Resource-aware Data Parallel Array Processing

Grelck, C.; Blom, C.

Noname manuscript No.
(will be inserted by the editor)

Resource-aware Data Parallel Array Processing

Clemens Grelck · Cédric Blom

the date of receipt and acceptance should be inserted later

Abstract Malleable applications may run with varying numbers of threads, and thus on varying numbers of cores, while the precise number of threads is irrelevant for the program logic. Malleability is a common property in data-parallel array processing. With ever growing core counts we are increasingly faced with the problem of how to choose the best number of threads.

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

Single Assignment C (SaC) is a purely functional, data-parallel array language [1,2] 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

University of Amsterdam
Amsterdam, Netherlands
E-mail: C.Grelck@uva.nl · Delft University of Technology
Delft, Netherlands
E-mail: CedricBlom@outlook.com

number of dimensions). A key motivation for functional array programming is fully compiler-directed parallelization for various architectures. From the very same source code the SaC compiler supports general-purpose multi-processor and multi-core systems [3], CUDA-enabled GPGPUs [4], heterogeneous combinations thereof [5] and, most recently, clusters of workstations [6].

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.

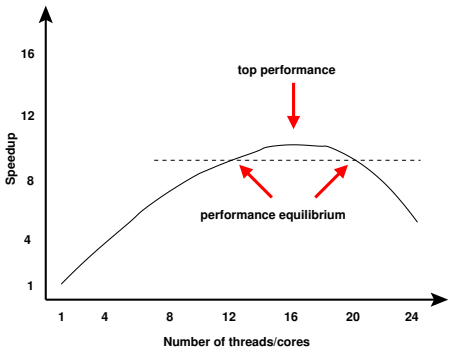

Fig. 1 Typical speedup graph observed for multi-core execution

This common behaviour [7–10] 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 it. Second, organisational overhead typically grows super-linearly.

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. As soon as computing resources are not considered available for free, it does make a big difference if we use 12 cores or 20 cores to obtain equvivalent performance. The 8 additional cores in the example of Fig. 1 could more

productively be used for other tasks or powered down to save energy. This observation leaves us with two application scenarios:

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

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

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 [11], 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 [12].

We illustrate multi-scale methods in Fig. 2 based on the example of the NAS benchmark MG (*multigrid*) [13,12]. In this so-called *vcycle* algorithm 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.

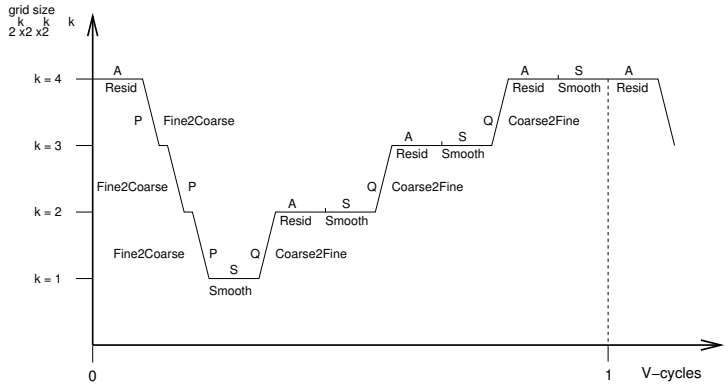

Fig. 2 Algorithmic vcycle structure of NAS benchmark MG as a representative of multi-scale methods, reproduced from [12]

Since the array's size determines the break-even point of parallel execution, the optimal number of threads is different on the various levels of the vcycle. Regardless of the overall problem size we started with, we will always reach problem sizes where using the total number of threads yields suboptimal performance before purely sequential execution becomes the best solution for very small data sizes.

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 application runs. In training mode compilation we instrument the generated code to produce an individual performance profile for each 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. One could think of a more seamless and transparent integration where applications silently store profiling data in a database all the time. 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 SAC and its compilation into multithreaded code. In Sections 3 and 4 we describe our proposed smart decision tool in detail: training mode and production mode, respectively. In Section 5 we outline necessary modifications of SAC's runtime system. Some preliminary experimental evaluation is discussed in Section 6. Finally, we sketch out related work in Section 7 before we draw conclusions in Section 8.

