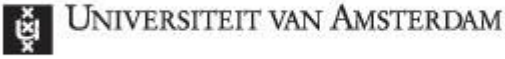
Using OpenMP as an Alternative Parallelization Strategy in SAC

Zheng Zhangzheng



January 2011

Using OpenMP as an Alternative Parallelization Strategy in SAC

Master's Thesis (Afstudeerscriptie)
written by
Zheng Zhangzheng

under the supervision of Dr. C. Grelck, and submitted in partial fulfillment
of the requirements for the degree of

Master of Science in Grid Computing
at the *University of Amsterdam*.

Date of the public defense:
9th February 2011

Members of the Thesis Committee:
Dr. C. Grelck
Dr. A.S.Z. Belloum
Dr. I. Bethke


UNIVERSITEIT VAN AMSTERDAM

Content

Chapter 1. Introduction

SINGLE ASSIGNMENT C — in the following referred to as SAC — is a purely functional array processing language which is especially designed for computationally intensive applications in fields such as scientific computing, image processing, simulation, or modeling**.** Its design aims at achieving excellent runtime performance and offering a high level of abstraction at the same time. And the functional property of SAC reveals the fact that SAC is based on the principle of context-free substitution instead of a stepwise modification of state.

SAC has the array as the first class object. This means that array is an entity that can be passed as a parameter, returned from a subroutine, or assigned into a variable. And every variable in SAC is regarded as an array. For the simplicity of the array operation, SAC provides a few built-in functions to manipulate on the shape of the array. And more complex array operations can be defined by so-called WITH-loop which is a versatile construct to define aggregate operations on arrays. The WITH-loop allows denoting operations which completely abstract from the concrete shapes and even ranks of the arrays involved. Due to its expressiveness, almost all array operations are implemented through WITH-loop.

Considering its functional semantics, SAC is well-suited for parallel execution. Previous work **[1, 3, 4, 5, 10]** has been done to use PTHREAD **[22]** to generate the multithread code and the performance for this parallelization strategy has been proven to be good. For nearly all typical computationally intensive applications, a considerable portion of execution time is spent on array operations. Thus the PTHRED parallel strategy to generate executable concurrent code is directed to the WITH-loop construct.

OpenMP **[14]** is a shared-memory application programming interface (API) to specify shared-memory parallelization in FORTRAN and C/C++ programs. Since its advent in 1990s, OpenMP has become the de-facto standard in writing shared-memory parallel programs and mainstream software and hardware vendors have been actively involved in the development of OpenMP.

OpenMP is comprised of a set of compiler directives, runtime library routines and environment variables. The principle of OpenMP is summarized as follows: OpenMP expects the application developer to give a high-level specification of the parallelization

in the program and the method for exploiting that parallelization by means of inserting compiler directives at appropriate places of the program and setting environment variables if necessary. And the responsibilities of the OpenMP runtime library routines are to sort out the low-level details of the parallel execution, such as creating a team of threads to execute the code in parallel, dividing the work between the threads according to a certain mechanism, synchronizing the threads at the end of the parallel execution.

OpenMP 2.5 “is somewhat tailored for large array-based applications” **[23]**. Thus OpenMP 2.5 is ill equipped to exploit the concurrency available in irregular and dynamic structures such as while loops and recursive routines. But with the advent of OpenMP 3.0 **[15, 16, 17]**, a task model is put forward to efficiently exploit less structured concurrency.

This thesis is about designing, implementing and evaluating an alternative OpenMP-based parallelization strategy for SAC. The idea is to create a new backend utilizing OpenMP to generate the multithread program for SAC on all the architectures that support OpenMP. After inserting correct OpenMP directives and clauses in the appropriate places of the intermediate C code, it is up to OpenMP to generate the correct and efficient multithread code. The performance from the experiments which will be explained in Chapter 5 demonstrates the suitability of this parallelization strategy in principle.

Considering the presence of PTHREAD parallelization strategy and the fact that PTHREAD strategy already provides good performance, the motivation of OpenMP parallelization strategy is the combination of the following factors:

- PTHREAD is restricted to the shared memory architecture. For the new rising architectures such as ASMP which stands for Asymmetric multiprocessing, the code generated from SAC program through PTHERAD parallel solution may not run on these architectures. But OpenMP is a widely used API for parallel programming. It is attractive to support OpenMP because programmers can continue using their familiar programming model, and existing code can be re-used **[18]**. Thus it is likely that OpenMP is adapted to the new rising architectures by the experts in the industry the moment the new architecture arises. Then SAC can rely on OpenMP to run in parallel on these architectures. For instance, Cell **[19]** is an ASMP architecture designed by IBM which does not support PTHREAD, thus it is not possible to run PTHREAD parallel solution code on it. But with the work presented in **[18]**, the OpenMP parallel solution code can run on Cell since additional work to support OpenMP on Cell architecture is already done. Another example is that OpenMP can be adapted to GPGPU **[20]** architecture which is designed by NVIDIA. Interested reader can find more information in **[21]**.

.

- SAC will support task parallelization in the near future, but how to integrate task parallelization with the current data parallelization elegantly and efficiently is still an open topic of research. As illustrated before, OpenMP 3.0 now supports task parallelization. So if we have an OpenMP backend, we can again rely on OpenMP to solve this tough problem for us.

- As an open research topic, it is interesting to figure out the efficiency discrepancy between OpenMP solution and the PTHREAD solution. Since OpenMP has more versatile scheduling techniques (such as static, dynamic and guided) and it has already been proven a success in the industry, we are curious to know whether OpenMP solution can outperform PTHREAD solution in efficiency or not.

The remainder of this thesis is organized as follows: An introduction to SAC is given in Chapter 2. Thereafter, Chapter 3 illustrates the essential ideas of OpenMP and introduces the syntax and semantics of the OpenMP constructs relevant to the generation of OpenMP multithread backend. The design issues for OpenMP multithread strategy is presented in detail in Chapter 4. Afterwards, in Chapter 5, the runtime performance achieved is investigated. Chapter 6 presents the potential optimization opportunities and outlines the directions of future work. Finally Chapter 7 concludes the work.

Chapter 2. SAC

SAC is short for Single Assignment C which is a functional array processing language designed for computationally extensive applications. The aims of SAC are to achieve FORTRAN-rivaling runtime performance while at the same time offer a high level of abstraction. And being an array language, SAC supports true multidimensional arrays as first class citizens.

This chapter serves as the background knowledge of SAC. It only illustrates the knowledge of SAC which is relevant to the OpenMP parallelization strategy. Section 2.1 explains a functional subset of C that forms the SAC kernel. In section 2.2 the array subsystem is presented. And since all aggregate array operations in SAC are defined in terms of WITH-loops, Section 2.3 provides detailed description of this versatile construct. Finally, brief introductions to the PTHREAD parallelization strategy, the memory management system of SAC and the compilation steps of SAC are given in Section 2.4, Section 2.5 and Section 2.6, respectively.

2.1 A functional subset of C

As illustrated by its name, the syntax of SAC has great similarity to ANSI-C **[8]**. In order to allow for purely functional interpretation, SAC eliminates all elements of C which can cause side-effects, such as pointers and global variables. Besides this, only the control flow instructions break, goto and continue must be left out. The semantics of the remaining language constructs is then given by a straightforward mapping to an applied Lambda-calculus **[2]**. And the meaning of programs is given by this mapping and by context-free substitutions, as defined by the applied Lambda-calculus. The detailed syntax description of SAC can be found on the SAC webpage **[9]**.

Besides the similarity to C, SAC also provides some renovations, such as support for functions with multiple return values.

2.2 Arrays in SAC

In SAC, every variable is regarded as an array which is represented by two vectors: a data vector containing all array elements in row major order and a shape vector specifying its structures. But the data and shape vectors could be chosen with a restriction: Assume A is a n-dimensional array with a shape vector sv consisting of integer scalars sv = [sv_0, ..., sv_{n-1}] and a data vector dv consisting of scalars of arbitrary type dv = [dv_0, ..., dv_{l-1}], then the length l of the data vector must be

$$l = \prod_{j=0}^{n-1} sv_j$$

Figure 2.1 shows several examples of this array representation.

A0 = 1	Shape Vector: [ ]	Data Vector: [1]
A1 = $(1 \quad 2 \quad 3)$	Shape Vector: [3]	Data Vector: [1, 2, 3]
A2 = $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$	Shape Vector: [2, 3]	Data Vector: [1, 2, 3, 4, 5, 6]

Figure 2.1: **SAC array representation: the data vector and the shape vector.** In SAC, an array is represented by a shape vector and a data vector.

SAC provides compound homogeneous array operations which are applicable to all array elements or to the elements of coherent subarrays. All these operations are defined shape invariantly, i.e., they can be applied to arrays of arbitrary shape and thus to arrays of arbitrary dimensionality **[3].**

For the simplicity of manipulations on the arrays, SAC also provides some primitive functions to retrieve the array's shape and content. For instance, let A denote an array, **dim(A)** retrieves the dimensionality of array A and **shape(A)** yields the shape vector of

array A. Additionally the more powerful construct to manipulate the arrays in SAC is WITH loop construct, which is the topic of Section 2.3.

2.3 The WITH loop

The WITH-loop is a versatile construct in SAC. It can be used to aggregate operations on arrays of given shape along with a specification of how to initialize each element depending on its index position. And the WITH-loop can also be used to define a reduction operation over a range of elements in the WITH-loop together with a specification of how to compute the set of fold operands. One of the biggest advantages of the WITH-loop is that it can define the specification of arbitrary, truly shape invariant and even rank invariant array operations. The syntax of the WITH-loop is outlined in Figure 2.2.

Essentially, a WITH-loop consists of an operation part and a generator part. The **generator** part defines the lower and upper bounds for a set of index vectors and an index variable, which represents an element of this set. The index variable makes it possible to reference a single element in the WITH-loop operation.

And the **operator** part is to define an operation to be applied to the elements of the index vector set. At present, there are three different kinds of operation parts whose functionalities are defined as follows. Let **shp** denotes an expression that evaluates to an integer vector; **expr** denotes the expression directly after the **generator** part; and let **fold_op** be the name of a binary commutative and associative function with neutral element **neutral**.

- **genarray(shp)** generates an array with the shape **shp**. All elements whose index positions are covered by the **generator** will get the value **expr**. Other elements get the default value 0.

- **modarray(shp)** defines an array which has the same shape as **shp**. All elements whose index positions are covered by the **generator** will get the value **expr**. Other elements get the value of the corresponding element in **shp**.

- **fold(fold_op, neutral)** specifies a reduction operations, setting out with the **neutral**, the value is computed for each position from the specified set **expr** and subsequently folded using **fold_op**.

```
Expr          => …
                 | WithExpr

WithExpr => with  {  ( Generator ) : Expr ;  [Assign]*  }:
                    Operation;

Generator   => Expr Relop Id Relop Expr

Relop         =>  < | <=

Operation    =>   genarray ( Expr )
                  |     modarray ( Expr )
                  |     fold ( Foldfun , Expr )

Foldfun       =>    Id | Foldprf

Foldprf       =>     + | * | && | || | max | min
```

Figure 2.2: **Syntax of WITH-loops.** The WITH-loop has two parts: the operation part and the generator part.

Figure 2.3 is an example which shows the use of these three different kinds of WITH-loops in SAC.

In Figure 2.3, the genarray WITH-loop will generate the array A which is a 6 × 6 array. All the elements from the position [1, 1] to position [4, 4] will be initialized to the value 3. And the other elements will have the default value 0.

```
int main()
{
    A =  with {
      ([1,1] <=iv<= [4,4]) : 3;
    }: genarray( [6,6]);
    print(A);

    B =  with {
      ([1,1] <=iv<= [2,2]) : 13;
    }: modarray(A);
    print(B);

    C = with { ([0,0] <= iv <= [2,2]): B[iv];} : fold(+,0);
    print(C);


    return (0);
}
```

Figure 2.3: **Example of the three kinds of WITH-loops in SAC.**

$$A = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 3 & 3 & 3 & 0 \\ 0 & 3 & 3 & 3 & 3 & 0 \\ 0 & 3 & 3 & 3 & 3 & 0 \\ 0 & 3 & 3 & 3 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

And the modarray WITH-loop will generate the array B, which has the same shape as the array A. Each element in B has the same values as the element of the same position in A; except the elements from position [1, 1] to positions [2, 2] which are set to 13.

$$B = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 13 & 13 & 3 & 3 & 0 \\ 0 & 13 & 13 & 3 & 3 & 0 \\ 0 & 3 & 3 & 3 & 3 & 0 \\ 0 & 3 & 3 & 3 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

And for the array C, the fold WITH-loop computes the sum of all elements from position [0, 0] to position [2, 2] in array B.

$$C = 52$$

2.4 Current multithreaded model in SAC

The current parallelization strategy in the framework of SAC is to generate multi-threaded target code based on POSIX-THREADS **[22]**, which provides operations to dynamically create new threads and to synchronize them upon termination.

At present, there are two models of multi-threaded execution based on PTHREAD. The first one is to execute each WITH-loop in parallel separately using a fork/join pattern, and an enhanced fork/join model is also provided in this model **[4, 26]**. The second one is a hybrid execution model that combines elements of traditional fork-join and single-program-multiple-data approaches **[5]**.

But the second model is currently not fully implemented yet. In Chapter 5, the performance of the first model is used to compare with the performance of OpenMP parallelization strategy.

2.5 Memory management in SAC

SAC also provides implicit memory management in the form of reference counting **[6]**. With this advanced feature, the programmers are not bothered with the details of memory allocation and de-allocation, but can concentrate on the implementation of the program.

Reference counting is used to keep track of the number of conceptual copies of the data. In SAC, every data is regarded as an array, thus each array has a reference counter. And when an array is passed as a parameter to the functions, its reference counter is incremented and decremented implicitly. If its reference counter drops to zero, it indicates that this array is no longer needed, and its memory will be automatically reclaimed by the memory management system.

In the implementation of SAC memory management system **[27]**, there is an important data structure named descriptor which is used to store all relevant information of the array, such as reference counter, dimension and shape vector.

And in order to support the multithreaded execution, memory management in SAC also provides the heap manager in the multithread environment **[28]**. To the best convenience of the programmers, SAC memory management system will automatically generate the different heap manager facility for single thread environment and multithread environment.

2.6 Compilation steps of SAC

Compiling high-level abstract SAC programs into efficiently executable low-level code requires complicated analysis techniques and various transformation steps. In the whole process of the compilation, numerous intermediate representations are introduced to accommodate knowledge about program properties. This section sketches only the major compilation steps which are relevant to OpenMP multithread solution. Figure 2.4 presents the major compilation steps.

- Scanner / Parser: Generates the syntax tree whose structure reflects the original program.

- Code Simplification: Reduces both the variety as well as the complexity of the language constructs to simplify the compilation in the subsequent phases. In this phase for-loops, do-while loops, and while-loops are transformed into equivalent tail-end recursive functions. And also it is in this phase that the code is transformed into the representation of Single Assignment Form.

