

How the request is serviced depends on the memory type. Accesses to core
SRAM and external DRAM are implemented as native load and store operations
(the target core need not be invoked). Memory-mapped registers are a little
trickier. All writes to such registers are serialized on the target core, which is
accomplished with one write slot, a mutex, and a condition variable. Reads to
special core registers are implemented via normal load instructions. Since reads
to memory-mapped, reads to general-purpose registers are only allowed when
the target core is inactive; we check core status before allowing the request.

We created a backend for the Epiphany hardware abstraction layer (*e-hal*).
Using the client API lets us compile Parallella host applications natively for
x86_64 without code modification\(^2\). We experimented with cross-compiling pro-
grams (from the *epiphany-examples* repository on the Adapteva github ac-
count) with generally good results. Obviously, programs that use implicit syn-
chronization might not work, and programs that use core functionalities not yet
supported will not work.

We have also extended the mesh simulator with networking support im-
plemented in MPI [7]. We use MPI’s RMA (remote memory access) API to
implement normal memory accesses (core SRAM and external RAM). We im-
plement all register accesses with normal message passing and a helper thread
on the remote side. We implement *TESTSET* with MPI _compare_and_swap()
, which is only available in MPI-3.0 [5]. Since we use both threads and MPI-3.0
functionalities, we require a fairly recent MPI implementation compiled with
MPI _THREADS_MULTIPLE support.

### 4.3 epiphany-elf-sim

As noted, the simulator is process-based, i.e., one simulated core maps to one
system process. When we began development, we started these processes by hand
(which is cumbersome and does not scale). We therefore created a command-
line tool that makes it easy to launch simulations. Mesh properties and the
program(s) that should be loaded onto the cores are given as command-line
arguments, and the tool spawns and manages the core simulator processes.

\(^2\) All data structures must have the same memory layout in both the 32-bit and 64-bit
versions.
5 Limitations and Future Work

The current simulator is purely functional, and thus it lacks a timing model. Adding such a model will enable more accurate performance analysis.

Before a simulation starts, all core simulators synchronize on a barrier, but thereafter the cores can “drift apart”. Even though the simulator remains functionally correct, making implicit timing assumptions can still render programs faulty. The core simulators thus require a sense of global time.

The two most notable missing CPU features are DMA and event timers. We believe that DMA functionality can be implemented adequately with the current eMesh simulator design. For the event timers, some of the sources are harder to implement than others. Some require a more accurate pipeline timing model, which CGEN supports to some extent. Other events are generated when packets flow through the NoC-router connected to the CPU.

We would like to add support for more advanced mesh features like multicast and user-defined routing. Message passing presents one obvious solution for more accurate routing. We could make MPI a hard dependency and spawn an extra router thread in each simulator process. Alternatively, we could use event queues (like \texttt{MPI\_Send()} but without MPI), or we could model the mesh as a directed graph and let the initiating core route the request all the way to the target core, using locks on the edges for sequencing messages. This should trigger fewer context switches.

6 Preliminary Results

The simulator currently supports most features not marked as LABS in the reference manual (i.e., untested or broken functionalities), with the exceptions of DMA and event timers. The simulator executes millions of instructions per second (per core) and scales up to 4095 cores running concurrently on a single computer. With respect to the networking backend, we have run tests with up to 1024 simulated cores spread over up to 48 nodes in an HPC environment. In larger single-node simulations the memory footprint averages under 3MB per simulated core. Note that the simulator design requires that writes from non-local (external) cores flush the entire semantics cache (see algorithm 1, line 21-23) rather than just invalidating the affected region, which may hamper performance.

6.1 Matrix multiplication

The parallel matrix multiplication application from \texttt{epiphany-examples} uses many CPU functions and performs all interesting work on the Epiphany chip, and thus we choose it for initial studies of simulator behavior. For simplicity, we move all host code to the first core process. We implement our own data transfer functions, since the simulator does not yet support DMA. Our port revealed a race condition in the \texttt{e-lib} barrier implementation, which we attempted to fix (see the discussion of the 1024-core simulation, below).
For the tests, we allocated 32 nodes on an HPC cluster. Nodes contain two Intel Xeon processors with 16 physical cores (eight per socket) connected by a Mellanox Infiniband FDR. Hyper-threading is disabled. Tables 1-4 present results for 16, 64, 256, and 1024 cores per computation, respectively.

