Resource-aware Data Parallel Array Processing

Grelck, C.; Blom, C.

Publication date
2018

Document Version
Final published version

Published in
Tagungsband des 35ten Jahrestreffens der GI-Fachgruppe "Programmiersprachen und Rechenkonzepte" = Proc. of the 35th Annual Meeting of the GI Working Group “Programming Languages and Computing Concepts”

35tes Jahrestreffen der GI-Fachgruppe "Programmiersprachen und Rechenkonzepte"

Bad Honnef, 2.–4. Mai 2018
Tagungsband

Resource-aware Data Parallel Array Processing

Clemens Grelck[1] and Cédric Blom[2]

[1] University of Amsterdam
Amsterdam, Netherlands
C.Grelck@uva.nl
[2] Delft University of Technology
Delft, Netherlands
CedricBlom@outlook.com

Abstract. Malleable applications may run with varying numbers of threads, and thus on varying numbers of cores, while the exact number of threads/cores is irrelevant for the program logic. Malleability is a common property in data-parallel array processing, and with continuously growing core counts we are increasingly faced with the problem of how to choose the best number of threads/cores.
We propose a compiler-directed, almost automatic tuning approach for the functional array processing language SaC. Our approach consists of an offline training phase during which compiler-instrumented application code systematically explores the design space and accumulates a persistent database of profiling data. When generating production code our compiler consults this database and augments each data-parallel operation with a recommendation table. Based on these recommendation tables the runtime system chooses the number of threads individually for each data-parallel operation.
With energy/power efficiency becoming an ever greater concern, we explicitly distinguish between two application scenarios: aiming at best possible performance or aiming at a beneficial trade-off between performance and resource investment.

1 Introduction

SaC (aka Single Assignment C) is a purely functional, data-parallel array language [17, 18, 14] with a C-like syntax (hence the name). SaC features homogeneous, multi-dimensional, immutable arrays and supports both shape- and rank-generic programming: SaC functions may not only abstract from the concrete shapes of argument and result arrays, but even from their ranks (i.e. the number of dimensions). A key motivation for functional array programming is fully compiler-directed parallelization for various architectures starting from exactly the same one source. Currently, the SaC compiler supports general-purpose multi-processor and multi-core systems [13], CUDA-enabled GPGPUs [19], heterogeneous combinations thereof [8], the Amsterdam MicroGrid general-purpose many-core processor [15] or, most recently, clusters of workstations [25].

For the purpose of this paper we focus on general-purpose multi-core systems with cache-coherent shared memory. One of the advantages of a fully compiler-directed approach to parallel execution is that compiler and runtime system are technically free to choose any number of threads for execution, and by design the choice cannot interfere with the program logic. We call this characteristic property *malleability*. Malleability raises the central question of this paper: what would be the best number of threads to choose for the execution of a data-parallel operation? This choice depends on a number of factors, including but not limited to

- the number of array elements to compute,
- the computational complexity per array element and
- architecture characteristics of the compute system used.

For a large array of computationally challenging values making use of all available cores is a rather trivial choice on almost any machine. However, smaller arrays or less computational complexity or both inevitably leads to the observation illustrated in Fig. 1. While for a small number of threads/cores we often achieve almost linear speedup, the additional benefit of using more of them increasingly diminishes until some (near-)plateau is reached. Beyond this plateau using even more cores often shows a detrimental effect on performance.

This common behaviour [26, 28, 29, 27] can be attributed to essentially two independent effects. First, on any given system off-chip memory bandwidth is limited, and some number of actively working cores is bound to saturate the available bandwidth. Second, the organisational overhead for synchronisation, communication workload scheduling, etc, typically grows super-linearly in practice.

We can distinguish two scenarios for choosing the number of threads. From a pure performance perspective we would aim at the number of threads that yield the highest speedup. In the example of Fig. 1 that would be 16 threads. However, we typically observe a performance plateau around that optimal number. In the given example we can observe that from 12 to 20 threads the speedup obtained only marginally changes. Hence, finding a sub-optimal, but sufficiently near-optimal, number of threads suffices in practice.

However, as soon as computing resources are not considered free of charge, it does indeed make a big difference if we use 12 cores or 20 cores to obtain extremely similar performance. The 8 additional cores clearly fail to deliver any additional performance in the example of Fig. 1. Thus, they could more productively be used for other tasks. In the absence of any useful work, they could be powered down to save energy. This observation leaves us with two possible usage scenarios:

- aiming at best possible performance, the traditional HPC view, or
- aiming at a favourable trade-off between resource consumption and performance delivered.

Outside extreme high performance computing (HPC) the latter policy becomes more and more relevant. Here, we are looking at the gradient of the speedup

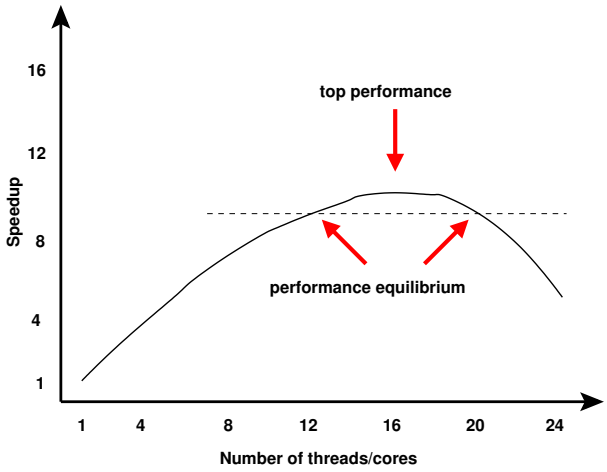

Fig. 1. Typical speedup graph observed for multi-core execution

curve. If the additional performance benefit of using one more core/thread drops below a certain threshold, we constitute that we have reached the optimal (with respect to the chosen policy) number of threads. Where exactly this threshold lies, is highly application- and situation-dependent.

In classical, high performance oriented parallel computing our issues have hardly been addressed because in this area users have typically strived for solving the largest possible problem size that still fits the constraints of the computing system used. In today's ubiquitous parallel computing [3], however, the situation has completely changed, and problem sizes are much more often determined by problem characteristics than machine constraints. But even in high performance computing some problem classes inevitably run into the described issues: multi-scale methods. Here, the same function(s) is/are applied to arrays of systematically varied shape and size. We illustrate multi-scale methods in Fig. 2 based on the example of the NAS benchmark MG (*multigrid*) [2, 11]. In this so-called *vcycle* algorithm (A glimpse at Fig. 2 should suffice to understand the motivation behind this name.) we start the computational process with a 3-dimensional array of large size and then systematically reduce the size by half in each dimension. This process continues until some predefined minimum size is reached, and then the process is sort of inverted and array sizes now double in each dimension until the original size is reached again. The whole process is repeated many times until some form of convergence is reached.

Since the array's size determines the break-even point of parallel execution in a data parallel array comprehension, it is clear that the optimal number of threads is different on the various levels of the vcycle. Regardless of the overall

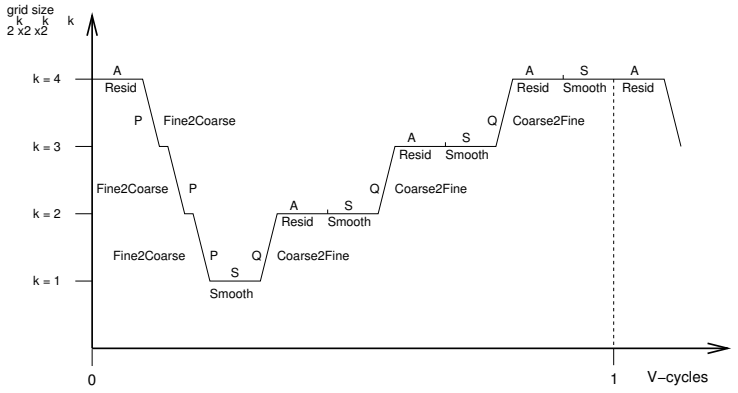

Fig. 2. Algorithmic vcycle structure of NAS benchmark MG as a representative of multi-scale methods, reproduced from [11]

problem size started with, we will always reach problem sizes where using varying fractions of the total number of threads will yield the best possible performance before for very small data sets purely sequential execution is the best solution.

