Delft University of Technology

Towards Dynamic Adaptation of Parallelism for Better Performance and Less Energy Consumption

Cedric Blom

November 9, 2016

Supervisors dr. Clemens Grelck
University of Amsterdam

prof. Henk Sips
Delft University of Technology

Abstract

It is possible to reduce the computation time of data parallel programs by dividing the computation work among different threads and executing them on multiple compute cores. Some data parallel programs are functionally correct only when they are solved with a specific number of threads. For these programs, the question whether or not parallelization reduces computation time, can be answered with a simple "yes" or "no". Other data parallel programs can be solved with any number of threads. This study focusses on these other data parallel programs which can use any number of threads and are still able to function correctly. With those data parallel programs, there is not a single answer to the question whether or not parallelization reduces computation time, it depends on the number of threads used.

There are many things that impact the computation time when data parallel programs are executed. For example: overhead, the cache size, and/or the data size. Therefore, the relation between the number of threads and the computation time of a data parallel program is not linear. A typical time-thread curve has a first part where it decreases towards a certain minimum time, and a second part where the curve increases. In the first part, there is sufficient data to make efficient use of the extra threads. In the second part of the curve, threads can no longer be used efficiently, therefore there is no further decrease in execution time possible. Since, the overhead still increases, the curve increases as well. In order to make the computation time of data parallel programs as short as possible, it is essential to determine the optimal number of threads.

There is another reason not to use just the maximum number of threads. To execute multiple threads in parallel, extra hardware is needed. Therefore, the energy consumption increases, when more threads are used. By using less than the maximum number of threads energy can be saved by deactivate threads that are not used for computations.

We describe a method for automated determination of the optimal number of threads. We have developed a technique that we call the smart decision tool. With the smart decision tool the number of threads needed for maximum speedup can be determined for every parallel computation task in a program.

The program language Single assignment C (SaC) was used as a host for our research. SaC is a functional programming language with a syntax that is similar to C. The programming language has a special syntactical construction (array comprehension) with which data parallel operations can be described. The SaC compiler can turn these array comprehensions into program parts that can be executed on any number of threads. With the add-in of the smart decision tool the number of threads that are actually involved in a parallel computation can be optimized.

Single assignment C puts threads that are not used for computations in a spinlock. These threads in spinlock consume energy. No energy is saved when a part of the treads is hold in spinlock. We have developed barriers that deactivate threads that are not used for computations. In this way energy can be saved.

Contents

1 **Introduction** 5

2 **Introduction to SaC** 11

3 **Runtime system of SaC** 18

4 **Implementation** 21

4.1 Smart decision tool – training mode 21

4.2 Smart decision tool – decision mode 23

4.3 Implementation of barriers 25

5 **Evaluation** 33

5.1 Evaluating speedup model 33

5.2 Scheduler 37

5.3 Evaluation of barriers 39

6 **Simple energy model** 43

7 **Related work** 49

8 **Discussion** 53

9 **Conclusions** 57

References 59

1 Introduction

Increasing numbers of computer systems feature multi-core processors. Using a multi-core processor, a computing task can run on multiple cores by dividing the task into a number of parts. Each core then solves a part of the computing task. Since the cores work in parallel the computing time can be shortened. If we refer to the number of cores as p, in the most favourable instance the computing time is p times shorter than if the computing task were to be executed sequentially. We speak of a speed-up of p. [1]

Unfortunately, the speedup is not always equal to p, the number of cores or threads. The speedup can even decrease if too many cores or threads are used. An endeavour will be made to reveal this by use of the curves in Figure 1 and a model based on the definition of speedup. Because the speedup can reduce, it is of importance to the computing time of a parallel program to select the optimal number of cores or threads. Examined in this study is whether the finding of the optimal number of threads can be automatically determined. We shall explain the object of this study in more detail later, using Figure 1 and a speedup model.

So far, we more or less implicitly assumed that a computing task can be divided into any number of parts, that can be executed in parallel. This is not true in general. Some computing tasks can only be solved with a specific number of threads. However, in this thesis, we will only focus on data parallel computing tasks that can, theoretically speaking, be divided in any number of parts. In this specific class of programs, we can study the effect of the number of threads on speedup.

Speedup is defined as in equation (1).

$$S = \frac{T_s}{T_p} \tag{1}$$

In this, T_s is the time needed to solve a problem using the fastest known sequential algorithm. T_p is the time needed to solve the same problem in parallel. Suppose that a problem consists of n sub-problems and that each sub-problem can be solved in a fixed amount of time. If we introduce a new unit that is precisely the same size as the time needed to solve a single sub-problem, then measured in that unit it takes $T_s = n$ time to solve the entire problem sequentially. Supposing that the problem can be parallelized perfectly, then equation (2) indicates the time in which the entire problem can be solved by parallelization, with p being the number of cores or threads that are used to solve the problem.

$$T_p = \frac{n}{p} \tag{2}$$

The speedup in this example is thus equal to equation (3) [2]. p is the upper limit for speedup. This upper limit can also be obtained using Amdahl's law when the parallel fraction f is equated to 1. In his law Gene Amdahl put forward that speedup is limited by the sequential fraction $(f-1)$ of a parallel program. After all, the computing time taken in the execution of sequential fraction cannot be shortened by using more processors/threads (p). Amdahl's law is shown in equation (4) [3] [4].

$$S = \frac{n}{\frac{n}{p}} = p \tag{3}$$

$$S = \frac{1}{(1-f) + \frac{f}{p}} \tag{4}$$

There are countless elements that can make a negative contribution to the speedup upper limit. For example, no account is taken of the communication overhead that occurs when multiple cores or threads are used. Various studies demonstrate that models can be made to estimate the amount of communication overhead that accompanies each thread. [5] [2] If we call this estimated overhead c and work on the basis of the "perfect machine", then the total communication overhead for p threads is equal to cp. Equation (5) is an indication for what T_p may become when the communication overhead is taken into account. Equation (6) is a simple model for the anticipated speedup when a problem is solved by parallelization and the communication overhead is taken into account.

$$T_p = \frac{n}{p} + cp \tag{5}$$

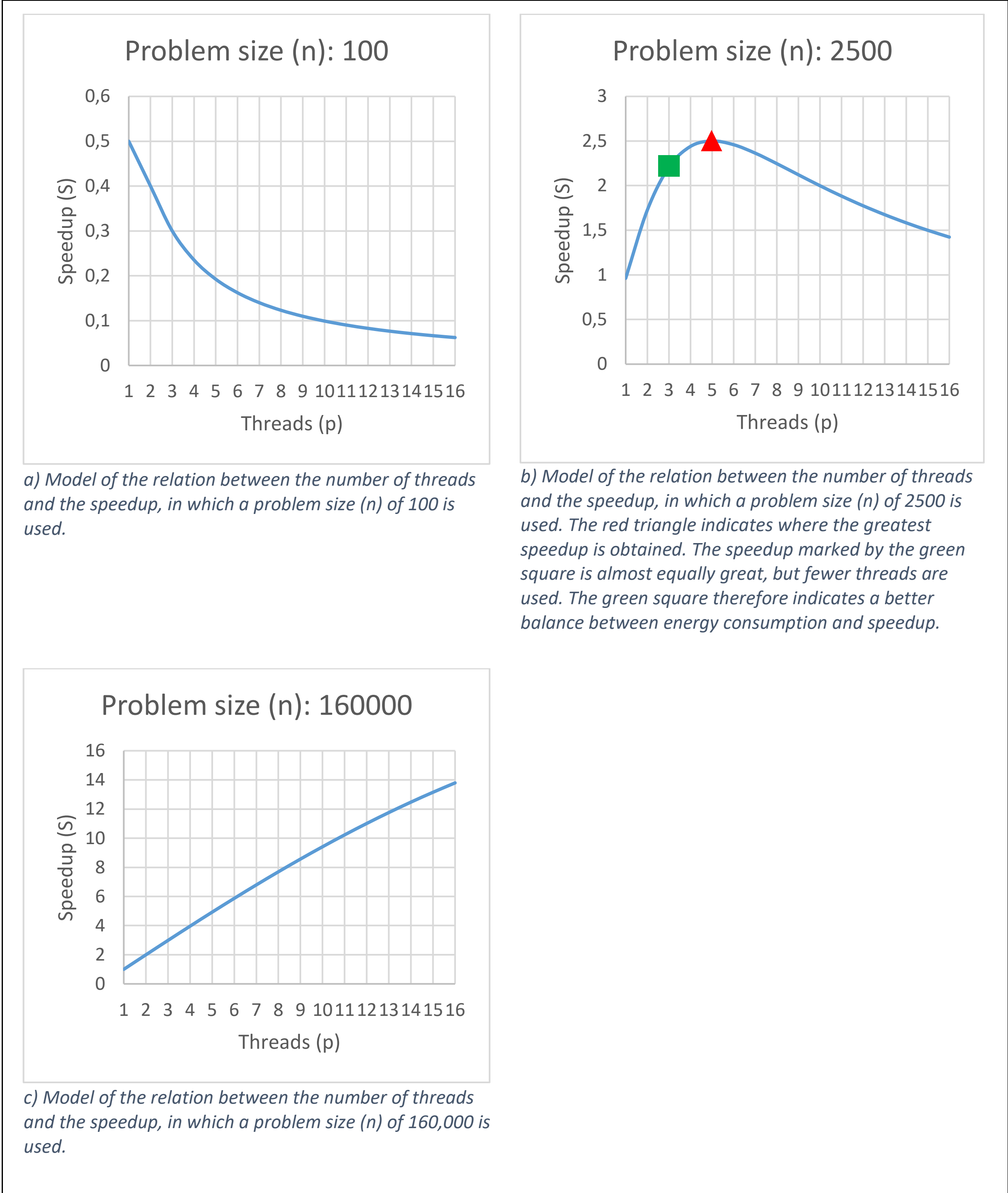
$$S = \frac{n}{\frac{n}{p} + cp} \tag{6}$$

In Figure 1 the relation is established between the number of threads (p) and the speedup (S) by making use of the above model. Used for the graphs are, respectively, a small (100), a medium (2,500), and a large value (160,000) for the input size (n). For each graph, the same value of (100) is used as the amount of overhead costs (c). The graphs show mostly increasing curves except for the first graph where a small value for n was used. With a problem size of 100 the use of more threads only creates worse results. With a problem size of 2,500, there is a maximum at 5 threads. The maximum speedup is $S = 2.5$. With a problem size of $n = 160{,}000$ the maximum even lies outside the range of the graph, and the maximum speedup that still remains within the range of the graph is equal to $S = 13.8$.

Figure 1 demonstrates that the greatest speedup is not always obtained using a maximum number of threads. It is therefore important to select the optimal number of threads to obtain the maximum performance. That is not always easy. Figure 1 is based on a simple model with 3 parameters. In reality there are many more parameters that can influence the relation between the number of threads and the speedup. One can think here of cache size, the data size but also of the limitations of the algorithm itself. The model implicitly assumes that a problem can be divided in n equal sub-problems, but not each algorithm can be subdivided in this manner. With other algorithms, a problem can be subdivided into a limited number of sub-problems only. Yet other algorithms do not scale because their complexity increases with the number of threads used.

An algorithm does not generally work in isolation, but is part of a program. Such a program does not have to limit itself to the execution of a single parallel algorithm only. A parallel program consists of sequential and parallel code sections that alternate during execution. The relation between the number of threads and the speedup can be different for every parallel piece of code. If one chooses to let all parallel pieces of code work with the same number of threads, a maximum speedup will not be achieved for each part. If a maximum performance is required for all parallel pieces of code, then the number of thread must be adjusted dynamically. The number of threads necessary to achieve the maximum speedup must be determined for every parallel piece of code. As has been said, it can be difficult to determine this for a single parallel section. If a program has many parallel sections, the

determination of the number of threads can be so complicated that it is no longer possible to do this manually. Bear in mind that a number of parameters, such as the earlier-mentioned cache size, can be machine-dependent. The determination of the number of threads for maximum speedup must be done for each type of machine.



a) Model of the relation between the number of threads and the speedup, in which a problem size (n) of 100 is used.

b) Model of the relation between the number of threads and the speedup, in which a problem size (n) of 2500 is used. The red triangle indicates where the greatest speedup is obtained. The speedup marked by the green square is almost equally great, but fewer threads are used. The green square therefore indicates a better balance between energy consumption and speedup.

c) Model of the relation between the number of threads and the speedup, in which a problem size (n) of 160,000 is used.

Figure 1: The above curves indicate the relation between the number of processors/threads (p), the speedup (S) and the problem size that is obtained using equation (6).

In this study, research was carried out to discover whether the determination of the number of threads for maximum speedup could be computerized. This research has led to the development of the smart decision tool. The smart decision tool uses an offline profile-based technique to determine the number of threads for maximum speedup. With the smart decision tool, a program can be set up

in two steps to use the number of threads for maximum speedup. In the first step the program is run in training mode. The relation between the number of threads and the speedup is then determined for each parallel piece of the program's code. This is achieved by running the program in the environment in which the optimized version for speedup will ultimately be run. The computation time for each parallel piece of code is measured during this test run. This is not carried out just once, but a number of times. Each time, more threads are deployed for the computation, beginning with 1 thread. In this way, the speedup can be set against the number of threads for each parallel piece of code.

In the second step, the number of threads for maximum speedup is determined for each parallel executable piece of code on the basis of the thread-speedup relations. A user defined gradient is used to help the smart decision tool determine what number of threads to use. The first point at which the gradient of the thread-speedup curve is smaller or equal to the user-defined gradient is seen as being the user-required point. The number of threads at this point is used in the computation concerned. The user can select a gradient of 0, by which the smart decision tool is instructed to find the maximum speedup under the given circumstance.

By use of the system with gradients it is not only possible for a user to instruct the smart decision tool to find the maximum speedup, he can also choose a point at which a balance is achieved between energy consumption and speedup. The more threads that are used, the more energy is consumed. This is because threads can only run in parallel if extra hardware is used. By keeping this extra hardware active energy is consumed. When the energy consumption has to be minimal, it is best to use the least number of threads. However, that can be at the expense of speedup. A balance must be sought when both energy consumption and speedup are important. By using the gradient system the user can define at what point the increase in speedup gets to small in respect to the increase in energy consumption.

The programming language Single assignment C (SaC) is used as a model language to enable testing of the smart decision tool. The smart decision tool was integrated in the compiler of SaC. SaC is a functional programming language with a syntax that is comparable to C. [6] The programming language has a special syntactical construction (array comprehension) with which data parallel operations can be described. This array comprehension is known as the with-loop. Shown in Figure 2 are two examples of functions written in SaC. [7]

```
double sum(double[.] x) {
      len = shape(x)[0];
      s = 0.0;

      for (i = 0; i < len; i++) {
            s = s + x[i];
      }

      return s;
}


double[.,.] matmul(double[.,.] a, double[.,.] b) {
      n = shape(a)[0];
      m = shape(b)[1];

      d = with {
            ([0,0] <= [i,j] < [n,m]) : sum(a[i,.] * b[.,j]);
      } : genarray([n,m], 0.0);

      return d;
}
```

Figure 2: Example functions written in Single assignment C (SaC). The piece of code highlighted in green is an example of a with-loop construction. It describes a data parallel operation.

The function “sum” is sequential. It calculates the sum of the elements of the vector “x” by adding together all the numbers of the vector in a for-loop. With the exception of a few differences, the syntax of the sum function is equal to C. The data type of the parameter “x” is “double[.]”. This data type is used to declare a one-dimensional array (vector) of an as yet unspecified length. With SaC it is not possible to refer to locations in the computer’s register using pointers. [6] For this reason, there is a special syntax to declare data types with one or more dimensions. Within functions in SaC it is not necessary to indicate types for variables. SaC calculates the types itself as soon as a value is assigned to a variable. For this reason, the data types of the variables “len”, “s” and “i” are omitted. Unlike C, SaC has the shape function. [8] This is a standard function of SaC. The shape function returns a vector with in it the length of each dimension of a given array. In the example function, shape is used to determine the length in the first and only dimension of the vector “x”.

With the second example function (“matmul”) the inner product of two matrices can be calculated. The function therefore has as its parameters two two-dimensional arrays (matrices) named “a” and “b”. The function uses the with-loop construction to describe a data parallel operation that will compute the inner product. In Figure 2 the with-loop construction is highlighted in green. Written in the last line of the with-loop construct is that a two-dimensional array of “n” by “m” elements must be created. After this, the two-dimensional array is filled by multiplying all the columns of matrix “a” by all the rows in matrix “b” and to calculate the sum of each column-row product. The example function “sum” is used for the computation of the sum. Thanks to the with-loop construct the localization of parallel executable pieces of code is simple. During compilation, the smart decision tool can relatively easily modify the with-loops such that pieces of code for time measurements or thread number modification are added. This makes SaC very suitable as a host language for our research.

In order to improve the performance of SaC programs, threads are not repeatedly recreated for executing each with-loop. Instead, a user-defined maximum number of threads is created when a SaC program is started. The threads remain active during the entire execution of the program. [7] [9] Spinlock barriers are used to block threads as sequential tasks are being executed. One disadvantage of spinlocks is that threads consume energy even if they are not being used in solving computations. The smart decision tool gives the user the opportunity to use fewer threads than the number that is needed for the greatest speedup. In this way, energy can be saved, but only as the excess threads are actually deactivated.

For this reason, time was spent during the research not only on the development of the smart decision tool, but also on the development of new barriers for SaC. These new barriers use techniques by which threads can be deactivated temporarily. An important aspect of the new barriers is that the overhead must remain limited. If the overhead is very large this will be at the expense of the speedup of the SaC program. In order to discover which barrier technique provides the least overhead three different barriers were implemented and tested.

Figure 22 is a schematic comparison between the execution model of a SaC program using the spinlock barriers and the execution model of a SaC program using one of the new barriers. The figure shows that by using the spinlock barrier all threads remain active. The new barriers on the other hand deactivate threads that are not being used in any computation. As a result energy can be saved both during sequential execution phases of a program as well as during parallel execution phases of a program.