- Type Inference: Extracts the type information of the variables as concrete as possible because specific knowledge of the array is very essential in generating the efficient code. For instance, if an array whose dimension is 0 is known at

compile time, the memory management will store this array as a variable in the stack instead of allocating and deallocating memory for it on the heap. The type information includes the dimensionality of the array and also the exact shape of the array such as int [10, 10].

- High-level Optimization: Various optimization techniques are provided to optimize the code. For instance, function inline, loop invariant removal, common sub-expression elimination, dead code removal and loop unrolling.

- Automatic Parallelization: Automatic generates the PTHREAD strategy parallelization code. In order to reuse the common multithreaded facilities which are currently only used in PTHREAD parallelization strategy as much as possible, two sub phases in this phase are reused by OpenMP parallelization solution. The first one is the **cost model** which is used to determine whether the WITH-loop is worthwhile to be executed in parallel or not based on the size of the iteration space of the WITH-loop. The second one is the **create ST/MT function** which is used to tag the function either as ST (single thread) if the function is called in a sequential context or as MT (multiple thread) if the function is called in a parallel context.

- Precompilation: Converts tail-end recursive functions back into loops, which is crucial for the runtime performance of the compiled code.

- Code Generation: Generates the C code as the intermediate code.

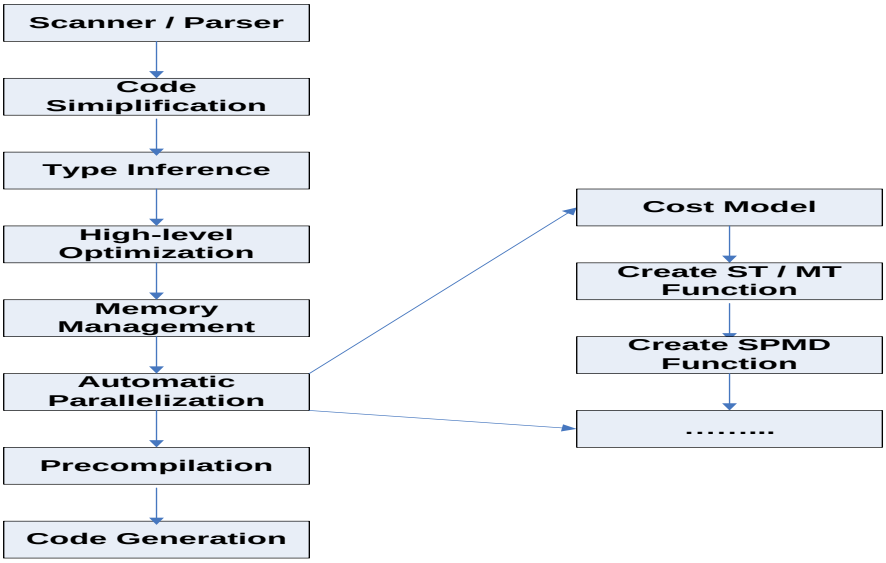


Figure 2.4: **Major phases of the compilation process of SAC program.**

Chapter 3. OpenMP

OpenMP **[14]** is a shared-memory application programming interface (API) which is used to write parallel program in FORTRAN and C/C++. Since it was developed in 1990s, OpenMP has become a de facto standard in the industry and is supported by various architectures and compilers.

This chapter introduces the background knowledge of OpenMP. And it only presents the knowledge of OpenMP constructs which are used in the OpenMP parallelization strategy. Section 3.1 explains the background which OpenMP was developed. In Section 3.2 the data parallelization of OpenMP is introduced. Section 3.3 introduces briefly the task parallelization of OpenMP.

3.1 Background of OpenMP

Since the first day of the appearance of the computer, our appetite for the more powerful hardware has always been insatiable. With the advent of the new generation hardware which comprises the faster CPU and more memory storage spaces, the new application software will always quickly exhaust the enhanced hardware resources and in turn will incur the requirement of a much more powerful computer. In order to satisfy the growing eagerness of the software, many researches have been done in the field of computer architecture to design a faster computer.

Among them, superscalar architecture **[11]** has been successfully designed and deployed in the industry. The principle of the superscalar architecture is that it is made up of multiple functional units with different and specific purposes which can operate simultaneously. For instance, in the superscalar architecture, there is one component especially for adding two integer numbers, one component especially for determining whether a value is greater than zero or not, and one component especially for fetching a data from the memory. This low level of parallelism is often referred to as “instruction-level parallelism”.

Through some optimizations in the compiler **[12]** to reorder the instructions in one application to best utilize the superscalar architecture and keep all the components in the system busy as much as possible, a certain degree of speed up can be achieved in the superscalar architecture. But unfortunately some studies **[13]** showed that typical applications are not likely to contain more than three or four different instructions that can be fed to the computer at a time in the superscalar architecture.

Then researchers have come up with an alternative strategy. From the 1980s, the shared memory architecture has been developed and dominated the market. In the remainder of this paper, we will refer to shared-memory parallel computers as SMPs. The idea of SMPs is that multiple processors which share the same memory space are configured in a single machine and, increasingly, on a single chip. Thus several jobs could be dispatched to different processors and executed simultaneously. And in order to make all the computing power exploited by the applications in the SMPs, the appropriate support from the software to describe the concurrency must also be provided. And this is the background that OpenMP was created.

OpenMP is a shared-memory application programming interface (API) to specify shared-memory parallelization in FORTRAN and C/C++ programs **[14]**. It was jointly defined by a group of major computer hardware and software vendors. Because OpenMP is comparatively easy to use in writing parallel programs and it is strongly supported by nearly all the mainstream hardware and software vendors, it is a now de facto standard in the industry.

3.2 Data parallelism in OpenMP

3.2.1 Introduction

The principle of OpenMP is that OpenMP expects the application developer to give a high-level specification of the potential parallelization in the program and the method for exploiting that parallelization by means of inserting compiler directives in certain places and setting environment variables. And OpenMP runtime library routines sorts out the low-level details of actually creating independent threads to execute the code and to assign work to them according to the strategy specified by the programmer.

Since in the process of compiling SAC program, C is used as the intermediate language in order to achieve the portability on different architectures and reuse the current compiler technology, in the remainder of this chapter we will only focus on the syntax and semantics of C in OpenMP. For C programs, pragmas which are called

directives in OpenMP are provided by the OpenMP API to express the parallelism. These directives always start with **#pragma omp**, followed by a specific keyword that identifies the directive, with possibly one or more so-called clauses, each separated by a comma. These clauses are used to express the detailed information of the parallelism. The formal syntax description is described in Figure 3.1.

#pragma omp *directive-name [clause[[,] clause]. . .]*
new-line

Figure 3.1: **General form of an OpenMP directive for C program**. Directive name is a keyword that defines the behavior. For instance, **parallel** directive is used to define a region that is executed by multiple threads in parallel. Clause is used to express the detailed behavior.

One of the most powerful features in OpenMP is that programmers can write a parallel program, and at the same time preserve the original sequential version. This is the case because OpenMP multithreading version is triggered by using an OpenMP flag in the compiler. If the programmer does not compile using the OpenMP option or uses a compiler that does not support OpenMP, the OpenMP directives will be simply ignored, and sequential code will be generated.

3.2.2 The parallel construct

Before we dive into the details of other characteristics of OpenMP, we will first see the most essential directive: the **parallel** construct. The syntax of parallel is described in Figure 3.2:

#pragma omp parallel *[clause[[,] clause]. . .]*
structured block

Figure 3.2: **Syntax of parallel construct in C**. The **parallel** construct is used to specify that the computation inside the parallel region should be executed in parallel. The parallel region implicitly ends at the end of the structured block. In most cases, it is a closing curly brace (}).

The parallel construct has the characteristics of “fork-join” model, which is illustrated in Figure 3.3. In this programming model, the program will start up with a single thread which executes sequentially until it encounters the parallel construct. Then this single thread, which is named as “master thread” will create a group of threads (this is called “fork”) and collaborate with them to execute the code inside the structured parallel block in parallel. At the end of the parallel region, there is an implicit barrier which forces all threads to wait until the work inside the region has been completed. Afterwards only the master thread continues to execute sequentially while the rest of threads will be terminated (this is called “join”).

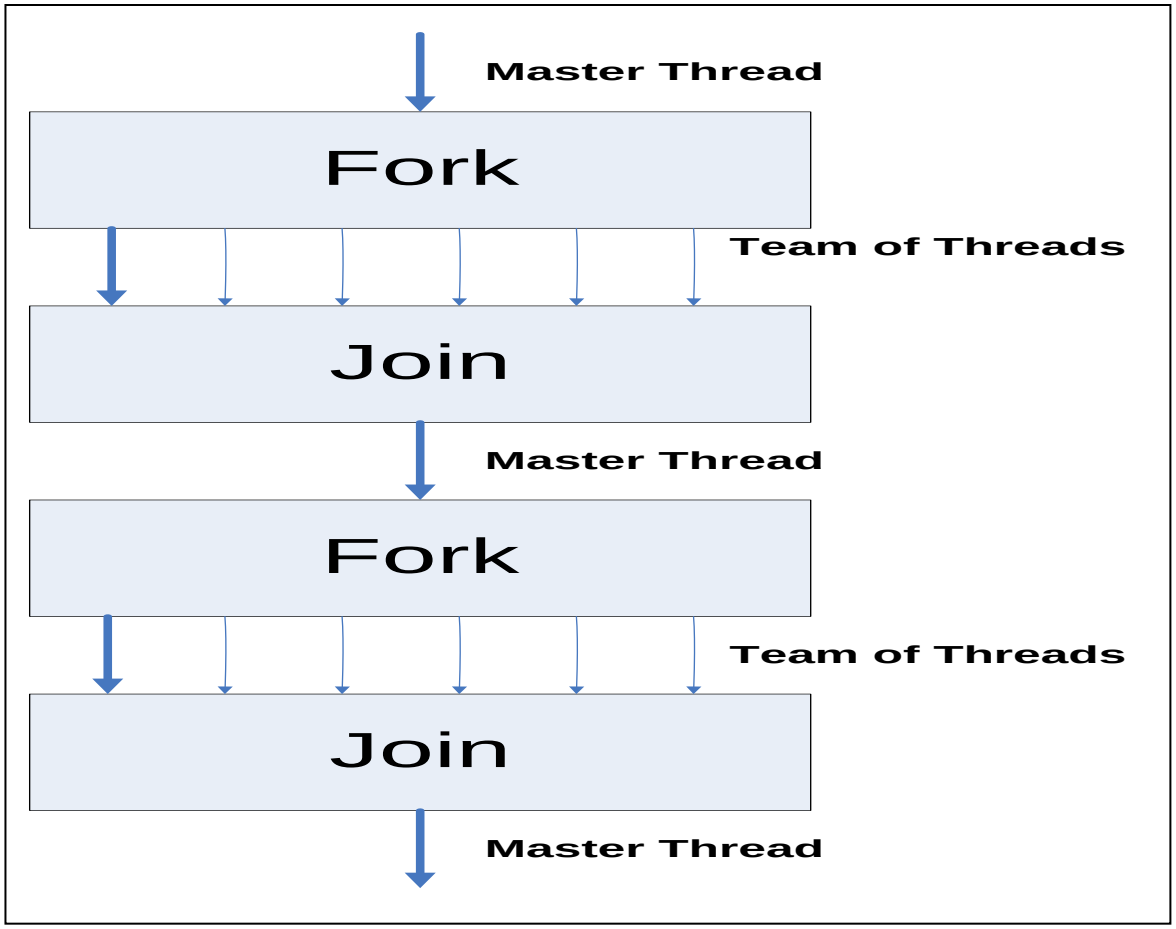


Figure 3.3: **The fork-join programming model supported by OpenMP.** The program starts as a single thread of execution, the master thread. A team of threads is forked at the beginning of a parallel region and joined at the end.

Although the **parallel** construct ensures that computations are performed in parallel, it does not distribute units of the work to the threads in a team. This means that the work inside the parallel construct will be replicated by all threads. Thus we need some

appropriate directives, such as the **for** construct, to dispatch the different work to different threads in the team.

3.2.3 The for construct

The responsibility of the **for** construct is to divide the iteration space of the for loop immediately following it. At run time, the loop iterations are distributed across the threads. Each iteration is assigned to a specific thread in the team and is executed by the same thread from the start to the end. The syntax of for construct is illustrated by Figure 3.4.

```
#pragma omp for [clause[[,] clause]. . . ]
                for-loop
```

Figure 3.4: **Syntax of for construct in C**. The **for** construct is to distribute the iteration space of the for loop into different threads.

In C programming language, there is a restriction in the **for** construct to be used in OpenMP. That is the number of the iterations of the for loop can be counted at runtime. Thus the loop must have an integer whose value is incremented by a fixed number at each step until some specified upper bound is reached. Thus the **for** construct in OpenMP can be only applied to the for loop with the format presented in Figure 3.5 .

```
for ( init ; var relop b ; increment )
{
}
```

Figure 3.5: **Format of C loop.** The OpenMP for construct is only applied to this kind of for loop in C. The **init** is the initialization of the loop counter variable via an integer expression, **var** is the loop counter variable, **b** is also an integer expression, and relop is one of the following: <, <=, >, >=. The **increment** is a statement that increments or decrements the loop counter variable **var** by an integer amount using a standard operator (++, – , +=, -=). Alternatively, it may take a form such as var = var + incr.

With the example in Figure 3.6 the use of the **for** construct in OpenMP can be illustrated clearly. As explained in Section 3.2.2, the **parallel** construct is to instruct the compiler to implicitly generate a group of threads and the **for** construct is to distribute the work of each iteration to different threads in the team according to a certain mechanism which will be illustrated in Section 3.2.5. The function **omp_get_thread_num**() is a OpenMP standard library function to get the ID of the thread. And we will postpone the explanation of clause “private” to Section 3.2.4.

```
#pragma omp parallel private(i)
{
    #pragma omp for
     for (i=0; i <9; i++)
     {
        printf("Thread %d executes loop iteration %d\n", omp_get_thread_num(), i);
     }
 }
```

Figure 3.6: **Example of a work-sharing loop.** Each thread executes a subset of the total iteration space i = 0, . . . , 9.

Figure 3.7 shows one possible output produced when we executed the code of Figure 3.6 using four threads. The output is non-deterministic and may change from run to run.

```
Thread 0 executes loop iteration 0
Thread 0 executes loop iteration 1
Thread 0 executes loop iteration 2
Thread 3 executes loop iteration 7
Thread 3 executes loop iteration 8
Thread 2 executes loop iteration 5
Thread 2 executes loop iteration 6
Thread 1 executes loop iteration 3
Thread 1 executes loop iteration 4
```

Figure 3.7: **Output from the example shown in Figure 3.6.** The example in Figure 3.6 is executed with four threads and the iteration space of the for loop is distributed to four threads according to a certain mechanism.

3.2.4 The private clause and the shared clause

In the previous example in Figure 3.6, we have seen the clause **private**. In fact, clause **shared** and **private** are the two most indispensable clauses in OpenMP to specify whether one variable is shared by all the threads in the team or each thread has a private copy of this variable during the process of the parallel execution.