Things to consider:

1. Using more nodes creates more network traffic (versus direct accesses to memory within a node) for handling simulated memory accesses. This is orders of magnitude slower, even with FDR Infiniband.
2. More nodes means more physical cores. If all cores only accessed their local memory, the ideal number of nodes would be where there was a one-to-one mapping between physical cores and simulated cores.
3. The barrier implementation does busy-waiting, and thus waiting for another core increases total instructions executed. This also shows a clear limitation of the simulator. Because there is no global time or rate limiting, the number of executed instructions could differ significantly between the simulator and real hardware.
4. The nodes are allocated with exclusive access, but the network is shared between all users of the cluster. This means that there might be network contention or other system noise that could qualitatively affect results. Another error factor is the network placement of the nodes for a requested allocation, which is also beyond our control.

For matmul-16 and matmul-64, results are understandable: the number of instructions executed per (simulated) core per second increases until we reach a 2:1 mapping between physical and simulated cores. The reason that 2:1 outperforms 1:1 is likely because all simulator processes running on a node can be pinned to (physical) cores on the same socket, which means more shared caches and less cache coherence communication between sockets.

Execution time increases for a modest number of simulated cores when we go from one to two nodes due to network communication. We get a nice speedup in execution time for matmul-64.

For matmul-256 two results stand out. The jump in execution time from one to two nodes is much higher compared to matmul-16 and matmul-64. This data point might be due to a system/network anomaly: unfortunately, we only ran this test once (to date), so we cannot yet explain the observed behavior. For matmul-256 running on 32 nodes, execution time jumps from 44 seconds on 16 nodes to 521 seconds on 32 nodes. We could expect execution time to increase a bit, since there is a 1:1 mapping between physical and simulated cores on 16 nodes, and we get more network traffic with 32 nodes. We repeated the test a few times (on the same allocation) with similar results. This behavior, too, requires further study.

When we tried to scale up to 1024 simulated cores, the program could not run to completion. Attaching the debugger revealed that all simulated cores

\[\text{In truth, our allocation of CPU hours expired before we could repeat all our experiments. We are in the process of repeating all experiments on multiple clusters, but results will not be available before this paper goes to press.}\]
were stuck waiting on the same barrier. It is likely that we hit the e-lib race condition and that our fix proved insufficient. As a limit study, we removed all barrier synchronization. This means that program output is incorrect, and the results for matmul-1024 are not directly comparable to the other runs. Since all synchronization is removed, we expect execution time to be lower than it would have been in a correct implementation. But the program still exhibits a similar memory access pattern, so it is fair to assume that the instruction rate tells something about performance even with synchronization back in place. From the previous results, we would expect running on 128 nodes to yield the lowest execution time and peak instruction rate; running on a larger allocation is part of future work.

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Execution time</th>
<th>Total insns</th>
<th>Min insns</th>
<th>Max insns</th>
<th>Avg. insns/core/s</th>
<th>Avg. insns/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>32.0 s</td>
<td>2.28E+09</td>
<td>1.40E+08</td>
<td>1.44E+08</td>
<td>4.45E+06</td>
<td>7.12E+07</td>
</tr>
<tr>
<td>2</td>
<td>37.9 s</td>
<td>4.22E+09</td>
<td>2.07E+08</td>
<td>3.29E+08</td>
<td>6.96E+06</td>
<td>1.11E+08</td>
</tr>
<tr>
<td>4</td>
<td>53.2 s</td>
<td>5.13E+09</td>
<td>2.49E+08</td>
<td>4.68E+08</td>
<td>6.03E+06</td>
<td>9.65E+07</td>
</tr>
<tr>
<td>8</td>
<td>65.8 s</td>
<td>5.51E+09</td>
<td>2.42E+08</td>
<td>5.55E+08</td>
<td>5.24E+06</td>
<td>8.38E+07</td>
</tr>
</tbody>
</table>