All the above examples and discussions lead to one insight: in most non-trivial applications we cannot expect to find the one number of threads that is best across all data-parallel operations. This is the motivation for our proposed *smart decision tool* that aims at selecting the right number of threads for execution on the basis of individual data-parallel operations and user-configurable general policy in line with the two usage scenarios sketched out above. The smart decision tool is meant to replace a much more coarse-grained solution that SaC shares with many other high-level parallel languages, namely that based on heuristic methods some data-parallel operations in an application program may be run entirely sequential.

Our smart decision tool is based on the assumption that for many data-parallel operations the effectively best choice in either usage scenario neither is to use all cores for parallel execution nor to only use a single core for completely sequential execution. We follow a two-phase approach that distinguishes between offline training runs and online production runs of the same application. In training mode compilation our compiler instruments the generated code to produce an individual performance profile for each individual data-parallel operation. In production mode compilation we associate each data-parallel operation with an oracle that based on the performance profiles gathered offline chooses the number of threads based on the array sizes encountered at application production runtime.

The distinction between training and production mode has the disadvantage that users need to explicitly and consciously use the smart decision tool for man-

ual optimisation. One could think of a more seamless and transparent integration where applications as silently as continuously produce profiling data stored in a database and dynamically consult this database to make educated decisions. We rejected such an approach because of its adverse effects on production runtime performance. In contrast, our proposed approach inflicts minimal runtime overhead in production mode.

The remainder of the paper is organised as follows. Section 2 provides some background information on our functional array language SaC while Section 3 sketches out compilation into multithreaded code and the SaC runtime system for multi-core machines. In Sections 4 and 5 we describe our proposed smart decision tool in detail: training mode and production mode, respectively. In Section 6 we outline necessary modifications of SaC's runtime system. Some preliminary experimental evaluation is discussed in Section 7. Finally, we sketch out related work in Section 8 before we draw conclusions in Section 9.

2 Introducing SAC

As the name "Single Assignment C" suggests, SaC leaves the beaten track of functional languages with respect to syntax and adopts a C-like notation. This is meant to facilitate adoption in compute-intensive application domains, where imperative concepts prevail. Core SaC is a functional, side-effect free variant of C: we interpret assignment sequences as nested let-expressions, branches as conditional expressions and loops as syntactic sugar for tail-end recursive functions; details can be found in [17].

Despite the radically different underlying execution model (context-free substitution of expressions vs. step-wise manipulation of global state), all language constructs adopted from C show exactly the same operational behaviour as expected by imperative programmers. This allows programmers to choose their favourite interpretation of SaC code while the compiler exploits the benefits of a side-effect free semantics for advanced optimisation and automatic parallelisation.

On top of this language kernel SaC provides genuine support for processing truly multidimensional and truly stateless/functional arrays using a shape-generic style of programming. Any SaC expression evaluates to an array. Arrays may be passed between functions without restrictions. Array types include arrays of fixed shape, e.g. `int[3,7]`, arrays of fixed rank, e.g. `int[.,.]`, and arrays of any rank, e.g. `int[*]`. The latter include scalars, which we consider rank-0 arrays with an empty shape vector. For convenience and equivalence with C we use `int` rather than the equivalent `int[]` as a type notation for scalars. The hierarchy of array types induces a subtype relationship, and SaC supports function overloading with respect to subtyping.

SaC only features a very small set of built-in array operations, among others to query for rank and shape, or to select individual arry elements. Aggregate array operations are specified in SaC itself using WITH-loop array comprehensions:

```
with {
  ( lower_bound <= idxvec < upper_bound) : expr;
   ...
  ( lower_bound <= idxvec < upper_bound) : expr;
}: genarray( shape, default)
```

Here, the keyword `genarray` characterises the WITH-loop as an array comprehension that defines an array of shape *shape*. The default element value is *default*, but we may deviate from this default by defining one or more index partitions between the keywords `with` and `genarray`.

Here, *lower_bound* and *upper_bound* denote expressions that must evaluate to integer vectors of equal length. They define a rectangular (generally multidimensional) index set. The identifier *idxvec* represents elements of this set, similar to loop variables in FOR-loops. Unlike FOR-loops, we deliberately do not define any order on these index sets. We call the specification of such an index set a *generator* and associate it with some potentially complex SAC expression that is in the scope of *idxvec* and thus may access the current index location. As an example, consider the WITH-loop

```
A = with {
      ([1,1]<=iv<[4,5]): 10*iv[0]+iv[1];
      ([4,0]<=iv<[5,5]): 42;
    }: genarray( [5,5], 99);
```

that defines the 5×5 matrix

$$A = \begin{pmatrix} 99 & 99 & 99 & 99 & 99 \\ 99 & 11 & 12 & 13 & 14 \\ 99 & 21 & 22 & 23 & 24 \\ 99 & 31 & 32 & 33 & 34 \\ 42 & 42 & 42 & 42 & 42 \end{pmatrix}$$

WITH-loops in SAC are extremely versatile. In addition to the dense rectangular index partitions show above SAC supports also strided generators. In addition to the `genarray`-variant used here, SAC features further variants, among others for reduction operations. Furthermore, a single WITH-loop may define multiple arrays or combine multiple array comprehensions with further reduction operations, etc. For a complete, tutorial-style introduction to SAC as a programming language we refer the interested reader to [14].

3 Compiling SAC

Compiling SAC programs into efficiently executable code for a variety of parallel architectures is a non-trivial undertaking. While this clearly is not the place to explain the compilation process in any detail, we still sketch out a few areas most relevant for our current work.

It probably comes at no surprise to the experienced reader that WITH-loops, as introduced in the previous section, play a central role in the compilation of SAC programs. Any aggregate array operation in SAC, in one way or another, is expressed by means of WITH-loops, may it be explicitly in the application program or implicitly through composition of basic operations from the SAC standard array library. Thus, we can expect that almost any SAC program spends almost all execution time in WITH-loops.

Many of our optimisations are geared towards the composition of multiple WITH-loops into one [16]. These compiler transformations systematically improve the ratio between productive computing and organisational overhead. Consequently, when it comes to generating multithreaded code for parallel execution on multi-core system, we can focus on individual WITH-loops. WITH-loops are data-parallel by design. Thus, any WITH-loop can be executed in parallel. The subject of our current work is: should it?

So far, the SAC compiler has generated two alternative codes for every WITH-loop: a sequential and a multithreaded implementation. The choice which route to take is made at runtime based on two criteria:

- If program execution is already in parallel mode, we evaluate nested WITH-loops sequentially.
- If the size of an index set is below a configurable threshold, regardless of the computational complexity per element, we evaluate the WITH-loop sequentially.

Multithreaded program execution follows an offload model (or fork/join-model), as illustrated in Fig. 3. Program execution always starts in single-threaded mode and only if the execution reaches a WITH-loop for which both above criteria are met, worker threads are created that join the master thread in the execution of the data-parallel WITH-loop. A WITH-loop-scheduler assigns array elements for computing to among the worker threads according to one of several policies that range from static scheduling to dynamic self-scheduling with and without affinity control. In fact, the SAC compiler implements various scheduling technique to be selected by the programmer, but all this is irrelevant for our current work. When no more work is available, the worker threads terminate and, having waited for the last worker thread, the master thread resumes single-threaded execution.

The total number of threads, eight in the illustration of Fig. 3, is once determined at program startup and remains the same throughout program execution. This number is typically motivated by the hardware resources of the deployment system. Due to the malleability property of the data-parallel applications concerned, application characteristics are mostly irrelevant. While it would be technically simple to determine the number of available cores at application start, SAC for the time being expects this number to be provided by the user, either through a command line parameter or through an environment variable.