The syntax of the **shared** clause is: **shared**(variable_list). All the variables in the list in the bracket are the variables shared among the threads during the parallel execution. Simply stated, there is one unique instance of the shared variable in the memory and each thread can freely read or modify its values. The example in figure 3.8 demonstrates the use of **shared** clause.

```
#pragma omp parallel for shared(b) private(i)
  for (i=0; i<n; i++)
  {
        b[i] += i;
  }
```

Figure 3.8: **Example of the shared and private clause.** All threads are able to read and write elements of **b** which is declared as a shared variable. And each thread has its own copy of **i** which is declared as a private variable. The modification of its private **i** by each thread is invisible to the other threads in the team.

And in the example of Figure 3.8, we can notice the loop iteration variable **i** is declared as **private**. The clause **private** means that each thread has its own copy of this variable and the modification made by one thread to its private variable will not be visible to the other threads. Similar to clause shared, the syntax for clause private is: **private**(variable_list).

And in OpenMP, there are also two more clauses to express the attribute of the variable. One is **firstprivate** and the other is **lastprivate**. Since neither clause is necessary in the OpenMP parallelization strategy for SAC, we will not introduce them here. Interested readers can find more information in **[14]**.

3.2.5 The schedule clause

The **schedule** clause in OpenMP is used to control the manner that the iteration spaces of the for loop is to be distributed over the threads. And the appropriate selection of the schedule technique can have a crucial impact on the performance of the program. One of the biggest advantages of OpenMP is that the programmers can experiment with these numerous scheduling techniques provided by OpenMP to win the utmost performance.

The syntax of schedule is: **schedule** (kind [,chunk_size]).

The **kind** specifies how the iterations of the loop are assigned to the threads in the team. And the **chunk_size** is the granularity of this workload distribution, which means a contiguous and nonempty subset of the iteration space. There are four schedule kinds supported now in schedule clause: static, dynamic, guided and runtime. We will not present the detailed descriptions of these scheduling techniques here. Interested readers can find in depth explanations in **[14]**.

3.2.6 The critical construct

As we point out in Section 3.2.4, each thread has access to the variable if it is declared as shared. This has an important implication that multiple threads might attempt to update the same memory location simultaneously or that one thread might try to read from a location on which another thread is updating at the same time. Thus OpenMP provides **critical** construct to ensure that multiple threads do not attempt to update the same shared data simultaneously **[14]**. The syntax of critical in C is given in Figure 3.9.

```
#pragma omp critical [(name)]
        structured block
```

Figure 3.9: **Syntax of the critical construct in C.** The structured block is executed by all threads, but only one at a time executes the block.

The principle of critical construct is that when one thread encounters a **critical** construct, it waits until no other thread is executing the critical region with the same name. In other words, it is impossible that multiple threads execute the code in the

critical region simultaneously. The example in Figure 3.10 shows the usage of the **critical** construct

In the example shown in Figure 3.10, if without the critical region, one thread may read the value of sum while the other thread is still updating it. This race condition will definitely lead to chaos and nondeterministic result from run to run.

```
   sum = 0;

#pragma omp parallel shared(n, b, sum) private(THREAD_ID, sumLocal)
{
   THREAD_ID = omp_get_thread_num();
   sumLocal = 0;

   #pragma omp for
        for (i=0; i<n; i++)
            sumLocal += b[i];

   #pragma omp critical (update_sum)
    {
      sum += sumLocal;
      printf("THREAD_ID=%d: sumLocal=%d sum = %d\n", THREAD_ID,   sumLocal,sum);
    }
}

   printf("Value of sum after parallel region: %d\n", sum);
```

Figure 3.10: **Example of critical construct.** The **critical** construct guarantees that one thread at a time enters the critical regions and avoids the race condition which will definitely lead to chaos and nondeterministic result.

3.2.7 The reduction clause

In the previous example, we have seen that with critical construct in OpenMP, the summation operation can be executed in parallel. But for some recurrence calculations which involve associative and commutative mathematical operations, OpenMP provides a much easier and more efficient clause to parallelize the operation. This operation is named **reduction**.

The syntax of the reduction clause in C is: **reduction**(operator :list). The programmers need to specify the type of the operation and the variable that stores the value after the calculation. And the compiler will implement the parallel execution in an efficient way, such as binary tree. The example in Figure 3.11 demonstrates how to use reduction clause to implement the example in Figure 3.10.

```
#pragma omp parallel for default(none) shared(n, a) reduction(+:sum)
{
        for (i=0; i<n; i++)
            sum += a[i];
}

printf("Value of sum after parallel region: %d\n", sum);
```

Figure 3.11**: Example of the reduction clause.** Using reduction clause is more efficient than using critical region.

The only disadvantage for reduction clause is that it only supports simple mathematic operations such as add, multiplication, and, or. So in some cases it is unavoidable that we have to use critical construct.

3.2.8 Active nest level

OpenMP 3.0 also provides an advanced feature named nested parallelism to improve the performance for some recursive algorithms. Simply speaking, nested parallelization means that if one thread in a team which is executing a parallel region encounters another **parallel** construct, it creates a new team of threads and becomes the master of that new team. Figure 3.12 is one example of an OpenMP program utilizing nested parallel feature.

And due to the fact that creating the parallel region incurs additional overhead, sometimes it is unknown that creating the nested parallel region is worthwhile or not. Thus OpenMP provides a flexible way to control the active nest level. The library function **omp_set_max_active_levels**() can specify the levels of the active parallel regions.

```
#pragma omp parallel num_threads(3)
{
    printf("Thread %d executes the outer parallel region\n",omp_get_thread_num());

    #pragma omp parallel num_threads(2)
    {
        printf(" Thread %d executes inner parallel region\n",omp_get_thread_num());
    } /*-- End of inner parallel region --*/
} /*-- End of outer parallel region --*/
```

Figure 3.12**: Example of the nested parallelism.** The **num_threads()** is used to configure the number of threads in the parallel region.

For instance, in the example above, if the active nested level is set to one, the inner parallel directive will be ignored by the compiler and the output is show in Figure 3.13; and if the active nested level is set to two, both parallel directives will take effect and the output is shown in Figure 3.14.

```
Thread 0 executes the outer parallel region
   Thread 0 executes inner parallel region
Thread 1 executes the outer parallel region
   Thread 0 executes inner parallel region
Thread 2 executes the outer parallel region
   Thread 0 executes inner parallel region
```

Figure 3.13: **Output of the nested parallelism program with active level set to one.** The inner parallel region is ignored by the compiler since there no more threads generated from the inner parallel region. But the **omp_get_thread_num()** still yields the thread ID in the current inner –most team. That is why there are three “Thread 0 executes inner parallel region” in the output.

```
Thread 0 executes the outer parallel region
Thread 1 executes the outer parallel region
Thread 2 executes the outer parallel region
   Thread 0 executes inner parallel region
   Thread 1 executes inner parallel region
   Thread 0 executes inner parallel region
   Thread 1 executes inner parallel region
   Thread 1 executes inner parallel region
   Thread 0 executes inner parallel region
```

Figure 3.14: **Output of the nested parallelism program with active level set to two.** The inner parallel region is active and for each thread in the outer parallel region, two more threads are generated in the inner parallel region.

3.3 Task parallelism in OpenMP

In 2008, OpenMP v3.0 was released. The biggest advance from OpenMP 2.5 to OpenMP 3.0 is that OpenMP 3.0 supports task parallelism. The reason that task parallelism is essential is that although OpenMP 2.5 is very successful in exploiting structured parallelism like for loops, it is very difficult to parallelize the unstructured applications such as while loop, link list traversal and tree traversal. In **[15]**, the authors have listed a few examples which are difficult and inefficient to parallelize using the OpenMP 2.5 construct such as critical, section and single.

The basic ideas of OpenMP task parallelism can be summarized as follows. In OpenMP 2.5, there are implicit tasks generated in the parallel region and each task is assigned to one thread. And in OpenMP 3.0, the explicit task is introduced. If one segment of code is encapsulated in the task directive region, this part of code will be used to initialize a task together with the data environment assigned by the OpenMP data attribute directive. Then OpenMP has its own implementation about when to schedule the task to a specific thread, whether the task can be suspended by one thread and later on resumed by another thread or not and when to synchronize the task. The advantage of task parallelism is that it can be helpful to the parallelization of irregular data structures.

Chapter 4. Implementation of OpenMP parallelization strategy

This chapter introduces the detailed implementation aspects of the OpenMP parallelization strategy. Section 4.1 introduces the compilation phases added or modified in order to generate the OpenMP parallelization strategy code. Since almost all array operations in SAC are transformed into WITH-loops which exhibit a large degree of fine grained concurrency, Section 4.2 to 4.4 focus on the compilation scheme of the genarray and modarray WITH-loop and also two variants of fold WITH-loop. But rather than giving abstract compilation schemes, they are illustrated by some tiny examples. Section 4.5 discusses how to incorporate the feature of OpenMP nesting parallel regions. And in Section 4.6, the false sharing problem is addressed and also the solution is presented. Finally the design issues of how to make OpenMP parallelization strategy and PTHREAD parallelization strategy co-exist concisely is described in Section 4.7.

4.1 Compilation steps

Unlike PTHREAD multithread implementation whose code differ substantially from its sequential code, one of the biggest advantages of OpenMP is that it is easy to maintain the sequential version and the multithread version of program simultaneously since the multithread version of code is quite similar to the sequential version, with only some OpenMP directives and pragmas inserted at appropriate places of sequential version of source code. This fact means that for the OpenMP multithread strategy, we only need to do a few changes to adapt to the code generation of sequential C code with OpenMP directives. These changes include the following four steps below:

Step 1: Add the necessary operation to get the operator of the fold WITH-loop in the early compilation phase.

As already discussed in Chapter 3, OpenMP **reduction** clause only supports limited simple mathematical operators such as add and multiplication. If the operator of the fold WITH-loop is one of these simple operators, OpenMP **reduction** clause can be used in the generation of the OpenMP code for fold WITH-loop without any problem. But for the operators in the fold-WITH loop, it could be more versatile constructs such as user-defined functions. In the context of SAC, these user-defined functions are often declared as inline functions and the function call will be replaced by the code inside the function body. Figure 4.1 illustrates this transformation. Please note that in the remainder of the thesis, the C code generated from SAC code in the examples is not exactly the same as the real C code generated from the compiler. The real C code generated is much more complicated, containing many redundant brackets and being a bit ugly in the layout of the program. Thus simplified C code is used in the example to make the explanations more easily and clearly.

Then in this situation, the **reduction** clause is difficult to use since the knowledge of the operation inside the body of the user-defined function has to be retrieved. Take the program in Figure 4.1 as an example, if we want to use the **reduction** clause in this case, we have to step into the body of the function add_add, analyze each operation inside and finally know that the operator is '+'. This is a bit difficult especially if there are more operations inside the user-defined function.

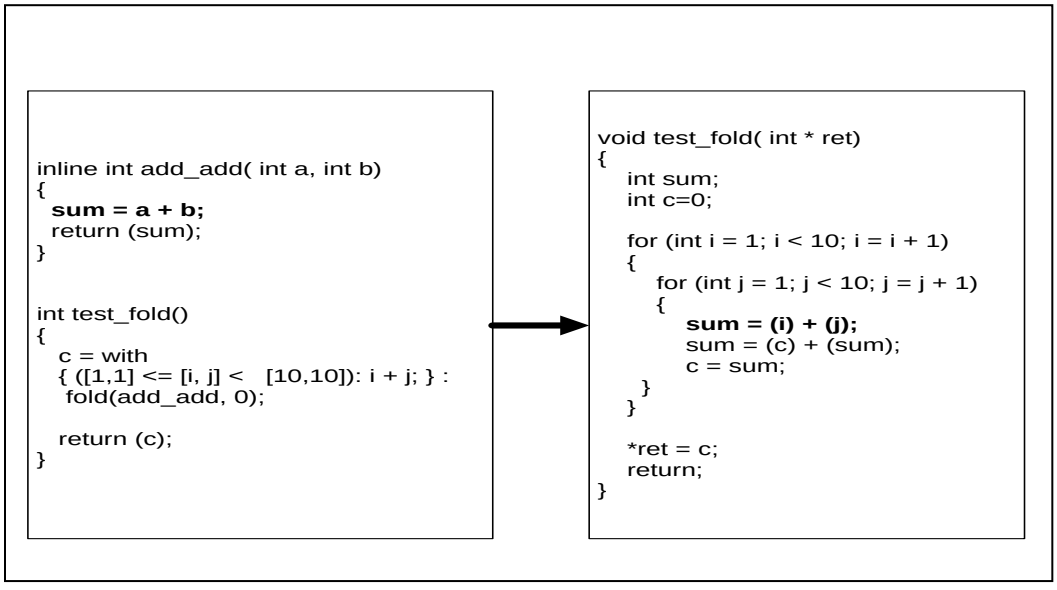


Figure 4.1: **Sequential code transformed from the fold WITH-loop with user-defined function.** The body of the user-defined function which is used as the operator of the WITH-loop will be inlined.

Thus OpenMP **critical** construct has to be used under these circumstances. In order to make a distinction between these two different compilation schemes in the code

generation phase, the information of the fold operators has to be inferred. This can be more easily done in the earlier compilation phase since in later phase even the simple operator such as '+' will be transformed into more complicated wrapper functions.

Step 2: Reuse two sub phases in the current automatic parallelization phase.

As already briefly discussed in Section 2.6, in the compilation process from SAC code into C code, there is a phase in which PTHREAD parallelization strategy is implemented. This phase is only triggered when the PTHREAD strategy is activated from the command line at the compile time of the program. The functionalities of the first sub phase and the second sub phase, however, are not only restricted to the PTHREAD implementation, but could be utilized in all multithread strategies as well. Thus it is a good idea to reuse these two sub phases to make the implementation of OpenMP strategy more concise.

In SAC program, the WITH-loop will consume most of the execution time, thus the parallelization strategy focuses on the WITH-loop. In the first sub phase, based on the cost model which checks whether the size of the iteration space of the WITH-loop exceeds the threshold or not, the compiler will categorize the WITH-loop into three different kinds: the ones which will be executed in parallel; the ones which will be executed sequentially; and the ones which may either run in parallel or sequentially because the size of the iteration space is unknown at compile time and the decision has to be postponed until the run time. Then OpenMP parallelization strategy could reuse this sub phase without any modification.

In the second sub phase, the compiler needs to decide the execution mode of each function. There are two different modes: **ST** mode and **MT** mode. ST mode means the function is called in the sequential context. But the function itself may contain part of code which could be executed in parallel. For instance, the function may has a parallelized WITH-loop. On the contrary, MT mode means the function is called in parallel context and thus does not aim at exploiting further concurrency. For example, the function which is called inside a parallelized WITH-loop is a MT mode function. Thus it is also obvious that OpenMP parallelization strategy could reuse this sub phase as well.

Step 3: Add one sub phase to find all OpenMP private variables in pre-compile phase.

The most important issue for OpenMP parallelization strategy is to determine which variable is the OpenMP private variable. Before inserting the OpenMP directives in the

code generation phase, we need a separate phase to find all the OpenMP private variables and store them somewhere so that this information could be retrieved in the later phase.

The first question is where to store this information. Since the information of other attributes of the variables is not needed in OpenMP, we only need to know the names of these variables in the code generation phase. Then we decide to add an “string” attribute into the syntax tree. This attribute is connected to the WITH-loop. And all OpenMP private variables inside the WITH-loop are then constructed into this single string.