2 SAC: Language and Compiler

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.

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 [1]. Despite the radically different underlying execution model, 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 absence of side-effects 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.

SaC only features a very small set of built-in array operations, among others to query for rank and shape, or to select array 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

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

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 [2].

Compiling SaC programs into efficiently executable code for a variety of parallel architectures is a challenge, where WITH-loops play a critical role. Many of our optimisations are geared towards the composition of multiple WITH-loops into one [14]. 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, we evaluate the WITH-loop sequentially, regardless of the computational complexity per element.

Multithreaded program execution follows an offload model (or fork/join-model). 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 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 [3].

3 Smart decision tool training mode compilation

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. 3 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;

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

Fig. 3 Compiled pseudo code of the example WITH-loop from Section 2 in smart decision training mode. The variable `max_threads` denotes a user- or system-controlled upper limit for the number of threads used.

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 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 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 to obtain more reliable timing data. A-priori we have no 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 an 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, by default 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 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.

Possibly in contrast to readers' expectations we do not systematically vary the problem size, although quite obviously the problem size has a major impact on execution time as well as on the optimal number of threads to be used. Our rationale is twofold: firstly, it is quite possible (and hard to rule out for a compiler) that the problem size affects the program logic (not so in our simplistic running exmple, of course). For example, the NAS benchmark MG, that we referred to in Section 1, assumes 3-dimensional argument arrays whose extents along each of the three axes are powers of two. Silently running the code for problem sizes other than the ones prescribed by the application, may lead to unexpected and undedired behaviour, including runtime errors.

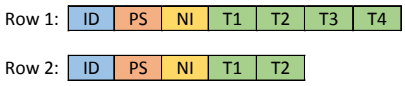

Fig. 4 Illustration of training database rows: ID: unique WITH-loop identifier, PS: index set size, NI: number of repetitions, Tn: measured time using n threads

Secondly, only the user application knows the relevant problem sizes. Unlike the number of threads, whose alternative choices are reasonably restricted by the hardware under test, the number of potential problem sizes is theoretically unbounded and practically too large to systematically explore.

The organisation of the binary database file in rows of data is illustrated in Fig. 4. 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. 4 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. 4 accordingly stems from a training of the same application with the maximum number of threads set to two.

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

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

Both *`name`* and *`architecture`* are set by the user through corresponding compiler options when compiling for training mode. Otherwise, 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.

4 Smart decision tool production mode compilation

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. Therefore, we decided to 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;
2. it merges information from several database files;
3. it creates a recommendation table for each WITH-loop.

These recommendation tables are compiled into the SaC code; they are used by the compiled code at runtime to determine the number of threads to execute each individual WITH-loop.

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.

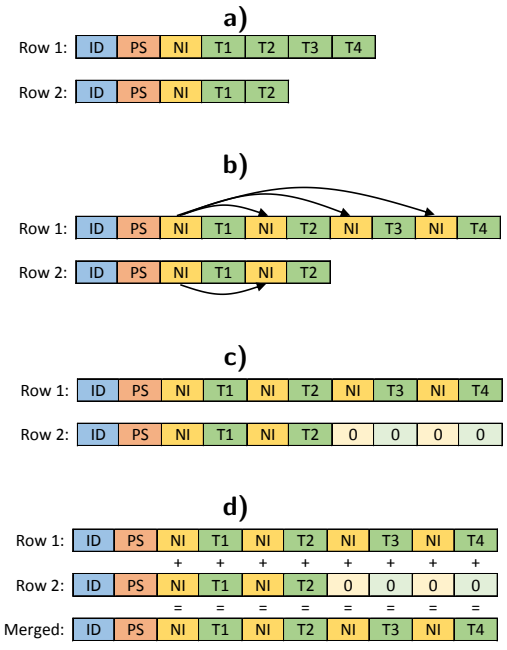

Fig. 5 Illustration of database row merging

Database rows are merged pairwise, as illustrated in Fig. 5. First, a mini-database is created in memory to store the merged rows. Second, the rows from the subselection 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. 5b). 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. 5c). 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. 5d). 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.

