International Journal of Parallel Programming (2020) 48:652–674
https://doi.org/10.1007/s10766-020-00664-0

Resource-Aware Data Parallel Array Processing

Clemens Grelck[1] · **Cédric Blom**[2]

Received: 15 October 2019 / Accepted: 27 May 2020 / Published online: 9 June 2020

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 [17, 19] 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 parallelisation for various architectures. From the very same source code the SaC compiler

✉ Clemens Grelck
C.Grelck@uva.nl

Cédric Blom
CedricBlom@outlook.com

[1] University of Amsterdam, Amsterdam, Netherlands

[2] Delft University of Technology, Delft, Netherlands

supports general-purpose multi-processor and multi-core systems [16], CUDA-enabled GPGPUs [21], heterogeneous combinations thereof [11] and, most recently, clusters of workstations [26].

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, less computational complexity or both inevitably lead 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 [28–30, 32] 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

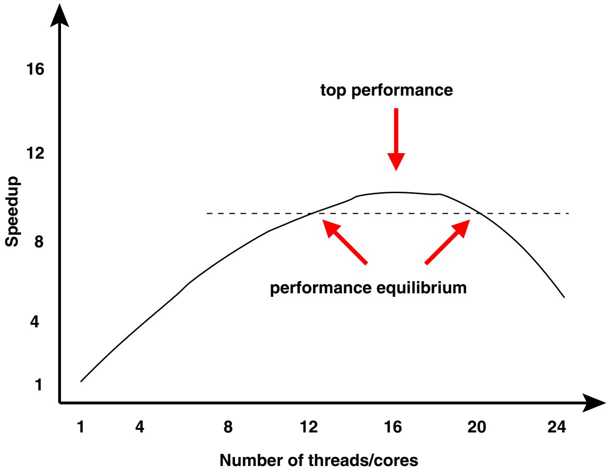

Fig. 1 Typical speedup graph observed for multi-core execution

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 equivalent 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 [4], 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 [15].

We illustrate multi-scale methods in Fig. 2 based on the example of the NAS benchmark MG (multigrid) [3, 15]. In this so-called *v-cycle* algorithm (A glimpse at Fig. 2 suffices to understand the motivation of the 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 an array's size determines the break-even point of parallel execution, the best number of threads is different on the various levels of the v-cycle. Regardless of the overall problem size, we always reach problem sizes where using the total number of threads yields suboptimal performance before purely sequential execution becomes the best choice.

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

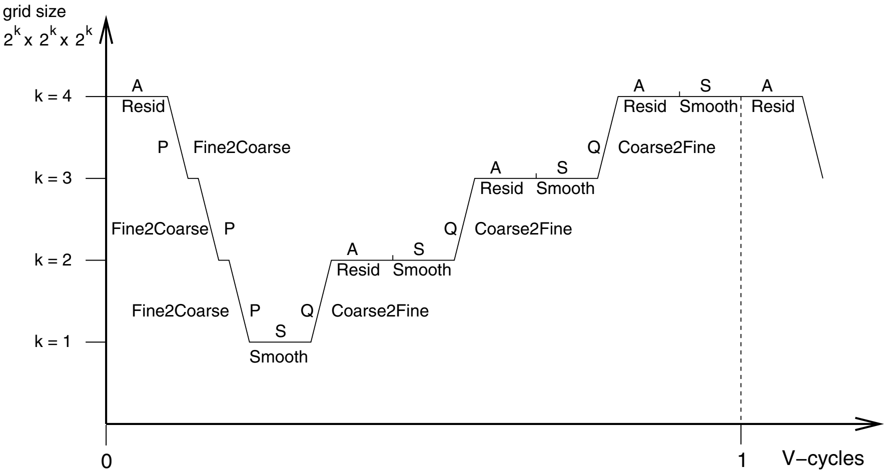

Fig. 2 Algorithmic v-cycle structure of NAS benchmark MG as a representative of multi-scale methods, reproduced from [15]

shares with many other high-level parallel languages: a rather simple heuristic decides whether to effectively execute a data-parallel operation in parallel or sequentially.

Our smart decision tool is based on the assumption that for many data-parallel operations the best choice in either of our two 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 an application. 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 production runtime.

The distinction between training and production modes 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 due to its inevitable adverse effects on production mode performance.