During this research answers were sought for the following questions:

1. With respect to speedup, how can an optimum level of parallelism be found for any given SaC program?
2. What consequences does the smart decision tool have on the computation time needed to solve parallel computations?
3. How can barriers be developed that have limited overhead but with the capacity to deactivate threads so that less energy is consumed?
4. What influence will the barriers that are developed have on the overhead and the energy consumption of SaC programs?

The remainder of this thesis is organized as follows. In chapter 2 a basis is laid for the programming language SaC. To integrate the smart decision tool with SaC, some parts of the runtime system were changed. The parts of the runtime system that are most critical for understanding how the smart decision tool is able to control SaC, are explained in more detail in chapter 3. The implementation of the smart decision tool in SaC is explained in chapter 4.1 and 4.2. The first step, the training mode, is discussed in chapter 4.1. In chapter 4.2 the second step, the decision mode, is discussed. A description is made how the number of threads is determined from the time-thread curve obtained in this step. In chapter 4.3 there is detailed discussion as to which techniques are used for the new barriers. Chapter 5 is an evaluation of the techniques discussed in chapter 4. In chapter 5.1 and 5.2, on the basis of a number of examples, insight is provided into the working of the smart decision tool under different circumstances. In chapter 5.3 the working of the various barriers is evaluated. In chapter 6 a simple energy model is introduced. The model is used to predict whether in practice the new barriers can be used to lower energy consumption. Chapter 7 provides an overview of research related to this study. Chapter 8 is a discussion based on the results obtained in the evaluation (chapter 5). The conclusions that we draw from this study are given in chapter 9.

2 Introduction to SaC

Single assignment C (SaC) is a functional programming language, but the language's syntax is very similar to C (hence the name Single assignment C). The syntax is deliberately similar to C, so that a changeover to SaC is less difficult for programmers who are used to imperative programming languages. Just as in C, variables are typed and a program can be described using statements such as the if-branch, the while-loop and the for-loop.

Nonetheless, there are differences. Because SaC is a functional programming language, the state of a program cannot be changed by adapting variables (no side effects). Variables are not references to places in memory, but represent a certain value. As a result, it is not possible to change the value of a variable after a value has been assigned to that variable (single assignment). This has implications for the programming language. [6] [10]

Where in C it is normal to use pointers, this is not possible in SaC. Also, SaC has no global variables. However, in SaC it is permitted to use an identical variable name on each side of an assignment. In this way, the expression "x += x" is simply permitted in SaC. However, the expression is interpreted differently in SaC than in C. If the expression is executed in C, the register value "x" is doubled. In SaC the values before and after the assignment are seen as two different variables that happen to have the same name. In such a case, SaC creates a new scope with a variable "x", the value of which is twice as large as that of the old variable "x". By applying this strategy consistently, the C syntax is preserved and at the same time the requirements of SaC as a functional language are met. [6]

SaC is designed for the execution of data-parallel computations. The data that is necessary to enable the parallel computations must be prescribed with help of multi-dimensional arrays. Multidimensional arrays are also the only data structure that SaC has. All variables or constants in SaC are arrays. Scalar values are arrays without dimensions and a single element, vectors are arrays with one dimension and zero or more elements, matrices are arrays with two dimensions and multiple elements, etc. For the sake of completeness, in SaC it is also possible to define arrays without elements, these are known as empty arrays. Empty arrays can have multiple dimensions. In this way, SaC vectors or matrices can occur without elements. With the exception of scalar values, all arrays are instantiated with square brackets and the values are separated by a comma, as shown in Figure 3.

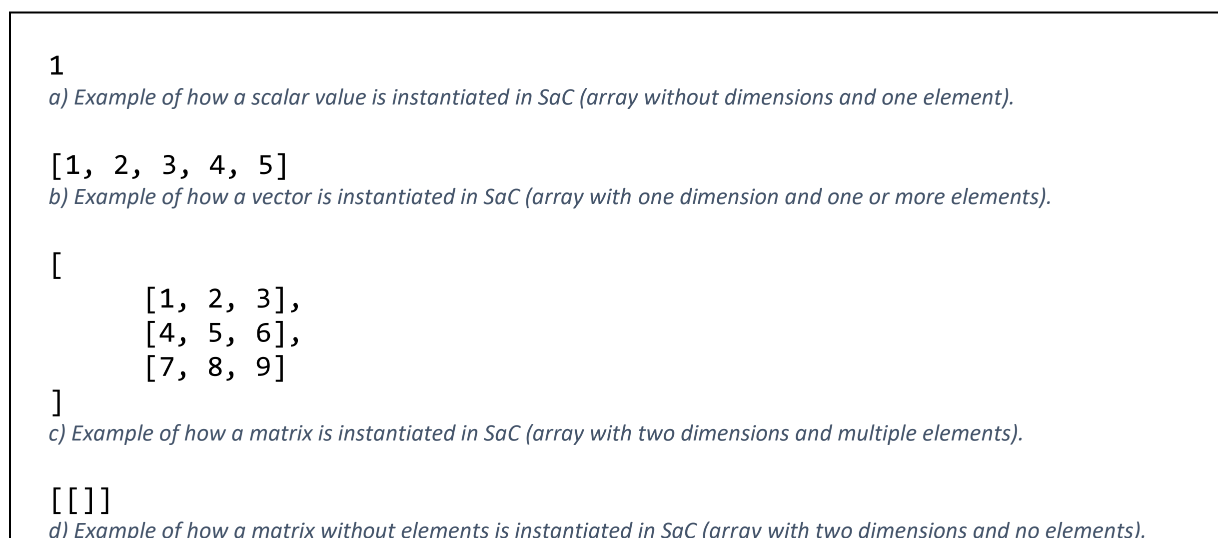


Figure 3: Examples of how arrays can be instantiated in SaC.

Just as in C, variables are typed. Nevertheless, in many instances it is not necessary to declare variables with SaC. For example, with SaC it is not necessary to declare variables that are only used within a function. This is because SaC can determine the type on the basis of the array that is assigned to the variable. Functions and function arguments do need a type. In SaC variables are declared by giving the type of the array, followed by the name of the variable. SaC has five basic types of array: int, float, double, char and bool. The five basic types can be used to declare scalar variables. For every other type of array the dimensionality of the array must be given with the help of a vector. In Figure 4 can be seen how this works for different type of arrays.

In Figure 4a is shown a method for the declaration of a scalar variable. In Figure 4b is shown how a variable for an array with one dimension and five elements can be declared. The length of the vector determines the number of dimensions of the array, and the size of the values determines the number of elements in each dimension. A variable for a matrix can also be declared in the same manner (see Figure 4c). The values of the vector can also be replaced by dots, if the number of dimensions of the array is known only (see Figure 4d). The length of the vector still determines the number of dimensions of the array, but by the use of the dots the setting of the number of elements is postponed until an array is assigned to the variable. If the number of dimensions is also unknown, the method shown in Figure 4e can be used. Finally, if a variable may contain a scalar value or another type of array, the notation that is shown in Figure 4f can be used. [8]

```
int a;
```

a) Example for the declaration of a variable for a scalar value.

```
char[5] b;
```

b) Example for the declaration of a variable for a vector with 5 elements.

```
float[3,3] c;
```

c) Example for the declaration of a variable for a 3x3 matrix.

```
double[.,.] d;
```

d) Example for the declaration of a variable for a matrix, in which the number of elements is not determined until an array is assigned to the variable.

```
int[+] e;
```

e) Example for the declaration of a variable for an array, in which both the dimensionality and the number of elements is not determined until an array is assigned to the variable.

```
double[*] f;
```

f) Example for the declaration of a variable for an array, in which the choice of a scalar value or an array of a higher dimensionality is not made until an array is assigned to the variable.

Figure 4: Example of how variables can be declared in SaC.

SaC has a notation designed especially for the selection of array elements. This notation looks like “a[iv]”, in which “a” denotes an arbitrary array and “iv” a vector. The values of the vector “iv” indicate the position of the elements to be selected in each dimension of the array “a”. The length of the vector “iv” must be smaller than or equal to the number of dimensions of array “a”. If the vector is smaller than the number of dimensions of the array, then all elements of the unspecified dimensions are selected. In this way, whole sub-arrays can be selected. In Figure 5 there are four examples of selection operations in which use is made of the selection notation.

```
M = [
    [1,2,3],
    [4,5,6],
    [7,8,9]
];

A = M[[0,0]];
```
a) Example of selection of the element at position 0 of row 0 of the matrix M, in which use is made of the selection notation. The result of this selection is the scalar value 1. This scalar value is assigned to variable A.

```
B = M[[2]];
```
b) Example of selection of the elements of row 2 of the matrix M, in which use is made of the selection notation. The result of this selection is the vector [7,8,9]. This vector is assigned to the variable B.

```
C = M[[]];
```
c) Example of selection of all elements of the matrix M, in which use is made of the selection notation. The result of this selection is the entire matrix M. This matrix is assigned to the variable C.

```
D = M[[1]][[0]];
```
d) Example of selection in which firstly row 1 of the matrix M is selected, after which the element that is at position 0 of the selected row is choosen. The result of this selection is the scalar value 4. This value is assigned to the variable D.

Figure 5: Examples of how elements of an array "M" can be selected in SaC.

With the described vector “iv” it is not possible to select all elements of each and every dimension. For example, try to select the first column of matrix “M” of Figure 5; this is not possible. In order to make it possible to select all elements of each and every dimension, the dot was devised. In the vector “iv”, numbers can be replaced by a dot in order to select all the elements of the array in the relevant dimension. Figure 6 is an example in which elements of arrays are selected with the help of the selection notation and the dot. [11]

```
M = [
    [1,2,3],
    [4,5,6],
    [7,8,9]
];

A = M[[0,.]];
```
a) Example of selection in which firstly row 0 of the first dimension is selected after which all columns form the row of the second dimension are selected. The result of this selection operation is the vector [1,2,3]. This vector is assigned to the variable A.

```
B = M[[.,0]];
```
b) Example of selection in which firstly all the rows of the first dimension are selected, after which only the elements at position 0 from each row are selected. The result of this selection operation is the vector [1,4,7]. This vector is assigned to the variable B.

Figure 6: Example of how elements from array "M" can be selected in SaC, in which use is made of dots.

Sac has a special construct (array comprehension) by which parallel operations on arrays or parts of arrays can be described. This construct (array comprehension) is known as with-loop. Figure 7

contains an example of a so-called genarray with-loop. In total, there are three types of with-loops: the genarray with-loop, the modarray with-loop and the fold with-loop. The genarray with-loop is used to describe a data-parallel operation in which an array is constructed. With the modarray with-loop a data-parallel operation can be described in which the elements of an existing array are used to construct a new array. Finally, there is the fold with-loop. With the fold with-loop an operation can be described in which arrays or parts of arrays are combined into a single element. The calculation of the sum of all the numbers in an array is an example of an operation that could be described with the help of the fold with-loop. [9] [8]

The with-loops have a fixed structure. In Figure 7 various components of the structure are highlighted in colours. The generators, shown in blue, are used to generate a set of indices for the index vector “iv”. All indices in the set are assigned to the variable index vector “iv”. In Figure 7 the index set is defined by a range. Vectors are used to define the beginning and end of the range. In the figure these beginning and end vectors are hard coded, but it is also possible to use variables (see Figure 9a). If a genarray or modarray with-loop is used, it is possible to replace the vectors with dots. A dot at the left-hand side of the generator stands for the index of the first element of the array that is to be constructed. A dot at the right-hand side of the generator stands for the index of the last element of the array that is to be constructed. The symbol “<” indicates that the range is exclusive of the beginning/end vector. The symbol “<=” is used to indicate a range inclusive of the beginning/end vector.

With the generator-expressions, highlighted in green, a value can be assigned to all elements within the range of the generator. The number 1 has been used in the example, but the generator-expression can also be a complex computation. It is even possible to call a function. The variable “iv” from the generator can be used in the generator-expression to select elements from another array, for example (see Figure 8). The generator and generator-expression occur only as pair, and together they are known as the generator-rule. In the example in Figure 7 only one generator-rule is used, but in practice there are often more.

The operator, highlighted in orange, determines the type of the with-loop. In the example, the genarray operator is used. This operator has two parameters. The first parameter is a vector that determines the number of dimensions and the size of the array that is to be constructed. The second parameter is the value that each element of the array is given by default. In the array that is to be constructed with the with-loop of Figure 7, all the elements are given a default value of 0, but the elements in the range [1,1] to [4,4] are given the value 1.

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

Figure 7: Example of a genarray with-loop. The with-loop constructs a 5x5 array ("A"), in which the outermost elements have a value of 0 and the innermost elements a value of 1. Blue: generator with which a range can be described, green: generator-expression with which a value can be assigned to all the lements within the range, orange: operator by which the type of the with-loop is determined.

Shown in Figure 8 is a simple example of a modarray with-loop and a fold with-loop. The modarray operator needs only one parameter. The parameter is an already existing array. With the generator-rules a description can be made as to which part of the existing array another value be given in order to construct a new array. In the generator of the modarray example of Figure 8, the dot notation is used. One can use the dot notation to refer either to the first or the last element of the array. A dot

at the first position of the generator represents the first element of the array. A dot at the last position of the generator represents the last element of the array.

The fold with-loop has two parameters. The first is a fold function. The second parameter is the neutral element of that function. Firstly, in a fold operation all operations that are described by the generators must be applied, after which the fold operation is applied in order to combine all the values.

A fold function operates on two parameters, and calculated from these is a single value. An important condition for a fold function is that the value calculated must be of the same type as at least one of the parameters. The fold function can then be applied recursively. The outcome of the fold function can serve as a parameter for another fold operation. An example of a fold function is string concatenation for example. The parameters for this function consist of a string and a character. The fold function calculates as a value a new string that is equal to the old string to which the character has been added. A fold operation begins with the neutral element. In the case of string concatenation this can be an empty string for example. Characters can continuously be added to the string by applying the fold function recursively. If the fold operation is executed sequentially there are two natural ways in which this can occur. The character can be the first or the second parameter of the fold function. In other words, a character can be added to a string that has been calculated recursively (fold-right). Or a string can be determined with the help of recursion, after which a further character is added (fold-left).

The advantage of fold-right is the possibility of so-called lazy evaluation. In this, the recursive call can be omitted if the outcome of the computation can be determined from the first parameter only. This will not occur with string concatenation, but if for example multiplication is applied recursively to a list of numbers, the computation can then be stopped as soon as one of the numbers is 0. Multiplication with zero always produces 0. In fold-left lazy evaluation is impossible, because the first parameter is determined from the recursive call of the fold function.

SaC has no fold-left or fold-right operations, because with-loops must be able to execute in parallel. Fold-left and fold-right imply a strict sequential order in which elements are combined. SaC gives absolutely no guarantee as to the order in which the fold operation takes place. Therefore, in SaC no fold functions can be used in which the outcome is dependent on the order in which the fold operation is applied.

```
B = with {
          (. <= iv < .) : A[iv] + 1;
    } : modarray(A);


C = with {
          ([0,0] <= iv < [5,5]) : B[iv];
    } : fold(+, 0);
```

Figure 8: Example of a modarray and fold with-loop. The modarray with-loop construct a 5x5 array ("B") by adding all the elements in array "A". The fold with-loop then calculates the sum of all the elements of array "B".

The examples in Figure 7 and Figure 8 are simple with-loops, but with-loops can be much more complicated. Shown in Figure 9a is a with-loop in which a variable is used to indicate the beginning and end of a range. In Figure 9b is shown a method by which not only the index as vector can be

obtained, but also as individual values. Figure 9c is an example of a with-loop with which multiple generator-rules are used. The generator rules overlap one another. In overlap the value of the last generator-rule is assigned to the element. In the example shown in Figure 9d step and with are used. Step can be used in order not to apply a generator-expression operation to certain elements within the range. If step is set the generator-expression is applied to an element, after which a number of elements are skipped. The number of elements that are skipped in each dimension of the array is determined by the vector that follows the keyword step. With the width option the number of times that an operation is applied to consecutive elements is extended. If width and step are of equal size, no elements are skipped.

```
shp = [3,3];

D = with {
          (0*shp <= iv < shp) : 1;
    } : genarray(shp, 0);
```

a) Example of a genarray with-loop. The "shp" variable is used to indicate the beginning and end of the generator range. The same variable is also used in the operator to indicate the size of the array.

```
E = with {
          ([0,0] <= iv=[x,y] <= [1,1]) : D[iv] + shp[x*y];
    } : modarray(D);
```

b) Example of a modarray with-loop in which the index is obtained as the vector "iv", but also as separate values "x" and "y".

```
F = with {
          (. <= iv <= [1,1]) : 1;
          ([1,1] <= iv <= .) : 2;
    } : genarray([3,3], 0);
```

c) Example of a genarray with-loop in which multiple generators are used. The generators overlap one another in the element [1,1]. Because the generator-expression of the last generator-rule has the value 2, the value 2 will be assigned to the element [1,1].

```
G = with {
          (. <= iv <= . step [3,3] width [2,2]) : 1;
    } : genarray([5,5], 0);
```

d) Example of a genarray with-loop in which step and width are used. The step of [3,3] ensures that after the assignment of the value 1 to the first element, two elements, in both x-direction and y-direction, will immediately be skipped. However, because of the width of [2,2] the value 1 is also assigned to all elements that are adjacent to the element in which the value 1 is first assigned to. The result is a matrix in which all elements have the value 1, apart from two crossing rows of elements in the middle of the array that have the value 0.

Figure 9: Examples of more advanced with-loop operations.