5 Smart decision tool runtime system support

In this section we describe the extensions necessary to actually implement the decisions made by the smart decision tool at runtime. SaC programs compiled for multithreaded execution alternate 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. 6, start and stop barriers are responsible for the necessary synchronisation and communication, but likewise for the corresponding overhead. Hence, their efficient implementation is crucial.

SaC's standard implementations of start and stop barriers are based on active waiting, or *spinning*. More precisely, a waiting thread continuously polls a certain memory location until that value is changed by another thread. This choice is motivated by the low latency of spinning barriers in conjunction with

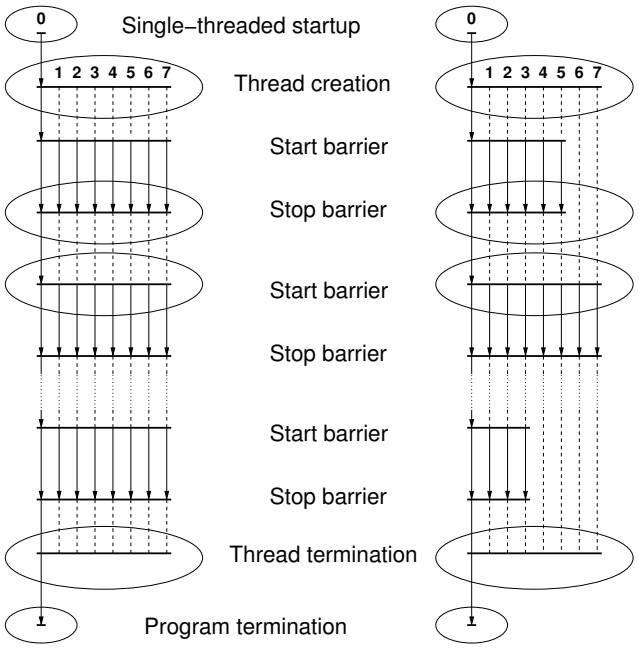

Fig. 6 Multithreaded execution: start/stop barrier model with fixed number of threads (left) and proposed model with fixed thread pool but tailor-made activations on a per WITH-loop basis (right)

the expectation that SAC applications typically spend most execution time within the multithreaded execution mode [3]. Thus, threads are expected to never wait long at a start barrier for their next activation while load balancing scheduling techniques ensure low waiting times at stop barriers.

In addition to the runtime constant `max_threads` that denotes the number of threads that exist for execution we introduce the runtime variable `num_threads` that denotes the actual number of threads to be used as determined by our oracle, as illustrated in Fig. 6. We modify start and stop barriers to limit their effectiveness to the first `num_threads` threads from the the thread pool. WITH-loop-schedulers do not assign any work to threads with ID beyond `num_threads`. These threads immediately hit the stop barrier and immediately continue to the subsequent start barrier.

Spinning barriers are of little use for the performance/energy trade-off scenario of our work. From the operating system perspective, it is indistinguishable whether some thread productively computes or whether it waits at a spinning barrier. While spinning at a barrier, a thread nonetheless consumes energy, pretty much at the same level as during productive computing phases. Therefore, we experiment with three non-spinning barrier types that suspend threads at the start barrier and re-activate them as needed: the built-

in PThread barrier, a custom implementation based on condition variables and one that is solely based on mutex locks, for details see [15].

6 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 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. 7 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.

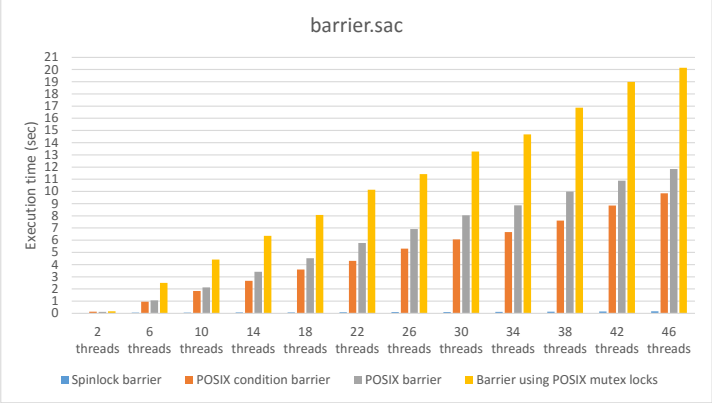

Fig. 7 Scalability of our four barrier implementations on the 48-core AMD system, results for the Intel system are nearly identical.