As illustrated on the right hand side of Fig. 3 does not literally implement the fork/join-model, but rather starts the worker threads right at the beginning and before the first WITH-loop is encountered during program execution. All worker threads are preserved until program termination. The conceptual fork/join model

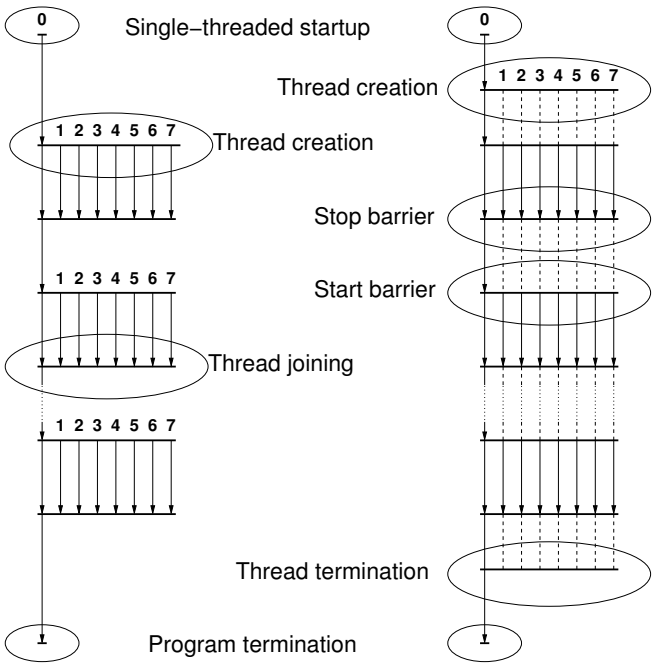

Fig. 3. Multithreaded execution models: conceptual fork-join model (left) and start/stop barrier implementation (right)

is implemented through two dedicated barriers: the *start barrier* and the *stop barrier*. At the start barrier worker threads wait for activation by the master thread. At the stop barrier the master thread waits for all worker threads to complete the parallel section while the worker threads immediately pass on to the following start barrier. We use highly efficient tailor-made implementations that exploit properties of the cache coherence protocol, but are essentially based on spinning. All details about our barrier implementations in particular and SAC's multicore implementation in general can be found in [12, 13].

4 Smart decision training mode

The proposed *smart decision tool* consists of two modes: we first describe the *training mode* in this section and then focus on the *production mode* in the following section. When compiling for smart decision training mode, the SAC compiler instruments the generated multithreaded code in such a way that

– for each WITH-loop we systematically explore the entire design space regarding the number of threads;
– we repeat each experiment sufficiently many times to ensure a meaningful timing granularity while avoiding excessive training times;
– profiling data is stored in a custom binary database.

At the same time we aim at keeping the smart decision training code as orthogonal to the existing implementation of multithreading as possible, mainly for general software engineering concerns. Fig. 4 shows pseudo code that illustrates the structure of the generated code. To make the pseudo code as concrete as possible, we pick up the example WITH-loop introduced in Section 2.

```
size = 5 * 5;

A = allocate_memory( size * sizeof(int));

spmd_frame.A = A;
num_threads = 1;
repetitions = 1;
init = 1;

do {
  start = get_real_time();

  for (int i=0; i<repetitions; i++) {
    StartThreads( num_threads,
                  spmd_fun, spmd_frame);
    spmd_fun( 0, num_threads, spmd_frame);
  }

  stop = get_real_time();

  repetitions, num_threads, init
    = TrainingOracle (init, unique_id, size,
                      repetitions, start, stop);
}
while (repetitions > 0);
```

Fig. 4. Compiled pseudo code of the example WITH-loop from Section 2 in smart decision training mode

The core addition to our standard code generation scheme is a `do-while`-loop plus a timing facility wrapped around the original code generated from our WITH-loop. Let us briefly explain the latter first. The pseudo function `StartThreads` is meant to lift the start barrier for `num_threads`-1 worker threads. They subsequently execute the generated function `spmd_fun` that contains most of the code generated from the WITH-loop, among others the resulting nesting of C `for`-

loops, the WITH-loop-scheduler and the stop barrier. The record `spmd_frame` serves as a parameter passing mechanism for `spmd_fun`. In our concrete example, it merely contains the memory address of the result array, but in general also all values referred to in the body of the WITH-loop would be made available to all worker threads via `spmd_frame`. After lifting the start barrier, the master thread temporarily turns itself into a worker thread by calling `spmd_fun` directly via a conventional function call. Note that the first argument given to `spmd_fun` denotes the thread ID. All worker threads require the number of active threads (`num_threads`) as input for the WITH-loop-scheduler.

Coming back to the specific code for smart decision training mode, we immediately identify the timing facility, which obviously is meant to profile the code, but why do we wrap the whole code within another loop? Firstly, the functional semantics of SaC and thus the guaranteed absence of side-effects in the WITH-loop allow us to actually execute the compiled code multiple times without affecting semantics. In a non-functional context this would immediately raise a plethora of concerns whether running some piece of code repeatedly may have an impact on application logic.

However, the reason for actually running a single WITH-loop multiple times is motivated by creating more reliable timing data. Take into account that we have very little a-priori insight into how long the with-loop is going to run. Shorter runtimes often result in greater relative variety of measurements. To counter such effects, we first run the WITH-loop once to obtain a rough estimate of its execution time. Following this initial execution a *training oracle* decides about the number of repetitions to follow in order to obtain meaningful timings while keeping overall execution time at acceptable levels.

In addition to controlling the number of repetitions our training oracle systematically varies the effective number of threads employed. More precisely, the training oracle implements a three step process:

Step 1: dynamically adjust the time spent on a single measurement iteration to match a certain pre-configured time range. During this step the WITH-loop is executed once by a single thread, and the execution time is measured. Based on this time the training oracle determines how often the WITH-loop could be executed without exceeding a configurable time limit, currently 500ms.

Step 2: measure the execution time of the WITH-loop while systematically varying the number of threads used. This step consists of many cycles, each running the WITH-loop as many times as determined in step 1. After each cycle the execution time of the previous cycle is stored, and the number of threads used during the next cycle is increased by one. Step 2 ends as soon as the number of threads reaches the preset maximum.

Step 3: collect measurement data to create a performance profile that is stored on disk. During this step all time measurements collected in step 2 are packaged together with three characteristic numbers of the profile: a unique identifier of the WITH-loop, the size of the index set and the number of repetitions in step 1. The packaged data is stored in the application-specific binary smart decision database file on disk.

Let us have a closer look into the last of the three steps. The SaC compiler determines the unique identifier at compile time, if in training mode, by simply counting all WITH-loops in the SaC module compiled. The resulting identifier is compiled into the generated code as one argument of the training oracle. Here, it is important to understand that we do not count the WITH-loops in the original source code written by the user, but those in intermediate code after substantial program transformations by the compiler.

The index set size may be known at compile time, as in our simple example, or may only be computed at runtime. In case of a `genarray` or `modarray` WITH-loop the size of the index set coincides with that of the array defined, and is already required for the purpose of memory allocation independent of our current work. However, in the case of a `fold`-WITH-loop we need to generate an expression that symbolically describes the index set size based on the generators' lower and upper bound specifications (and possibly their strides).

The organisation of the binary database file in rows of data is illustrated in Fig. 5. Each row starts with the three integer numbers that characterise the measurement: WITH-loop id, index set size and number of repetitions, each in 64-bit representation. What follows in the row are the time measurements with different numbers of threads. Thus, the length of the row is determined by the preset maximum number of threads. For instance, the first row in Fig. 5 contains a total of seven numbers: the three characteristic numbers followed by profile data for one, two, three and four threads, respectively. The second row in Fig. 5a accordingly stems from a training of the same application with the maximum number of threads set to two.

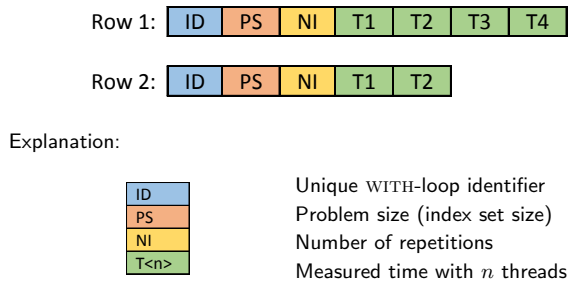

Fig. 5. Illustration of training database rows

The smart decision tool recognises a database file by its name. We use the following naming convention:

`stat.`*`name`*`.`*`architecture`*`.`*`#threads`*`.db`

and store database files in a specific subdirectory of the `.sac2c` subdirectory in the user's home directory. Both *`name`* and *`architecture`* are set by the user through corresponding compiler options when compiling for training mode. Oth-

erwise, we use suitable default values. The name field `#threads` is the preset maximum number of threads. The *`name`* option is mainly meant for experimenting with different compiler options and/or code variants and thus keep different smart decision databases at the same time. The *`architecture`* reflects the fact that the system on which we run our experiments and later the production code crucially affects our measurements. Profiling data obtained on different systems are usually incomparable. With this option we can distinguish data obtained on different execution systems, even if they end up on the same file system, e.g. a file server with user directories mounted on various different machines. In the long run, we would rather want to set this parameter automatically, but rather is an engineering than a research concern and so we postpone it for now.