Table 1: matmul-16

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Execution time</th>
<th>Total insns</th>
<th>Min insns</th>
<th>Max insns</th>
<th>Avg. insns/core/s</th>
<th>Avg. insns/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>81.6 s</td>
<td>6.01E+09</td>
<td>9.22E+07</td>
<td>9.52E+07</td>
<td>1.15E+06</td>
<td>7.37E+07</td>
</tr>
<tr>
<td>2</td>
<td>88.6 s</td>
<td>1.30E+10</td>
<td>1.26E+08</td>
<td>2.42E+08</td>
<td>2.30E+06</td>
<td>1.47E+08</td>
</tr>
<tr>
<td>4</td>
<td>46.5 s</td>
<td>1.29E+10</td>
<td>1.30E+08</td>
<td>2.52E+08</td>
<td>4.34E+06</td>
<td>7.68E+08</td>
</tr>
<tr>
<td>8</td>
<td>29.2 s</td>
<td>1.42E+10</td>
<td>1.33E+08</td>
<td>3.05E+08</td>
<td>7.60E+06</td>
<td>4.86E+08</td>
</tr>
<tr>
<td>16</td>
<td>29.9 s</td>
<td>1.46E+10</td>
<td>1.29E+08</td>
<td>3.16E+08</td>
<td>7.63E+06</td>
<td>4.88E+08</td>
</tr>
<tr>
<td>32</td>
<td>35.1 s</td>
<td>1.65E+10</td>
<td>1.52E+08</td>
<td>3.70E+08</td>
<td>7.35E+06</td>
<td>4.71E+08</td>
</tr>
</tbody>
</table>

Table 2: matmul-64

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Execution time</th>
<th>Total insns</th>
<th>Min insns</th>
<th>Max insns</th>
<th>Avg. insns/core/s</th>
<th>Avg. insns/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>344.3 s</td>
<td>2.10E+10</td>
<td>7.45E+07</td>
<td>8.51E+07</td>
<td>2.38E+05</td>
<td>6.10E+07</td>
</tr>
<tr>
<td>2</td>
<td>1007.3 s</td>
<td>1.52E+11</td>
<td>1.13E+08</td>
<td>9.18E+08</td>
<td>5.89E+05</td>
<td>1.51E+08</td>
</tr>
<tr>
<td>4</td>
<td>323.8 s</td>
<td>1.08E+11</td>
<td>1.51E+08</td>
<td>5.08E+08</td>
<td>1.31E+06</td>
<td>3.35E+08</td>
</tr>
<tr>
<td>8</td>
<td>103.0 s</td>
<td>6.98E+10</td>
<td>9.17E+07</td>
<td>3.53E+08</td>
<td>2.65E+06</td>
<td>6.77E+08</td>
</tr>
<tr>
<td>16</td>
<td>44.4 s</td>
<td>6.03E+10</td>
<td>1.36E+08</td>
<td>2.49E+08</td>
<td>5.31E+06</td>
<td>1.36E+09</td>
</tr>
<tr>
<td>32</td>
<td>520.9 s</td>
<td>1.58E+12</td>
<td>8.39E+07</td>
<td>6.32E+09</td>
<td>1.19E+07</td>
<td>3.04E+09</td>
</tr>
</tbody>
</table>

Table 3: matmul-256
<table>
<thead>
<tr>
<th>Nodes</th>
<th>Execution time</th>
<th>Total insns</th>
<th>Min insns</th>
<th>Max insns</th>
<th>Avg. insns/core/s</th>
<th>Avg. insns/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>262.8 s</td>
<td>1.50E+11</td>
<td>3.12E+07</td>
<td>1.47E+08</td>
<td>5.59E+05</td>
<td>5.73E+08</td>
</tr>
<tr>
<td>32</td>
<td>135.5 s</td>
<td>1.50E+11</td>
<td>3.12E+07</td>
<td>1.47E+08</td>
<td>1.08E+06</td>
<td>1.11E+09</td>
</tr>
</tbody>
</table>

Table 4: matmul-1024*

7 Conclusions

We have modified a single-core Epiphany simulator, integrating it with our own mesh simulator that models the Epiphany network-on-chip. This enables full chip simulations modeling large numbers of cores. Although most of the necessary basic functionality is in place, the tool is still a work-in-progress, and we welcome others who would like to contribute to its further development. Source code is available from https://github.com/adapteva/epiphany-binutils-gdb/tree/epiphany-gdb-7.6-multicore-sim (the epiphany-gdb-7.6-multicore-sim branch). It is included as an experimental feature in the Epiphany SDK as of version 2014.11.

References