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 dif-

ference 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 throughout 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. 8 and with suspending barriers in Fig. 9.

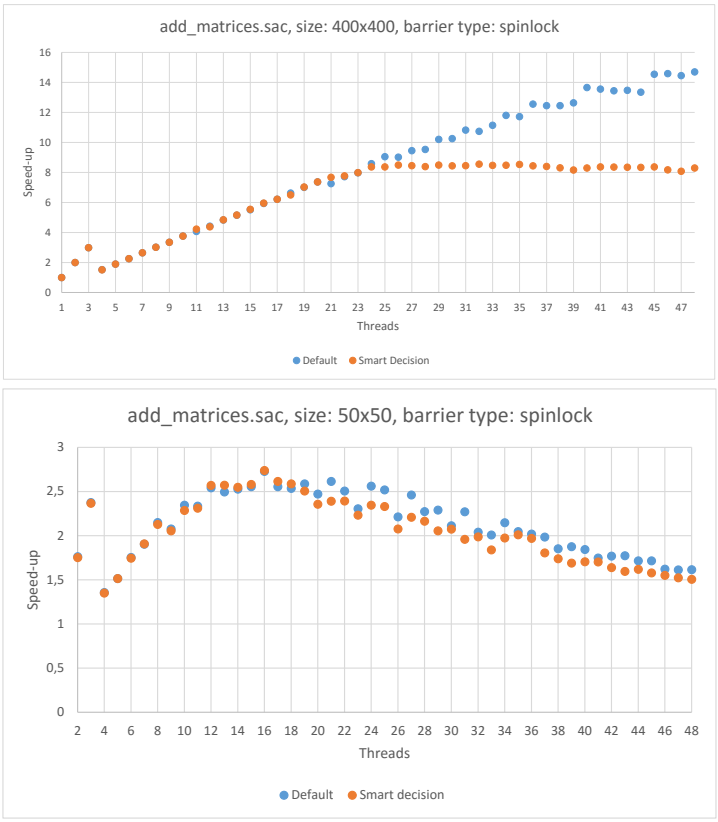

Fig. 8 Performance on AMD 48-core system with and without the proposed smart decision tool for two different problem sizes and spinning barrier implementation; smart decision tool recommendations: 26 and 24

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 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.

When using suspending barriers instead of spinning barriers, as shown in Fig. 9, we see a similar picture. Only the considerably higher overhead of suspending barriers, as already observed in Fig. 7, 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 4 literal, but rather as a rough indication.

We now present experimental results obtained on the Intel system with spinning barriers in Fig. 10 and with suspending barriers in Fig. 11. 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.