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

2 SAC: Language and Compiler

As the name “Single Assignment C” suggests, SaC combines a purely functional semantics based context-free substitution of expressions with a C-like syntax and overall look-and-feel. This design is meant to facilitate adoption in compute-intensive application domains, where imperative concepts prevail. We interpret assignment sequences as nested let-expressions, branches as conditional expressions and loops as syntactic sugar for tail-recursive functions; details can be found in [19]. All syntactic constructs adopted from C show precisely the same operational behaviour as in C proper. This allows programmers to choose their favourite interpretation of SaC code while the SaC compiler exploits the absence of side-effects for advanced optimisation and automatic parallelisation.

SaC provides genuine support for truly multidimensional and truly stateless/functional arrays using a shape-generic style of programming: any SaC expression evaluates to an array, and arrays are passed to and from functions call-by-value. 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 induction variables in for-loops. In contrast to 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], 10);
```

that defines the 5×5 matrix

$$\begin{pmatrix} 10 & 10 & 10 & 10 & 10 \\ 10 & 11 & 12 & 13 & 14 \\ 10 & 21 & 22 & 23 & 24 \\ 10 & 31 & 32 & 33 & 34 \\ 42 & 42 & 42 & 42 & 42 \end{pmatrix}$$

With-loops are extremely versatile. In addition to the dense rectangular index partitions, as shown above, SaC supports also strided generators. In addition to the `genarray`-variant, 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 [17].

Compiling SaC programs into efficiently executable code for a variety of parallel architectures is a challenge, where with-loops play a vital role. Many of our optimisations are geared towards the composition of multiple with-loops into one [20]. 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 systems, we focus on individual with-loops. With-loops are data-parallel by design: 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 each with-loop: a sequential and a multithreaded implementation. The choice which route to take is made at runtime based on two criteria:

- If the size of an index set is below a configurable threshold, we evaluate the with-loop sequentially.
- If program execution is already in parallel mode, we evaluate nested with-loops sequentially.

Multithreaded program execution follows an offload (or fork/join) model. Program execution always starts in single-threaded mode. Only when execution reaches a with-loop for which both above criteria for parallel execution are met, worker threads are created. These worker threads join the master thread in the data-parallel execution of the with-loop. A with-loop-scheduler assigns index space indices to worker threads according to one of several scheduling policies. At last, the master thread resumes single-threaded execution following a barrier synchronisation. We refer the interested reader to [16] for all details.

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 and each problem size found in the code 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. Figure 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 Sect. 2.

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` worker threads. They subsequently execute the generated function `spmd_fun` that contains most

```
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 Sect. 2 in smart decision training mode. The variable `max_threads` denotes a user- or system-controlled upper limit for the number of threads used

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 are 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 500 ms.

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 (problem size) 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 third step. In training mode the SaC compiler determines the unique identifier of each with-loop by simply counting all with-loops in a SaC module. 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 the intermediate representation 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 does affect the program logic (not so in our simplistic running example, of course). For example, the NAS benchmark MG, that we referred to in Sect. 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 undesired behaviour, including runtime errors. 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, problem size (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 profiling 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 field `#threads` is the preset maximum number of threads.

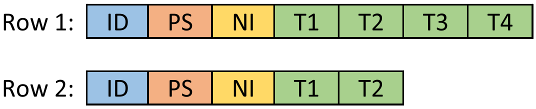

Fig. 4 Illustration of training database rows: ID: unique with-loop identifier, PS: problem size (index set size), NI: number of repetitions, T*n*: measured time using *n* threads

The *name* option is primarily meant for experimenting with different compiler options and/or code variants and, thus, allows us to keep different smart decision databases at the same time. The *architecture* option 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 every instance of a WITH-loop encountered during program execution. However, locating the right database 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.

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 number of repetitions (NI) of the row is 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 (PS) 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 repetition numbers and time measurements of the corresponding columns (Fig. 5d). Finally, all time