SaC has a number of I/O functions, which will appear familiar to C programmers. For example, SaC has a print function, with which arrays can be written to the standard output buffer, so that they can be displayed on the monitor. There is also a scanf function, to read-out keyboard input. In addition, SaC also has functions such as: fopen, fread, fwrite, and fclose, with which data can be read from or written to a file. These and other functions are available via the StdIO module of SaC. There are also other standard modules for SaC, such as Array, which contains a collection of function for frequently-used array operations. [8] [12]

"Single Assignment C – tutorial" [13] is a manual in which the basic principles of SaC are explained. In this, all the subjects mentioned in this chapter are explained in more detail. Additionally, it provides

information about the SaC module system. In chapter 5 of the manual it is explained how existing modules in SaC can be used and how one's own modules can be made. "Single Assignment C (SaC) – High Productivity meets High Performance" [8] is an extensive introduction to SaC. Explained in the document, among other things, is how SaC interprets the different I/O functions so that the C-syntax is retained, but at the same time the requirements of SaC as a functional language are satisfied. In "Shared Memory Multiprocessor Support for Functional Array Processing in SaC" [9] is described how the with-loops of SaC are translated to C-code.

3 Runtime system of SaC

SaC was used as a host language for the smart decision tool. To integrate the smart decision tool with SaC, some parts of the runtime system were changed. Two parts of the runtime system deserve some extra attention: the scheduler and the barrier system. They are most critical for understanding how the smart decision tool is able to control SaC's runtime system. In this chapter the concepts of the scheduler and barrier parts of the runtime system of SaC are explained.

The scheduler is a part of SaC that ensures that the work is divided among the threads. [9] The scheduler has been adapted to enable setting the number of threads dynamically. The object of the smart decision tool is namely to enable setting the optimum[1] number of threads per parallel executable part of the code. The scheduler has a global variable that stores the total number of threads. The scheduler uses this variable to determine for how many threads work must be divided.

In the original implementation of the scheduler, the variable was set when all the threads were created, and it remained constant thereafter. However, this behaviour was changed to make dynamic adaptations on the number of used threads possible. With the current implementation, the value of the global variable is set dynamically by the smart decision tool. By adapting this parameter dynamically, the scheduler may divide the work among fewer threads than there are in reality. As a result, the scheduler may give work to only some of the threads, which means that the rest of the threads are not used.

SaC offers several scheduling techniques for dividing the work among the threads. There are both static and dynamic techniques available. Static techniques employ an a-priori association of work with threads. One example of a static scheduling technique is block scheduling. With block scheduling each thread is associated with a block of consecutive array elements to operate on. The blocks are all equal in size. SaC uses the block scheduling method by default.

With dynamic scheduling techniques, work is assigned to threads on the go. An example of a dynamic scheduling method is self-scheduling. With this technique, a central task queue is maintained. Each thread receives a bit of work by invoking the task queue. When a thread has completed the work, it has to seek for another job by invoking the task queue again. SaC offers self-scheduling among other dynamic scheduling techniques. More about the SaC scheduler can be found in "Shared Memory Multiprocessor Support for Functional Array Processing in SaC" [9]. In the document, information can be found about the different scheduling techniques that are supported by SaC.

To integrate the smart decision tool with SaC, the barrier system had to be changed as well. The number of threads selected by the smart decision tool can be less than the maximum number of threads. By using less threads energy consumption may reduce. However, in order to achieve energy reduction, threads have to be deactivated. The original SaC compiler uses a system that keeps threads alive even when they are not used. As a result, the energy consumption is constantly at a maximum. In the paragraphs below the technique used by the original (before the integration with the smart decision tool took place) SaC compiler is discussed.

In SaC there exist parallel sections and sequential sections. These sections alternate, therefor SaC has to switch from a sequential execution mode into a parallel execution mode or vice versa. These switches are a source of overhead. For example, to start a parallel section, a scheduler is needed to divide parallel work over the threads. Another source of overhead happens at the end of a parallel section. The program has to wait until all threads are completed, before execution of the sequential

section. The wait time can become a serious source of overhead, when work is not equally divided over the threads. One computation might take more time than another.

SaC has optimizations to keep the overhead small. One such an optimization is that threads are only created at the start of a program. Threads are kept alive until the whole program terminates. An implication of this technique is that SaC has to rely on barriers to be able to begin and end parallel sections. SaC has two types of barriers: start barriers that are used at the begin of a parallel section, and stop barriers that are used at the end of a parallel section. Figure 12a is a schematic representation of SaC's multithreaded execution model with the start and stop barriers. [7] [9]

As can be seen in Figure 12a, sequential sections only have one active thread. This thread is called the master thread. The other parked threads are called the worker threads. The master thread schedules the work over the worker threads and lifts the start barrier at the moment of switch between sequential and parallel execution. When the start barrier is lifted all worker threads become active. The master thread acts as worker thread during parallel execution. With the exception of the master thread, all threads pass the stop barrier and are parked at the next start barrier after completing their work. The master thread hits the stop barrier and waits at the spot until all worker threads have passed. After the passage of the last worker thread the master continues with sequential computing.

The execution model does change when the smart decision tool is integrated in SaC. The main difference is that the smart decision tool can decide to use less than the maximum available number of threads for parallel sections.The threads that are not actively used are parked by the start barrier which is implemented as a spinlock barrier. A spinlock barrier is an active barrier. As a result, threads consume power even while they are parked. We have developed three new barriers to reduce the energy consumption of parked threads.

4 Implementation

The smart decision tool is a technique to find the number of threads that solve a given parallel algorithm with the highest speedup. Alternatively, a user can setup the smart decision tool to find a more energy-efficient number of threads. As shown in Figure 1 highest speedup is not always achieved by using the largest number of threads possible. The smart decision tool uses a profile based technique to find the number of threads of interest. Because of the offline approach, profiling and decision making have to be separated. In SaC this is realized by creating two different modes for the compiler, the training mode and the decision mode. When the compiler is in training mode, it generates a binary file that is meant for profiling. By execution of the binary file a profile is generated. When profiling is completed, the decision mode of the compiler can be used. In decision mode, the compiler will use the above-mentioned profile information to generate an optimized binary file. By having two compilation modes, the profile information can be analyzed at compile time.

4.1 Smart decision tool – training mode

The purpose of compiling a SaC program in training mode and executing the resulting binary is to gain insight in the time-thread relation of each parallel section of the program. In SaC, with loops are the only source for parallelism. There are a few notes. First, a SaC program may contain several with loops, and each with loop may have its own time-thread relation. Second, with loops can be executed multiple times, and each time the problem size of the loop may change. For every problem size the time-thread relation may be different. Finally, a with loop can also be executed multiple times using the same problem size. Executing a single with loop multiple times using the same problem size will result in time-thread relations which appear similar.

At compile time, the smart decision tool gives each with loop its own identifier, by increasing a counter for each with loop that is found in the SaC program. When the size of the with loop is known at compile time, the problem size is computed as well. If the size of the with loop is not known at compile time, the problem size is computed at runtime. A time-thread relation is measured for each unique combination of identifier and problem size. When the combination of the identifier and the problem size are not unique, time-thread relations are merged. This is done to improve accuracy. If a with loop has more than once the same problem size, accuracy may be improved by summarizing the measured times, so double measurements are not wasted.

To get the time-thread measurements, a profiler is included in the binary file that is generated when the SaC compiler is used in training mode. Furthermore, a code construct is wrapped around each with loop in the program, in the way that is visualized in Figure 10. The entire code construct is repeated for a number of cycles, due to the do-while loop shown in Figure 10. Each cycle begins by setting the number of threads that must be used to execute the with loop (see blue highlighted code). Next, the execution time of the with loop is measured (see green highlighted code). Finally, the measurement results are stored by the profiler and the number of threads are increased (see orange highlighted code). This process continues until a user defined maximum number of threads is reached. The profiler has stored the time-thread relation by the time the maximum is reached.

```
int with_id = 12345; // the with_id is generated at compile time
int with_problem_size = get_problem_size(with_id);
int num_repetitions = 1;
int num_threads = 1;

bool another_cycle;
int time_begin;
int time_end;

do {
      set_num_threads_scheduler(num_threads);

      time_begin = get_realtime();
      for (i = 0 to num_repetitions) {

            a = with { ... } // some with loop

      }
      time_end = get_realtime();

      (another_cycle, num_repetitions, num_threads) = profiler(
                                                         with_id,
                                                         with_problem_size,
                                                         time_begin,
                                                         time_end);
}
while (another_cycle);
```

Figure 10: Example of code structure that is wrapped around each with loop by the compiler component of the smart decision tool when the training mode is active (blue: set number of threads for executing with loop, green: measure execution time of with loop, orange: store measurement results and increase number of threads).

The profiler is a state-machine that controls many of the loop constructs. For example, it does increase the number of threads, it stores measurements, and it can decide when to terminate the do-while loop. When the code of Figure 10 is observed closely, one would notice that the construct that is used for time measurements, contains a for loop. The for loop is used to improve the accuracy of the time measurements. The number of repetitions of this loop is determined dynamically by the profiler. During the first do-while loop cycle, the execution time of a single with loop iteration is measured. This first measurement is not stored. The profiler uses this time measurement to compute the maximum number of repetitions that fit within a user defined preset time. The number of repetitions is made dynamic, to ensure enough but not too much measurement time.

The time-thread profile is written to a database that is stored on disk, just before the do-while loop terminates. The database is a binary formatted matrix. Each row contains a profile of a with loop. The first three columns of each profile are the with loop identifier, the problem size, and the number of repetitions. The remaining columns contain the time measurements. The name of the database file is automatically generated, and consist of a user defined filename, a user defined computer architecture, and a number which is the maximum number of threads. The name of the database functions as an identifier. This has two reasons. First, one can choose to repeat the training of a SaC program. If the name, architecture, and the number of threads of the database file match those of

the new training, new measurements will be merged into the database file, similar to how profiles are merged. Second, the profiles are used when the SaC compiler is in decision mode. The SaC compiler is able to recognize database files, due to their specific naming.

4.2 Smart decision tool – decision mode

In Figure 1 there are examples of a typical speedup-thread relations observed when executing a parallel program on a multi-core machine. In Figure 1b two points on the curve are marked: highest speedup and a point that is more of a balance between energy consumption and speedup. These points can also be described using the curve gradient. For example, highest speedup can be observed when the curve gradient is zero. This concept is used to predict the number of threads of interest for each with-loop when the SaC compiler is in decision mode.

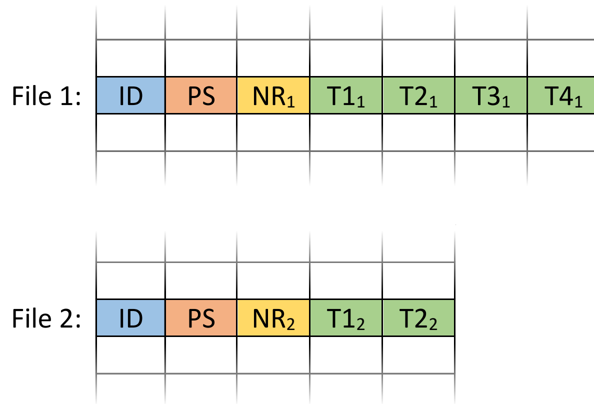
In Figure 1 there are examples of typical speedup-thread relations observed when the measurement circumstances are perfect. In reality one could expect to find outliers, caused by for example hardware limitations and load balancing. To diminish the effect of outliers on the predictions made during decision mode, a fourth order polynomial interpolation of the measurement results is used. The typical speed-thread relation (as shown in for example Figure 1b) is not a parabolic curve. A third order polynomial would not fit the typical curve. We use a fourth order polynomial to improve the curve fitting. With a fourth order polynomial, there can be more than one point that matches the user defined gradient. We use the point with the smallest number of threads as our prediction for the optimal[1] number of threads.

The decision mode is used to create an optimized binary for a specific architecture using a known maximum number of threads. However, nothing is preventing a user from creating profile databases for several architectures or for different maximum numbers of threads. This creates a problem as only one profile database can be used. The problem is reduced by letting the user define the filename and architecture for the profile database. However, in SaC the maximum number of threads is only known at runtime. Since the maximum number of threads is unknown during the compilation and optimization of a SaC program, predictions are made based on the database profile that was made using the largest maximum number of threads. If at runtime it appears that the predicted optimal[1] number of threads is above the maximum number of threads allowed, the maximum number of threads will be used for the computation.

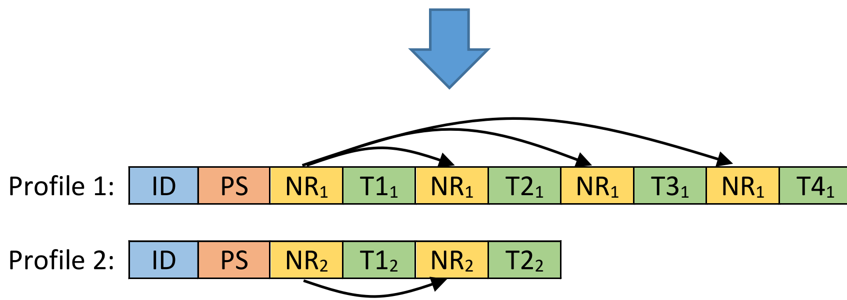
Using only the information that is stored inside the profile database that was made using the largest maximum number of threads, would be a waste of the effort and time that was spent in collecting the profiles for the other databases. Instead, we merge the databases to improve the accuracy of the profiles. Figure 11 is an example of the merge process using two database files. In the figure two profiles are merged. First a profile match is found (Figure 11a). A match means that the identifiers and problem sizes of the two profiles are identical. Next the value for the number of repetitions is being copied as shown in Figure 11b. The profile that contains the smallest number of measurements is being padded with zeros as shown in Figure 11c. Finally, the number of repetitions and the time measurements of the two profiles are being added as shown in Figure 11d. This process is repeated until all profiles of the two files are merged. The resulting file can be merged with another file, until all files are merged. As a final step, each time measurement is divided by its number of repetitions, to make it possible to compare the measurement results of each profile.

---

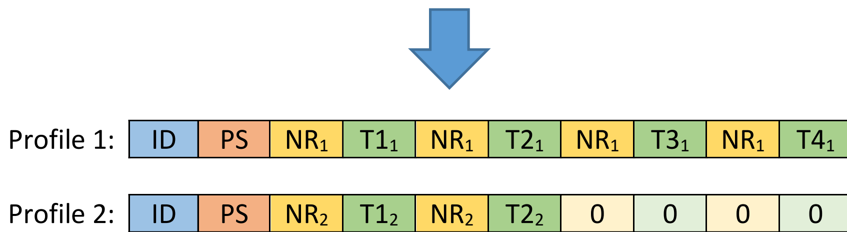
[1] In this context, optimum is looked upon from the perspective of the user. If the user considers that maximum speedup is what is needed, then this is looked upon as being optimum. However, if the user chooses a more energy-efficient approach, optimum is then a balance between speedup and energy consumption.



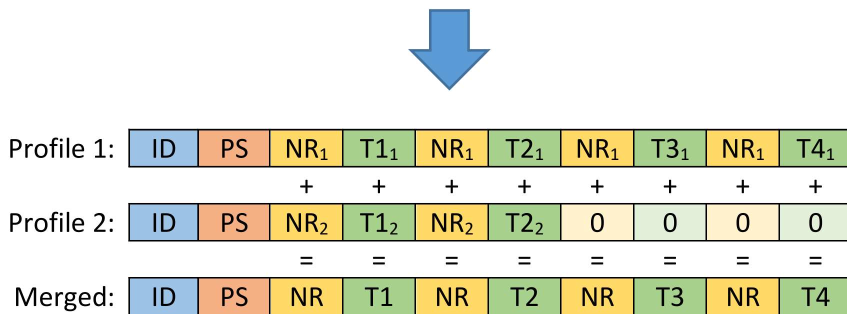
a) A profile match is found. The highlighted profiles of file 1 and file 2 have the same identifier and problem size.



b) For each profile, the value for the number of repetitions is copied and inserted just before each time measurement.



c) The profile that contains the smallest number of time measurements is padded with zeros until its size is identical to the size of the largest profile.



d) The profiles are merged by adding all values except for the identifier and problem size, which remain unchanged.

ID	Identifier
PS	Problem size
NR	Number of repetitions
T<n>	Time measurement with <n> threads

Figure 11: Example of how two profiles from different databases are being merged. The profiles have the same identifier (ID) and problem size (PS).

The merged profiles are analyzed to gain predictions for the number of threads for each with loop. These predictions are stored in recommendation tables, one table for each with loop. A recommendation table consist of two columns. The first column is a list of different problem sizes. The second column of the table is the predicted number of threads for the given problem size. The recommendation tables are compiled into the binary, before each with loop. At runtime, the problem size of each with loop are compared with the problem sizes in the recommendation tables. If there is a match, the recommended number of threads are used for the with loop computation. If the runtime problem size is in between two adjacent problem sizes in the recommendation table, linear interpolation is used to estimate the optimal[1] number of threads. The result is rounded to the nearest number of threads. No extrapolation is used. If the problem size is smaller than any value in the recommendation table, the number of threads given by the smallest problem size in the table is used. In case of a larger value the largest problem-size is used.

Extrapolation is subject to greater uncertainty then interpolation. As explained in chapter 1, there are many parameters that effect the speedup gained by executing a piece of code in parallel. Therefore, extrapolation results on the speedup curve can easily become meaningless. More importantly, the user can easily avoid the need for extrapolation. When the smart decision tool is trained using data that is of a similar problem size as the data that is used in the productive environment, no extrapolation is needed. Training the smart decision tool using data with a problem size that is very different from what is used in the productive environment makes little sense. Therefore, instead of performing complicated extrapolation computations, we assume that the user provides meaningful training data.

4.3 Implementation of barriers

Figure 12a is a schematic representation of SaC's multithreaded execution model when the smart decision tool is not being used. Figure 12b shows the execution model of SaC when the smart decision tool is used. The main difference between the two models is that the smart decision tool can decide to use less than the maximum available number of threads for parallel sections. The threads that are not actively used are parked by the start barrier which is implemented as a spinlock barrier. A spinlock barrier is an active barrier. As a result, threads consume power even while they are parked.