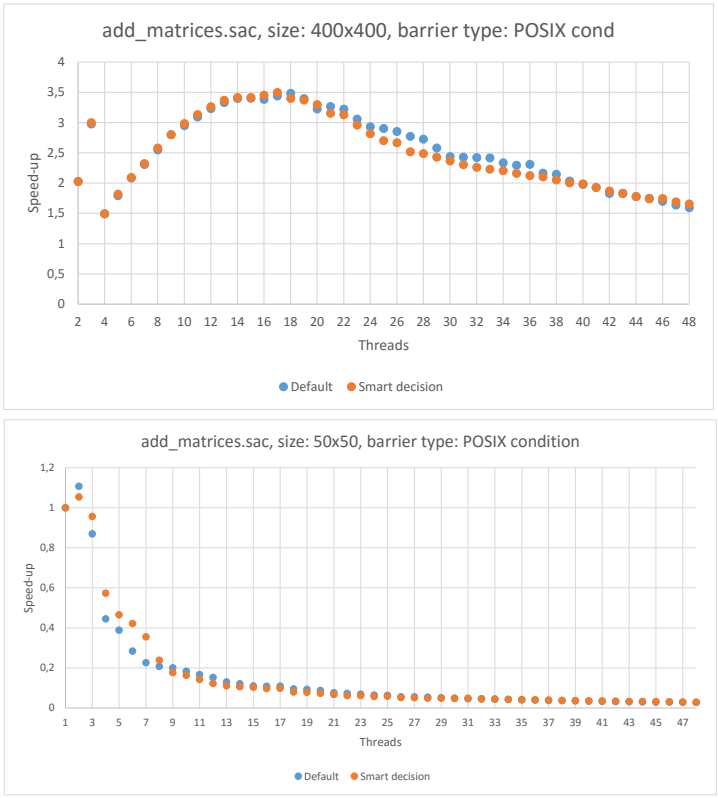

Fig. 9 Performance on AMD 48-core system with and without the proposed smart decision tool for two different problem sizes and suspending barrier implementation; smart decision tool recommendations: 24 and 1

7 Related work

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 [16], 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. Another similar example is the `--dataParMinGranularity` command line option of Chapel [17], which likewise allows the user to exercise control over the mostly implicit data parallel constructs of the Chapel language. 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

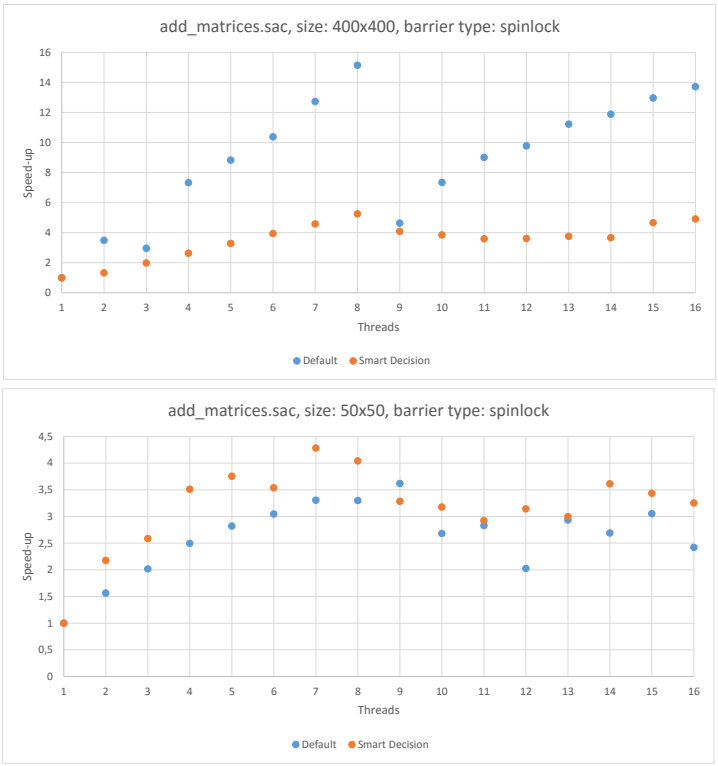

Fig. 10 Performance on Intel 8-core hyperthreaded system with and without the proposed smart decision tool for two different problem sizes and spinning barrier implementation; smart decision tool recommendations: 9 and 9

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 [3]. The SaC compiler makes use of a suitable default minimum index set size if the user does not provide specific information.

All approaches so far share a fundamental restriction: 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.