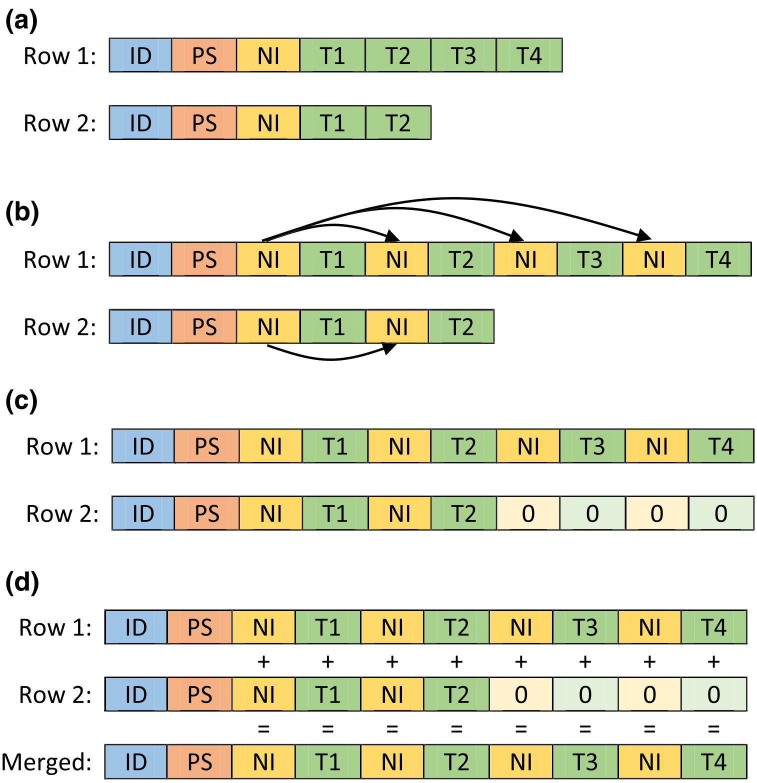
(a)
Row 1: ID PS NI T1 T2 T3 T4
Row 2: ID PS NI T1 T2
(b)
Row 1: ID PS NI T1 NI T2 NI T3 NI T4
Row 2: ID PS NI T1 NI T2
(c)
Row 1: ID PS NI T1 NI T2 NI T3 NI T4
Row 2: ID PS NI T1 NI T2 0 0 0 0
(d)
Row 1: ID PS NI T1 NI T2 NI T3 NI T4
+ + + + + + + +
Row 2: ID PS NI T1 NI T2 0 0 0 0
= = = = = = = =
Merged: ID PS NI T1 NI T2 NI T3 NI T4

Fig. 5 Illustration of database row merging

measurements are divided by the corresponding problem sizes to compute the average execution time of the WITH-loop, which is likewise stored in the mini-database.

Following the merge process, the compiler creates a recommendation table for each WITH-loop, 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 and normalising it to the range zero to one.

To diminish the effect of outliers we use fourth-order polynomial interpolation of the measurement results. Then, we determine the gradient between any two adjacent numbers of threads and compare it with a configurable threshold gradient (default: 10°). The recommended number of threads is the highest number of threads for which the gradient towards using one more thread is above the gradient threshold. The gradient threshold is the crucial knob whether to tune for performance alone 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 as 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 the expectation that SAC applications typically spend most execution time in multithreaded execution mode [16]. 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 proceed 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 a thread consumes as much energy as during productive computing. Therefore, we experiment with three non-spinning barrier implementations that suspend and re-activate threads as needed: the built-in PThread barrier, a custom

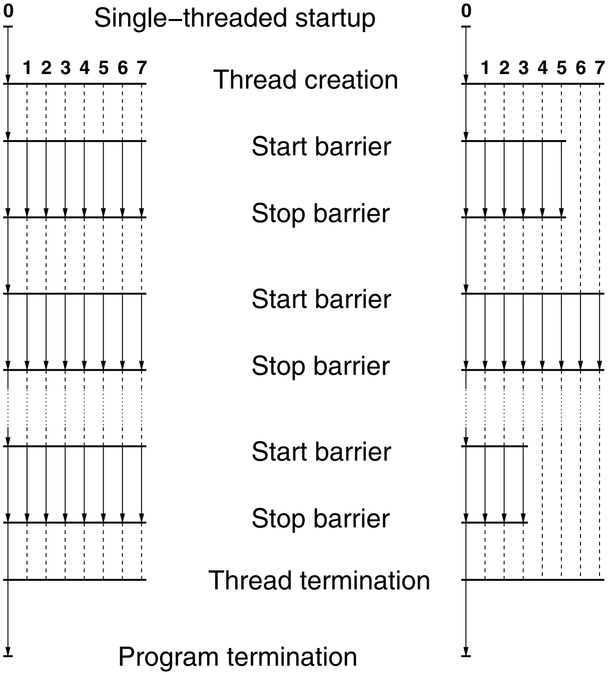

Fig. 6 Multithreaded execution: start/stop barrier model with fixed number of threads (left) and proposed model with fixed thread pool but tailor-made activations on a per WITH-loop basis (right)

implementation based on condition variables and one that is solely based on mutex locks; for details see [18].