To limit the power consumption of the threads, three new start barriers are designed. These new barriers can deactivate/hibernate threads. The barriers are implemented as part of the runtime library to test their impact on the performance of SaC. One can select a barrier type using the –mt_barrier_type compiler flag in SaC.

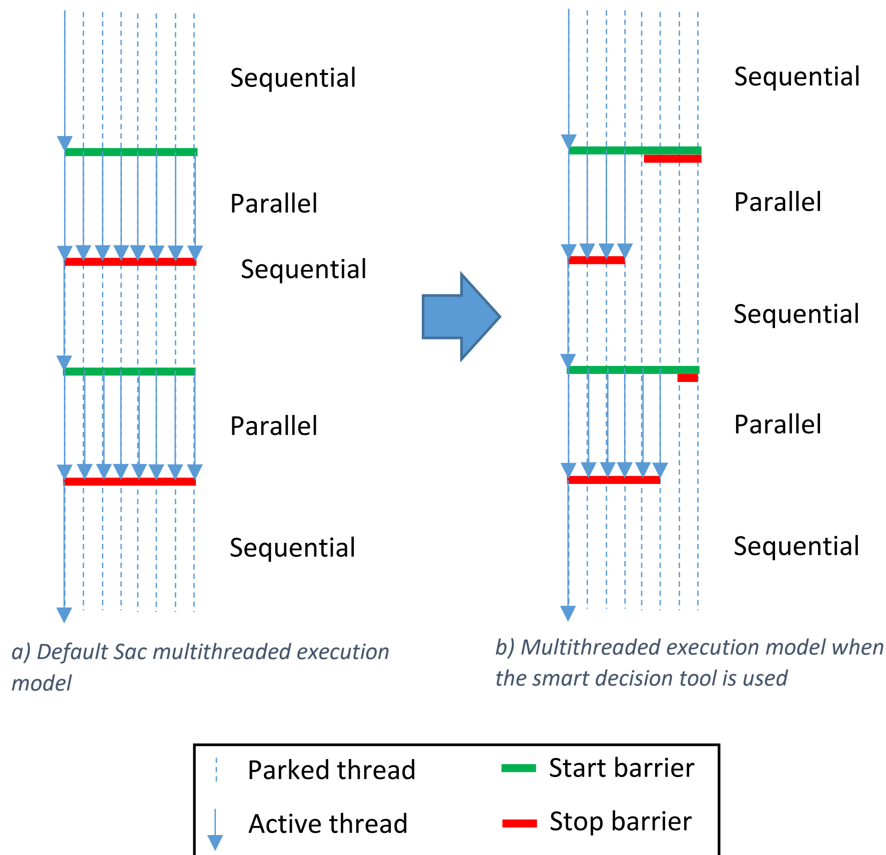


Figure 12: Schematic representation of SaC's multithreaded execution model.

```
n = 0

function barrier_wait() {
      n = n + 1
      if (n == N_t) {
            awake()
            n = 0
      }
      else {
            sleep()
      }
}
```

Figure 13: The figure above shows the principle of the barrier_wait function. In order to improve the readability of the function, all protection against, for example, race conditions and deadlocks are omitted. In reality multiple threads can call the function at the same time without the occurrence of complications.

Of the three new start barriers, the so-called POSIX-thread barrier has the simplest implementation. The barrier is founded completely on functionalities that are described in the POSIX standard (hence the name POSIX-thread barrier). Specifically, use is made of the barrier_wait function of the POSIX standard. The function is called at the moment that threads pass the start barrier. That is at the beginning of a parallel executable part of the code in the case of the master thread. When the

master thread calls the function barrier_wait, all worker threads are activated. The worker threads reach the start barrier at the end of a parallel executable part of the code. They call the barrier_wait function in order to deactivate themselves temporarily, until the master thread reactivates them. The call for the master thread and the worker threads is precisely the same. The barrier_wait function keeps count of the number of threads that have called the function. This is compared with a preset number (for the sake of clarity we shall call this number N_t). On the basis of this comparison the barrier_wait function makes a decision as to whether one thread should be deactivated or all the threads activated. The barrier_wait function will deactivate threads as long as the function has been called fewer times than is indicated by the preset number N_t. As soon as the number of calls equals the preset number N_t the barrier_wait reactivates all threads and the whole procedure starts again from the beginning. [14]

The principle of the barrier_wait function as described in the paragraph above is translated in Figure 13 into pseudocode. For the sake of the readability of the code, all protection against race conditions and deadlocks, for example, have been omitted from the figure. In the figure, the number of threads that have called the function is registered in the variable n. The variable is compared with N_t in order to determine whether the threads should be activated or indeed deactivated. The function awake ensures that all threads are activated. The current thread is deactivated by the function sleep.

The POSIX-thread barrier sets N_t in such a way that it is equal to the total number of threads (the master thread plus the number of worker threads). Note that with this setting all threads can never be deactivated. There is always at least one active thread. This is the master thread. If the last active thread (the master thread) uses barrier_wait, the number of times that the function has been called is equal to the total number of threads, as a result of which all threads are activated. Because of the stop barrier the master thread cannot carry on at the end of a parallel piece of code until all worker threads have reached the start barrier. As a result, the master thread is always the last active thread. At a following parallel part the master thread can call barrier_wait in order to reactivate all worker threads without any problems.

One disadvantage of the POSIX-thread barrier is that the function barrier_wait, on which it is dependent, is part of the IEEE Std 1003.1j-2000 version of the POSIX standard. Not all operating systems support this version of the POSIX standard. [15] For example, Apple does not support this version of the POSIX standard, as a result of which barrier_wait is not available on OSX. However, an advantage is that when the function does become supported, the technology will be very reliable. The POSIX-thread barrier consists of nothing other than the call for the barrier_wait function. This function comes from the POSIX standard. We therefore assume that the functionality has been applied and tested on the most systems in countless programs.

```
function deactivate_thread(cv, mutex) {
      lock(mutex)
                                    ⎧ unlock(mutex)
      cond_wait(cv, mutex)          ⎨ sleep(cv)
                                    ⎩ lock(mutex)
      unlock(mutex)
}


function wake_all_threads(cv, mutex) {
      lock(mutex)
      cond_broadcast(cv)
      unlock(mutex)
}
```

Figure 14: Shown in the figure above is the principle of the POSIX-conditional barrier. The function deactivate_thread is used by the worker threads to deactivate themselves. The deactivation of the threads is carried out by the cond_wait function that is called by deactivate_thread. The principle of cond_wait is shown to the right of the function. The master thread uses the function wake_all_threads in order to activate all the threads at the beginning of a parallel executable piece of code. The function cond_broadcast activates all threads. With the functions lock and unlock a mutex can be acquired or released, respectively.

An alternative to the POSIX-thread barrier is the POSIX-conditional barrier. The barrier makes use of two functions of the POSIX standard: cond_wait and cond_broadcast. Using cond_wait, threads can be deactivated if a certain condition is met. The user himself can determine what the condition is. The deactivated threads become active again if cond_broadcast is called. The master thread can use cond_broadcast in order to activate all worker threads at the beginning of a parallel executable piece of code. The worker threads can deactivate themselves at the end of a parallel executable piece of code by using cond_wait. To do this it is only necessary to create a condition that is always met.

One can ask oneself why the POSIX-conditional barrier does not suffer from race conditions. It is of course possible that the master thread activates all threads while the process for the deactivation of threads is still operating on a number of threads. To prevent the occurrence of such a situation, cond_wait and cond_broadcast are protected by POSIX-mutex locks. The mutex can never be acquired by more than one thread at a time. As a result, multiple threads can never make use of the function cond_wait at the same time. The function cond_wait can never be called at the same time as cond_broadcast. Only when cond_wait or cond_broadcast have been processed completely is the mutex released again and another thread can use the functions.

The principle of the POSIX-conditional barrier is translated into pseudocode in Figure 14. The function deactivate_thread is used by worker threads to deactivate themselves. If the function is called, the mutex must first be acquired. As a result, only one thread at a time can execute the function. A condition determines whether this thread calls the function cond_wait. In the figure a condition has been formulated that is always true, by which each thread can call the function cond_wait. This function ensures that the mutex is released again so that other threads can also call cond_wait. Immediately after this the threads are deactivated. The principle of the cond_wait is shown in the figure to the right of the function.

The master thread is the only thread to use the wake_all function. This function calls cond_broadcast, reactivating all the threads. The cond_broadcast function is protected by a mutex, which is only released when all the threads have been reactivated. As a result, it is impossible for a

worker thread to use the function deactivate_thread before all the other threads have been activated. This prevents race conditions and deadlock situations.

Lastly there is the POSIX-mutex barrier. As the name suggests this is a barrier based on POSIX-mutexes. When a mutex is acquired by a thread this automatically implements a barrier for other threads because they cannot acquire the mutex at that moment. The threads that must wait are temporarily deactivated by the POSIX-mutex system. The barrier continues to exist as long as the thread that has the mutex does not release it. If the thread does release the mutex another thread can take over the mutex. If all threads release the mutex, the barrier is lifted. The only thing needed to implement a barrier with mutexes is a logic that ensures that the mutex is retained and released at the right moments. As long as a program is occupied in executing a sequential computation the mutex must be retained. As soon as a sequential computation changes into a parallel computation, all the threads must release the mutex, if they have it.

For the implementation of the POSIX-mutex barrier, we made two assumptions on mutexes, although they are not allowed according to the latest POSIX-standard. [14] First, we assume that a thread may ask for a mutex twice, were it will only get the mutex at the first call, and as a consequence it will lock the second time it tries to obtain the mutex. Second, we assume that the mutex can be released by another thread than the thread that took possession of the mutex. Although, these assumptions do not match the POSIX-standard, they seem to hold on all systems we have used for testing (see chapter 5.1 for a detailed description of the machines we used for testing). Therefore, we have decided to keep the POSIX-mutex barrier as one of our experimental barriers.

The POSIX-mutex barrier first checks the total number of threads. In the most trivial case, in which there is in total only 1 thread, the POSIX-mutex barrier does nothing. The mutex barrier mechanism is only applied if there are 2 or more threads. The logic of the POSIX-mutex barrier consists of an addition system, a subtraction system and three possible actions, one of which is selected. The addition system is comparable with that of the barrier_wait function. The system is used to register how many threads have used the barrier. Every time a thread uses the barrier a counter (“n”) is raised by one.

On the basis of the status of the counter (“n”), one of the three possible actions is selected. If the counter (“n”) is at 1, the first action is then executed. In this action an attempt is made to acquire the mutex twice. Because of the assumptions we have made, this is successful only once. As a result of this the thread itself throws up a barrier that it cannot pass, unless another thread releases the mutex. After the two attemps to acquire the mutex there follows a step by which the mutex is released once more. The thread cannot reach this step directly, but when it does, it releases a chain reaction in which all other threads are released.

If the counter (“n”) indicates that 2 or more threads have used the barrier, the second action is then executed. The thread concerned then attempts to acquire the mutex that has already been given to the first thread. As a result, threads that execute the second action also implement a barrier themselves. After the barrier, there again follows a step in which the mutex is released. Each thread that executes the second action becomes part of the chain that the very first thread started. They are locked by the threads that came before, but if only one of these threads is released, all other threads will follow.

If the counter (“n”) equals the total number of threads, the last action is then executed. This action consists of the lowering of the counter (“n”) and the releasing of the mutex. The chain reaction is now released by which all other threads are freed from their barrier. From this point the counter is deployed to determine how many threads have been freed from their barrier. This works as follows:

when threads are freed they execute the steps that follow after the barrier. For all the remaining threads these steps consist of lowering the counter (“n”) and releasing the mutex so that a following thread can pass its barrier.

The thread that is the last to pass its barrier executes one extra step. The counter (“n”) is used to determine which thread is the last to pass its barrier. As an extra step, this last thread releases a second mutex. This is a different mutex from the one that has been used until now. In order to avoid further confusion, we shall refer to this second mutex from now on as mutex2. The mutex that has been used until now we shall call mutex1.

Mutex2 is used to protect the logic of the POSIX-mutex barrier. The mutex ensures that only one thread at a time can execute the logic. The logic only functions well if only one thread at a time is processed. If all the threads were to execute the logic simultaneously, the counter (among other things) would be disrupted. Two threads could attempt to raise the counter at the same time. Or a thread is released that attemps to lower the counter, whilst at the same time another thread must raise the counter. A disrupted adder can lead to all manner of consequences. For example, that it can no longer be determined what action a thread must execute. As a result of all problems that could then arise, threads end up in a deadlock, for example.

In Figure 15 the POSIX-mutex barrier is written as pseudocode. The barrier makes use of two mutexes. In the figure, it can be seen that a mutex must first be acquired in order to be able to execute the logic of the barrier. As has been described earlier, the logic of the POSIX-mutex barrier consists of a counter system followed by three possible actions, one of which is selected. The first action is executed by the first thread that calls the mutex_barrier. The third and final action is carried out by the last thread that calls the mutex_barrier. All the other threads execute the second action. In the figure it can be seen that as the first action is carried out a barrier is implemented by calling lock(mutex1) twice. The threads that execute the second action are blocked by the barrier by calling lock(mutex1). In the last action, the barrier is lifted by calling unlock(mutex1).

```
// n is used as a counter, to keep track of the number of threads
// that have used/called the barrier.
n = 0

function mutex_barrier() {
      // If Nt ( the total number of threads) equals 1, the barrier
      // returns, because there is no need for a barrier.
      if (Nt == 1) return
      // This mutex lock makes sure that only one thread at the time
      // can execute the logic of the POSIX-mutex barrier
      lock(mutex2)
      // A thread is using the barrier, so the counter is increased.
      n = n + 1
      // If the counter is equal to 1, execute the first action. The
      // thread will call mutex1 twice. The thread will be locked.
      if (n == 1) {
            lock(mutex1)
            unlock(mutex2)
            lock(mutex1)

            n = n - 1
            if (n == 0) {
                  unlock(mutex2)
            }
            unlock(mutex1)
      }
      // Second action; try to get mutex1 which is already been taken
      // by the first thread. The thread will be locked.
      else if (n > 1 && n < Nt) {
            unlock(mutex2)
            lock(mutex1)

            n = n - 1
            if (n == 0) {
                  unlock(mutex2)
            }
            unlock(mutex1)
      }
      // Third action; unlock mutex1. This triggers a chain reaction
      // in which all threads are freed from their lock.
      else {
            n = n - 1
            unlock(mutex1)
      }
}
```

Figure 15: Shown in the figure above is the principle of the POSIX-mutex barrier. The barrier makes use of two mutexes. With lock(mutex1) and unlock(mutex1) the first mutex is acquired or released respectively. Lock(mutex2) and unlock(mutex2) are used for the second mutex.

5 Evaluation

5.1 Evaluating speedup model

In order to find out what the curves of Figure 1 look like in practice, and to understand what influence this has on the predictions made by the smart decision tool, the program shown in Figure 16 has been developed. There are existing benchmark suites for measuring parallel performance, however they are not designed to work with SaC. With the program shown in Figure 16 two matrices with the same dimensions are added repeatedly, inside a loop. When the time is measured before and after the program, the execution time of the program of Figure 16 can be determined. By repeating the addition of the matrices in a loop a large number of times, the time needed to add up the matrices is greatly enlarged. The time that is taken in overhead costs (such as the function call and the declaration of memory for the matrices) becomes immeasurably small as a result. The loop also ensures that, when the execution time of the program is measured, the duration of the program comes out (far) above the margin of error of the time-measuring software and hardware. By measuring the execution time of the program by different numbers of threads, the relation between the number of threads and the speedup can be established. This experiment can be repeated for matrices of different problem sizes. The influence of the problem size on the speedup can be studied by comparing these experiments with one another.

```
int[.,.] add_matrices(int[.,.] A, int[.,.] B, int iterations) {
      for (i = 0; i < iterations; i++) {
            A += B;
      }

      return A;
}
```

Figure 16: Using the program above, two matrices with the same dimensions can be added repeatedly, inside a loop. By determining the execution time of this program by different numbers of threads, the relation between the number of threads and the speedup can be established. By carrying out experiments with matrices of different sizes (problem sizes) and comparing these experiments with one another, the influence of the problem size on the speedup can be studied.

In Figure 17 the relation is established between the number of threads and the speedup (solid blue spheres) with a problem size of $n = 100$ (10×10), a problem size of $n = 2{,}500$ (50×50) and a problem size of $n = 160{,}000$ (400×400) by carrying out time measuring using the program shown in Figure 16. The measurements were carried out on an Intel Xeon machine with 2 processors, both equipped with 4 cores. The measurements were repeated on an AMD Magny Cours machine with 4 processors, where each processor has 12 cores.

The Intel machine is a NUMA system with 2 processors. Each processor has 4 physical cores with support for hyper-threading, as the result of which the machine has at its disposal in total 16 virtual cores (2 threads per core). The 2 processors each have per physical core 64 KB L1 cache (of which 32 KB is instruction cache and 32 KB data cache) and 256 KB L2 cache. The 12 MB L3 cache, which each processor has at its disposal, is shared by the 4 physical cores. The Intel machine has 24 GB of RAM memory. [16] [17]

The AMD Magny Cours machine is also a NUMA system, but with 4 processors. Each processor consists of 2 nodes, each with 6 cores. In total the AMD Magny Cours machine has 48 physical cores. Each core has 128 KB L1 cache (of which 64 KB is instruction cache and 64 KB data cache) and 512 KB

L2 cache. Each node shares 6 MB L3 cache. Each processor therefor has in total 12 MB L3 cache at its disposal. However, of the 12 MB cache 2 MB is used as directory cache. The directory cache is a data structure that keeps track of which cachelines the other nodes have. The AMD Magny Cours machine has 128 GB of RAM memory. [17]

The program in Figure 16 describes an addition operation of two matrices that have the same size. This has been chosen because the addition of two matrices can be parallelized very well. If x and y are the dimensions of the matrices, the computation of adding those matrices can be divided in $n = x \cdot y$ sub-problems, each of which can be solved separately (parallel). Furthermore, the program is simple to implement and the addition of matrices has practical applications. The program in Figure 16 perhaps does not give the greatest possible speedup, but because of its practical applicability time measurements with this program are not only representative in experimental context, but also for numerous parallel algorithms that do not per se have to be of an experimental nature.