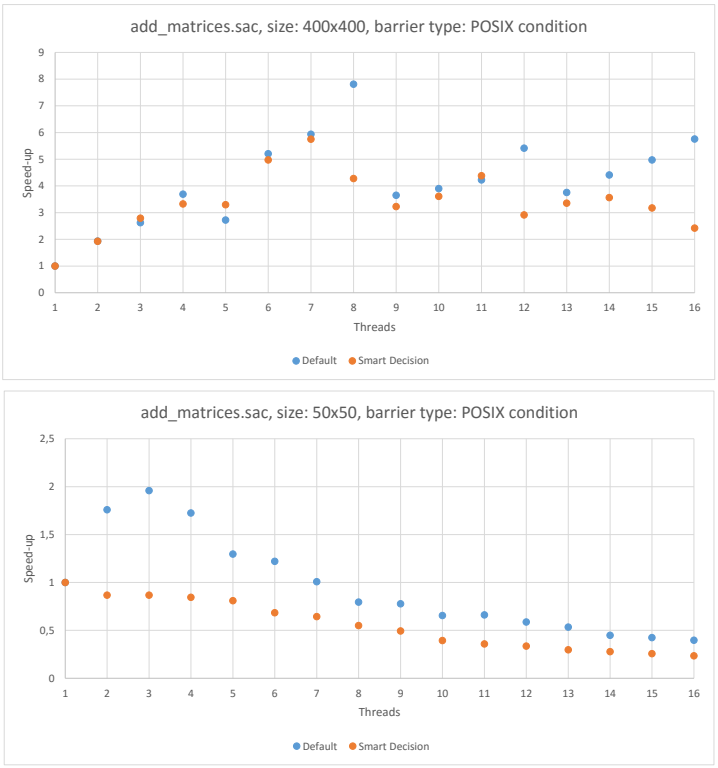

Fig. 11 Performance on Intel 8-core hyperthreaded system with and without the proposed smart decision tool for two different problem sizes and suspending barrier implementation; smart decision tool recommendations: 9 and 1

This gap is filled by a multitude of performance analysis and tuning tools as for example Intel's VTune [18]. Corresponding guidelines [19] explain the issues involved. These tools and methodologies 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.

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 even goes a step further as a completely implicit solution [9]: 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. Despite 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. Moreover, they rely on the assumption that the initial iterations of a parallelised loop are representative for all remaining iterations, whereas we always measure the entire data-parallel operation.

Pusukuri et al proposed *ThreadReinforcer* [10]. 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* [20]. Here, the emphasis lies on *weaving threads together* 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 [21] or [22], 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 [23, 24]. 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 [25]. 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.

8 Conclusions and future work

Malleability of data-parallel programs offer interesting opportunities for compilers and runtime systems alike to adapt the effective number of threads separately for each data-parallel operation. This could be exploited to achieve best possible runtime performance or a good trade-off between runtime performance and resource investment alike. 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.

Following our experimental evaluation in Section 6, additional research is needed. We must make our approach more robust to training data that does not expose a shape as characteristic as in Fig. 1 and develop robust methods against 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.

For the time being, our work is geared at SAC's shared memory code generation backend; other backends, as mentioned in Section 1, cannot benefit. While our technical solutions do not directly carry over to other code generation backends, both the problem addressed as well as the principle approach of automated offline training and online database consultation do very well. For example, in our latest distributed memory cluster backend [6] the problem aggravates to two questions: how many nodes in a cluster to use and how many cores per node to use. In our CUDA-based GPGPU backend [4] the question boils down to when to effectively use the GPU and when it would be beneficial to rather use the host CPU cores, for instance in the case of onfortunate ratios of data transfer overhead and GPGPU compute times. Last not least, our heterogeneous systems backend [5] would strongly benefit from automated support as to how many GPGPUs and how many CPU cores to use for optimal execution time or performance/energy trade-off. In other words, the work described in this paper could spawn a plethora of further research projects.

References

1. Grelck, C., Scholz, S.B.: SAC: A functional array language for efficient multithreaded execution. International Journal of Parallel Programming **34** (2006) 383–427
2. Grelck, C.: Single Assignment C (SAC): high productivity meets high performance. In Zsók, V., Horváth, Z., Plasmeijer, R., eds.: 4th Central European Functional Programming Summer School (CEFP'11), Budapest, Hungary. Volume 7241 of Lecture Notes in Computer Science., Springer (2012) 207–278
3. Grelck, C.: Shared memory multiprocessor support for functional array processing in SAC. Journal of Functional Programming **15** (2005) 353–401
4. Guo, J., Thiyagalingam, J., Scholz, S.B.: Breaking the GPU programming barrier with the auto-parallelising SAC compiler. In: 6th Workshop on Declarative Aspects of Multicore Programming (DAMP'11), Austin, USA, ACM Press (2011) 15–24
5. Diogo, M., Grelck, C.: Towards heterogeneous computing without heterogeneous programming. In Hammond, K., Loidl, H., eds.: Trends in Functional Programming, 13th Symposium, TFP 2012, St.Andrews, UK. Volume 7829 of Lecture Notes in Computer Science., Springer (2013) 279–294
6. Macht, T., Grelck, C.: SAC Goes Cluster: Fully Implicit Distributed Computing. In: 33rd International Parallel and Distributed Processing Symposium (IPDPS'19), Rio de Janeiro, Brazil, IEEE Computer Society Press (2019)
7. Nieplosha, J., et al.: Evaluating the potential of multithreaded platforms for irregular scientific computations. In: Computing Frontiers. (2007)
8. Saini, S., et al.: A scalability study of Columbia using the NAS parallel benchmarks. Journal of Computational Methods in Science and Engineering (2006)
9. Suleman, M., Qureshi, M., Patt, Y.: 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, ACM (2008) 277–286
10. Pusukuri, K., Gupta, R., Bhuyan, L.: Thread Reinforcer: Dynamically determining number of threads via OS level monitoring. In: IEEE International Symposium on Workload Characterization (IISWC'11), Austin, TX, USA, IEEE Computer Society (2011) 116–125
11. Catanzaro, B., Fox, A., Keutzer, K., Patterson, D., Su, B.Y., Snir, M., Olukotun, K., Hanrahan, P., Chafi, H.: Ubiquitous Parallel Computing from Berkeley, Illinois, and Stanford. IEEE Micro **30** (2010) 41–55