The second question is which variable should be private. From the observation of the sequential version of C code generated, we find that there are two categories of variables that should be declared as OpenMP private. Figure 4.2 illustrates both categories.

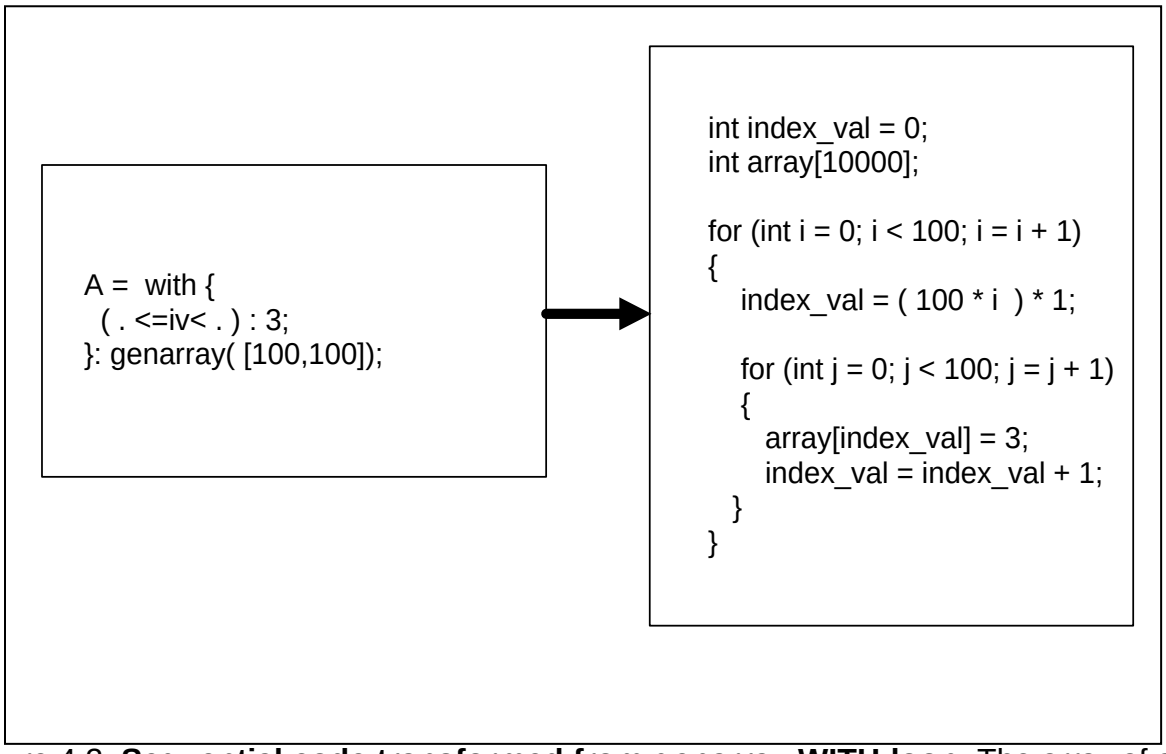


Figure 4.2: **Sequential code transformed from genarray WITH-loop.** The array of any dimension in SAC will be transformed to one dimension array in C.

The first category of variables is the variables which are updated inside WITH-loop body. For those variables which appear on the left hand side of the assignments, each thread could update this variable independently. Thus it is obvious that these variables

should be declared as private variables. For instance, the variable '**index_val**' in Figure 4.2 falls into this category of OpenMP private variables. Inferring this category of OpenMP private variables is trivial since it is easy to justify the assignment operation inside the WITH-loop and the variable on the left hand side of the assignment operation.

And from the introduction in Section 3.2.4, we know that the loop variables should be regarded as OpenMP private variables since each thread will have its own iteration space which will not overlap with each other. Thus the variable '**i**' and '**j**' in Figure 4.2 are OpenMP private variables. And during the transformation from the WITH-loop in SAC to the for loop in C, there is an internal data structure in the syntax tree which stores the names of all loop variables from the dimension 0 to the highest dimension. For instance the WITH-loop in Figure 4.2 has the internal data structure as follows.

iv = [**i, j**]

Thus if we do a simple traversal on this data structure, we will get all the loop variables which should be declared as OpenMP private. And after combining the both categories of OpenMP private variables, the OpenMP private list could be constructed now.

Step 4: Generate OpenMP code.

Before this phase, all the sufficient information has been inferred and the OpenMP solution code is generated in this phase. Considering the characteristics of OpenMP programming, the code resembles the sequential C code a lot. From Section 4.2 to 4.4, we will present the different detailed OpenMP code transformed under different situations.

4.2 Genarray and modarray

In this section, we will present the detailed C code transformed for both genarray and modarray together since the code generated from these two operations is almost the same. As we already introduce in the previous sections, the OpenMP multithread code only differs slightly from the sequential code, then the most important thing is to insert the OpenMP directives and clauses into the appropriate places of the sequential C code.

For the **parallel** directive which is used to inform the OpenMP compiler to generate multiple threads to execute the code in parallel, it will be inserted to encapsulate all C

for loops generated from the SAC WITH-loop which is worthwhile to be executed in parallel.

And since the compiler has already constructed the OpenMP private variable list in the previous compilation phase, now what we need to do is just put this OpenMP private variable list in the OpenMP **private** clause which is right after the **parallel** directive.

For the **for** directive which is used to control the manner to distribute the iteration space between the different threads, it is obvious that it has to be put right before each C for loop. And of course, the parameters for the OpenMP **for** directives, such as the scheduling technique and the chunk size can be retrieved from the global structure which is initialized explicitly at compile time. The default scheduling technique is **static**. For the chunk size, it will not be set in default and the OpenMP standard library will choose an appropriate number.

After all these work, the generated C code can be executed in parallel by the multiple threads generated automatically by OpenMP standard library according to the parallelization strategy described by OpenMP directives and clauses.

Figure 4.3 presents the OpenMP code transformed for the WITH-loop which is presented in Figure 4.2.

```
    int index_val = 0;
    int array[10000];

#pragma omp parallel private (i, j, index_val)
{
  #pragma omp for schedule (static)
    for (int i = 0; i < 100; i = i + 1)
    {
      index_val = ( 100 * i  ) * 1;

      for (int j = 0; j < 100; j = j + 1)
      {
        array[index_val] = 3;
        index_val = index_val + 1;
      }
    }
}
```

Figure 4.3: **OpenMP code transformed from the WITH-loop in Figure 4.2.** The code in bold is the OpenMP directives and clauses inserted and the rest code is the code generated from sequential code.

4.3 Fold WITH-loop with simple operator

In this section, we will present the detailed OpenMP code transformed for fold WITH-loop which has the simple mathematical operator supported by OpenMP **reduction** clause as the fold operator. Like the transformation for genarray WITH-loop, the transformation for fold WITH-loop with simple operator should also use the **parallel** construct to encapsulate all C for loops generated, insert OpenMP **for** directive before each for loop and use the **private** clause. The only difference is that for fold WITH-loop with simple operator, we have to find which variable should be regarded as the reduction variable.

Figure 4.4 illustrates the sequential code transformed for the fold WITH-loop that uses the simple mathematical operator '+' which is supported by OpenMP reduction clause.

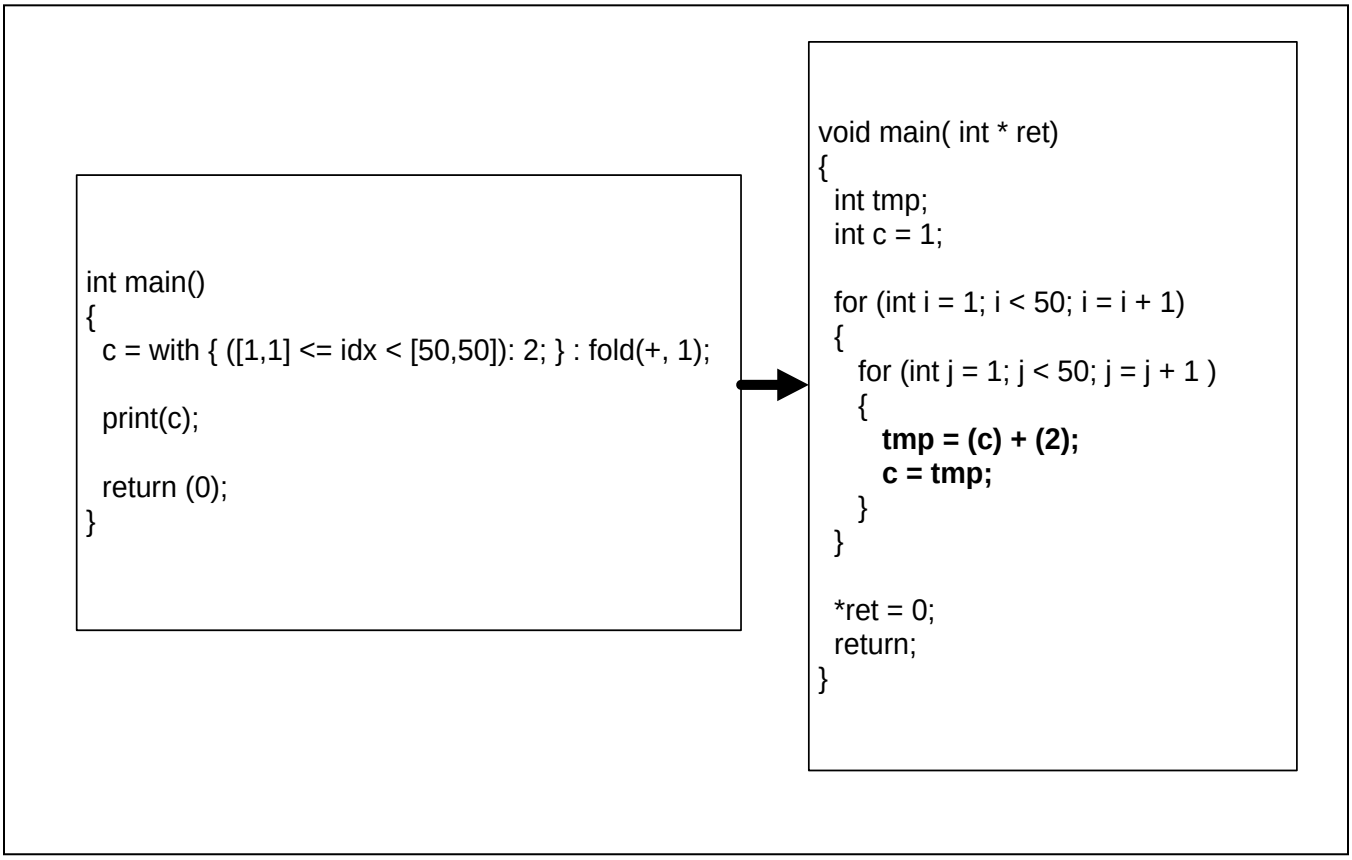


Figure 4.4: **The sequential code transformed from the fold WITH-loop with simple operator.** '+' is a mathematically associative and commutative operator and is supported by OpenMP **reduction** clause.

From the Figure 4.4, we know that the variable **c** holds the result of recurrence calculations. Thus it is obvious that it should be the OpenMP reduction variable. But here comes the question that where is **c** stored in the syntax tree.

In the intermediate representation of the SAC code, there is an internal representation of **accu** function. And in the code block there is an explicit reduction operation called. The pseudo C code of the intermediate representation is illustrated in Figure 4.5. And we can find easily that the variable which holds the return value of **accu**() function is the OpenMP reduction variable. Then after copying the name of this variable into the () bracket of the **reduction** clause, the transformation for fold WITH-loop with simple operators is complete. And last but not least, the variable '**c**' should be removed from the OpenMP private variable list even if '**c**' is updated inside the for loop since '**c**' is already a reduction variable. Figure 4.6 illustrates the complete OpenMP code transformed from the example in Figure 4.4.

```
c = accu();
tmp = i+ j
tmp = c +  tmp;
c = tmp;
```

Figure 4.5: **Internal representation of accu() function.** The variable which holds the return value of accu() function is OpenMP reduction variable.

```
void main( int * ret)
{
  int tmp; int c = 1;
#pragma omp parallel reduction( + : c) private (i, j)
{
   #pragma omp for schedule (static)
    for (int i = 1; i < 50; i = i + 1)
    {
      for (int j = 1; j < 50; j = j + 1 )
      {
        tmp = (c) + (2);
        c = tmp;
      }
    }
}
   *ret = 0; return;
}
```

Figure 4.6: **OpenMP code transformed for the example in Figure 4.6.**

4.4 Fold WITH-loop with user-defined function

Figure 4.7 illustrates a bit more complicated fold WITH-loop that uses a user-defined function as the fold operator. As already explained in Section 4.1, it should be transformed into C code with OpenMP **critical** construct. But the question is what code needs to be encapsulated within the **critical** region.

```
inline int add_add( int a, int b)
{
   sum = a + b - 2;
   return (sum);
}

int main()
{
   c = with { ([1,1] <= [i, j] < [500,500]): i + j; } : fold(add_add, 0);
   return (0);
}
```

Figure 4.7: **The fold WITH-loop with a bit more complicated user-defined function.** In SAC, the user-defined function in fold operator will be inlined.

The most direct way is to encapsulate all the code inside the function body within the critical region. This will of course yields very bad performance since all the operations inside the function body could not be executed in parallel. But in fact only the attempt to update the shared data should be protected.

Thus instructed by what we know from Section 3.2.6, we should find the variable which holds the final value of the fold WITH-loop. Then find the first assignment whose right hand side has this variable, before this read operation should be the place that OpenMP critical region starts. And finally find the assignment whose left hand side has this variable, after this write operation is the place that OpenMP critical region ends.

Figure 4.8 show the complete OpenMP code transformed for the example in Figure 4.7. The variable '*c*' holds the final value and the OpenMP **critical** region starts before the first read of '*c*' and end after the assignment of '*c*'. Thus the operation '*var = i + j*' could still be executed in parallel.

```
void main(int *ret)
{
   int c = 0;

   #pragma omp parallel private ( i,  j, sum )
   {
     #pragma omp for schedule (static)
      for (int i = 1; i < 500; i = i + 1)
      {
          for (int j = 1; j < 500; j = j + 1)
          {
            sum = i + j;

            #pragma omp critical
            {
              sum = c + sum;
              sum= sum + (-2);
              c  = sum;
            }
         }
      }
   }

    *ret = c;
    return;
}
```

Figure 4.8: **The OpenMP code transformed for the example in Figure 4.7.** The code in bold is the inserted OpenMP constructs.

4.5 Using OpenMP nested parallelism

One of the advanced features provided by OpenMP is the support of nested parallelism which means if a thread in a team executing a parallel region encounters another parallel construct; it creates a new team of threads and becomes the master of that new team. The nested parallelism could achieve good performance especially for recursive algorithms. Thus in OpenMP parallelization strategy, nested parallelism is supported.

As discussed in Section 3.2.8, OpenMP has a way to configure the number of the active parallel regions. This parameter can be set on the command line when the SAC program is compiled and the default number is set to one. And if this parameter is larger than one, say ***N***, and the dimension of the WITH-loop is ***M***, the number of ***min(N, M)*** OpenMP parallel regions will be generated.