Users may choose to repeat the training of a SAC program. If name, architecture and number of threads match those of an existing database, new measurements are merged into that file. For every new measurement the smart decision tool first checks if there are already similar measurements in the database file based on WITH-loop identifier and problem size. In case of a match, existing and new measurements are pairwise added, and the resulting values are written back into the corresponding database row.

5 Smart decision production mode

Continuous training leads to a collection of database files. In an online approach running applications would consult these database files in deciding about the number of threads to use for each and any instance of a WITH-loop encountered during program execution. However, locating the right data base in the file system, reading and interpreting its contents and then making a non-trivial decision would incur considerable runtime overhead that we would need to compensate first before realising any advantage through smarter decisions regarding the effective number of threads.

Therefore, we decided to take a different route and actually consult the database files created by training mode binaries when compiling production binaries. This way we can move almost all overhead to production mode compile time while keeping the actual production runtime overhead minimal. In production mode the SAC compiler does three things with respect to the smart decision tool:

1. it reads the relevant database files identified by *`name`* and *`architecture`*;
2. it applies a merge operation to combine information from several database files;
3. it creates a recommendation table for each WITH-loop.

These recommendation tables are compiled into the SAC code and used at runtime component to decide for each instance of a WITH-loop how many threads to actually.

The combination of *`name`* and *`architecture`* must match with at least one database file, but it is well possible that a specific combination matches with

several files, for example if the training is first done with a maximum of two threads and later repeated with a maximum of four threads. In such cases we read all matching database files for any maximum number of threads and merge them. The merge process is executed for each WITH-loop individually. As in training mode we identify each WITH-loop by a unique identifier. Since training and production mode compilation does not lead to different intermediate code representations otherwise, we are guaranteed to obtain the same unique identifier for each WITH-loop in either mode. These unique identifiers are matched with the identifiers in the database files to create subselections of database rows.

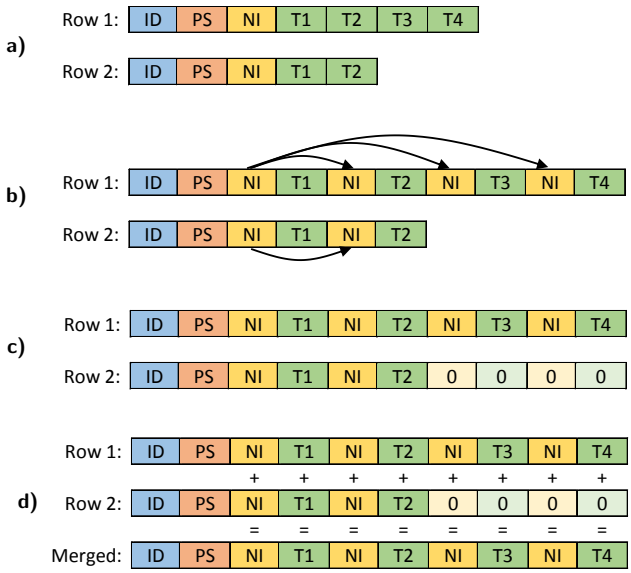

Fig. 6. Illustration of database row merging

Database rows are merged pairwise, as illustrated in Fig. 6. First, a mini-database is created in memory to store the merged rows. Second, the rows from the sub selection are read one by one and prepared for merging: The index set sizes of the row are copied in front of each time measurement (Fig. 6b). Rows are padded with empty entries where needed to make all rows as long as the one resulting from running the largest number of threads (Fig. 6c). Third, the position of each row in the mini-database is determined using rank sort. The problem size of each row is used as index for the rank sort algorithm. Rows with the same index become new rows in the mini-database. If two or more rows have the same index (e.g. they have the same problem size), they are merged by

simply adding the problem sizes and time measurements of the corresponding rows (Fig. 6d). Finally, all time measurements are divided by the corresponding problem sizes to compute the average execution time of the WITH-loop, which are likewise stored in the mini-database.

Following the merge process, the compiler creates a recommendation table, based on the in-memory mini-database. This recommendation table consists of two columns. The first column contains the different problem sizes encountered during training. The second column holds the corresponding recommended number of threads. Recommendations are computed based on the average execution times in relation to the problem sizes. Average execution times are turned into a performance graph by taking the inverse of each measurement. The performance graph is then normalized to the range zero to one. Then, we determine the gradient between any two adjacent numbers of threads in the performance graph and compared it with a configurable threshold gradient (default: 10 degrees). The recommended number of threads is the highest number of threads for which the gradient towards using one more thread is below the gradient threshold. In other words, the gradient threshold is the crucial knob by which users control whether to tune for performance or for performance/energy trade-offs. At last, the entire recommendation table is compiled into the production SAC code, just in front of the corresponding WITH-loop.

The runtime component of the smart decision production code is kept as lean and efficient as possible. When reaching some WITH-loop during execution, we compare the actual problem size encountered with the problem sizes in the recommendation table. If we find a direct match, the recommended number of threads is taken from the recommendation table. If the problem size is in between two problem sizes in the recommendation table, we use linear interpolation to estimate the optimal number of threads. If the actual problem size is smaller than any one in the recommendation table, the recommended number of threads for the smallest available problem size used. In case the actual problem size exceeds the largest problem size in the recommendation table, the recommended number of threads for the largest problem size in the table is used. So, we do interpolation, but refrain from extrapolation beyond both the smallest and the largest problem size in the recommendation table.

6 Smart decision runtime

In this section we describe the extensions necessary to actually implement the decisions made by the smart decision tool at runtime. As outlined in Section 3, a SAC program compiled for multithreaded execution alternates between sequential single-threaded and data-parallel multithreaded execution modes. Switching from one mode to the other is the main source of runtime overhead, namely for synchronisation of threads and communication of data among them. As illustrated in Fig. 7, start and stop barriers are responsible for the necessary synchronisation and communication, but likewise for the corresponding overhead. Hence, their efficient implementation is crucial.

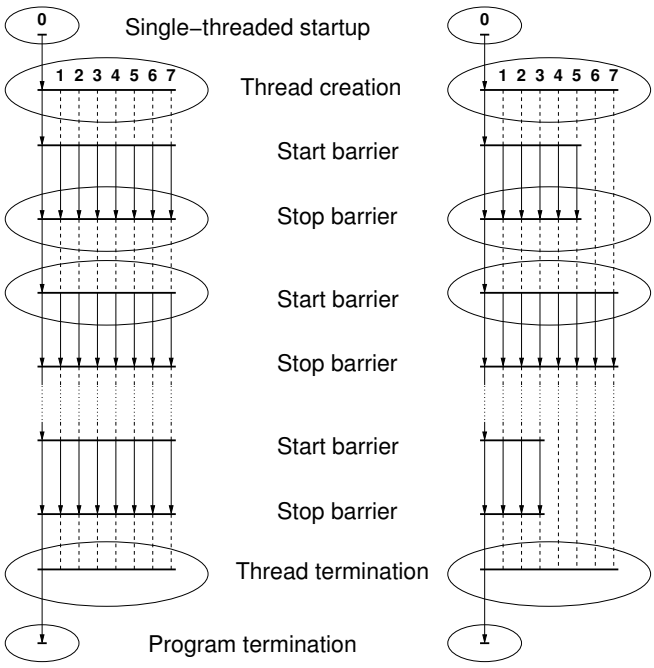

Fig. 7. Multithreaded execution: start/stop barrier model with fixed number of threads (left) and proposed model with fixed thread pool but tailor-made activations on a per WITH-loop basis (right)

SAC's standard implementations of start and stop barriers are based on active waiting, or *spinning*. This choice is motivated by the low latency of spinning barriers in conjunction with the expectation that SAC applications typically spend most execution time within the multithreaded execution mode. Thus, threads are expected to never wait long at a start barrier for their next activation while load balancing scheduling techniques make sure that waiting times at stop barriers are kept low.