6 Experimental Evaluation

We evaluate our approach with a series of experiments using two different machines. The smaller one is equipped with two Intel Xeon quad-core E5620 processors with hyperthreading enabled. These eight hyperthreaded cores run at 2.4 GHz; the entire system has 24 GB of memory. Our larger machine features four AMD Opteron 6168 12-core processors running at 1.9 GHz and has 128 GB of memory. Both systems are operated in batch mode giving us exclusive access for the duration of our experiments. We refer to these systems as the Intel and as the AMD system from here on.

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.

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 thread counts. Therefore, we restrict all further experiments

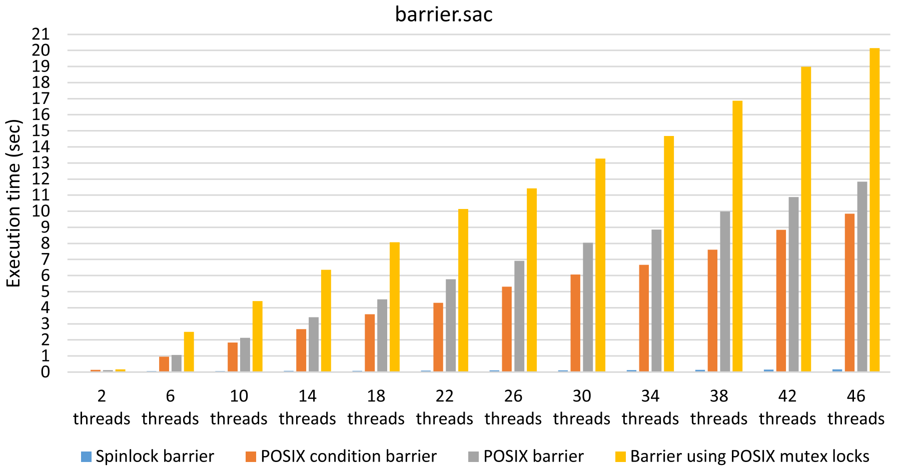

Fig. 7 Scalability of our four barrier implementations on the 48-core AMD system; our results for the Intel system are nearly identical

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. This experiment 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 both spinning and non-spinning barriers. We first present experimental results obtained on the AMD system with spinning barriers in Fig. 8.

For the larger problem size of 400×400 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 (or not so smart) decision tool choose to limit parallel activities at this level. For the smaller problem size of 50×50 we indeed observe the expected performance plateau, and the smart decision tool decides to limit parallelisation to 24 cores. Subjectively, this appears to to be on the high side as 12 cores 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

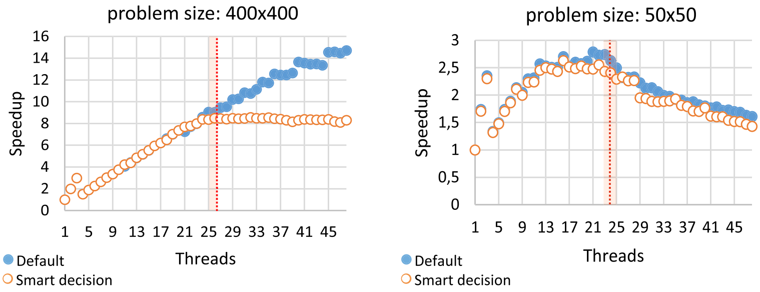

Fig. 8 Performance on AMD 48-core system with and without the proposed smart decision tool for two different problem sizes and spinning barrier implementation; smart decision tool recommendations: 26 and 24

WITH-loop-scheduler, which divides the available work evenly among a smaller number of active threads. We presume that this implementation choice inflicts too much overhead in relation to the fairly small problem size, and synchronisation cost dominate our observations.

We repeat the same experiments on the Intel system and show the results in Fig. 9. In the 400 × 400 experiments we can clearly identify the hyperthreaded nature of the architecture. For example in the 400 × 400 experiment with spinning barriers speedups continuously grow up to eight threads, dramatically diminish for 9 threads and then again continuously grow up to 16 threads. What strikes us on the Intel architecure is that for the 400 × 400 experiment we observe a substantial overhead due to the smart decision tool. This is not the case on the AMD architecture, and we have no tangible explanation for this observation. At the same time we can observe for the 50 × 50 experiment that the smart decision tool leads to generally improved performance, for which we likewise lack a convincing explanation.