The measurements are intended to study how well the smart decision technique is capable of determining the maximum speedup in practice. That is why there is, in addition to the solid blue spheres that represent the relation between threads and the speedup, a vertical dotted red line drawn in each graph of Figure 17. This line indicates the number of threads for maximum speedup calculated by the smart decision tool. This number of threads is determined by having the smart decision tool determine the number of threads for maximum speedup 7 times after which the median value is calculated. In the figures, there are also open orange spheres. These give the relation between the number of threads and the speedup, after optimizing the program of Figure 16 using the smart decision tool.

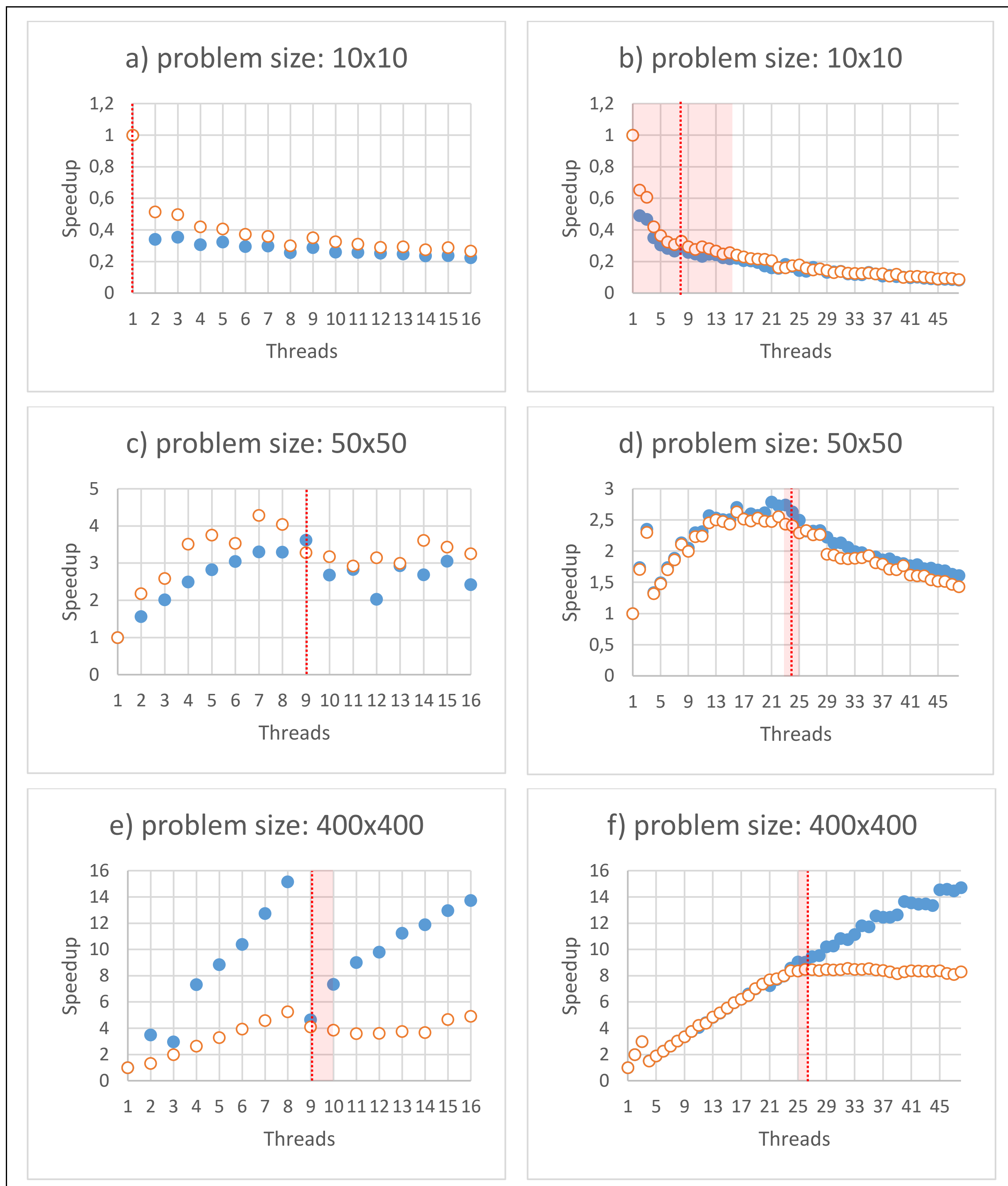


Figure 17: In the figures above is shown the relation between the number of threads and the speedup with different problem sizes. The relations were determined by carrying out time measurements using the program in Figure 16. The time measurements were been carried out on an Intel 2x 4 core hyper-threading Xeon processor (left) and repeated on an AMD 4x 12 core Magny Cours machine (right). In the graphs the solid blue spheres are the speedups obtained without the use of the smart decision tool. The open orange spheres are the speedup with the use of the smart decision tool. The dotted red line indicates he median of the number of threads for maximum speedup calculated by the smart decision tool. The transparent zone indicates he area between the first and third quartile of the calculated maximum.

The relations measured between the number of threads and the speedup follow the course of the model curves in Figure 1. The graphs also show clearly the effects that are the consequences of the design choices of the machines used. These effects result in some variation being shown in the graphs in comparison with the model curves. The 2x 4 core Intel Xeon processor has a reduction in performance if more than 8 threads are used. This is especially easy to see in Figure 17e, but it can

also be observed in Figure 17c. The reduction in performance is caused because the Intel processor only has 8 physical cores (2 processors, each with 4 cores), as a result of which with more than 8 threads, use is made of the hyper-threading technology. [18] By means of the hyper-threading technology the number of cores are virtually doubled, but the resources remain the same. If a thread has to wait for some time, for example because there is a cache mismatch, hyper-threading can provide more speedup by quickly activating another scheduled thread. This new thread must itself then not have to wait, otherwise the switch between the threads was a pure waste of time. Because the program in Figure 16 is an example of a memory-bound problem, this is precisely what happens on the Xeon processor in executing this program. SaC divides the amount of work equally among all the threads. But because in hyper-threading there can be fewer cores than threads, some cores get more work than others. For example, with 9 threads one core must carry out twice as much work than the other cores. The entire computation is then slowed down by one core that must carry out twice as much work. Notable is that in Figure 17e there is clear evidence of a superlinear rise in the first part of the graph. What this superlinear rise causes is a matter for further research.

The Magny Cours machine also gives deviations of the model curve that are the result of the design choices. In Figure 17d and Figure 17f it can be seen clearly that the increase in the speedup up to 3 threads is greater than the increase with the use of more than 3 threads. The great increase is visible in many measurement results, but in none of the experiments carried out in this study does the great increase carry through to more than 4 threads. The Magny Cours machine makes use of 4 separate processors. Whether this can explain the great increase in the first part of the measurement results is left to future research.

The changes in relation to the model curves, which are the result of the design choices of the machines, influence the predictions of the smart decision tool. The consequences of the machine specific design differences are particularly visible in Figure 17e. That the smart decision tool selects here 9 threads and not 8 threads is, among other things, due to interpolation. The polynomial estimate that was intended to reduce the influence of local differences appears in many instances to have an unintended side effect, such as shown in Figure 17e. Because of the polynomial estimate, the influence of sudden changes in the curve, which are more structural in nature, is also reduced. As a result, the maximum for the smart decision tool comes to lie at another point.

The consequences of the polynomial estimate on the predictions of the smart decision tool can be seen in all the graphs shown in Figure 17. Nevertheless, the influence of the machine-specific design differences on the polynomial estimates is limited. The manner in which the smart decision tool represents time measurements internally has a much greater influence on the form of the polynomial estimates and the position of the maximum.

Internally the smart decision tool uses a time-thread graph instead of a speedup-thread graph. Through this difference it is possible to explain why in the experiments of Figure 17e and f the smart decision tool did not select the number of threads for maximum speedup. As an example for this explanation we take the graph as shown in Figure 17f, but the same reasoning is also valid for the experiment in Figure 17e. Within the interval studied in Figure 17f can be seen an almost monotonically rising graph. There is no maximum in such a graph, so one could be inclined to expect that the tool indicates the most extreme point on the graph as maximum. However, as can be seen the tool indicates 26 threads. Shown in Figure 18 is the time-thread graph that belongs to Figure 17f. In this graph can be seen that the time in the interval shown initially begins to fall quickly and soon flattens out to an almost flat line. This occurs because the time-thread graph is inversely proportional to the speedup-thread graph. If in the interval studied the speedup rises almost linearly, then the

time-thread relation is a descending hyperbola with the x-axis and y-axis as asymptote. With such a hyperbola, the graph in the interval studied is almost flat and parallel to the x-axis.

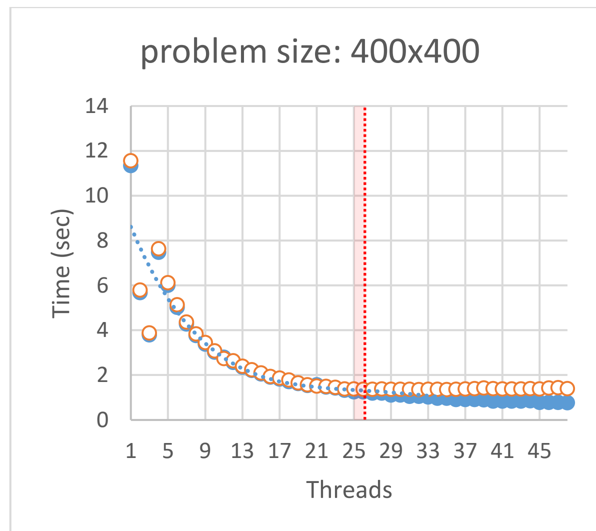


Figure 18: In the figure above is shown the relation between the number of threads and the execution time with a problem size of 400x400. The relations are established by carrying out time measurements on the AMD Magny Cours machine with the program form Figure 16. The figure is almost similar to Figure 17f, but with the difference that execution time is represented on the y-axis instead of the speedup. As a result, the figure above corresponds more with the internal data representation of the smart decision tool than Figure 17f. In the graph the solid blue spheres show the execution time obtained without the use of the smart decision tool. Open orange spheres show the execution time when the smart decision tool is used. The dotted red line indicates the median of the number of threads for maximum speedup calculated by the smart decision tool. The red transparent zone shows the area between the first and third quartile of the calculated maximum.

It is difficult to determine a maximum on an (almost) flat curve. The smallest fluctuations in the measurements can produce a local maximum. In this, the polynomial estimate is of little added value, and can even produce unexpected maxima. The smart decision tool finds a maximum at 26 threads. Whether this is the best choice is a point for discussion. In any event, 26 threads do not give the greatest speedup. On the other hand, the addition of more threads has virtually no influence of the time needed to solve the computation (see Figure 18).

5.2 Scheduler

The scheduler is a part of SaC that ensures that the work is divided among the threads. [9] It has been adapted to enable setting the number of threads dynamically. The adaptations to the scheduler have been kept as simple as possible in order to keep changes on existing, correctly working codes as limited as possible. That is why only one adaptation has been applied. The scheduler has a global variable that stores the total number of threads. The scheduler uses this parameter to determine for how many threads work must be divided. Originally, this variable was a runtime constant, however it is changed such the value of the variable is dynamically set by the smart decision tool. By adapting this parameter dynamically, the scheduler divides the work among fewer threads than there are in reality. As a result, the scheduler gives work to only some of the threads, which means that the rest of the threads are not used.

This adaptation has however unintended implications. Although the scheduler does not give work to some of the threads, they are still physically present. When the master thread lifts the start barrier all the threads become active, including the threads that have not been given work. As a result, the threads that have no work immediately reach the stop barrier and are switched off again. As a

consequence, these threads make no contribution in solving the computation, but overhead cost are paid for this.

In all the graphs of Figure 17 there are vertical lines that indicate the number of threads that will be used by the smart decision tool in the circumstance of the graph concerned. In many of the graphs shown, the smart decision tool has clearly not selected the number of threads for maximum speedup. As has been said, in Figure 17e and f this choice was particularly influenced by the manner in which the smart decision tool represents its data internally. The smart decision tool uses the same time-thread representation in the experiments of Figure 17a to d inclusive. There however, the predictions are particularly influenced by the scheduler not assigning work to some of the threads.

The scheduler does not assign work to all the threads, but the threads to which no work is assigned do create overhead. The total overhead costs are therefore always dependent on the total number of threads present, irrespectively of whether they are used or not. As a result, in training mode the smart decision tool always measures an almost constant overhead. The tool measures the increase in the speedup, but does not measure any reduction that would normally occur if more threads were used. As a result, the graph therefore goes flat in the area where it would normally descend. As mentioned earlier, the tool can easily find a relative maximum in such a flat area, as the result of which the number of threads can come out on the high side.

In Figure 19 the connection is established between the number of threads and the speedup (solid blue spheres) at a problem size of $n = 2{,}500\ (50\times50)$ by having the smart decision tool (in training mode) carry out time measurements using the program shown in Figure 16. The measurement is carried out on the AMD Magny Cours circuit. Figure 19 is thus comparable with Figure 17d, with the difference that the time measurements are carried out by the smart decision tool. Internally the smart decision tool represents the data as a time-thread graph, but in order to be able to compare Figure 19 better with Figure 17d the data in the figure is converted to a speedup-thread relation.

Just as in Figure 17d the graph in Figure 19 is ascending to approximately 21 threads. But in contrast to Figure 17d, Figure 19 has no maximum, because the graph remains flat at more than 21 threads. As has been said, this is caused because the smart decision tool measures an overhead that is virtually independent of the number of threads that is used for a computation.

The scheduler technique is not only used in the training mode, but it is used in the decision mode of the smart decision tool as well. When executing a program in decision mode the scheduler technique is used to ensure that, for a parallel computation, no more threads can ever be used than are selected by the smart decision tool in training mode. In this way, it is endeavoured not to allow the speedup to slow down if the total number of threads is larger than the number of threads for maximum speedup. But as explained, fixing the number of threads using the scheduler technique does not prevent overhead to rise when the total number of threads is increased. As has been said, the scheduler technique only ensures that some of the threads cannot take part in a computation, but the unused threads are switched on and off with unrelenting frequency, as the result of which the overhead increases as the maximum number of threads is increased. Because overhead is paid non-the less, in this situation, performance wise it makes little difference as whether to use the number of threads for maximum speedup or all the threads together.

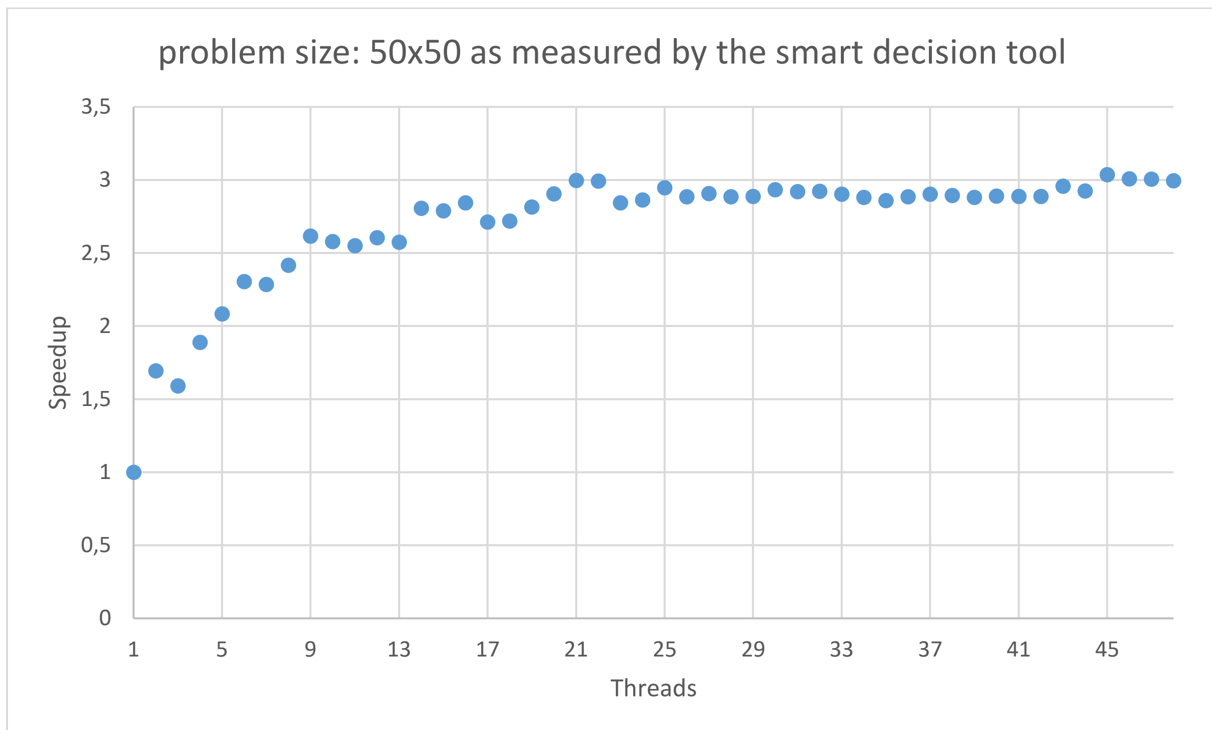


Figure 19: In the figure above time measurements made by the smart decision tool are represented as a speedup-thread relation. The data is obtained by having SaC execute the program in Figure 16 in training mode with matrices of 50x50. In training mode, a performance profile of the program is made from which the data for the figure above is obtained. The measurement is carried out on the AMD Magny Cours machine.