For the high performance application scenario of our work, we ignore all energy-saving opportunities and stick to spinning barriers as outlined in Fig. 8. Our implementation makes use of two counters: the `local` counter is stack-allocated and private to each thread, whereas the volatile `global` counter is shared by all threads. As long as their values coincide, execution is captured in the `while`-loop. The master thread releases the worker threads by incrementing the global counter. For a more detailed discussion of this synchronisation mechanism we refer the interested reader to [13].

```
void BarrierWait( volatile unsigned int *global,
                            unsigned int *local)
{
  while (*global == *local) {
    // spin
  }
  (*local)++;
}

void BarrierRelease( volatile unsigned int *global)
{
  (*global)++;
}
```

Fig. 8. Start barrier implementation with thread spinning

For the purpose of our current work we introduce an additional high watermark for threads, as illustrated in Fig. 7. This high watermark achieves that only the recommended number of threads is actually activated in the start barrier and waited for in the stop barrier. We always use those threads with thread ID below the high watermark. With minimal change of existing implementation this can be achieved by using the changing high watermark instead of the fixed total number of threads in all scheduling decisions, regardless of the actual scheduling policy chosen. This way WITH-loop-schedulers do not assign any work to threads with ID beyond the high watermark, and these threads then immediately hit the stop barrier and subsequently the start barrier.

Spinning barriers are of course of little use for the performance/energy trade-off scenario of our work. Therefore, we introduce three further non-spinning barrier types to be selected by the user via a corresponding command line option of the SAC compiler. These barriers suspend threads at the start barrier and re-activate them as needed. The second barrier implementation is the built-in PThread barrier, the third is based on condition variables and shown in Fig. 9 and the fourth barrier implementation is a variant solely based on mutex locks.

The thread suspending start barrier implementation shown in Fig. 9 takes in principle the same approach as the spinning barrier of in Fig. 8, but additionally uses a mutex lock and a condition variable. Instead of spinning on the equality condition of the two counters, we now call `pthread_cond_wait` to suspend. Analogously, we call `pthread_cond_broadcast` to wake up the suspended worker threads for action.

7 Experimental evaluation

We evaluate our approach with a series of experiments using two different machines of the DAS-4 research cluster. The smaller of our test systems is equipped with two Intel Xeon quad-core E5620 processors with hyperthreading enabled. These eight hyperthreaded cores run at 2.4 GHz and the entire system has 24GB

```
void BarrierWait( volatile unsigned int *global,
                            unsigned int *local)
{
  pthread_mutex_lock( &barrier_mutex);
  if (*global == *local) {
    pthread_cond_wait( &barrier_condvar,
                       &barrier_mutex);
  }
  pthread_mutex_unlock( &barrier_mutex);
  (*local)++;
}

void BarrierRelease( volatile unsigned int *global)
{
  pthread_mutex_lock( &barrier_mutex);
  (*global)++;
  pthread_cond_broadcast( &barrier_condvar);
  pthread_mutex_unlock( &barrier_mutex);
}
```

Fig. 9. Start barrier implementation with thread suspension

of memory. The larger of our test system features four AMD 6100 12-core processors and has 128GB of memory. Both systems are operated in batch mode giving us exclusive access for the duration of our experiments. In the sequel we will refer to these systems as the Intel or as the AMD system, respectively.

Before exploring the actual smart decision tool, we investigate the runtime behaviour of the four barrier implementations sketched out in the previous section. In Fig. 10 we show results obtained with a synthetic micro benchmark that puts maximum stress on the barrier implementations. We systematically vary the number of cores and show actual wall clock execution times of the micro benchmark.

Two insights can be gained from this initial experiment. Firstly, from our three non-spinning barrier implementations the one based on condition variables clearly performs best across all levels of concurrency. Therefore, we restrict all further experiments to this implementation as the representative of thread-suspending barriers and relate its performance to that of the spinning barrier implementation. Secondly, we observe a substantial performance difference between the spinning barrier on the one hand side and all three non-spinning barriers on the other hand side. Positively, this experiment nicely demonstrates how well tuned the SAC synchronisation primitives are. Nevertheless, the experiment also shows that the performance/energy trade-off scenario we sketched out earlier is not easy to address.

Before exploring the effect of the smart decision tool on any complex application programs, we need to better understand the basic properties of our approach. Therefore we use a very simple, almost synthetic benchmark through-

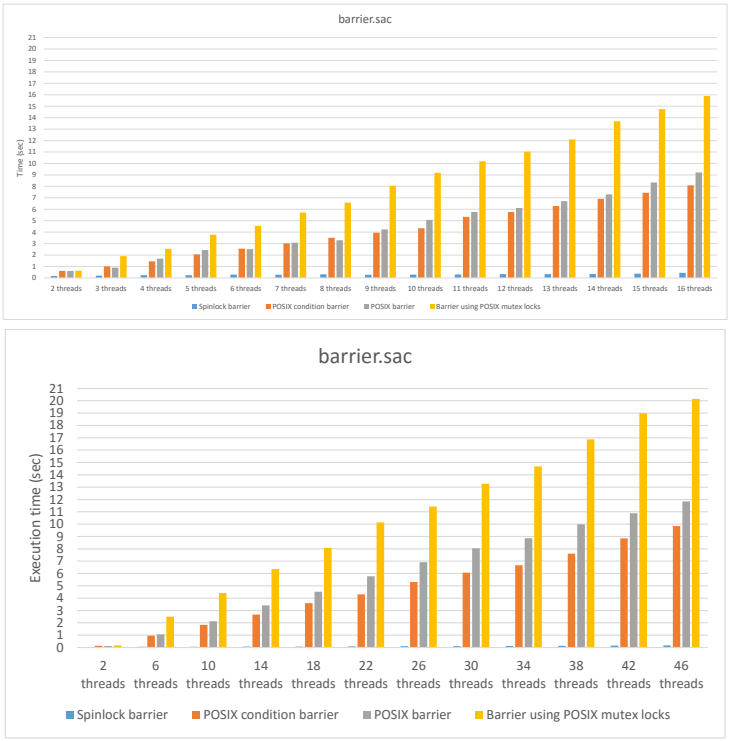

Fig. 10. Scalability of our four barrier implementations on the 8-core hyperthreaded Intel system and on the 48-core AMD system

out the remainder of this section: repeated element-wise addition of two matrices. We explore two different problem sizes, 50×50 and 400×400, that have proven to yield representative results for spinning and non-spinning barrier implementations, respectively. We first present experimental results obtained on the AMD system with spinning barriers in Fig. 11 and with suspending barriers in Fig. 12.

Frankly speaking, we cannot be happy with the results reported. For the larger problem size of 400x400 the human eye easily identifies that no fundamental speedup limit is reached up to the 48 cores available. Nonetheless, an intermediate plateau around 26 cores makes the smart decision tool (or not so smart decision tool) choose to limit parallel activities at this level.