In the 400 × 400 experiment in Fig. 9, however, we must observe that our tool does not properly detect the obvious sweet spot of using 8 cores. This must be attributed to our fourth-order polynomial interpolation (see Sect. 4) of the discrete measurements. The choice of fourth-order polynomial interpolation was

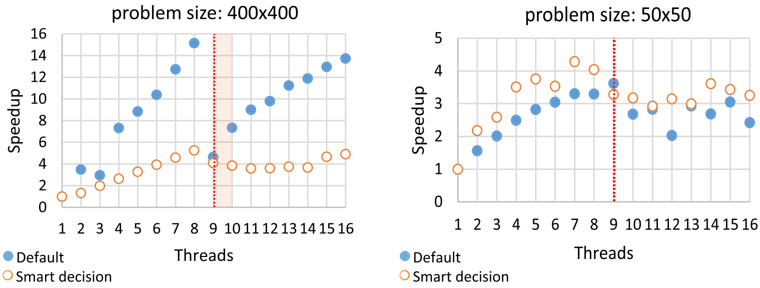

Fig. 9 Performance on Intel 8-core hyperthreaded system with and without the proposed smart decision tool for two different problem sizes and spinning barrier implementation; smart decision tool recommendations: 9 and 9

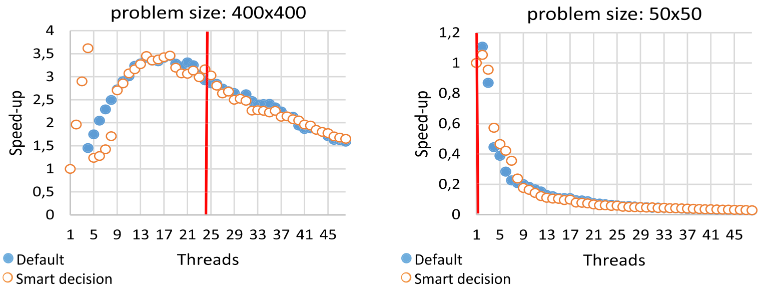

Fig. 10 Performance on AMD 48-core system with and without the proposed smart decision tool for two different problem sizes and suspending barrier implementation; smart decision tool recommendations: 24 and 1

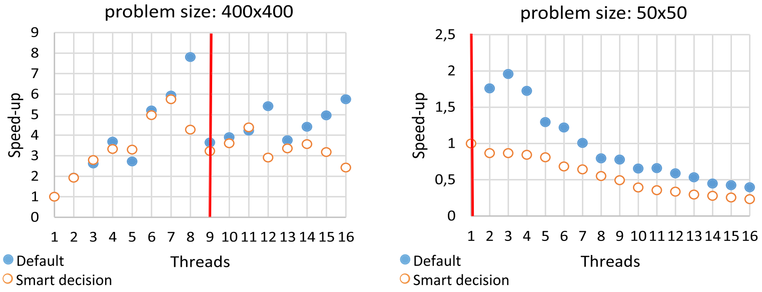

Fig. 11 Performance on Intel 8-core hyperthreaded system with and without the proposed smart decision tool for two different problem sizes and suspending barrier implementation; smart decision tool recommendations: 9 and 1

made in anticipation of speedup curves like the one shown in Fig. 1, but not as the one actually observed in Fig. 9. The interpolation suggests a speedup when going from 8 cores to 9 cores that we (unsurprisingly) cannot observe in practice. Knowing the architecture of the Intel system it is of little comfort that the prediction for the 50×50 experiment turns out to be spot-on, at least based on the default figures without using the smart decision tool.

We repeat all experiments using suspending barriers instead of spinning barriers and report the results in Fig. 10 for the AMD system and in Fig. 11 for the Intel system, respectively. We can immediately recognise the considerably higher overhead of suspending barriers (see Fig. 7) that makes the 400×400 graph for suspending barriers resemble the 50×50 graph for spinning barriers. Again, the fourth-order polynomial interpolation leads to a slight misprediction of the optimal thread count.

Running the 50×50 experiment with suspending barriers even results in a severe slowdown. 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.

On the Intel system (Fig. 11) we observe variations of similar behaviour as in our previous experiments. The use of suspending barriers instead of spinning barriers induces major overhead, and the unanticipated speedup curve on the hyperthreaded architecture irritates our smart decision tool such that it is again off by one core or thread for the larger problem size 400 × 400.

Although the observation is more pronounced on the Intel system, we see a similar unanticipated speedup curve on the AMD system: linear speedup up to four threads followed by a sharp performance drop and more speedups when increasing the thread count further. While this again can be explained by the underlying system configuration of the AMD system with four processors with 12 cores each, this unanticipated behaviour irritates our tool.