In Figure 17 the open orange spheres indicate the speedup that is achieved after use is made of the smart decision tool. As can be seen, after use is made of the smart decision tool the speedup is in all the graphs shown equal to or even considerably lower than the speedup obtained without the use of the smart decision tool. The increase in overhead is therefore not prevented by use of the scheduler technique. If the smart decision tool predicts the number of threads for maximum speedup well, after optimization there is little change in the course of the graph. But if the smart decision tool selects a different number of threads, the tool achieves even less speedup than when the smart decision tool is not used. This can be seen clearly in Figure 17e and Figure 17f. Because in these the smart decision tool has selected a number of threads that does not result in a maximum speedup, computations take longer than necessary.

5.3 Evaluation of barriers

As discussed in chapter 4.3, SaC makes use of barriers in order to enable switching rapidly between sequential and parallel executable code. The existing barrier technique in SaC makes use of spinlock to park threads when they are not needed in carrying out computations. With the spinlock technique, threads remain active even if they are not used. As a result, threads can quickly be deployed for computations, but the threads also constantly consume energy. Three new barrier techniques have been developed for SaC with the object of finding a better balance between speed and energy consumption. The new barrier techniques pause threads at the moment that they are not in use, resulting in possible energy saving. In total there have been three new barriers developed: the POSIX-thread barrier, the POSIX-conditional barrier and the POSIX-mutex barrier. With different experiments and a model a study is made here of what the influence of the different barriers is on speed and energy consumption.

For the four different barriers in Figure 21 the relation has been established between the number of threads and the time, by carrying out time measurements using the program that is shown in Figure 20. The program in Figure 20 carries out a simple parallel computation (the adding of 3x4 matrices). Without modification to the source code, in SaC, it is not possible to perform a parallel matrix addition on smaller matrices. Because of the simplicity of the computation, in the execution of the program particularly large amounts of overhead costs are made. The time that SaC is occupied in solving the computation is hardly measurable. Because of the for-loop in the program, the overhead that is created by the constant starting and parking of threads is so great that all other overhead costs are immeasurably small. The loop also has the effect that when the execution time of the program is measured, the duration of the program comes out (far) above the fault margin of the time measuring software. The execution time of the program in Figure 20 is thus an indicator of the overhead that is created by the staring and parking of the threads. This overhead consists of more than just the barrier overhead, and as a result the execution time of the program is not an absolute indicator of the barrier overhead. However, the program in Figure 20 can be used for relative time measurements. It is therefore possible to carry out time measurements in which only the type of barrier is changed, after which an assessment can be made as to which type of barrier produces the least amount of overhead.

It is important to note that the program in Figure 20 performs a stress test on the barriers, which is a quite artificial scenario. With this kind of program the barrier overhead might look (much) better or worse than it is in reality. For example, the spinlock barrier, is really made to be called really often and in short periods of time. Therefore, the spinlock barrier might perform better with the stress test, then it would do in any realistic scenario, where the barrier is called at a much lower frequency. The new barriers on the other hand, are made to be rarely called. With the stress test, they may look worse than they are in reality.

```
int main() {
      a = with {} : genarray([3, 4], 2);

      for (i = 0; i < 100000; i++) {
            a += a;
      }

      return a[0,0];
}
```

Figure 20: With the program above, the barrier overhead for the four different barriers that are implemented is SaC can be determined. The program carries out a simple computation (the adding up of 3x4 matrices). Because of the simplicity of the computation there is hardly any work for the threads, as a result mainly overhead costs are created. This barrier overhead is further increased by the for-loop. As a result, the fraction of the program that consists of all the other costs is so small that it has virtually no influence on the measurements.

In Figure 21a and Figure 21b the bars show the execution time of the program in Figure 20 on the y-axis for the number of threads shown on the x-axis. The colour of the bar indicates what type of barrier technique is used. Study has been made of four types of barriers; for this reason, there are four bar colours. The four barrier techniques are: the spinlock barrier, the POSIX-thread barrier, the POSIX-conditional barrier, and the POSIX-mutex barrier. The experiment was carried out on the Intel Xeon processor (Figure 21a) and repeated on the AMD Magny Cours processor (Figure 21b).

As more threads are added, the amount of work with regard to, for example, the division of tasks among the threads or the staring and pausing of threads increases. In other words, the more threads, the more overhead. This can be seen clearly in Figure 21 because the bars that represent time become longer as the number of threads increases. But Figure 21 also shows that when the spinlock barrier is used the overhead does increase to a much smaller extent than when one of the other barriers is used. The blue bars that represent the spinlock barrier are in both Figure 21a and b much shorter than the other bars. The fastest alternative to the spinlock barrier is the POSIX-conditional barrier. The difference between the two barriers is however so great that, on the basis of Figure 21 it may be concluded that all new barriers, including the POSIX-conditional barrier, are slower than the spinlock barrier. Even if we consider that the program used to produce the results in Figure 21 is artificial, and way more beneficial for spinlock barriers than for any of the new barriers.

Another way of seeing this is on the basis of the results that are show in Figure 23. Figure 23 is a partial repetition of the experiment in Figure 17. However, in the experiments in Figure 23 the standard spinlock barrier is replaced with the POSIX-conditional barrier. For the measurements, deliberate use was made of the same problem sizes as those used in Figure 17, however the experiment with a problem size of $n = 100\ (10\times10)$ was not repeated. By using the same sizes, the effect of the POSIX-conditional barrier on the speedup is clear.

When, for example, Figure 23b is compared with Figure 17d, it immediately becomes clear that the forms of the graphs are totally different. Where in Figure 17d a speedup of almost 3x is reached, Figure 23b shows that, with the same problem size and with the POSIX-conditional barrier, the addition of more threads only leads to more overhead and thus less speedup. The graph in Figure 23b actually shows many more similarities with Figure 17b, but for that experiment a much smaller problem size was used. It is possible to obtain a graph similar to that shown in Figure 17d using the POSIX-conditional barrier, but a much larger problem size is needed for this. Figure 23d is an example of a graph with a similar speedup as shown in Figure 17d, but with the problem size of $n = 160{,}000\ (400\times400)$ a much greater speedup can be obtained with the spinlock barrier (see Figure 17f). In other words, under equal circumstances the spinlock barrier provides a greater speedup than can be realized with the POSIX-conditional barrier. The spinlock barrier is thus faster.

However, the new barriers were not intended to be a faster alternative to the spinlock barrier, but to find a better balance between energy consumption and speed. Figure 22 is a schematic representation of the execution model of SaC, with two different barriers. Figure 22a is the execution model with the spinlock barrier. Figure 22b represents the execution model when one of the new barriers is used. The new barriers are relatively slow and use a relatively large amount of energy. In Figure 22b this is represented by making the start and stop barrier wider than those in Figure 22a. There is no compensation for the time that is taken when the barriers are executed, because the execution of the rest of the program takes just as long as when the spinlock barrier was used. The energy use of the new barriers can however be compensated. Some of the threads can be deactivated by the new barriers, as a result of which energy is saved. If there are sufficient long periods in which fewer threads are active, it is possible that the program as a whole is more energy-efficient than when the spinlock barrier was used.

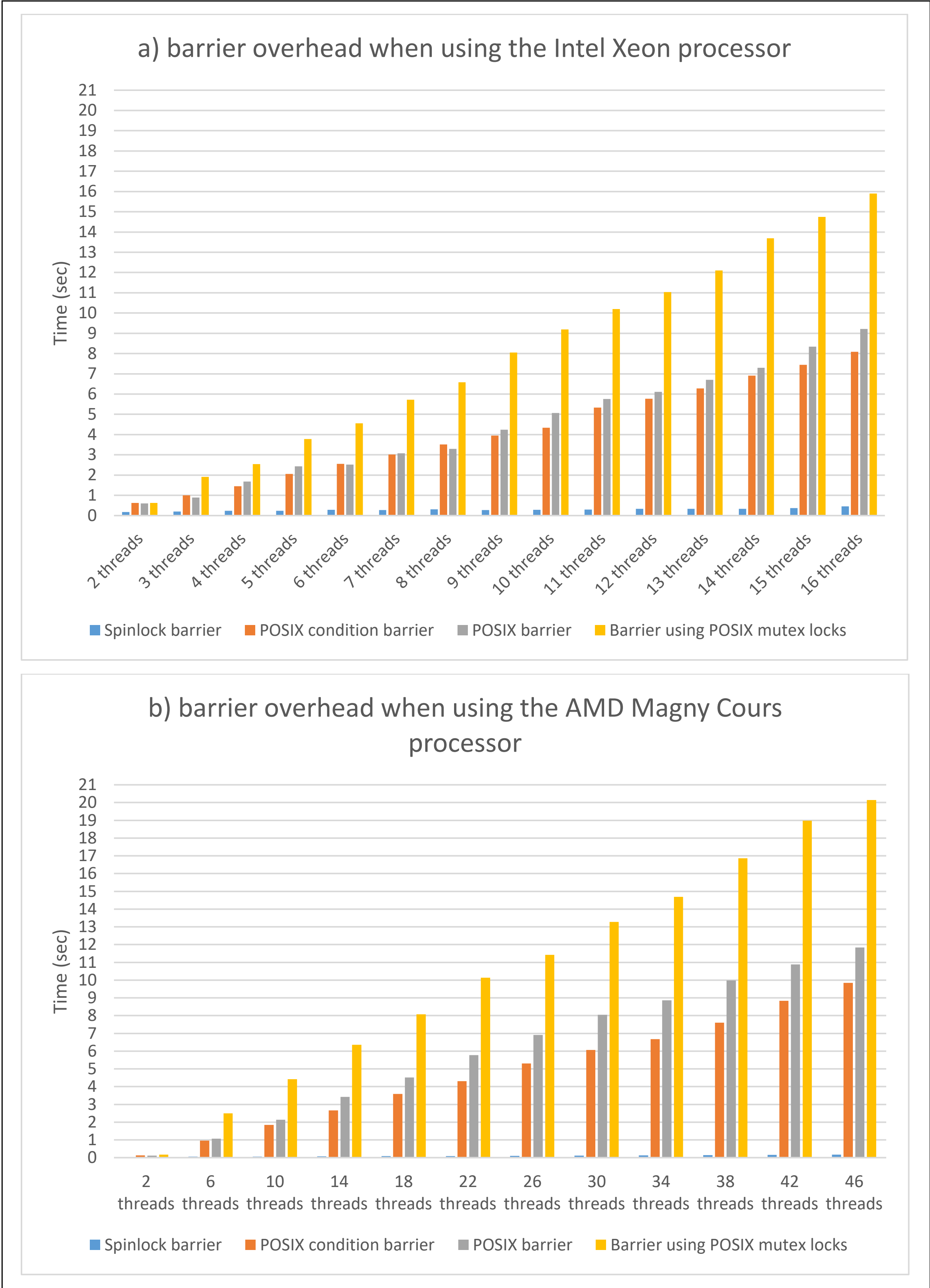
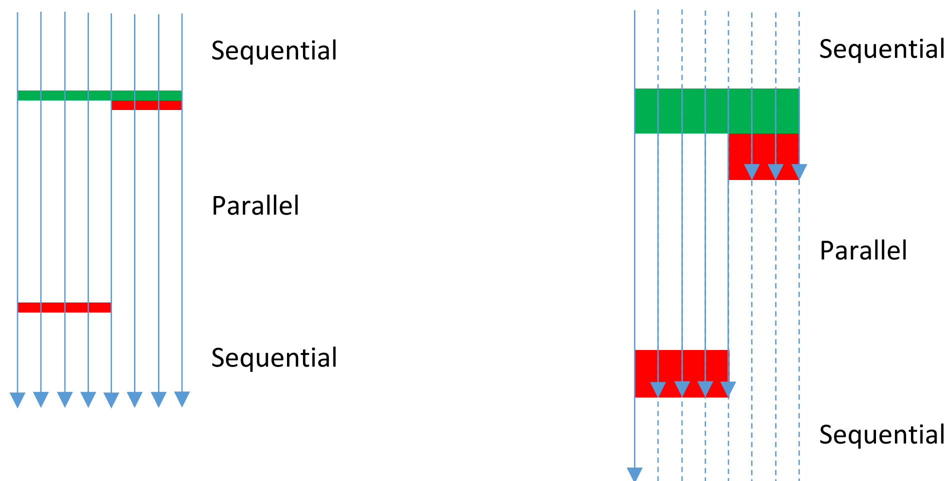


Figure 21: In the figures above are shown the relation between the barrier overhead time, the number of threads and the type of barrier. The barrier overhead is determined by carrying out time measurements using the program in Figure 20. The time measurements are carried out on a) the Intel Xeon processor and b) the AMD Magny Cours machine.

6 Simple energy model

With the new barriers energy can be saved if there are sufficient long periods in which less than the total number of threads are active. The question is whether in practice programs have sufficiently long periods in which fewer threads are active to enable energy to be saved. We cannot answer this question in general, but by using an energy model it is possible to predict whether the program shown in Figure 16 is more energy-efficient through the use of the new barriers. We shall therefor use this section to introduce an energy model. The program shown in Figure 16 is used as an example to gain an idea as to which type of program the new barriers produce energy savings. We hope in this way to be able to demonstrate whether the new barriers have any practical relevance. We shall analyse the model in order to find out what influence the different parameters have on energy consumption and what the most important parameters are. By identifying these parameters we hope to gain an idea as to how easy or how difficult it is in practice to save energy by the use of the new barriers.



a) SaC multithreaded execution model when using spinlock barriers

b) SaC multithreaded execution model when using one of the three new barriers that are able to deactivate/hibernate threads.

Deactivated thread — Start barrier
Active thread — Stop barrier

Figure 22: Schematic comparison between a) SaC's multithreaded execution model when using spinlock barriers and b) the execution model when using one of the new barriers that are able to deactivate/hibernate threads.

A model was created in order to enable a judgement to be made about energy consumption. Firstly, we study energy use in the unlikely event that 10 threads are used to execute a fully sequential program. In this instance the barriers are used to park 9 threads, after which the rest of the program runs on 1 thread. In order to gain an idea of the energy consumption of such a program, for the purpose of convenience we assume for now that each active thread uses a fixed amount of energy per µs. We call this fixed amount of energy a unit. If 10 threads take 100 µs there are thus 1,000 units of energy used. Let us assume that the computation time of the sequential program, which is

processed by 10 threads, amounts to 2,000 µs. When the spinlock is used as a barrier technique, all the threads remain active throughout the entire program and thus 20,000 units of energy are used.

If one of the other barriers is used, after some time 9 threads are paused, after which only 1 thread consumes energy. If we want to make a judgement as to how great the energy consumption is in this situation, we must know how long it takes before the 9 threads are paused. In order to give the sample computations any practical relevance, we use the data from Figure 18 and Figure 21b. In Figure 18 it can be seen that with 1 thread it takes approximately 12 seconds to add up two matrices of 400x400 elements. However, this time is measured over a large number of iterations, namely 6,000. One addition takes approximately 2,000 µs (2 ms). In Figure 21b it can be seen that the overhead of the POSIX condition barrier for the starting and pausing of 10 threads on the AMD Magny Cours takes approximately 2 seconds. It should be mentioned here that this overhead was calculated over 100,000 iterations. In reality the overhead amounts to approximately 20 µs.

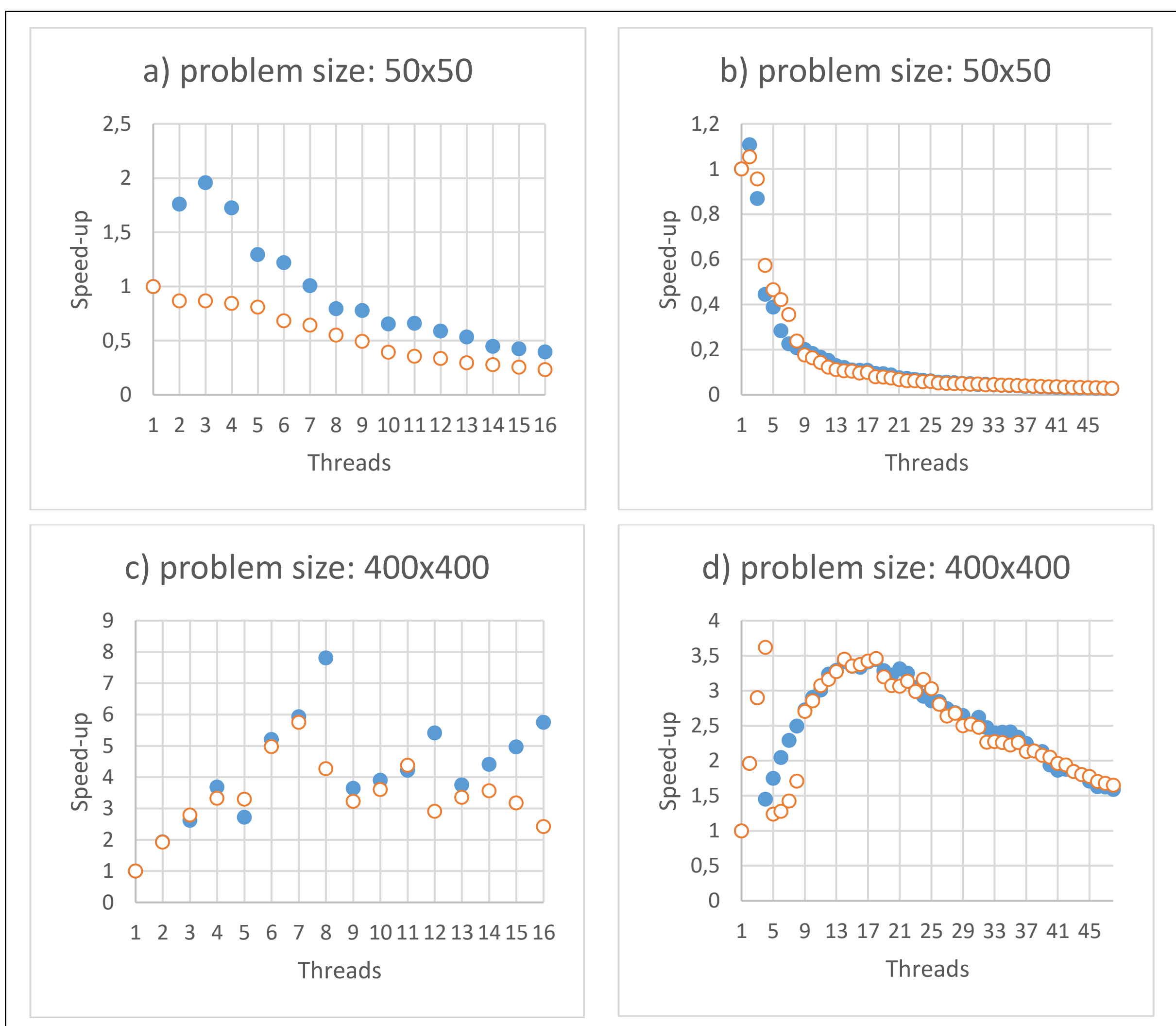


Figure 23: The figure above is obtained by repeating the experiment in Figure 17, in which the spinlock barrier is replaced with the POSIX-conditional barrier. The time measurements are carried out on a 2x 4 core hyper-threading Intel Xeon processor (left) and repeated on a 4x 12 core AMD Magny Cours circuit (right). In the graph the solid blue spheres show the speedup that is obtained without the use of the smart decision tool. The open orange spheres show the speedup when the smart decision tool is used.