For the smaller problem size of 50x50 we indeed observe the expected performance plateau, and the smart decision decides to limit parallelisation to 24 core. Subjectively, this appears to to be on the high side as 12 core already achieve a

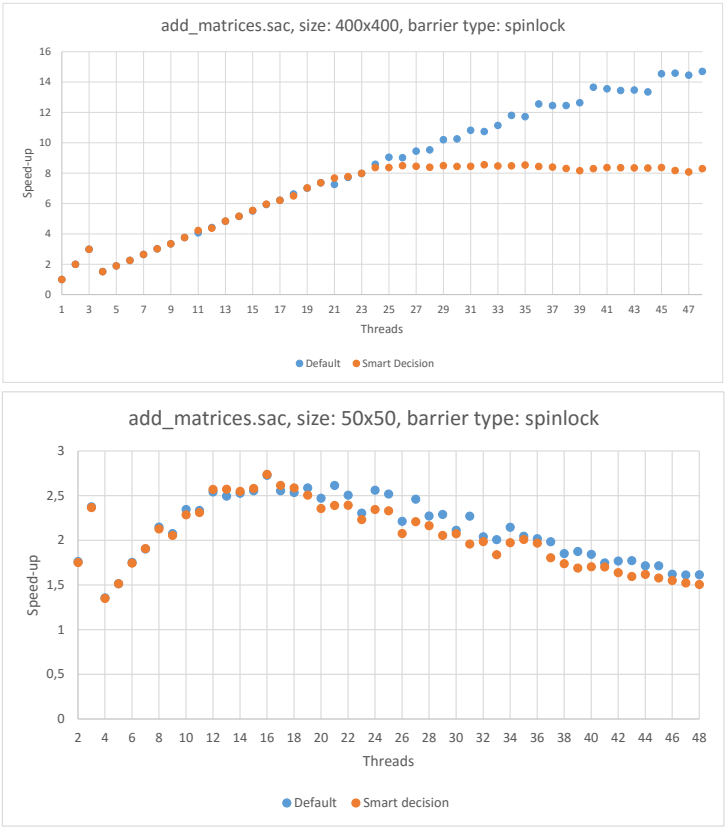

Fig. 11. Performance on AMD 48-core system with and without the proposed smart decision tool for two different problem sizes and spinning barrier implementation; smart decision tool recommendations: 26 and 24

speedup of 2.5, which is very close to the maximum. Trouble is we cannot realise the expected performance for higher thread numbers. Our expectation would be to keep a speedup of about 2.5 even if the maximum number of threads is chosen at program start to be higher.

We attribute this to the fact that our implementations of the start barrier always activate all threads, regardless of what the smart decision tool suggests. Its recommendation merely affects the WITH-loop-scheduler, which divides the available work evenly among a smaller number of active threads. We presume

that this implementation choice inflicts two much overhead in relation to the fairly small problem size, and synchronisation cost dominate our observations.

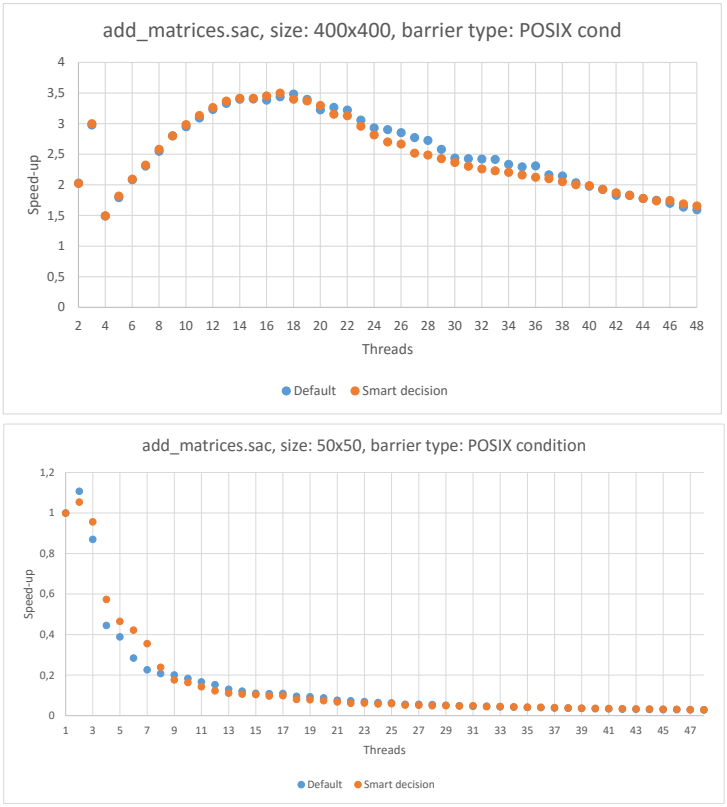

Fig. 12. Performance on AMD 48-core system with and without the proposed smart decision tool for two different problem sizes and suspending barrier implementation; smart decision tool recommendations: 24 and 1

When using suspending barriers instead of spinning barriers, as shown in Fig. 12, we see a similar picture. Only the considerably higher overhead of suspending barriers, as already observed in Fig. 10, makes the 400x400 graph for suspending barriers resemble the 50x50 graph for spinning barriers. Accordingly, running the 50x50 experiment with suspending barriers results in a major slowdown due to parallel execution. Still, we can observe that our original, motivating

assumption indeed holds: the best possible performance is neither achieved with one core nor with 48 cores, but in this particular case with two cores.

A general observation across all experiments so far is that the runtime behaviour with up to four threads is rather unexpected. This can best be seen in the graphs for the 400x400 matrices (top). With up to three threads we see expected speedups, but with four threads performance considerably goes down. From the low basis of four thread performance we can then observe an continuous incline again. This behaviour appears to be related to the system configuration of four processors with 12 cores each, but we do not have a plausible explanation of the concrete behaviour observed.

What is clear, however, is that the reproducible local performance maximum at low thread counts irritates our heuristics in constructing recommendation tables. To cope with such training data we decided not to take the angle as mentioned in Section 5 literal, but rather as a rough indication.

We now present experimental results obtained on the Intel system with spinning barriers in Fig. 13 and with suspending barriers in Fig. 14. Overall these figures confirm the results obtained on the 48-core AMD system. In the 400x400 experiments we can clearly identify the hyperthreaded nature of the architecture. For example in the 400x400 experiment with spinning barriers speedups continuously grow up to eight threads, dramatically diminish for nine threads and then again continuously grow up to 16 threads.

8 Related Work

It is a well known fact that not every parallelisable loop should indeed be run in parallel for optimal performance. Successful parallelisation not only depends on functional properties of the loop body, which may be known at compile time, but to a large extent on runtime parameters such as data set sizes or even data values themselves. Therefore, many parallel programming approaches provide basic means to switch off parallel execution in generally parallelised loops based on runtime values.

An example of this kind of mechanism is the `if`-clause of OpenMP [7], which allows programmers to postpone the decision whether or not to execute a parallel section of code actually in parallel by multiple worker threads or still sequentially by the single master thread until runtime. The `if`-clause contains a predicate that may be based on runtime variables of the programmer's choice. Analogous to our own motivation, the `if`-clause reflects the fact that parallel execution may not always be beneficial for runtime performance, but whether or not it actually is depends on information only available at application runtime.

Another similar example is the `--dataParMinGranularity` command line option of Chapel [4, 5], which likewise allows the user to exercise control over the mostly implicit data parallel constructs of the Chapel language. Unless instructed otherwise, Chapel automatically uses all available cores for implicit parallelisation. The above command line option is automatically added to every Chapel compiled code and allows the user of an application to a-posteriori

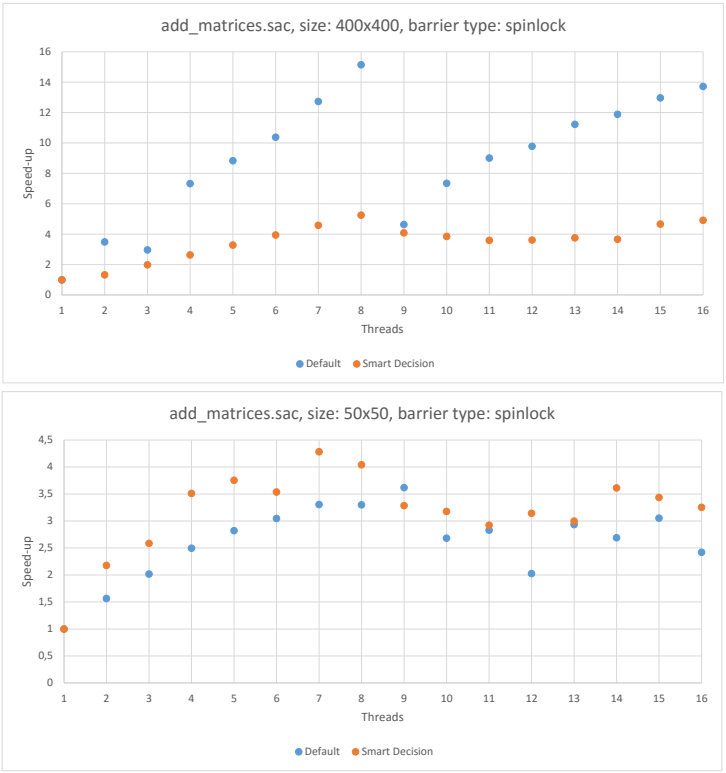

Fig. 13. Performance on Intel 8-core hyperthreaded system with and without the proposed smart decision tool for two different problem sizes and spinning barrier implementation; smart decision tool recommendations: 9 and 9

predicate the decision to parallelise or not to parallelise on the size of the index space (or *domain* in Chapel speak) of each data-parallel operation. In contrast to the `if`-clause of OpenMP only one value can be set across the entire application, whereas the optimal value obviously depends on the compute intensity per element of each individual data-parallel operation.