But here comes the question whether the OpenMP private variable list of one parallel region is different from the private variable list of another parallel region or not. As it is illustrated in Section 4.1, the OpenMP private variables fall into two categories. The first category is the variable which is updated inside the WITH-loop; and the second one is the loop variable of the C for loop.

The current transformation for sequential C code has the characteristics which makes it convenient for the code generation of OpenMP parallelization strategy. Let assume the dimension of the WITH-loop is **D**. On the first hand, before the **(D - 1)** th C for loop, there is an assignment operation which will calculate the index of the array. On the second hand, all other operations will be encapsulated into the innermost C for loop. Figure 4.9 illustrates this transformation.

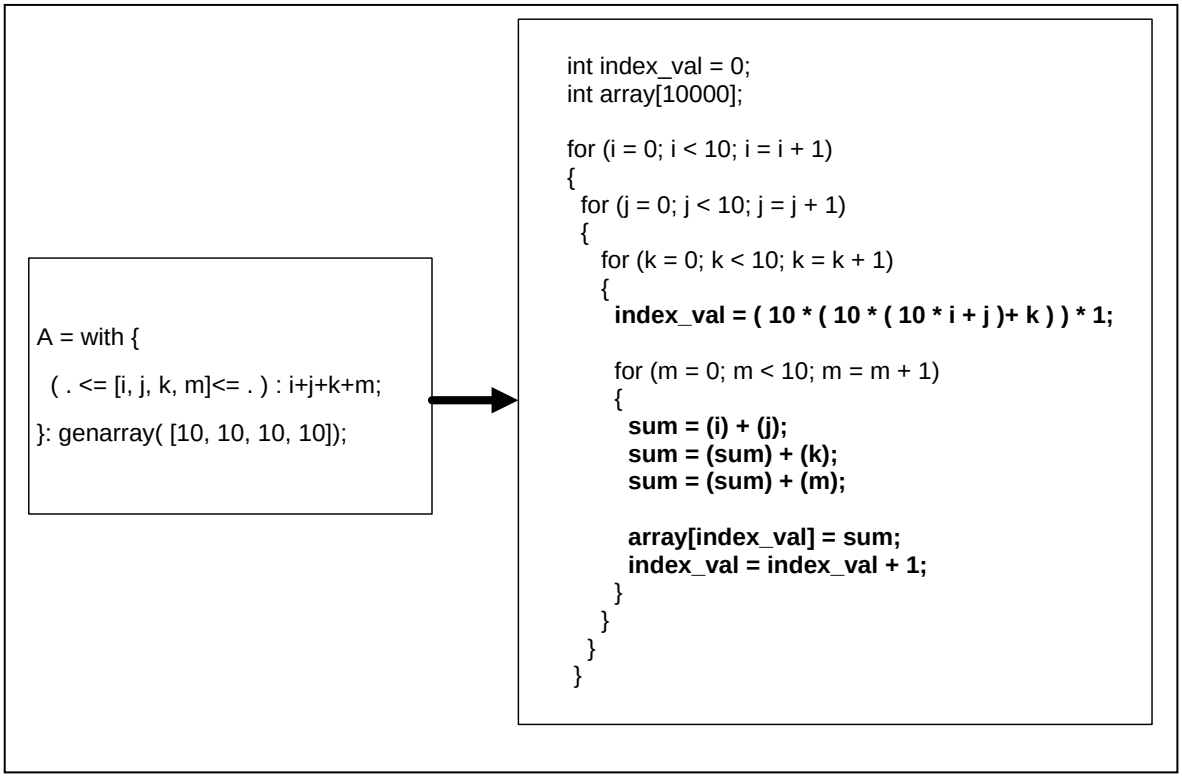


Figure 4.9: **Transformation of sequential C code.** An assignment operation is inserted before the innermost for loop and all other operations are encapsulated into the innermost loop.

Thus if we enable the number of active nested parallel regions to **(D - 1)**, which in most cases will be sufficient to improve the efficiency, all OpenMP parallel regions will share the same private variables in the first category.

```
    int index_val = 0;
    int array[10000];

#pragma omp parallel private ( index_val, sum, i, j, k, m )
{
    #pragma omp for schedule (static)
     for (i = 0; i < 10; i = i + 1)
     {
    #pragma omp parallel private ( index_val, sum, j, k, m )
    {
      #pragma omp for schedule (static)
       for (j = 0; j < 10; j = j + 1)
       {
         for (k = 0; k < 10; k = k + 1)
         {
           index_val = ( 10 * ( 10 * ( 10 * i + j )+ k ) ) * 1;

           for (m = 0; m < 10; m = m + 1)
           {
             sum = (i) + (j);
             sum = (sum) + (k);
             sum = (sum) + (m);

             array[index_val] = sum;
             index_val = index_val + 1;
           }
         }
       }
     }
  }
}
```

Figure 4.10: **The OpenMP code transformed from the WITH-loop in Figure 4.9 if the active parallel region is configured to 2**. The OpenMP private list for the inner parallel region is the subset of the one for the outer parallel region.

But for the C loop variables, it is obvious that the private variables of the inner parallel region should be the subset of the ones of the outer parallel region. The principle is as the follows: For the outermost C for loop, all loop variables should be included in the OpenMP private list of its parallel region; for the second outermost for loop, the loop variable from the outermost for loop should be eliminated from the private list of the second parallel region, etc.

Thus for the WITH-loop presented in Figure 4.9, the maximum active nested parallel region is limited to 3 and the OpenMP code transformed is illustrated in Figure 4.10, if the active OpenMP parallel region is two.

4.6 Index vector problem

As explained in the previous section, index vector is an array to store the offset in each dimension during the traversal of the array. After some optimization cycles in SAC, index vector could be removed. But there is one scenario in which index vector could not be removed: index vector used as a parameter of a function which is inside the WITH-loop. Under this situation, index vector is visited by a pointer. Therefor even if each thread generated by OpenMP standard library has a private copy of the pointer, every thread will still visit the same memory location, and concurrent update of the same memory location by different threads will lead to chaos during the execution. Figure 4.11presents the scenario in which the index vector could not be removed.

```
A = with  {
      ( . <= iv < . ) : Foo(iv);
  }: modarray(A);
```

Figure 4.11*:* **Index vector could not be removed.** The function **Foo** uses the index vector as the parameter, thus the index vector could not be removed.

The similar problem also exists for the descriptor of the index vector, which is already explained in Section 2.5.

One reasonable solution is to allocate additional memory in a continuous memory space for each thread to store its own private index vector. Usually the length of the index vector is very small. For instance, the index vector of three dimension WITH-loop is 12 bytes. But the length of the cache line is much larger than the length of the index

vector. Thus it is quite likely that the index vectors for all threads which are allocated in a contiguous memory space happen to reside in the same cache line. This will lead to the false sharing problem **[24]** which will have extremely negative impact on the performance.

In SMP architecture, the memory system guarantees cache coherence. And false sharing is the case in which simultaneous updates of individual elements in the same cache line coming from different processors invalidates entire cache lines, even though these updates are logically independent from each other. When one processor updates an individual element in a cache line, that cache line will be marked as invalid. Afterwards when the other processors access a different element in the same line marked as invalid, they are forced to fetch a fresh copy of the cache line from memory, even though the element they access has not been modified. Figure 4.12 illustrates the false sharing problem.

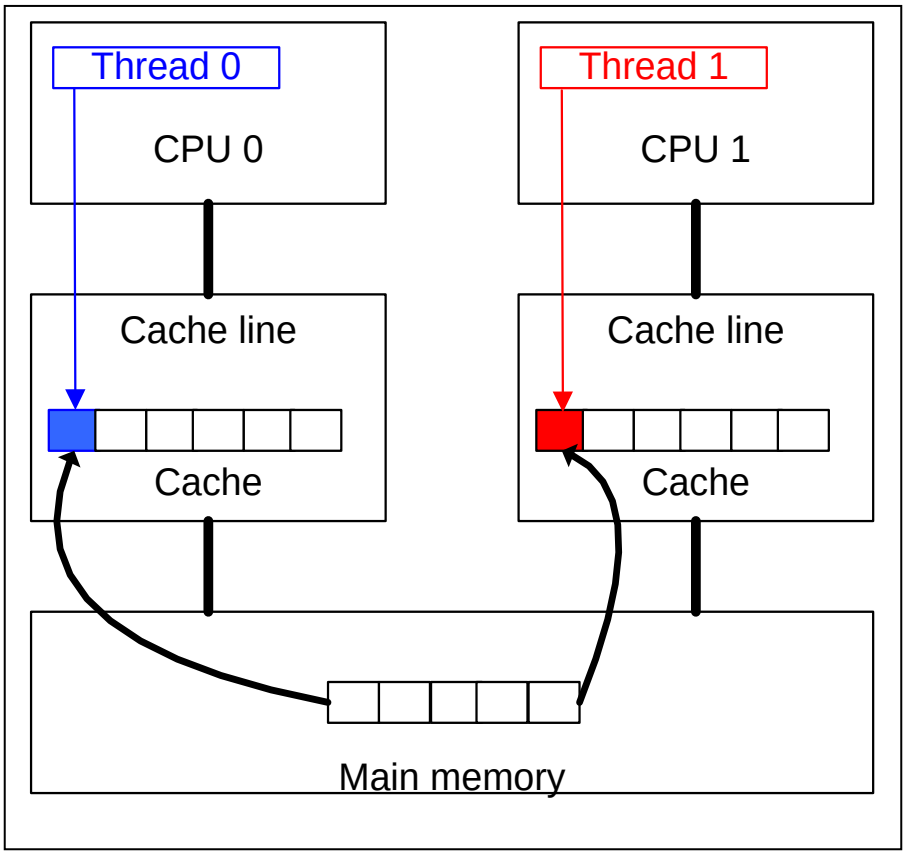


Figure 4.12: **Identification of the false sharing problem.** False sharing occurs when threads on different processors modify different elements that reside on the same cache line. This invalidates the whole cache line and forces a memory update to maintain cache coherency.

In order to improve the performance of OpenMP solution, some smart techniques must be implemented to avoid the false sharing problem. One of the most direct solution, and also the solution implemented in this thesis is to allocate a large chunk of memory for each thread to hold the values of the index vector for each thread. The memory of private index vector is large enough to avoid the situation that they reside in the same cache line. So the false sharing is avoided and whenever one thread updates its own index vector, it will not invalidate the cache line which holds the index vector for other threads. The same solution applies to the index vector descriptor, the data structure in SAC implementation which we already discuss in Section 2.5. Figure 4.13 illustrates this technique.

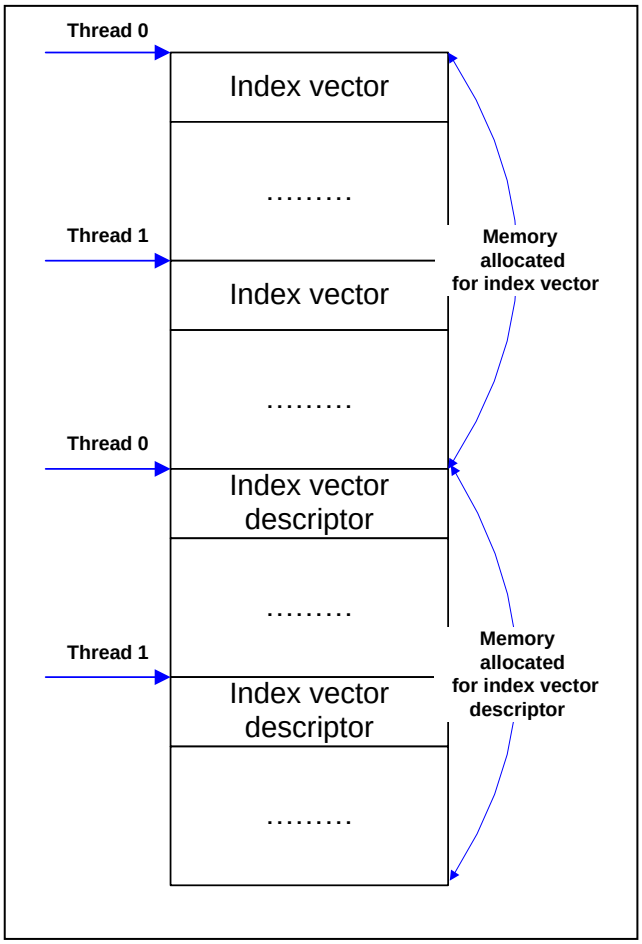


Figure 4.13: **Solution for the false sharing problem.** The memory allocated for the index vector for each thread is large enough to avoid the situation that different index vector for different threads reside in the same cache line.

Based on this technique, it is time now to present the detailed code generated to solve the false sharing problem. The first step is to show the necessary operations

needing to be done before the OpenMP parallel region. Let us assume that ***N*** is the number of threads; ***Len_cache*** is the length of a cache line of a certain architecture; ***Len_iv*** is the length of the index vector and ***Len_iv_desc*** is the length of the index vector descriptor. Figure 4.14 illustrates these operations.

```
int  thread_len_iv = (Len_iv / Len_cache + 1 + 1 ) * Len_cache;
int  thread_len_iv_desc = (Len_iv_desc / Len_cache + 1 + 1 ) * Len_cache;

int* iv_ptr = (int*)malloc( N * thread_len_iv );
int* iv_desc_ptr = (int*)malloc( N * thread_len_iv_desc );

for (int i = 0; i < N; i++)
{
      *(iv_desc_ptr + thread_len_iv_desc * i) = 1;
}

int *thread_prv_iv;
int *thread_prv_iv_desc;
```

Figure 4.14: **Allocation of a continuous memory to hold the index vector and its descriptor for each thread.** This code should be inserted before the C for loop generated from WITH-loop.

In Figure 4.14, the value of **thread_len_iv** is the size of the memory allocated for each thread to hold its private index vector. And the value of **thread_len_iv_desc** is the size of the memory allocated for each thread to hold its private index vector descriptor. In most cases, the value of ***Len_iv*** / ***Len_cache*** is 0 since the length of the cache line is much longer than the length of the index vector. Thus it is not difficult to understand the first '**+ 1**' operation. But why do we need the second '**+ 1**' operation?

Let us think about the situation presented in Figure 4.15. If the memory allocated for one thread to hold the index vector is one cache line, the operation by this thread is still possible to invalidate the adjacent cache line which holds the index vector for another thread. So that is why the second '**+ 1**' operation is needed here.

The value of **iv_ptr** and **iv_desc_ptr** is the start address of the memory space allocated for all threads' index vector and their descriptor respectively. Inside the OpenMP parallel region, each thread will first calculate its own offset to these two addresses so that they can access to their own private memory space. And the operation carried in the for loop is to initialize the reference counter of the index vector to one since it is the first time each private index vector is referenced.

And the variable **thread_prv_iv** and **thread_prv_iv_desc** is used by each thread to point to its private index vector and descriptor.