12. Grelck, C.: Implementing the NAS Benchmark MG in SAC. In Prasanna, V.K., Westrom, G., eds.: 16th International Parallel and Distributed Processing Symposium (IPDPS'02), Fort Lauderdale, USA, IEEE Computer Society Press (2002)
13. Bailey, D., Barszcz, E., Barton, J., Browning, D., Carter, R., Dagum, L., Fatoohi, R., Frederickson, P., Lasinski, T., Schreiber, T., Simon, R., Venkatakrishnam, V., Weeratunga, S.: The NAS Parallel Benchmarks. International Journal of Supercomputer Applications **5** (1991) 63–73
14. Grelck, C., Scholz, S.B.: Merging compositions of array skeletons in SAC. Journal of Parallel Computing **32** (2006) 507–522
15. Grelck, C., Blom, C.: Resource-aware Data Parallel Array Processing. In: 35th GI Workshop on Programming Languages and Computing Concepts, Bad Honnef, Germany. Volume 482 of Research Report., Department of Informatics, University of Oslo, Norway (2019) 70–97
16. Dagum, L., Menon, R.: OpenMP: An Industry-Standard API for Shared-Memory Programming. IEEE Transactions on Computational Science and Engineering **5** (1998)
17. Chamberlain, B., Callahan, D., Zima, H.: Parallel programmability and the Chapel language. International Journal of High Performance Computing Applications **21** (2007) 291–312
18. Intel: Threading Methodology: Principles and Practices. Intel Corp. (2003)
19. Gillespie, M., Breshears, C.: Achieving Threading Success. Intel Corp. (2005)
20. Lee, J., Wu, H., Ravichandram, M., Clark, N.: Thread Tailor: Dynamically weaving threads together for efficient, adaptive parallel applications. In: International Symposium on Computer Architecture (ISCA'10). (2010)
21. Jung, C., Lim, D., Lee, J., Han, S.: Adaptive execution techniques for SMT multiprocessor architectures. In: ACM SIGPLAN Symposium on Principles and Practice of Parallel Processing (PPoPP'05), ACM (2005)
22. Agrawal, K., He, Y., Hsu, W., Leiserson, C.: Adaptive task scheduling with parallelism feedback. In: ACM SIGPLAN Symposium on Principles and Practice of Parallel Processing (PPoPP'06), ACM (2006)
23. Li, J., Martinez, J.: Dynamic power-performance adaptation of parallel computation on chip multiprocessors. In: Symposium on High Performance Computer Architecture (HPCA'06). (2006)
24. Curtis-Maury, M., Dzierwa, J., Antonopoulos, C., Nikolopoulos, D.: Online power-performance adaptation of multithreaded programs using hardware event-based prediction. In: International Conference on Supercomputing (ICS'06). (2006)
25. Gordon, S., Scholz, S.: Dynamic adaptation of functional runtime systems through external control. In Lämmel, R., ed.: Implementation and Application of Functional Languages, 27th International Symposium (IFL'15), Koblenz, Germany, ACM (2015)