Prior to our current work, the SaC compiler has adopted a similar strategy as Chapel: at compile time the user may set a minimum index size for parallelisation, and the generated code at runtime decides between sequential and parallel execution based on the given size [12, 13]. The sac compiler makes use of a suitable default minimum index set size if the user does not provide specific information.

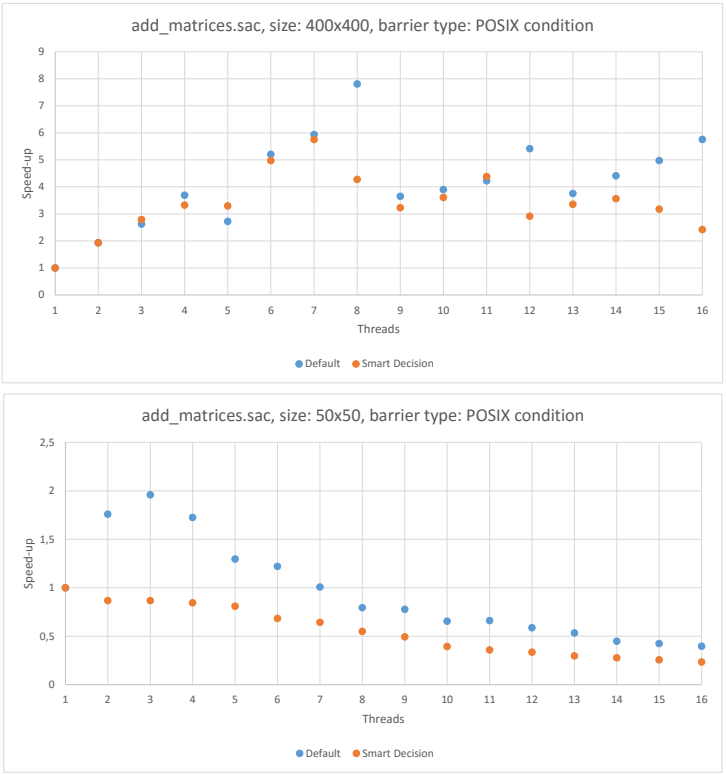

Fig. 14. Performance on Intel 8-core hyperthreaded system with and without the proposed smart decision tool for two different problem sizes and suspending barrier implementation; smart decision tool recommendations: 9 and 1

All approaches mentioned so far share a fundamental shortcoming: any data-parallel operation can only either be executed using all available worker threads or completely sequential by a single thread: a classical all-or-nothing decision.

The latest versions of OpenMP [] go one step beyond and introduce the `num_threads`-clause, which allows programmers to precisely specify the number of threads to be used for each parallelised loop, if they wish so. Like in the `if`-clause, the `num_threads`-clause contains an arbitrary C or Fortran expression that may access all program variables in scope. While the `num_threads`-clause is mainly motivated as a vehicle to express nested parallelism, it could also be used to explicitly set the number of threads in relation to problem set sizes or similar

runtime parameters of an application. However, programmers are completely on their own when using this feature of OpenMP.

This gap is filled by a multitude of performance analysis and tuning tools as for example Intel's VTune [21]. Corresponding guidelines [20, 9] explain the issues involved. These tools and methodologies indeed allow performance engineers to manually adapt effective parallelism in individual data-parallel operations to data set sizes and machine characteristics, but the process is highly labour-intensive if not to say painful. Furthermore, it needs to be repeated for every new architecture and data set size.

In contrast, our proposed approach for compiler-directed parallelisation in SaC works almost entirely automatically with the sole exception that users must explicitly compile their source for training and production mode and run training codes on representative input data. We would also like to remind the reader that our implicit approach works on the data-parallel operations in intermediate code *after* far-reaching code restructuring compiler optimisation. In contrast, any manual tuning approaches operate on the level of source code and either restrict the effectiveness and scope of compiler transformations or suffer from any decoupling between source and binary code.

In this sense, *feedback-driven threading* proposed by Suleman et al goes even one step further as a completely implicit compiler-based solution [29]. In an OpenMP-parallelised loop they peel off up to 1% of the initial iterations. They are executed by a single thread while hardware performance monitoring counters collect information regarding off-chip memory bandwidth and cycles spent in critical section. After this initial training phase the generated code evaluates the hardware counters and predicts the optimal number of threads to be used for the remaining bulk of iterations based on a simple analytical model. Apart from the obvious beauty of being completely transparent to users, this approach has some disadvantages. Considerable overhead is introduced at production runtime for choosing the (presumably) best number of threads. This needs to be offset first by more efficient parallel execution before any total performance gains can be realised. Peeling off and sequential execution of up to 1% of the loop iterations restricts potential gains of parallelisation according to Amdahl's law. This is a (high) price to be paid for every data-parallel operation, including all those that would otherwise perfectly scale.

In contrast to our approach that continuously accumulates insight into the scaling behaviour of individual data-parallel operations with every program run in training mode, Suleman et al do not carry over any information from one program run to the next and, thus, cannot reduce the overhead of feedback-driven threading. Last not least, they rely on the assumption that the initial iterations of a parallelised loop are representative in runtime behaviour for all remaining iterations, whereas we always measure the entire data-parallel operation. Therefore, we only reach the limits of reproducibility and predictability if the runtime behaviour critically depends on varying data being processed instead of data set sizes. But even in this adverse scenario increasing the number of training runs in

conjunction with the accumulation and averaging of profile data, we may come to meaningful conclusions.

Pusukuri et al proposed *ThreadReinforcer* [27]. While their motivation is similar, their proposed solution again differs in many aspects from what we propose. Most importantly, they treat any application as a black box and determine one single number of threads to be used consistently throughout the application's life time. In contrast, we approach the problem on the granularity of individual data-parallel operations and systematically vary the number of threads during an application's execution. Similar to Suleman et al, ThreadReinforcer integrates learning and analysis into application execution. This choice creates overhead, that only pays off for long-running applications. At the same time, having the analysis on the critical path of application performance immediately creates a performance/accuracy trade-off dilemma. In contrast, we deliberately distinguish between training mode and production mode in order to train an application as long as needed and to accumulate statistical information in persistent storage without affecting application production performance.

Another approach in this area is *ThreadTailor* [23]. Here, the emphasis lies on *weaving threads together* in order to reduce synchronisation and communication overhead where available concurrency cannot efficiently be exploited. This scenario differs from our setting in that we explicitly look into malleable data-parallel applications. Therefore, we are able to set the number of active threads to our liking and even differently from one data-parallel operation to another.

Much work on determining optimal thread numbers on multi-core processors, such as [22] or [1], stems from the early days of multi-core computing and are limited to the small core counts of that time. Furthermore, there is a significant body of literature studying power-performance trade-offs, among others [24, 6]. In a sense we also propose such a trade-off, but in our model performance and energy consumption are not competing goals. Instead, we aim at achieving runtime performance within a margin of the best performance observable with the least number of threads.

Coming back to SAC at last, we mention the work by Gordon and Scholz [10]. They aim at adapting the number of active threads in a data-parallel operation to varying levels of competing computational workload in an interactively configured multi-core system. The goal is to avoid thread context switches and thread migration and rather vacate cores that turn out to be oversubscribed by unrelated applications at program runtime. For this purpose they continuously monitor execution times of data-parallel operations, and upon observing significant changes in the performance level they adapt the number of threads of the running SAC application accordingly. Their work differs from our's not only in the underlying motivation, but likewise in the pure online approach. In contrast, we distinguish between training and production mode, focus on our own application's characteristics and assume that we face no significant external competing computational workload on our system, i.e. we rather look at batch-operated machines.