7 Related Work

Many parallel programming approaches provide basic means to switch off parallel execution of otherwise parallel loops. For example, Chapel[5] provides the command line option `-dataParMinGranularity` to set a minimum problem size for parallel execution of implicitly data-parallel operations on a per program basis. Prior to our current work, the SaC compiler adopted a similar strategy: at compile time users may set a minimum index set size for parallel execution of with-loops, and the generated code decides at runtime between sequential and parallel execution based on the given threshold [16].

The `if`-clause of OpenMP[9] allows programmers to toggle the execution of parallel code regions between sequential execution by the master thread and fully parallel execution by all threads based on the runtime values of variables specifically for individual parallel regions. OpenMP 2.0 introduced the `num_threads`-clause, which allows programmers to precisely specify the number of threads to be used for each parallelised loop. Like in the `if`-clause, the `num_threads`-clause contains an arbitrary C or Fortran expression that may access all program variables in scope. However, programmers are completely on their own when using these features of OpenMP.

This gap is filled by a multitude of performance analysis and tuning tools as for example Intel's VTune [22]. Corresponding guidelines [13] explain the issues involved. These tools and methodologies allow performance engineers to manually tune 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 approach is automatic with the sole exception that users must explicitly compile their source for training and for production mode and run training codes on representative input data. Furthermore, our approach works on intermediate code *after* far-reaching code restructuring through compiler optimisation, whereas manual tuning operates on source code, which restricts the effectiveness of compiler transformations and suffers from decoupling between source and binary code.

With respect to automation, *feedback-driven threading* [32] proposed by Suleman et al. even goes a step further than we do and advocates a completely implicit solution: In an OpenMP-parallelised loop they peel off up to 1% of the initial iterations.

These initial iterations are executed by a single thread while hardware performance monitoring counters collect information regarding off-chip memory bandwidth and cycles spent in critical sections. Following 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 beauty of being completely transparent to users, the approach of feedback-driven threading has some disadvantages. Considerable overhead is introduced at production runtime for choosing the (presumably) best number of threads. This needs to be offset by more efficient parallel execution before any total performance gain 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 when simply using all available threads.

In contrast to our approach, 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.

Cytowski and Szpindler [8] pursue a similar approach as Suleman et al. with their SOMPARlib. Restricted to a specific application design with a central, identifiable simulation loop they run and profile the first N iterations of that loop with different numbers of threads, starting with the maximum number and then systematically reducing the number dividing by two. All remaining iterations are then run with the best performing number of threads. This approach has about the same drawbacks as that of Suleman et al. Moreover, we understand that thread counts are not individually determined per OpenMP parallel code region.

Pusukuri et al. proposed *ThreadReinforcer* [29]. While their motivation is similar to ours, their proposed solution differs in many aspects. 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. Similar to feedback-driven threading by 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* [24]. 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 [23] 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, e.g. [7, 25]. In a sense we also propose such a trade-off, but for us 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.

Specifically in the context of OpenMP Sreenivasan et al. [31] propose an autotuning framework. This work goes beyond thread counts and includes scheduling technique and granularity (chunk size) in the parameter set, all on the basis of individual parallel loops. Combining a compact parameter space description with model-based search Sreenivasan et al. demonstrate the performance impact of (auto-)tuning.

In the area of parallel code skeletons Collins et al. [6] use machine learning techniques to find near-optimal solutions in very large parallelisation parameter search spaces. They demonstrate their tool MaSiF both on Intel TBB and on FastFlow [2]. Like with Sreenivasan et al.'s autotuning framework, MaSiF is designed to facilitate application tuning for expert programmers, whereas our approach in the context of SaC is explicitly geared at the casual user who expects decent performance (almost) without extra effort.

Another parallel skeleton library is SkePU [12]. SkePU exploits the C++ template mechanism to provide a number of data-parallel skeletons with implementations for sequential execution, multi-core execution using OpenMP as well as single- and multi-GPU execution based on either CUDA or OpenCL. SkePU applies a similar offline/online approach as we do for auto-tuning a range of execution parameters from OpenMP thread count for CPU-computing to grid size and block size for GPU-computing to automatic selection of the best-suited hardware in relation to the problem size [10]. SkePU does this on a per skeleton basis for the selected hardware, but does not take application-specific argument functions into account.

Wang and O'Boyle apply machine learning techniques, namely artificial neural networks, to find near-optimal OpenMP thread counts and scheduling techniques [33]. Their approach is based on extracting a number of characteristic code features at IR-level. They extract these feature vectors for a variety of representative benchmarks to train the neural network following a supervised learning approach. Training is based on potential thread counts and scheduling policies on one side and on their corresponding experimentally determined performance on the other side. The resulting performance model can be integrated into a compiler to make static decisions regarding thread count and scheduling policy based on the extracted features of compiled programs and the trained neural network.