In our model it therefore takes us approximately 20 µs to create 10 threads and then to pause 9 threads. The program then compute a further 2,000 µs. If 10 threads are active for 20 µs, this costs 200 units of energy. After that a further 2,000 units of energy are used to solve the computation. It therefore costs in total 2,200 units of energy to carry out a sequential computation of 2,000 µs with 10 threads using the POSIX-conditional barrier. This is less than the 20,000 units of energy that it would cost were the spinlock to be used. It is therefore probable that the use of the POSIX-conditional barrier will work energy efficiently when a sequential computation with 10 threads is carried out.

It is unlikely that a purely sequential program will be executed with 10 threads. When 10 threads are used, it is much more likely that a program runs both sequential and parallel code. Let us take as an example a program that consists first of a sequential part, followed by a parallel part. When the spinlock barrier is used and the program runs 2,000 µs, once again 20,000 units of energy will be used. With the spinlock barrier, all the threads remain active during the 2,000 µs.

With the POSIX-conditional barrier the model is slightly more complicated. By using this barrier less energy is used as long as the program runs sequentially. In order to be able to calculate how much energy is being saved if the program is carrying out the sequential part, we must know how large this sequential part is. To find this out we again take Figure 18 as the example. From Figure 18 we can see that when 10 threads are used, approximately four times less time is needed to solve the computation than when 1 thread is used. We therefore assume that the program of 2,000 µs is occupied for 1,500 µs with sequential tasks and 500 µs with a parallel computation. The program solves the same problem twice, first sequentially and after that in parallel.

Before the program can start the sequential computation, 10 threads must first be created, after which 9 are paused. We calculate 20 µs for this. In these 20 µs there are 200 units of energy consumed. After this the program runs sequentially for a further 1500 µs. This will take another 1500 units of energy. Thereupon 9 threads must be activated for the parallel computation to start. At the end of this computation the 9 threads must again be paused. We calculate for the starting and pausing of the 9 threads 20 µs once more. These are once again 200 units of energy. Subsequently a further 5,000 units of energy are used for the parallel computation. In total, needed to carry out the 2,000 µs duration program are therefore 6,900 units of energy paid for. This is still less than the 20,000 units of energy that would be paid for when the spinlock barrier is used. But it is considerably more than the 2,200 units of energy that are paid for a purely sequential program.

With a program that only carries out a parallel task, such as the program used to produce Figure 18, the new barriers create no energy saving. They consume energy, but the investment is not advantageous because the threads remain active. If it takes 500 µs to carry out the parallel computation with 10 threads, then the program with the spinlock barrier uses 5,000 units of energy. If instead of that the POSIX-conditional barrier is used, 5,200 units of energy are lost. 5,000 units for the computation and a further 200 for the barriers.

In this example however, we are working from the principle that all 10 threads are needed to carry out the computation in 500 µs. If it is possible to carry out the same computation with, for example, 9 threads in 500 µs, then 1 thread can be switched off. If the spinlock barrier is used the thread remains active and energy is therefore lost. With the new barriers, the thread can be switched off in order to save energy. If it takes 500 µs to carry out the parallel computation with 9 threads and 1 thread is parked, then 5,000 units of energy are lost when the spinlock barrier is used. With the POSIX-conditional barrier 4,700 units of energy are lost. 200 units are needed for the barriers and $9 \times 500 = 4{,}500$ units are needed for the computation. The program of Figure 18 that uses 25 out of

48 threads, will probably use less energy if it uses the POSIX-conditional barrier than when it uses the spinlock barrier. In practice, more complex programs are expected in which alternating sequential and parallel computations are used. In such a situation, the potential for energy saving, with for example the POSIX-conditional barrier, will only increase.

The models described above can be formulated as equations. We use the following parameters to formulate these equations:

Table 1: Description of the parameters that occur in the energy models for the four barriers implemented in SaC.

Parameter	Description
t	Total duration of the program (µs)
p	Number of threads/processes
T	Overhead per thread/process (µs)
f	Fraction of parallel executable code
n	Number of program parts that can be executed in parallel

The total number of units of energy that is calculated using these equations we call E. Indicated with E_{spin} is the total number of units of energy when the spinlock barrier is used. With E_{sleep} we indicate the total number of units of energy when one of the other barriers is used. We shall first derive the equation of E_{spin}.

With the spinlock barrier all the threads are always active. This means that as long as a program is running, p threads will consume energy. In total the program runs for t µs. Equation (7) therefore gives the total number of units of energy when the spinlock barrier is used. In the description of the model the overhead that the spinlock barrier creates is always omitted because it is relatively small. This overhead is therefore also not included in equation (7).

$$E_{spin} = t \cdot p \quad (7)$$

The equation for E_{sleep} is more complicated. In Figure 22b it can be seen that the number of active threads is dependent on which phase of the program is being executed: sequential or parallel. Three phases can be distinguished in Figure 22b. A sequential phase, in which only one thread is active. A phase in which the barriers are engaged in the activation or deactivation of threads. And, finally, there is the parallel phase in which some of the threads are active. We shall formulate the equation for E_{sleep} by writing down the three phases in terms of the parameters described in Table 1. Implicitly in Figure 22b a conservative assumption is made as regards these barriers. In the figure all the threads start immediately and simultaneously and so they are active during the entire barrier phase. This assumption implies that the barrier has maximum energy consumption. In reality, it is highly likely that the activation and deactivation of threads takes place gradually. This is, however, more difficult to express in a model. In order to simplify the model further we assume that during the parallel phase all the threads are active. As a result E_{sleep} gives an extent of energy consumption that is too high in many instances. Finally, we assume that a sequential phase precedes every parallel phase and that a program always ends with a sequential phase.

There is only 1 thread active during the sequential phases of the program. The duration of the sequential phases is thus proportionate to the energy consumption. The following equation is therefore a description of the energy consumption of the sequential phases: $(1 - f) \cdot t$. Put into words, the fraction of sequential code multiplied by the total duration of the program. The energy consumption of the barrier is equal to the time that the barrier uses multiplied by the number of

threads: $T \cdot p$. In total there are $n + 1$ barriers, namely one barrier for each parallel part of the program (which summarizes to a total of n) and one further barrier at the beginning of the program, because the threads must also be created. Therefore the energy consumption of all the barriers put together amount to: $(n + 1) \cdot T \cdot p$. During the parallel phases all the threads are active. The energy consumption of all parallel phases together is therefore equal to the time that the parallel phases are active multiplied by the total number of threads. Together, all the parallel phases cost $f \cdot t$ time and there are p threads. The energy costs for the parallel phases are therefore: $f \cdot t \cdot p$. If we add together the energy costs of the sequential phases, the barrier phases and the parallel phases, equation (8) is obtained. This equation gives the energy consumption for a program that uses one of the new barriers.

$$E_{sleep} = (n + 1) \cdot T \cdot p + (1 - f) \cdot t + f \cdot t \cdot p \qquad (8)$$

We can now ask ourselves what parameters determine the energy consumption of the new barriers. When do the new barriers consume less energy than the spinlock? The equation below gives us an answer to these questions.

$$\begin{aligned} &E_{sleep} < E_{spin} \\ &(n+1) \cdot T \cdot p + (1-f) \cdot t + f \cdot t \cdot p < t \cdot p \\ &(n+1) \cdot T \cdot p + t - f \cdot t + f \cdot t \cdot p < t \cdot p \\ &(n+1) \cdot T \cdot p + t - f \cdot t + f \cdot t \cdot p < 0 \\ &(n+1) \cdot T \cdot p + (f-1) \cdot (p-1) \cdot t < 0 \\ &(n+1) \cdot T \cdot p < -(f-1) \cdot (p-1) \cdot t \\ &(n+1) \cdot T \cdot p < (p-1) \cdot (1-f) \cdot t \end{aligned} \qquad (9)$$

$$(n+1)T\frac{p}{p-1} < (1-f)t \qquad (10)$$

Put into words, as long as the overhead is smaller than the sequential part of the program, it is more energy efficient to use the new barriers than to use the spinlock barrier. There are therefore two ways in which use of the new barriers can be made more energy efficient than the use of the spinlock barrier. The sequential part of the program can be made greater or the overhead can be made smaller. When the sequential part of the program is made greater the speedup measured over the whole program is lower. This is the opposite result to what one would want to achieve by writing an executable parallel program. It is therefore not a good idea to enlarge the sequential part of the program.

The alternative is to reduce the size of the overhead. This can be achieved by reducing the number of parallel executable pieces of code, or by reducing the size of the overhead per thread or (strangely enough) by increasing the number of threads. To begin with this last alternative, the overhead is not actually reduced by adding more threads, it is a relative difference. By adding more threads, the energy consumption of a program that makes use of the spinlock barrier is increased. However, when a program makes use of one of the new barriers, adding threads gives rise to more potential for energy saving. To be precise, energy is saved on $(p - 1)$ threads so long as sequential code is executed. That is why we find the factor $(p - 1)$ in equation (9); $(p - 1)$ threads multiplied by the time that sequential code is executed.

Unfortunately the barrier overhead actually increases with the number of threads (p). When equation (9) is written like equation (10), the barrier overhead increase p and the potential energy

saving $(p-1)$ form a fraction $\frac{p}{p-1}$. This fraction has a limit: $\lim_{p\to\infty}\frac{p}{p-1}=1$. Increasing the number of threads has therefore limited influence on the extent to which energy can relatively be saved. For this reason it is not a good idea to increase the number of threads with the objective of becoming more energy efficient than a program that uses a spinlock barrier. Furthermore, in absolute terms the energy consumption increases with each thread. Two threads use more energy than one. In addition, if we base ourselves on the ideal situation in which the smart decision tool takes optimum[1] decisions, with regard to computation speed and/or energy consumption, there is nothing to gain by adjusting the number of threads. After all, the decision made by the smart decision tool is optimal[1].

It may be possible to save energy by reducing the number of parallel executable pieces of code. This has already been done in SaC by, whenever this is technically possible, merging with-loops and executing them as single operations. The best way in which to save energy is possibly by reducing the overhead per thread. From this study, it appears that the POSIX-conditional barrier is the technique with the least overhead per thread. Whether a barrier can be found with a smaller amount of overhead per thread (and that still makes use of a technique with which threads are paused), must be evident from future study.

Although Figure 21 and Figure 23 show that the new barriers are slower than the spinlock barriers, the model (10) indicates that there is relatively little needed to make the new barriers more energy efficient than the spinlock barriers. Furthermore, the model does not take into account the influence of the smart decision tool. With the smart decision tool, less than the maximum number of threads may be used during parallel phases of the program. In order to introduce the model, an estimate was made earlier of what in reality the relation between overhead and the duration of the program could be. With this an overhead of 20 µs on a 2,000 µs-duration program was budgeted for. If we assume that switching off threads during a parallel section of the program saves energy in a similar manner as switching off threads during a sequential section, then switching off 1 thread should be more than sufficient to obtain a lower energy consumption by one of the new barriers than when the spinlock barrier were to be used. The total overhead over all the threads amounts to 20 µs, but by switching off 1 thread already 2,000 µs-worth of energy is saved. Whether this model augurs well for the energy consumption in practice must be researched.

7 Related work

Other techniques have been developed to estimate the number of threads that give maximum speed-up for a given algorithm on a given machine. [2] Abstract computer models such as "Parallel Random Access Machine" (PRAM) [19] can be used to estimate the performance of an algorithm when executed on a real machine. PRAM is a model for a theoretical, unlimited parallel machine with shared memory that is uniformly accessible from all processors. With PRAM one can compute the theoretically best possible performance of an algorithm.

"Bulk Synchronous Parallel" (BSP) [20] is another example of an abstract computer model. However, it is based on different assumptions. BSP does not take communication and synchronization for granted. BSP assumes that a parallel computation can be described as three phases. A local computation phase, a communication phase, and a synchronization phase. The local computation phase is the time where computations occur asynchronously on each processor. The duration of the local computation phase is equal to the time needed to complete the most time-consuming process. The communication phase is the time that is used for sharing data between processors. Finally, the synchronization phase is the overhead time that is caused by the need to synchronize computations between processors.

The performance estimations of abstract computer models such as PRAM and BSP are often used to predict the number of threads that give maximum speedup. The smart decision tool was developed to automate the estimation of the number of threads. In order to use abstract computer models such as PRAM and BSP a detailed knowledge about the algorithm, the machine and the to be processed data is required.

A considerable number of studies concerns the automatic determination of the number of threads that give maximum speed-up for a given program. Pusukuri et al. propose an offline profile-based technique called the thread reinforce framework. [21] The framework executes a program in two steps. During the first step the program is executed in short sequences collecting parameters that are marked as good indicators for predicting the number of threads that give maximum speed-up. Once the number of threads that give maximum speed-up is found, the program is fully re-executed using that number of threads.

The thread reinforce framework automates the determination of the number of threads that give maximum speed-up for a given program. The thread reinforce framework is not aware of what happens within the program and does not optimize individual parallel sections. When a program consists of many different parallel sections, a runtime constant number of threads is used, which might not be the best choice for each parallel section. We are again entitled to abstract computer models to find the number of threads that give maximum speed-up for the different parallel sections. The methods as described by Pusukuri et al. are therefore not a fully-fledged alternative for abstract computer models.

Gordon et al. have developed a technique to balance resources between multiple SaC programs. The number of threads that give maximum speed-up has to be user provided upon execution of the SaC program. Under normal circumstances this number of threads that give maximum speed-up is used to execute all the parallel sections that exist within a SaC program. However, using the technique as developed by Gordon et al. the performance of the parallel sections is constantly monitored. When a performance drop is detected threads may be taken from one program to free up resources for the other. [22]

Just like the smart decision tool, the technique as developed by Gordon et al. is perfectly able to optimize the different parallel sections that exists in a program. In addition, the number of threads used in each parallel section are adapted based on parameters that are often not included in computer models. The complicated effects of executing multiple programs on the same machine is just one such a parameter. However, the technique does not automate the estimation of the number of threads that give maximum speed-up. The number of threads that give maximum speed-up have to be provided manually. The number of threads are merely adopted based on realtime performance parameters.

OpenMP offers yet another technique to change the level of parallelism at runtime. Using the so-called if-clause, users are able to write conditions for parallel tasks. The tasks are parallelized only if the conditions are met. [23] Using the if-clause one could prevent a task from being executed in parallel for example when the data size is small. Although, the if-clause could be seen as a technique for changing the level of parallelism at runtime, there are only two choices: a task is either executed in parallel or not. Furthermore, the technique does not automate the estimation of the number of threads that give maximum speed-up.

Only a few studies concern automatic determination of the number of threads that give maximum speedup at the level of the individual parallel sections within a program. Suleman et al. have studied one such technique implemented in OpenMP. [24] They assume that the performance of most parallel applications is either data-synchronization limited (Synchronization-Aware Threading (SAT) applications) or bandwidth limited (Bandwidth-Aware Threading (BAT) applications).

Suleman et al. are using an online profile-based technique to compute the number of threads that give maximum speed-up. The main difference between an online technique and an offline technique, such as the smart decision tool is using, is that, with an online technique, measurements and optimization are all happening while the program is used for productive work. This means that with an online technique, the performance of the measurements and optimization steps are crucial. When the measurement and optimization investments take more time that the actual execution of the program, online techniques will only result in less performance.

On the other hand, offline techniques have the disadvantage of not operating on live data. As a result, the quality of any predictions very much depends on the quality of the training data that is provided. This introduces another disadvantage of the offline approach; the user has to provide good training data. When the training data is very different from the live data, the performance of the program acting in the production environment may not be optimal. Another argument against the offline approach is that the measurement and optimization steps can take a considerable amount of time, as there are no limits in how long this can take. However, the program that is used for the productive environment is completely free from overhead caused by measurement and optimization steps.

Suleman et al. are using a three-step online profile-based technique to compute the number of threads that give maximum speed-up for both SAT and BAT applications. All parallel computations are started single threaded. During this first step parameters such as execution time and cycle counts are collected. In a second step the number of threads that give maximum speed-up is computed based on the collected data. Suleman et al. are using models to estimate the number of threads that give maximum speed-up. There are separate models for SAT and BAT applications. As it is jet unknown whether the program is SAT of BAT limited, both models are used. The minimum is said to be the number of threads that give maximum speed-up. In the third step the remainder of the

parallel computation is executed using the computed number of threads that give maximum speed-up.

The smart decision tool and the technique developed by Suleman et al. differ in their approach. The smart decision tool is an offline profile based technique whereas Suleman et al. are using an online technique. Furthermore, Suleman et al. are using models to predict the number of threads that give maximum speed-up, while the smart decision tool uses the raw performance data.