9 Conclusions and Future Work

Malleable data-parallel application programs offer interesting opportunities for compilers and runtime systems alike to adapt the effective number of threads separately for each data-parallel operation in order to achieve best runtime performance. Alternatively, a favourable trade-off between runtime performance and resource investment appears equally relevant these days. We explore this opportunity in the context of the functional data-parallel array language SaC and propose a combination of offline training that builds up a persistent profiling database, production code that incorporates gathered profiling data into recommendation tables and a runtime system extension that actually implements such recommendations during parallel program execution.

We are particularly concerned with data-parallel operations that turn out to be too small to make efficient use of all available cores on the system while their purely sequential execution still foregoes considerable performance gains. There are two variations of this scenario. Firstly, we see entire applications that could successfully exploit a certain number of cores in parallel execution but not the tens and hundreds of cores that computer architecture roadmaps promise for the foreseeable future. Secondly, data-parallel operations that only benefit from a limited number of cores could make part of larger applications that as a whole scale much better. Here, executing certain data-parallel operation with a single thread just as with too many threads reduces parallelisation gains on the whole application level. Our approach to control the number of threads on a per-operation basis is exactly geared at such cases.

Following our experimental evaluation in Section 7, we must admit that we are not there yet. While we are still convinced by our general approach, not to mention the relevance of the underlying problem, additional research is need to make our approach succeed. We have identified the following directions to take immediate action. We must make our approach more robust to training data that does not expose the shape shown in Fig. 1 as characteristically as we would wish. In particular we must develop robust methods to detect outliers.

Furthermore, we must refine our barrier implementations to activate worker threads more selectively. And we plan to explore how we can speed up suspension-based barriers in comparison to spinning barriers. A likely option to this effect would be to use hybrid barriers that spin for some configurable time interval before they suspend.

A particular problem that we underestimated at the beginning of our work is the by nature short execution time of the data-parallel operations for which our work is particularly relevant. Consequently, overall performance is disproportionately affected by synchronisation and communication overhead, thus pushing our second research direction sketched out above. In addition, short parallel execution times likewise incur a large relative variation. Although our offline training approach does take this into account, there are still practical limits to execution times in training mode and thus to averaging over many measurements. Moreover, a large variation also means that the average or median often is not a good representative of actual behaviour.

References

1. K. Agrawal, Y. He, W. Hsu, and C. Leiserson. Adaptive task scheduling with parallelism feedback. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Processing (PPoPP'06)*. ACM, 2006.
2. D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, T. Schreiber, R. Simon, V. Venkatakrishnam, and S. Weeratunga. The NAS Parallel Benchmarks. *International Journal of Supercomputer Applications*, 5(3):63–73, 1991.
3. B. Catanzaro, A. Fox, K. Keutzer, D. Patterson, B. Y. Su, M. Snir, K. Olukotun, P. Hanrahan, and H. Chafi. Ubiquitous Parallel Computing from Berkeley, Illinois, and Stanford. *IEEE Micro*, 30(2):41–55, 2010.
4. B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.
5. Cray Inc. *Chapel Language Specification Version 0.98.* Cray Inc., Seattle, USA, 2015.
6. M. Curtis-Maury, J. Dzierwa, C. Antonopoulos, and D. Nikolopoulos. Online power-performance adaptation of multithreaded programs using hardware event-based prediction. In *International Conference on Supercomputing (ICS'06)*, 2006.
7. L. Dagum and R. Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Transactions on Computational Science and Engineering*, 5(1), 1998.
8. M. Diogo and C. Grelck. Towards heterogeneous computing without heterogeneous programming. In K. Hammond and H. Loidl, editors, *Trends in Functional Programming, 13th Symposium, TFP 2012, St.Andrews, UK*, volume 7829 of *Lecture Notes in Computer Science*, pages 279–294. Springer, 2013.
9. M. Gillespie and C. Breshears. *Achieving Threading Success.* Intel Corp, 2005.
10. S. Gordon and S. Scholz. Dynamic control of runtime systems through a common interface. In R. Lämmel, editor, *Draft Proceedings of the 27th International Symposium on Implementation and Application of Functional Languages (IFL'15), Koblenz, Germany.* University of Koblenz-Landau, 2015.
11. C. Grelck. Implementing the NAS Benchmark MG in SAC. In V. K. Prasanna and G. Westrom, editors, *16th International Parallel and Distributed Processing Symposium (IPDPS'02), Fort Lauderdale, USA*. IEEE Computer Society Press, 2002.
12. C. Grelck. A Multithreaded Compiler Backend for High-Level Array Programming. In M. H. Hamza, editor, *2nd International Conference on Parallel and Distributed Computing and Networks (PDCN'03), Innsbruck, Austria*, pages 478–484. ACTA Press, 2003.
13. C. Grelck. Shared memory multiprocessor support for functional array processing in SAC. *Journal of Functional Programming*, 15(3):353–401, 2005.
14. C. Grelck. Single Assignment C (SAC): high productivity meets high performance. In V. Zsók, Z. Horváth, and R. Plasmeijer, editors, *4th Central European Functional Programming Summer School (CEFP'11), Budapest, Hungary*, volume 7241 of *Lecture Notes in Computer Science*, pages 207–278. Springer, 2012.
15. C. Grelck, S. Herhut, C. Jesshope, C. Joslin, M. Lankamp, S.-B. Scholz, and A. Shafarenko. Compiling the Functional Data-Parallel Language SaC for Microgrids of Self-Adaptive Virtual Processors. In *14th Workshop on Compilers for Parallel Computing (CPC'09), IBM Research Center, Zürich, Switzerland*, 2009.

16. C. Grelck and S.-B. Scholz. Merging compositions of array skeletons in SAC. *Journal of Parallel Computing*, 32(7+8):507–522, 2006.
17. C. Grelck and S.-B. Scholz. SAC: A functional array language for efficient multithreaded execution. *International Journal of Parallel Programming*, 34(4):383–427, 2006.
18. C. Grelck and S.-B. Scholz. SAC: Off-the-Shelf Support for Data-Parallelism on Multicores. In *2nd Workshop on Declarative Aspects of Multicore Programming (DAMP'07), Nice, France*, pages 25–33. ACM Press, 2007.
19. J. Guo, J. Thiyagalingam, and S.-B. Scholz. Breaking the GPU programming barrier with the auto-parallelising SAC compiler. In *6th Workshop on Declarative Aspects of Multicore Programming (DAMP'11), Austin, USA*, pages 15–24. ACM Press, 2011.
20. Intel. *Developing Multithreaded Applications: A Platform Consistent Approach*. Intel Corp, 2003.
21. Intel. *Threading Methodology: Principles and Practices*. Intel Corp, 2003.
22. C. Jung, D. Lim, J. Lee, and S. Han. Adaptive execution techniques for SMT multiprocessor architectures. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Processing (PPoPP'05)*. ACM, 2005.
23. J. Lee, H. Wu, M. Ravichandram, and N. Clark. Thread Tailor: Dynamically weaving threads together for efficient, adaptive parallel applications. In *International Symposium on Computer Architecture (ISCA'10)*, 2010.
24. J. Li and J. Martinez. Dynamic power-performance adaptation of parallel computation on chip multiprocessors. In *Symposium on High Performance Computer Architecture (HPCA'06)*, 2006.
25. T. Macht and C. Grelck. SAC goes cluster: From functional array programming to distributed memory array processing. In J. Knoop, editor, *Tagungsband des 18. Kolloquiums Programmiersprachen und Grundlagen der Programmierung (KPS 2015), Pörtschach am Wörthersee, Austria*. Technical University of Vienna, 2015.
26. J. Nieplosha et al. Evaluating the potential of multithreaded platforms for irregular scientific computations. In *Computing Frontiers*, 2007.
27. K. Pusukuri, R. Gupta, and L. Bhuyan. Thread Reinforcer: Dynamically determining number of threads via OS level monitoring. In *IEEE International Symposium on Workload Characterization (IISWC'11), Austin, TX, USA*, pages 116–125. IEEE Computer Society, 2011.
28. S. Saini et al. A scalability study of Columbia using the NAS parallel benchmarks. *Journal of Computational Methods in Science and Engineering*, 2006.
29. M. Suleman, M. Qureshi, and Y. Patt. Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on CMPs. In *13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIII), Seattle, WA, USA*, pages 277–286. ACM, 2008.