Highly optimising compilers work with numerous heuristics and thresholds that are hard, if not impossible, to statically determine in any near-optimal way across target architectures and applications. A comprehensive survey of machine learning applied to compilation technology can be found in [34].

Coming back to SaC we mention the work by Gordon and Scholz [14]. They aim at adapting the number of active threads in a data-parallel operation to varying levels of competing computational workload in a multi-core system that is not in exclusive use of a single application. The goal is to avoid context switches and thread migration and instead to vacate oversubscribed cores. For this purpose they continuously monitor execution times of data-parallel operations. When observing significant performance changes Gordon and Scholz adapt the number of threads of the running SaC application. Their work differs from ours not only in the underlying motivation, but likewise in the pure online approach.

Moore and Childers [27] similarly address the problem of multiple multi-threaded applications running the same multi-core system, but they use machine learning techniques. Based on offline profiling data they build application utility models using multi-dimensional linear regression. These application utility models are used online to determine the distribution of compute resources among independently running and scheduled applications, including dynamic reconfiguration of applications to change the number of threads.

8 Conclusions and Future Work

Malleability of data-parallel programs offer interesting opportunities for compilers and runtime systems 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.

We explore this opportunity in the context of the functional data-parallel array language SaC, for which we propose a combination of instrumentation for offline training, generation of production code with thread count oracles and runtime system support. Offline training runs with instrumented code create persistent profiling databases. Profiling data is incorporated into production code through recommendation tables. Last not least, the runtime system consults these recommendation tables to effectively employ the recommended number of threads individually for each data-parallel operation.

Following our experimental evaluation in Sect. 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. Here, the use of fourth-order polynomial interpolation is in particular critique.

Furthermore, we must refine our barrier implementations to activate worker threads more selectively instead of merely not assigning worker threads actual work to do by mean of with-loop-scheduling. And we plan to explore how we can speed up suspension 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 initially underestimated is the by nature short execution time of those data-parallel operations for which our work is particularly relevant. Consequently, overall performance is disproportionately affected by synchronisation and communication overhead. Short parallel execution times likewise incur large relative variation.

For the time being, our work is geared at SaC's shared memory code generation backend. While our technical solutions do not directly carry over to other 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 backend for cluster systems [26] the problem we investigate in this paper 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 [21] the question boils down to when to effectively use the GPU and when it would be beneficial to rather use the host CPU cores only. Last not least, our heterogeneous systems backend [11] 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.

Acknowledgements This work received partial funding from the Erasmus+ Strategic Partnership for Higher Education FE3CWS (*Focusing Education on Composability, Comprehensibility and Correctness of Working Software*), under Grant Number 2017-1-SK01-KA203-035402. We thank the Advanced School for Computing and Imaging (ASCI) for access to their compute infrastructure and the anonymous reviewers for their valuable comments, suggestions and pointers to additional related work.

References

1. Agrawal, K., He, Y., Hsu, W., Leiserson, C.: Adaptive task scheduling with parallelism feedback. In: 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Processing (PPoPP'06). ACM (2006)
2. Aldinucci, M., Danelutto, M., Kilpatrick, P., Torquati, M.: FastFlow: high-level and efficient streaming on multi-core. In: Pllana, S., Xhafa, F. (eds.) Programming Multi-core and Many-Core Computing Systems. Wiley, New York (2014)
3. Bailey, D., et al.: The NAS parallel benchmarks. Int. J. Supercomput. Appl. **5**(3), 63–73 (1991)
4. Catanzaro, B., et al.: Ubiquitous parallel computing from Berkeley, Illinois, and Stanford. IEEE Micro **30**(2), 41–55 (2010)
5. Chamberlain, B., Callahan, D., Zima, H.: Parallel programmability and the chapel language. Int. J. High Perform. Comput. Appl. **21**(3), 291–312 (2007)
6. Collins, A., Fensch, C., Leather, H., Cole, M.: MaSiF: machine learning guided auto-tuning of parallel skeletons. In: 20th International Conference on High Performance Computing (HiPC'13). IEEE (2013)
7. 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)
8. Cytowski, M., Szpindler, M.: Auto-tuning of OpenMP Applications on the IBM Blue Gene/Q. PRACE (2014)
9. Dagum, L., Menon, R.: OpenMP: an industry-standard API for shared-memory programming. IEEE Trans. Comput. Sci. Eng. **5**(1), 46–55 (1998)
10. Dastgeer, U., Enmyren, J., Kessler, C.: Auto-tuning SkePU: a multi-backend skeleton programming framework for multi-GPU systems. In: 4th International Workshop on Multicore Software Engineering (IWMSE'11). ACM (2011)