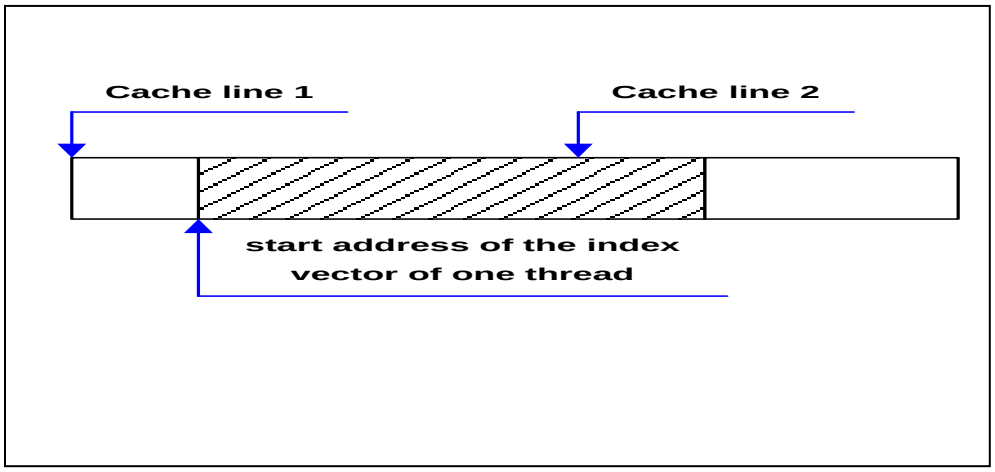


Figure 4.15: **Why the second '+ 1' is needed.** Without '+ 1', the false sharing problem still exists.

Then the second step is to explain the operations inside the parallel region. Figure 4.16 illustrates this work. The OpenMP private variables should include not only the loop variables but also the thread ID, the pointer to the index vector and also the pointer to the index vector descriptor. Inside the parallel region, every thread has to know its own ID in order to calculate the address of its private index vector and its descriptor. This is done by calling OpenMP library function **omp_get_thread_num()**. Then in the innermost for loop, each thread needs to write its private loop variable into its private memory space.

```
#pragma omp parallel private (iv0, iv1, thread_prv_iv, thread_prv_iv_desc,
thread_id, ... )
{
        int  thread_id = omp_get_thread_num();
        thread_prv_iv = iv_ptr + thread_id * thread_len_iv;
        thread_prv_iv_desc = iv_desc_ptr + thread_id * thread_len_iv_desc;

       #pragma omp for schedule (static)
        for (iv0 = 0; ; iv0 = iv0 + 1) {
              for (iv1 = 0; ; iv1 = iv1 + 1)  {
                          thread_prv_iv[0] = iv0;
                          thread_prv_iv[1] = iv1;
                           thread_prv_iv_desc[0] += 1;

                          /*
                          *  other C code generated from the
                          *  SAC code inside WITH-loop
                          */
              }
       }
}
```

Figure 4.16: **Inserted C code inside the parallel region.** Each thread will calculate the start address of its private memory space allocated.

4.7 The OpenMP version of SAC runtime library

One of the tough design issues in OpenMP parallelization strategy is how to make OpenMP parallelization strategy coexist with PTHREAD parallelization strategy more concisely and conform to the principle of software engineering. For the PTHREAD strategy, a run time library will be generated to do the job such as creating a Pthread. Similarly another run time library should be generated for OpenMP parallelization strategy.

But in PTHREAD strategy, there exist the implementations especially for PTHREAD strategy, such as using pthread API **pthread_create()** to create a new thread; and at the same time there also exist some data structures and environment variables that

should be regarded as the facilities for all multithread execution solutions instead of exclusively for PTHREAD multithread strategy.

For instance, when the program starts up, the number of threads will be configured to a global variable. This should not be the facility only for PTHREAD strategy. Another example is the SAC memory management system. In multithread execution environment, heap manager will provide the facility for allocating and de-allocating memory for multiple threads. This is also a facility that should not be restricted to PTHREAD strategy.

The problem now is that in the previous implementation, no-one could predict that in the near future another parallelism strategy would be implemented, so the PTHREAD specific implementation and the common multithread implementation are now coupled tightly in the same module.

Now the most direct way for OpenMP parallelism strategy is to create the similar module from scratch. Besides the OpenMP specific multithread implementation is encapsulated in this module, the module will also include the common multithread facility which perform the same functionalities, but just have different names from those facilities in the PTHREAD module.

The obvious disadvantage of this solution is that it is hard and tedious to maintain since if there is a new requirement for the common multithread environment, both PTHREAD module and OpenMP module have to be modified to adjust to the new requirement. The situation will become worse if in the future, there is the third multithread solution which is encapsulated in the third module.

Thus we come up with the solution to split the current module for PTHREAD multithread strategy into two separate modules, one of which will be responsible for the common multithread facilities and the other one is especially for the PTHREAD implementation. The specific OpenMP multithread implementation will be encapsulated into the new module created. Figure 4.17 illustrates this work.

After the previous PTHREAD multithread module is decomposed into two distinct modules, here comes the question how to generate two different SAC run time libraries for PTHREAD strategy and OpenMP strategy. The solution is summarized as follows and presented in the Figure 4.18.

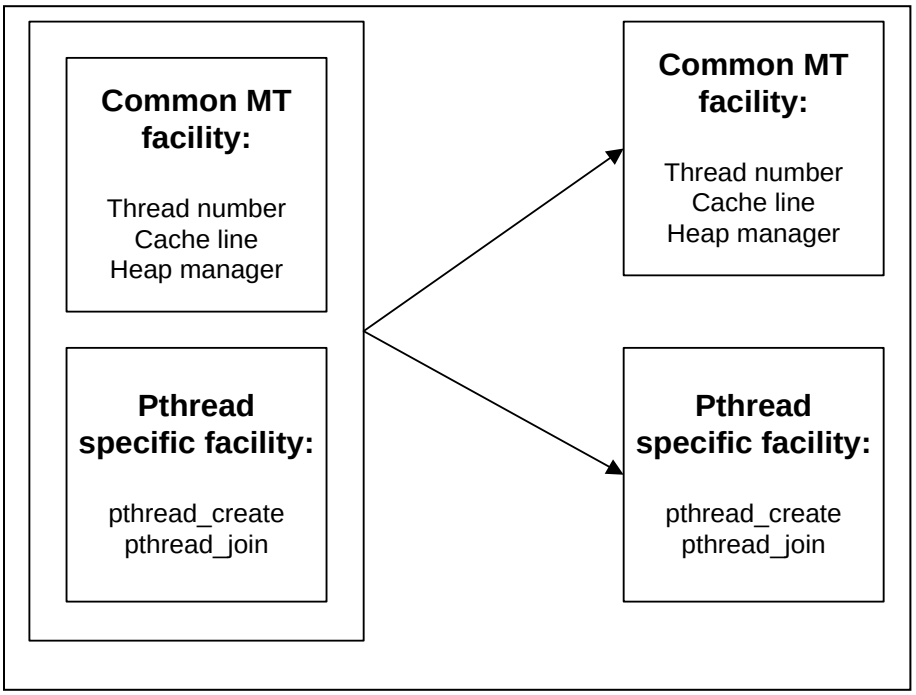


Figure 4.17: **Split original PTHREAD multithread module into two modules.** The common multithread facilities are encapsulated into a new module.

We introduce three MACROS to tailor the specific code to generate two run time libraries. One is libsac.mt.pth for PTHREAD parallelization strategy and the other is libsac.mt.omp for OpenMP parallelization strategy:

- **SAC_DO_MT**: The value of this MACRO is determined only by the number of threads configured at run time or compile time. If the number of threads is more than one, this MACRO is set to true and the common multithread facilities will be tailored to either OpenMP run time library libsac.mt.omp or PTHREAD run time library libsac.mt.pth.

- **SAC_DO_MT_PTHREAD**: The value of this MACRO is set to true if in the compile time the programmers choose the PTHREAD multithread strategy. With this MACRO set to true, the specific PTHREAD code will be tailored to PTHREAD run time library libsac.mt.pth.

- **SAC_DO_MT_OMP**: The value of this MACRO is set to true if in the compile time the programmers choose the OpenMP multithread strategy. With this MACRO set to true, the specific OpenMP code will be tailored to OpenMP run time library libsac.mt.omp.

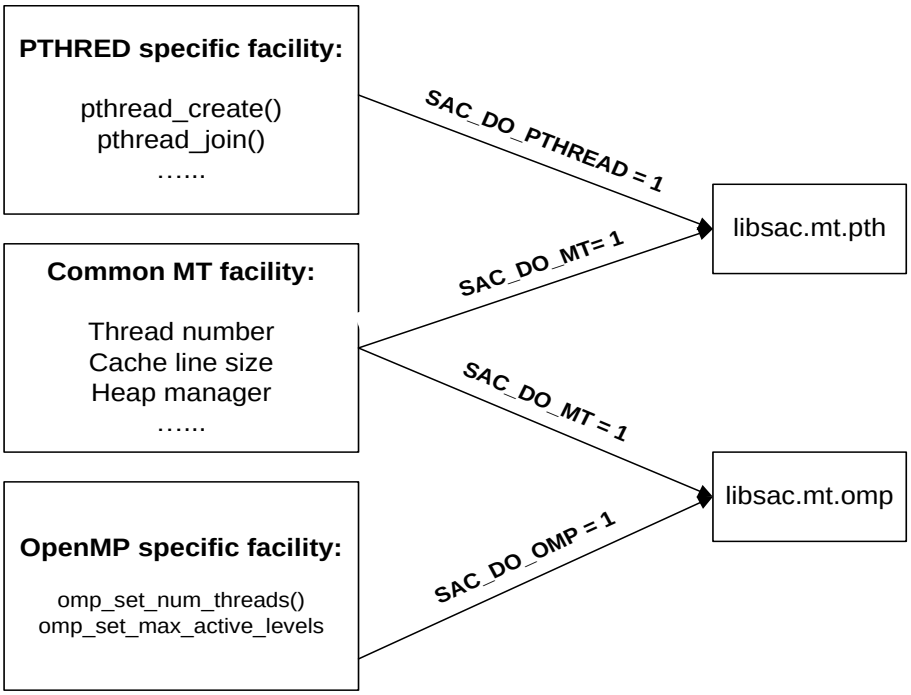


Figure 4.18: **Construction two different run time library from the same source code.** The combination of different MACROs will yield to different SAC runtime library.

Chapter 5. Performance evaluation

This chapter serves as the experimental investigations of the runtime performance achieved by the OpenMP parallelization strategy described in Chapter 4. Firstly Section 5.1 describes the performance of micro synthetic benchmarks to evaluate the performance achieved for genarray WITH-loop and two variants of fold WITH-loops. In Section 5.2 and Section 5.3, the performance for relax benchmark and the NAS MG benchmark are observed respectively.

Experimental investigations of the runtime performance are made on a 4-processor machine. Each processor is Quad-Core AMD Opteron(tm) Processor 8356. The cache size is 512 KB and the CPU is 2.3 GHz. And the methodology of the experiment is to first run the sequential version of program; then run the OpenMP parallelization strategy and Pthread parallelization strategy respectively from 2 threads to 16 threads, incremented by 2 threads each time. The figure of the execution time is achieved by the average number of three independent runs. And the speedups are achieved by the division between the sequential execution time and the multithread execution time.

5.1 Micro benchmarks

5.1.1 genarray WITH-loop

This micro benchmark serves as the benchmark to experiment on the performance of genarray WITH-loop. Figure 5.1 presents the code of the example. In this example, a 1000*1000 array is initialized and then 3000 iterations of calculation are carried out. In every iteration, each element is calculated to a certain new value. Figure 5.2 shows the results of the experiment.

```
int main()
{
    A =  with {
      (. <=iv<= .) : tod(iv);
    }: genarray( [1000,1000]);

    for (i = 0; i < 3000; i++)
    {
       A = with {
         (. <=iv<= .) : tod(iv+i) + A[iv];
       }: genarray( [1000,1000]);
    }
    print(A[1,1]);

    return (0);
}
```

Figure 5.1: **A micro benchmark to test the performance of genarray WITH-loop.**

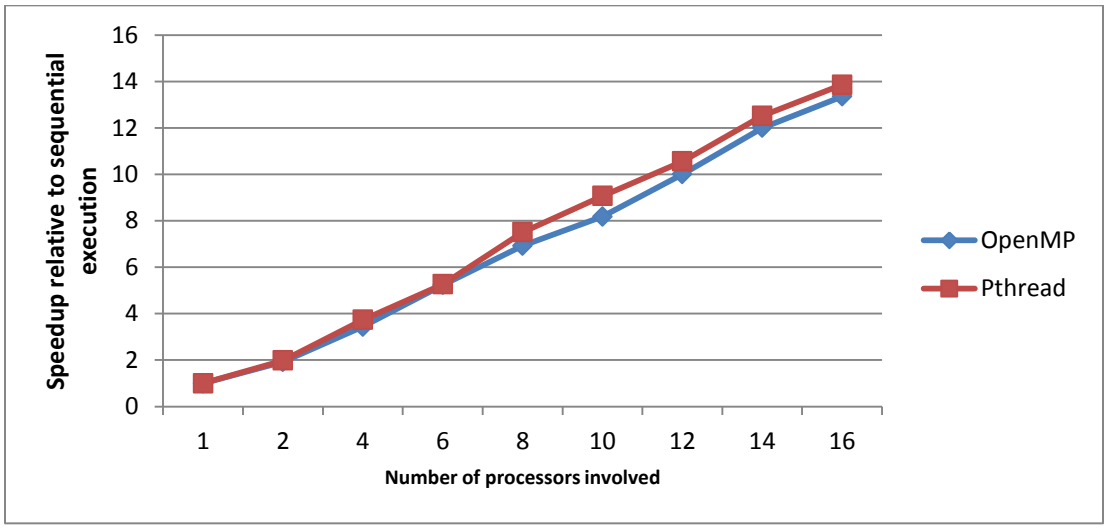


Figure 5.2: **Speedups of example in Figure 5.1.**

Figure 5.2 shows that the performance for both Pthread strategy and OpenMP strategy scale linearly. But the performance of Pthread strategy is a little better than the one of OpenMP strategy, especially when the number of threads scales above 8

5.1.2 fold WITH-loop with simple operator

This micro benchmark serves as the benchmark to experiment on the performance of fold WITH-loop which has the simple operator supported by OpenMP reduction clause. Figure 5.3 presents the code of the example. In this example, an array is first initialized and then 20000 iterations are carried out. In each iteration, fold WITH-loop is called once and the '+' operator is used to calculate the sum of all elements in the array. Figure 5.4 shows the results of the experiment.

```
int main()
{
  A =  with {
    (. <=iv<= .) : 1;
  }: genarray( [1000,1000]);

  c = 0;

  for(i = 0; i < 20000; i++)
  {
       c += with
       {
         ([0,0] < iv < [1000,1000]): i + A[iv];
       } : fold(+, 1);
  }

  print(c);
  return (0);
}
```

Figure 5.3: **A micro benchmark to test the performance of fold WITH-loop with simple operator.**

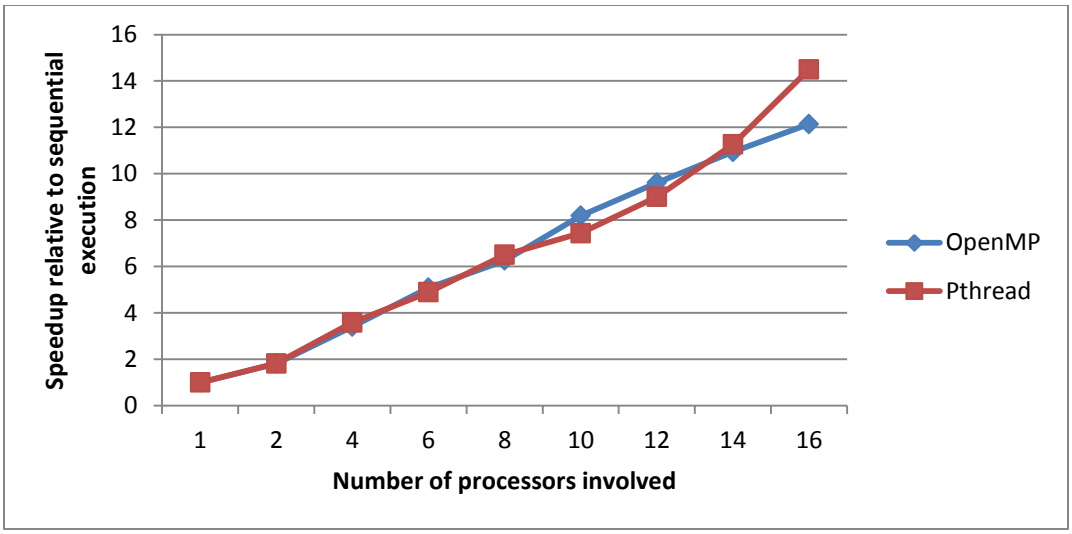


Figure 5.4: **Speedups of example in Figure 5.3.**

From Figure 5.4, we can observe that the performance here is similar to the performance we get in Section 5.1.1. Both Pthread strategy and OpenMP strategy achieve relatively good performance. But it is a bit hard to determine whether Pthread strategy outperforms OpenMP strategy or not. The performance of OpenMP strategy and Pthread strategy are very close from 2 threads to 8 threads; and OpenMP strategy performs a little better than Pthread strategy from 8 threads to 14 threads; but Pthread strategy achieves better performance when the number of threads involved is 16.