8 Discussion

SaC can be used to write fast multicore software without the need to know about hardware properties and multicore software design. Many program languages today are either close to the user's perspective but do not have the best possible multicore performance or they are closer to machine language but can be heavily optimized. The aim of SaC is to provide a language that is close to the user's perspective and still performs well on a wide area of machines, especially multicore machines. [7]

The smart decision tool is developed to improve the multicore performance, without changing the application code. The smart decision tool can be seen as the next step for program languages such as SaC, towards a language that is close to the user's perspective and still performs well. The smart decision tool tries to adapt the number of threads based on the requirements of multicore computations, instead of changing any application code. Figure 1 shows how speed-up is influenced by the number of threads. Adding more threads can result in more speed-up, but the speed-up can also drop when too many threads are used. To achieve maximum speed-up, it is very important to select the right number or threads.

In chapter 5.1 we have shown that the smart decision tool is able to determine the number of threads for maximum speed-up correctly for many of our test cases. In addition, the smart decision tool is able to adapt the number of threads dynamically while the application is running. In Figure 19 we show the relation between speed-up and the number of threads for one of our test applications. The number of threads for each measurement where configured by the smart decision tool. The figure does show that the number of threads effects the speed-up of the application. This indicates that the smart decision tool is indeed able to change the number of threads. The smart decision tool is thus able to both determine the number of threads for maximum speed-up and dynamically configure the running application such that it uses the optimal number of threads. Finally, as was explained in chapter 5.3 , energy consumption can be reduced when the smart decision tool is used in combination with the new barriers that we have designed for SaC.

However, the smart decision tool still has a number of issues that make the technique behave different from what it was designed for. The smart decision tool does not always predict the number of threads for maximum speedup correctly, as shown in chapter 5.1. The smart decision tool measures the time-thread relations of the parallel sections of a SaC program. Polynomial interpolation is used to minimize local differences in the time-thread relations. As a result, artefacts such as measurement errors have little effect on the predictions of the smart decision tool. But polynomial interpolation does also minimize local changes that should not be ignored, such as changes due to hyper threading. Furthermore, the performance graph flattens when approaching the maximum. On an almost flat curve local changes become important.

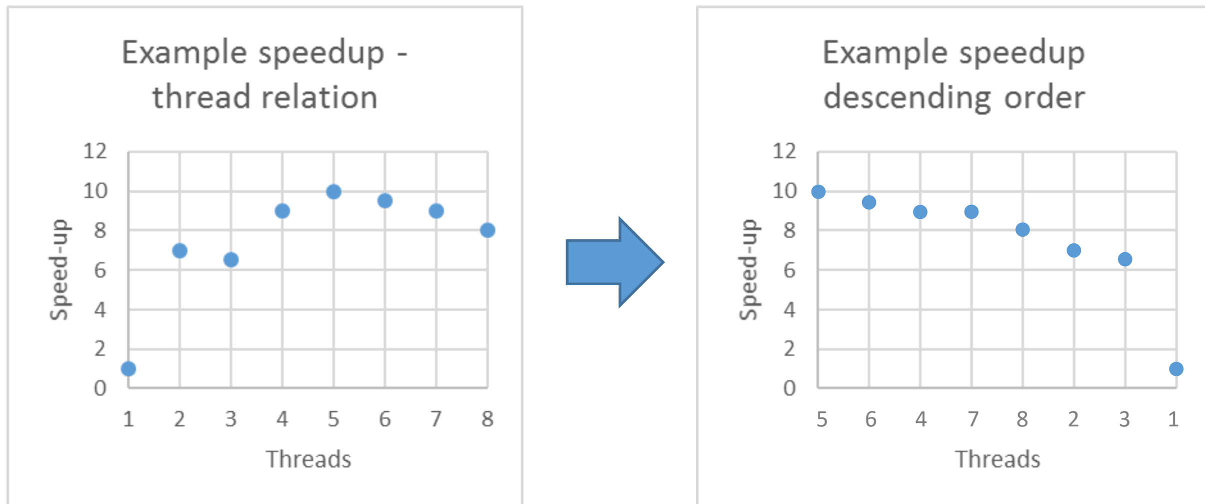


Figure 24: Alternative technique for predicting the number of threads for maximum speedup. The speedup-thread relation graph (left) is put in descending order based on the speedup (right). The first measurement in the sorted graph must bet he number of threads that result in the best performance.

To solve these issues a solution must be found that ignores local maxima as long as performance globally increases, but does not ignore local maxima when performance does no longer globally increase. Figure 24 shows an example of such a technique. The graph on the left is an example of a typical speedup-thread relation. A bit of noise has been added to this graph to make it appear more realistic. To compute the number of threads for maximum speedup, the graph on the left-hand side of the figure is put in descending order of speedup. The result of this operation is the graph that is shown on the right-hand side of Figure 24. The first measurement in the sorted graph must be the number of threads that result in the best performance.

When an energy-performance tradeoff has to be made, one could look for the first measurement in the sorted graph that uses less threads than the maximum. In Figure 24 this would be the performance that is obtained with 4 threads. This point is a candidate for giving the best energy-performance tradeoff. The gradient between the maximum and the candidate can be used as energy-performance ratio. When the gradient is above some user defined threshold, the maximum is marked as the best energy-performance tradeoff. When the gradient is below the user defined threshold, the candidate is temporarily marked as best energy-performance tradeoff and the search continues with the next candidate. In Figure 24 the next candidate would be the performance that is obtained with 2 threads. This technique could be subject for future research.

When the smart decision tool has selected the number of threads for maximum speed-up, the performance can still be less than expected. The smart decision tool uses the thread scheduler to divide the parallel work over the appropriate number of threads. This mechanism was explained in chapter 5.2. The remaining number of threads do not receive any work. However, when the barrier is lifted all threads are activated, including the threads without work. The threads that do not have work are not kept alive, but are immediately deactivated. As a result, overhead is paid to activate and deactivate threads that are not being used. No matter what number of threads are used, the overhead is always the same. The smart decision tool does not succeed in its goal to achieve the maximum performance for each parallel section.

The solution to this problem would be to only activate threads that are actually used. To accomplish this, the barrier system has to be redesigned such that individual threads can be activated. The scheduler and the new barrier system have to cooperate. The threads that are activated must be the

same threads that are provided with work, or the computations may produce wrong results. The barrier system has to be fast as well. Not activating a certain amount of threads may save time. However, if that time is spent while activating the remaining threads, there is no benefit.

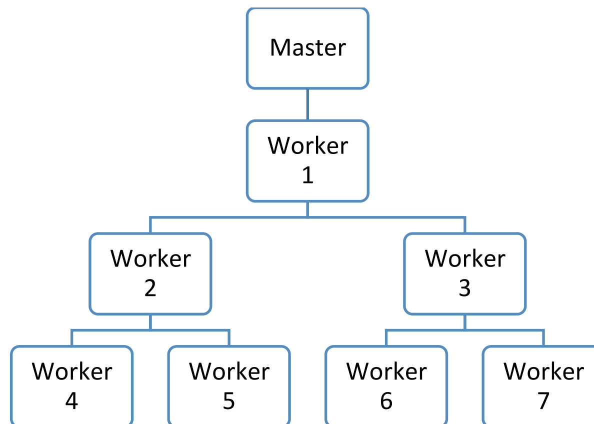


Figure 25: Suggested model for activating threads on individual basis. The master thread does activate the first worker thread. Each worker thread is responsible for activating two more worker threads until there are no threads left to cover.

With the current barrier system, threads can only be activated by a master thread. This approach may become very expensive when threads have to be activated on an individual basis. With the current barrier system, it can also be quite difficult to find and activate only those threads that are provided with work. Instead, each worker thread could be responsible for activating two more worker threads until there are no threads left to cover. The master thread is responsible for activating the first worker thread. Figure 25 illustrates such an activation scheme for an example of 8 threads. When the system requires 6 threads, the master thread activates the first worker thread. The first worker thread has to activate worker 2 and 3. And worker 2 has to activate worker 4 and 5. Due to the activation scheme, threads are activated in a very specific order. The setup for the activation scheme can be made such that the order in which threads are activated matches the order in which the scheduler provides work.

A similar tree like organization is used to create all threads when a SaC program is executed. The order in which threads are created is similar to the order used by the scheduler to provide work to the threads. The thread creation system is therefore very suitable to setup the thread activation scheme for the new barrier system.

To support the new activation scheme, the barriers have to be slightly modified. The POSIX-thread barrier for example can be set up such that it covers only two threads. When the function is called for the first time a thread is deactivated. When the function is called for the second time the deactivated thread is again activated. Such a POSIX-thread barrier has to be created for each thread. The call to the POSIX-thread barrier is shared with the parent worker thread, according to the scheme of Figure 25. Worker threads have to deactivate themselves by calling their own POSIX-thread barrier. They are reactivated when the parent thread calls their POSIX-thread barrier again. A similar setup is possible with other barrier techniques. The development of a fast new barrier system that can activate threads individually, may be a topic for future research.

In chapter 5.3 we show that despite all performance issues, energy can be saved when the smart decision tool is used. To reduce the energy consumption, the smart decision tool has to be combined with a barrier technique that can disable threads. For that purpose, three new barriers were designed: the POSIX-thread barrier, the POSIX-conditional barrier, and the POSIX-mutex barrier. In chapter 5.3 we prove that the original spinlock barrier technique performs much better than any of

the new barriers. However, the energy consumption per µs of a program that uses the spinlock technology is fixed. With the spinlock technique, a program that uses a single thread consumes just as much energy as a program that uses multiple threads. When the execution time of a program increases, the potential for saving energy by deactivating threads becomes larger. At some point less energy is consumed when using one of the new barriers than when using the spinlock barrier.

With the energy model introduced in chapter 5.3 it was demonstrated that the new barriers are beneficial in terms of energy consumption when the total thread inactivity time exceeds the barrier overhead time. It is not unlikely that this criterion is to be met in many cases. Although the new barriers cause quite some overhead, we can expect that in most realistic programs more time is spent on computations than waiting for barriers. These programs offer quite some potential to reduce energy consumption. If only a single thread can be deactivated during the computations, energy consumption is reduced. Reducing energy consumption can be made more easily when the barrier overhead is less. Therefore, future research regarding faster barriers that can deactivate threads, may result in SaC compiled programs that consume less energy. If barriers cannot be made faster when universal standards such as POSIX are used, faster barriers can possibly be created by using machine specific or operating system specific technologies.

9 Conclusions

It is possible to reduce the computation time of a program by dividing the computation work among different threads and executing them in parallel. We then speak of speedup of the program. With the help of a model and Figure 1 we demonstrated that when too many threads are used, speedup can also decrease. In order to make the computation time of a program as short as possible, it is essential to determine the right number of threads. Furthermore, energy consumption increases as more threads are used. For this reason, too, it is important to pay close attention to how many threads are used.

In this research, we have studied whether the right number of threads can be chosen automatically. To do this we have developed a technique that we call the smart decision tool. The technique first measures, for different numbers of threads, how much time is used in completing a computation task in parallel. After that is determined with how many threads a computation task runs best. This is dependent on the wishes of the user. For example, if the user wishes to minimize the computation time of the program, the number of threads with which the speedup of the computation task is the greatest is used. Within a program more than just one parallel computation can occur. The smart decision tool has been developed such that an individual time measurement can be executed for every computation task. In this way the smart decision tool can determine for every parallel computation task with how many threads the task in question will run best. SaC was used as host programming language and environment to enable the testing of the smart decision tool.

The smart decision tool appears to be capable of determining the number of threads that will produce the maximum speedup. In chapter 5.1 (Figure 17) we have demonstrated that in various tests the smart decision tool was successful in indicating correctly the number of threads with which the speedup is maximum. But as discussed in chapter 8 there are also circumstances in which the smart decision tool cannot determine (well) the number of threads for maximum speedup. This is so because the smart decision tool does not determine the number of threads for maximum speedup directly from the raw data. First the measurements are fit with a curve using polynomial interpolation. The number of threads at which the curve has a maximum is considered to be the number of threads for maximum speedup. This technique does not work well with sudden changes in the measurement data or when measurements remain almost constant. How this problem can be solved is open to further research. Retrospectively, polynomial interpolation could be seen as an unfortunate design choice.

Apart from determining the optimum number of threads, the smart decision tool also appears to be capable of adapting the number of threads in real-time. The real-time adapting of the number of threads is possible thanks to an integration of the smart decision tool with the SaC thread scheduler. Apparent in Figure 19 is that the smart decision tool is able to adapt the number of threads because the program’s speedup changes as more threads are added. The present integration of the smart decision tool with the SaC thread scheduler does however have a shortcoming. The technique only adapts how work is divided among the threads. However, when parallel tasks are executed, threads that are not provided with work are still activated only to detect that they are not needed. The unnecessary extra number of threads are then deactivated again. The activation and deactivation of threads creates a large amount of extra overhead. Future research must find a solution to this problem.

Using the smart decision tool in combination with the new barriers that we have developed it is possible to save energy. The new barriers were developed because the existing barriers in SaC make

use of a spinlock technique. This technique causes threads to remain active even when they are not used in computation. As a result, with the spinlock barriers it is not possible to save energy by deploying fewer threads in for certain computations. With the new barriers, threads can be deactivated when they are not needed and reactivated as soon as there is sufficient computation work available. However, this deactivation and reactivation takes time, as the result of which the new barriers are slower than the spinlock barrier. This difference was revealed in Figure 21. However, when the duration of the program run is longer than the time taken in deactivating and reactivating threads, the new barriers almost always result in energy savings.

The smart decision tool is capable to determine the optimal number of threads as well as adapting the number of threads in real-time. Furthermore, using the smart decision tool in combination with the new barriers it is possible to save energy. Therefore, the smart decision tool promises a great potential to be effectively used. However, further research is needed to determine the optimal number of threads in all circumstances and to develop a technique that prevents the scheduler from activating threads that are not being used to solve computations.

References

[1] S. Akhter and J. Roberts, Multi-Core Programming: Increasing Performance through Software Multi-threading, Intel Press, 2006.

[2] A. Grama, A. Gupta, G. Karypis and V. Kumar, Introduction to Parallel Computing, Harlow: Pearson, 2003.

[3] G. M. Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” in *Spring Joint Computer Conference*, Sunnyvale, California, 1967.

[4] J. L. Gustafson, “Reevaluating Amdahl's law,” *Communications of the ACM,* pp. Volume 31, Number 5, 1988.

[5] W. Nasri and K. Fathallah, “A Performance model for OpenMP programs on multi-core machines,” *Computer Applications Technology,* pp. 1-6, 2013.

[6] S.-B. Scholz, “Single Assignment C - Functional Programming Using Imperative Style,” in *Implementation of Functional Languages*, Norwich, 1994.

[7] C. Grelck and S.-B. Scholz, “SAC — A Functional Array Language for Efficient Multi-threaded Execution,” *International Journal of Parallel Programming,* no. Volume 34, Issue 4, p. 383–427, 2006.

[8] C. Grelck, “Single Assignment C (SAC) High Productivity Meets High Performance,” in *Central European Functional Programming School*, Berlin, Springer Berlin Heidelberg, 2012, pp. 207-278.

[9] C. Grelck, “Shared memory multiprocessor support for functional array processing in SAC,” *Journal of Functional Programming,* no. Volume 15, Issue 3, pp. 353-401, 2005.

[10] G. Michaelson, An Introduction to Functional Programming Through Lambda Calculus, Dover: Courier Corporation, 2011.

[11] C. Grelck and S.-B. Scholz, “Axis Control in SAC,” in *Implementation of Functional Languages*, Berlin, Springer, 2003, pp. 182-198.

[12] C. Grelck, S. Kuthe and S.-B. Scholz, “A Hybrid Shared Memory Execution Model for a Data Parallel Language with I/O,” *Parallel Processing Letters,* no. Volume 18, Issue 01,, 2008.

[13] S.-B. Scholz, S. Herhut, F. Penczek and C. Grelck, “Single Assignment C - tutorial,” 2010.

[14] IEEE Computer Society; The Open Group, “Standard for Information Technology - Portable Operating System Interface (POSIX),” 13 Sept 2016. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=7582338.

[15] Portable Application Standards Committee; IEEE Computer Society; The Open Group, “1003.1j-2000 - IEEE Standard for Information Technology- Portable Operating System Interface (POSIX)-Part 1: System Application Program Interface (API)-Amendment J: Advanced Real-time

Extensions [C Language],” 30 Jan 2000. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=859471.

[16] “Intel® Xeon® Processor E5620,” Intel, [Online]. Available: http://ark.intel.com/products/47925/Intel-Xeon-Processor-E5620-12M-Cache-2_40-GHz-5_86-GTs-Intel-QPI.

[17] H. Bal, D. Epema, C. de Laat, R. van Nieuwpoort, J. Romein, F. Seinstra, C. Snoek and H. Wijshoff, “A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term,” *IEEE Computer,* no. Vol. 49, No. 5, pp. 54-63, 2016.

[18] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller and M. Upton, “Hyper-Threading Technology Architecture and Microarchitecture.,” *Intel Technology Journal ,* no. Vol. 6 Issue 1, pp. 1-12, 2002.

[19] R. M. Karp, “A survey of parallel algorithms for shared-memory machines,” *University of California at Berkeley,* 1988.

[20] L. G. Valiant, “A bridging model for parallel computation,” *ACM,* no. Volume 33 Issue 8, pp. 103-111, 1990.

[21] K. K. Pusukuri, R. Gupta and L. N. Bhuyan, “Thread Reinforcer: Dynamically Determining Number of Threads via OS Level Monitoring,” in *IISWC 11*, Washington, 2011.

[22] S. Gordon and S.-B. Scholz, “Dynamic Control of Runtime Systems Through a Common Interface,” in *IFL '15 Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages*, New York, 2015.

[23] L. Dagum and R. Menon, “OpenMP: an industry standard API for shared-memory programming,” *Computational Science & Engineering,* no. Volume 5, pp. 46-55, 1998.

[24] M. A. Suleman, M. K. Qureshi and Y. N. Patt, “Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on CMPs,” in *ASPLOS*, New York, 2008.