11. Diogo, M., Grelck, C.:. Towards heterogeneous computing without heterogeneous programming. In: 13th Symposium on Trends in Functional Programming (TFP'12). LNCS 7829. Springer (2013)
12. Enmyren, J., Kessler, C.: SkePU: a multi-backend skeleton programming framework for multi-GPU systems. In: 4th International Workshop on High-Level Parallel Programming and Applications (HLPP'10). ACM (2010)
13. Gillespie, M., Breshears, C.: Achieving Threading Success. Intel Corporation, Santa Clara (2005)
14. Gordon, S., Scholz, S.: Dynamic adaptation of functional runtime systems through external control. In: 27th International Symposium on Implementation and Application of Functional Languages (IFL'15). ACM (2015)
15. Grelck, C.: Implementing the NAS benchmark MG in SAC. In: 16th IEEE International Parallel and Distributed Processing Symposium (IPDPS'02). IEEE (2002)
16. Grelck, C.: Shared memory multiprocessor support for functional array processing in SAC. J. Funct. Program. **15**(3), 353–401 (2005)
17. Grelck, C.: Single Assignment C (SAC): high productivity meets high performance. In: 4th Central European Functional Programming Summer School (CEFP'11). LNCS 7241. Springer (2012)
18. Grelck, C., Blom, C.: Resource-aware data parallel array processing. In: 35th GI Workshop on Programming Languages and Computing Concepts, Research Report 482, pp. 70–97. University of Oslo (2019)
19. Grelck, C., Scholz, S.-B.: SAC: a functional array language for efficient multithreaded execution. Int. J. Parallel Program. **34**(4), 383–427 (2006)
20. Grelck, C., Scholz, S.-B.: Merging compositions of array skeletons in SAC. J. Parallel Comput. **32**(7+8), 507–522 (2006)
21. 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). ACM (2011)
22. Intel: Threading Methodology: Principles and Practices. Intel Corporation, Santa Clara (2003)
23. Jung, C., Lim, D., Lee, J., Han, S.: Adaptive execution techniques for SMT multiprocessor architectures. In: 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Processing (PPoPP'05). ACM (2005)
24. Lee, J., Wu, H., Ravichandram, M., Clark, N.: Thread tailor: dynamically weaving threads together for efficient, adaptive parallel applications. In: 37th International Symposium on Computer Architecture (ISCA'10) (2010)
25. Li, J., Martinez, J.: Dynamic power-performance adaptation of parallel computation on chip multiprocessors. In: 12th IEEE Symposium on High Performance Computer Architecture (HPCA'06). ACM (2006)
26. Macht, T., Grelck, C.: SAC goes cluster: fully implicit distributed computing. In: 33rd IEEE International Parallel and Distributed Processing Symposium (IPDPS'19). IEEE (2019)
27. Moore, R., Childers, B.: Building and using application utility models to dynamically choose thread counts. J. Supercomput. **68**(3), 1184–1213 (2014)
28. Nieplosha, J., et al.: Evaluating the potential of multithreaded platforms for irregular scientific computations. In: ACM International Conference on Computing Frontiers. ACM (2007)
29. Pusukuri, K., Gupta, R., Bhuyan, L.: Thread reinforcer: dynamically determining number of threads via OS level monitoring. In: International Symposium on Workload Characterization (IISWC'11). IEEE (2011)
30. Saini, S., et al.: A scalability study of Columbia using the NAS parallel benchmarks. J. Comput. Methods Sci. Eng. (2006). https://doi.org/10.12921/cmst.2006.SI.01.33-45
31. Sreenivasan, V., Javali, R., Hall, M., Balaprakash, P., Scogland, T., de Supinski, B.: A framework for enabling OpenMP auto-tuning. In: OpenMP: Conquering the Full Hardware Spectrum (IWOMP'19). LNCS 11718. Springer (2019)
32. 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). ACM (2008)
33. Wang, Z., O'Boyle, M.: Mapping parallelism to multi-cores: a machine learning based approach. In: 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'09). ACM (2009)
34. Wang, Z., O'Boyle, M.: Machine learning in compiler optimization. Proc. IEEE **106**(11), 1879–1901 (2018)