According to the documentation of OpenMP, OpenMP provides a more efficient way to implement the reduction. For example, the final summation could be computed through a binary tree, which scales better than a naive summation. This is proved by the experiments in Section 5.1.1 and Section 5.1.2 since the OpenMP strategy always achieves worse performance than Pthread strategy for genarray WITH-loop; but for the fold WITH-loop which is transformed into the OpenMP code using OpenMP reduction clause, OpenMP strategy sometimes perform better performance.

5.1.3 fold WITH-loop with user-defined function

This micro benchmark serves as the benchmark to experiment on the performance of fold WITH-loop which uses the user-defined function in the operator. Thus the OpenMP critical construct has to be used in the code generation. Figure 5.5 presents the code of the example. In this example, a 1000 * 1000 array is first initialized and then 10000

iterations are carried out. In each iteration, fold WITH-loop is called once and the user-defined function **add_add** is used in the fold operator. Figure 5.6 shows the results of the experiment.

```
inline int add_add( int a, int b)
{
  min = a + b - 2;
  return(min);
}


int main()
{
   A =  with {
      (. <=iv<= .) : 1;
   }: genarray( [1000,1000]);

   c = 0;

   for(i = 0; i < 10000; i++)
   {
     c += with
          {
            ([0,0] < iv < [1000,1000]): i + A[iv];
          } : fold(add_add, 0);
   }

   print(c);
   return (0);
}
```

Figure 5.5: **A micro benchmark to test the performance of fold WITH-loop which uses user-defined function in the operator.**

Figure 5.6 shows that the performance for Pthread strategy scales linearly and achieves good performance from 2 threads to 16 threads. But the performance of OpenMP strategy is extremely bad. Instead of achieving some speedups against the sequential version, the OpenMP strategy is much slower than the sequential one and the performance becomes even worse with the augmentation of the number of threads involved. The reason is that the critical construct in OpenMP is relatively expensive.

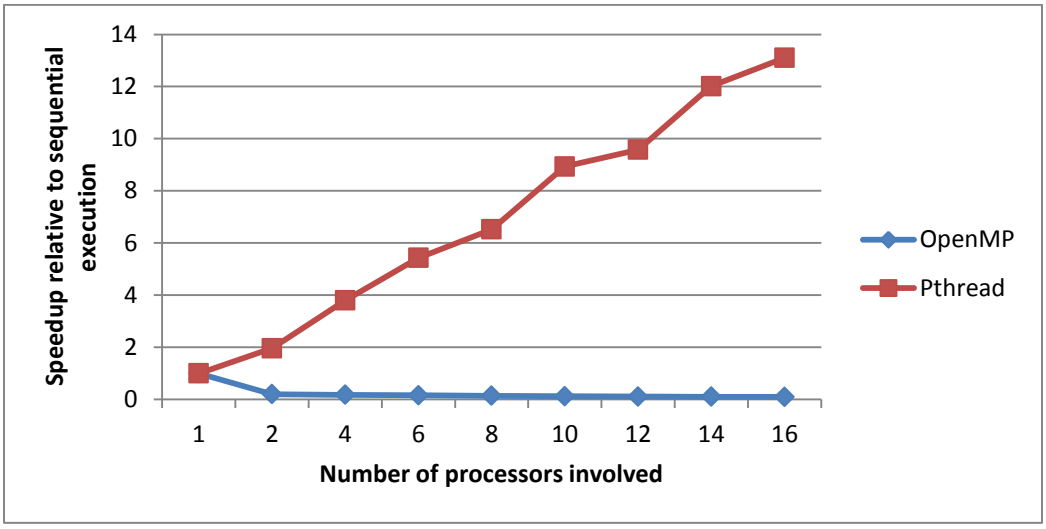


Figure 5.6: **Speedups of example in Figure 5.5.**

5.2 Relax benchmark

Relax benchmark is a typical numerical application which relaxes a 2-dimension grid. Figure 5.7 shows the relevant numerical kernel of SAC program. The function **relax** calls the function **onestep** for a pre-specified number of times. The function **onestep** is the computation carried out in one iteration. To be more precisely, each element of array B is set to the arithmetic mean of its 4 adjacent elements in array A. The specification of **0.25d** is to characterize double precision floating point constants.

The Relax benchmark can be applied to the 2-dimensional array with any shape. Thus it allows for systematic variations of the problem size. Similarly the number of iterations is another parameter in the experiment. Figure 5.8 presents the performance for the relaxation of a 4096 * 4096 array with 1000 iterations. Both strategies achieve relatively good performance and scale almost linearly. But Pthread strategy performs a little better performance after the number of processors scales up to 4.

```
double[+] onestep(double [+] B)
{
  A = with {
        ( . < x < . ) :  0.25d *(B[x+[1,0]]
                              + B[x-[1,0]]
                              + B[x+[0,1]]
                              + B[x-[0,1]]);
      } : modarray( B );

  return(A);
}

double[+] relax(double [+] A, int steps)
{
  for (k=0; k<steps; k++) {
    A = onestep(A);
  }

  return(A);
}
```

Figure 5.7: **Numerical kernel of Relax benchmark.**

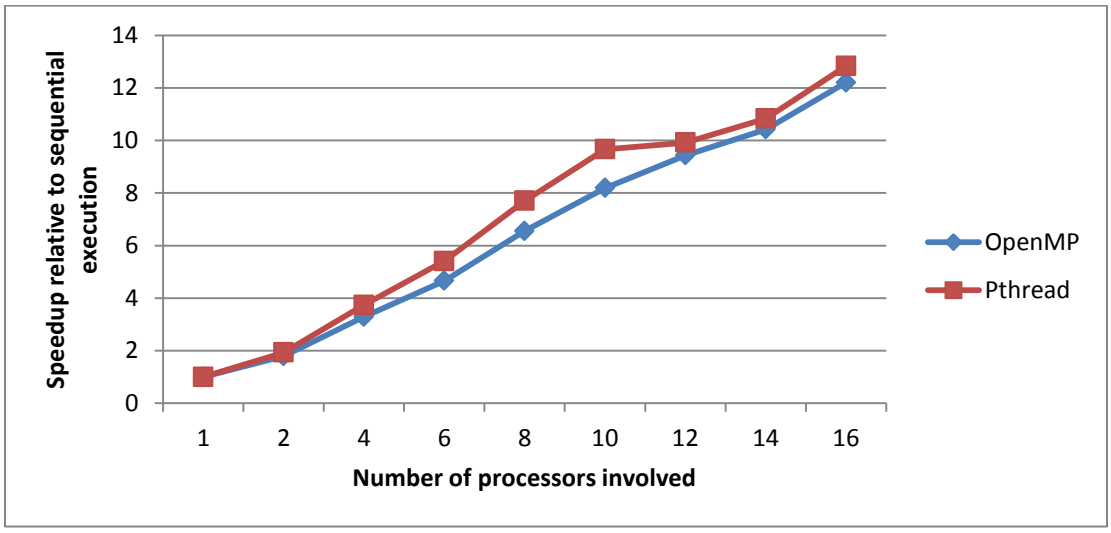


Figure 5.8: **Speedups of Relax benchmark.**

5.3 NAS MG benchmark

The NAS benchmark suite **[29]** has been developed at NASA Ames Research Center as part of the Numerical Aerodynamic Simulation Program. It consists of 5 application kernels and 3 small application programs, which are considered representative for large scale applications in computational fluid and aerodynamics. For the details of the SAC implementation of NAS MG benchmark, please look at **[10, 30]**.

The problem size we use in this experiment is 256 * 256 * 256 and the number of iterations is 2. Figure 5.9 presents the comparison of the performance between OpenMP strategy and Pthread strategy. OpenMP strategy is slightly less efficient than Pthread strategy, which is consistent with the micro benchmarks and the Relax benchmark. But the scalability for both strategies is quite poor: Pthread strategy achieves utmost 2.75 times of speedup and OpenMP achieves 2.48 times of speedup.

But in previous experiments **[30]**, the scalability for Pthread strategy is proven to be good. For the problem size 256 * 256 * 256, 7.6 times of speedup has been achieved when the number of threads involved is 10. Thus we speculate that there are now some problems inside the compiler which cause the scalability problem for both OpenMP strategy and Pthread strategy. Since these problems are out of our control, we can only expect the similar improvement on scalability for OpenMP strategy as for the Pthread strategy when these problems are solved.

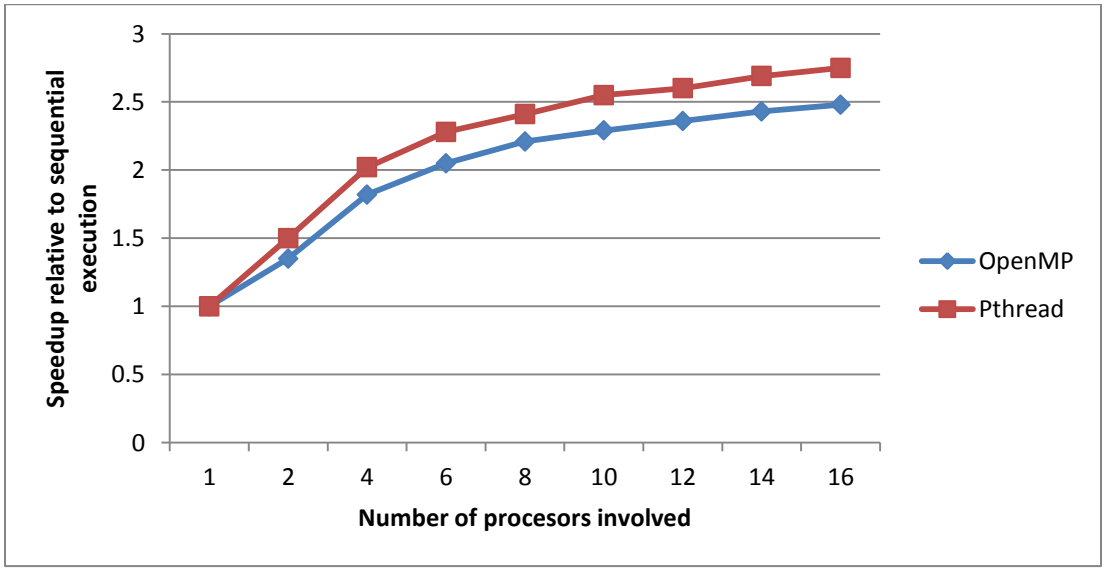


Figure 5.9: **Speedups of NAS MG benchmark.**

Chapter 6. Future work

6.1 OpenMP task parallelization in SAC

One of the most important future jobs is the implementation of OpenMP task parallelization in SAC. Due to the time constraint although the idea how to implement it is designed, it is not yet fully implemented.

The idea can be summarized as follows:

- In **[25]**, the functional parallelization of SAC is researched and implemented. A new key word **spawn** is introduced into the syntax of SAC. The use of **spawn** is to be inserted before one function call in the SAC program to inform the compiler that this function call should be regarded as a task. It is the programmers' responsibility to decide which function call to be regarded as the task because for some small function calls, it is not worthwhile to treat them as tasks since the initialization cost of the tasks is not trivial. The job of the OpenMP parallelization solution is to map the spawn into the task directive in OpenMP, as presented in Figure 6.1. After the map, the OpenMP implementation will have a certain mechanism to attach the task to one thread in the team and the thread will execute the task at a certain time.

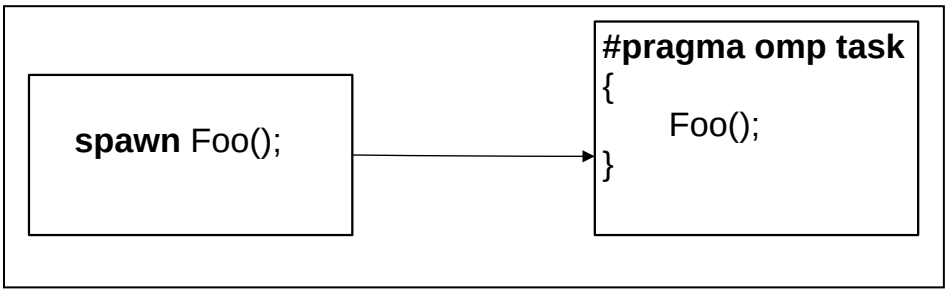


Figure 6.1: **Map from spawn in SAC into OpenMP task directive.**

- We need to determine whether a spawn task is inside the OpenMP parallel region or not. Consider the following two different situations presented respectively in Figure 6.2 and Figure 6.3. We know the fact that multiple threads in OpenMP are generated only if there is a **parallel** region. And the task directives are useless if it is not encapsulated within one **parallel** region. From previous examples, we know that the parallel directives are only used if there is a WITH-loop which is worthwhile to be executed in parallel. Thus for the SAC example presented in the left hand of Figure 6.2, the only thing we need to do is to map the spawn task into an OpenMP task directive, as presented in the right hand side of Figure 6.2. But for the SAC code presented in left hand side of Figure 6.3, we also need to create an additional OpenMP parallel region to encapsulate the two OpenMP task directives, as presented in the right hand side of Figure 6.3. Otherwise, both OpenMP task directives will be simply ignored by the compiler since there are no more threads generated by OpenMP parallel region to execute in the two tasks parallel. .

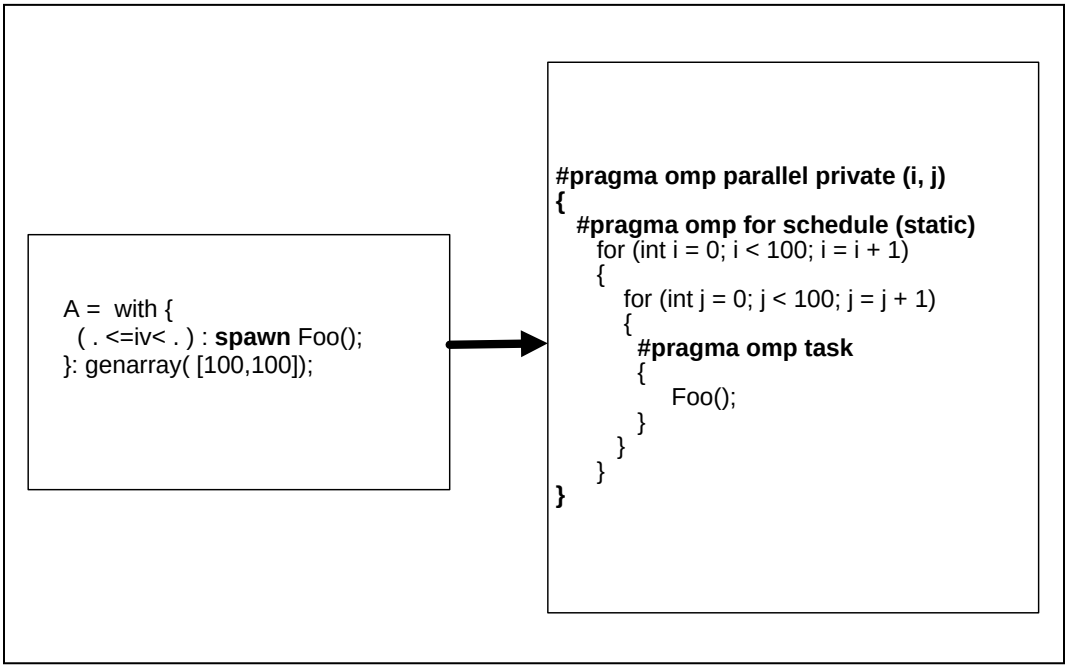


Figure 6.2: **spawn task inside a parallel region.** The spawning of a function is called in the context of multithread environment.

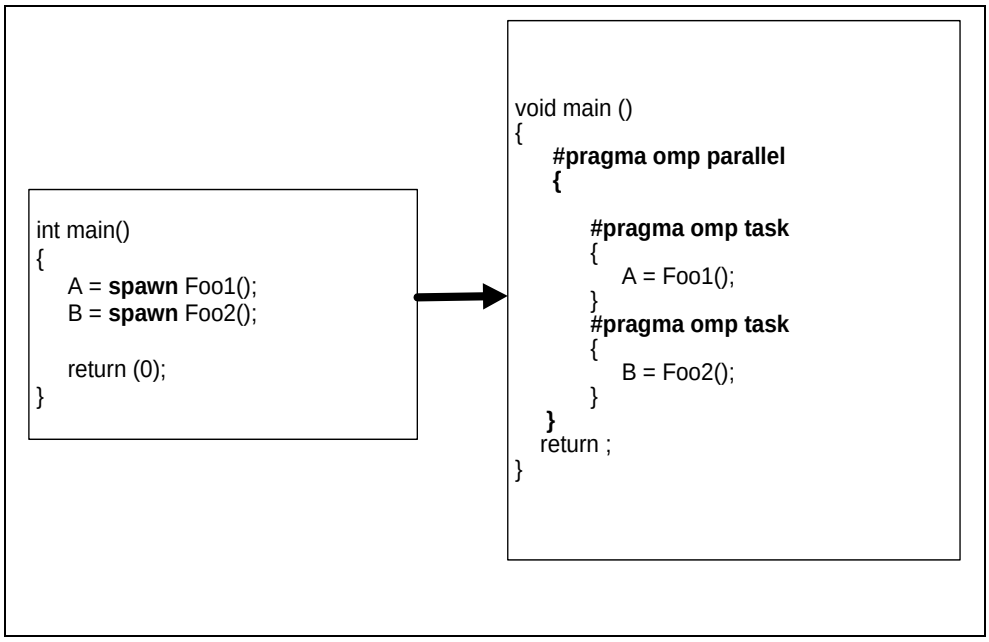


Figure 6.3: **spawn task not in parallel region.** The spawning of a function is called in the context of sequential environment. An additional OpenMP parallel directive needs to be inserted.

6.2 Optimization of OpenMP data parallelization in SAC

Another optimization opportunity is to reduce the unnecessary synchronization overhead. For instance the current parallelization focuses on each WITH-loop individually. Whenever a WITH-loop is encountered, the inserted parallel directive will guarantee that multiple threads are generated. At the end of the parallel region, there is default synchronization. But for some cases, this might lead to unnecessary overhead.

Take the situation illustrated in Figure 6.4 as an example, it might be possible to encapsulate the two adjacent WITH-loops into one OpenMP parallel region instead of two parallel regions and thus will reduce the unnecessary synchronization. Especially when the number of threads is large, the synchronization overhead could be very huge. This solution is presented in Figure 6.5.

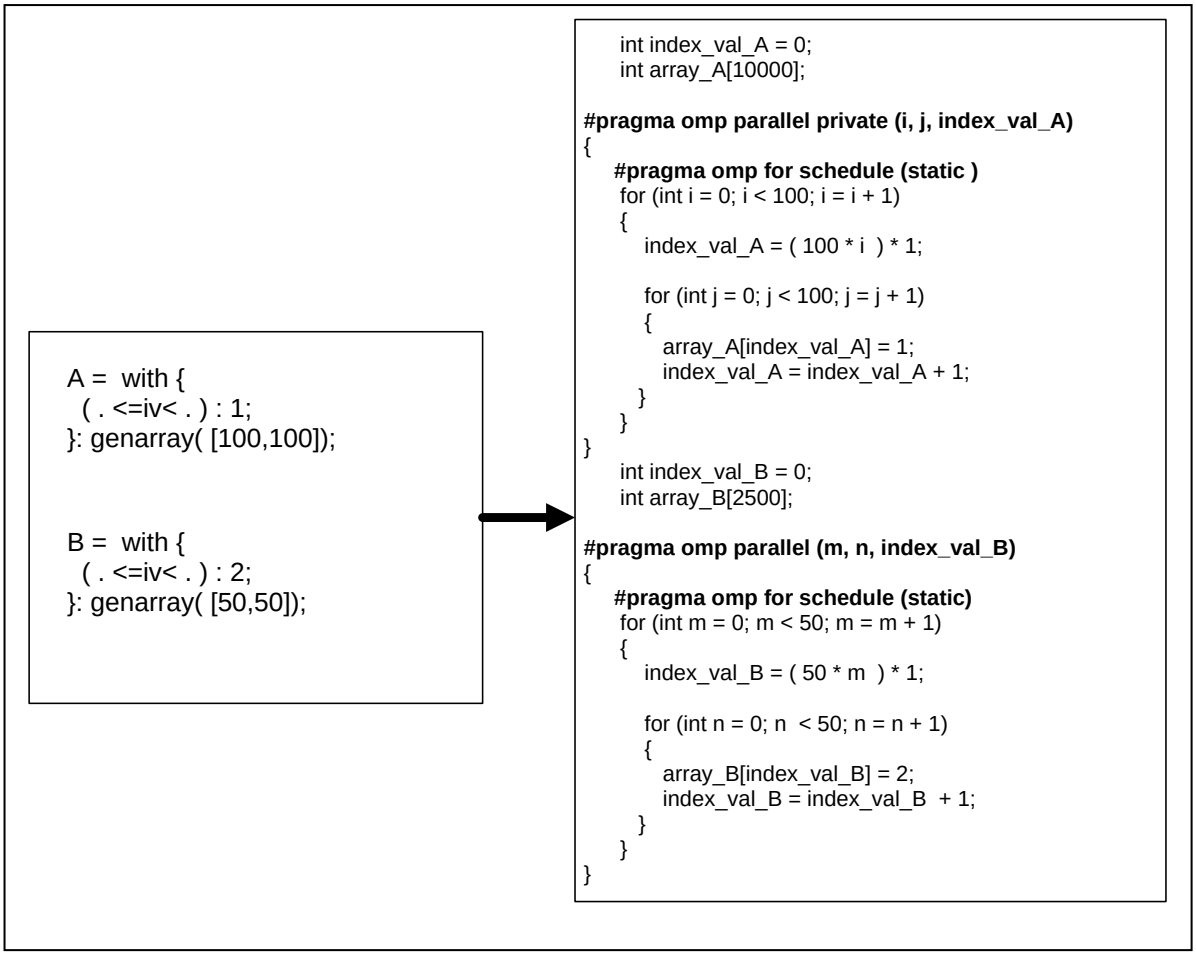


Figure 6.4: **Unnecessary synchronization overhead.** The first and the second OpenMP parallel region can be merged to only one.

The experiment results from Section 5.1.3 also shows the poor performance in the OpenMP parallelization strategy regarding the fold WITH-loop which uses a user defined function in the operator. This is because that using OpenMP critical construct is quite expensive. Thus this reveals another optimization opportunity that it is reasonable not to use the OpenMP critical construct. If we can infer the OpenMP reduction variable and the operator from the context of the user-defined function, the OpenMP reduction clause can be used, which will greatly improve the performance, as it is illustrated in Section 5.1.2.

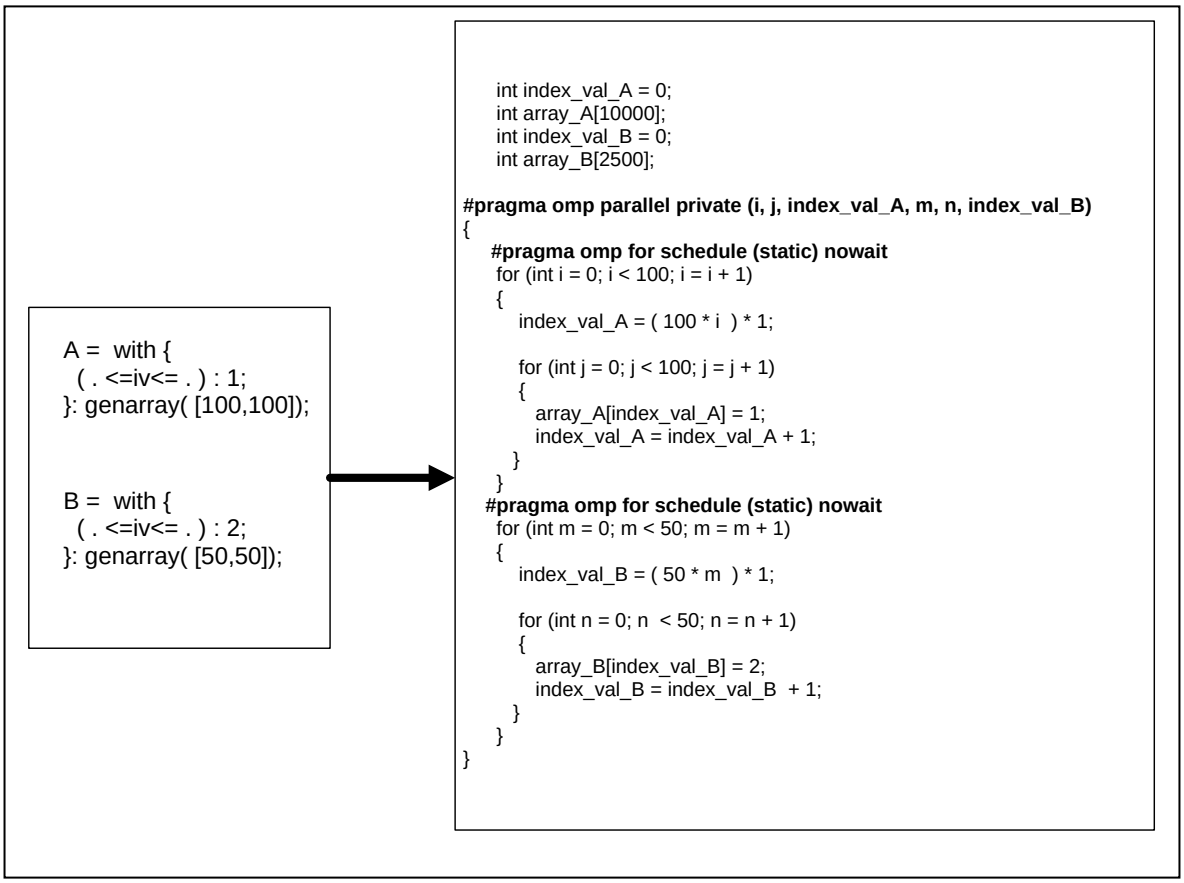


Figure 6.5: **Solution for reducing unnecessary synchronization.** The OpenMP clause **nowait** is to suppress the barrier of the associated work sharing construct **for**. In other words, when threads reach the end of the **for** construct, they will immediately proceed to perform other work.

Another trivial but also important optimization opportunity is to configure parameter of the OpenMP **for** construct at run time instead of compile time so that after an OpenMP code is generated, it is not necessary to recompile the code with another schedule technique or chunk size.

Finally, more experiments need to be done. On the first hand, as explained in Chapter 1, one of the motivations of OpenMP parallelization strategy is to expose the possibility to run SAC program on more architectures, such as Cell and GPGPU, on which are currently impossible to run Pthread parallelization strategy code. Theoretically, OpenMP parallelization can run on these architectures. But it still needs the experiments on these architectures. On the second hand, it is also interesting to see the performance of combining OpenMP task parallelization and data parallelization after OpenMP task parallelization is completely implemented.

Chapter 7. Conclusion

Single assignment C is a functional array processing language that is especially targeted to facilitate the design and implementation of computationally intensive applications in fields such as scientific computing, image processing, simulation, or modeling. It provides a high level of abstraction and at the same time offers competitive execution performance compared with the low level imperative programs such as C. Because scientific programs always perform complex operations on large arrays, which pose a possibility to execute the program in parallel, we could divide the array and distribute different parts of the array to different threads. The existing PTHREAD multithread strategy is a solution which uses the PTHERAD API to create multiple threads to divide the iteration space of WITH-loop and make multiple threads work on the different parts of WITH-loop concurrently.

OpenMP is a de-facto multithread standard in the industry which is widely supported by hardware and software vendors. It provides the C/C++ and FORTRAN programmers an easy way to write the multithread program and could maintain the sequential version of code and the parallel version of code simultaneously. The idea of OpenMP is that the programmers will describe the concurrency of the application with the syntax in OpenMP and the OpenMP compilers will sort out the details of the concurrency, such as generating multiple threads, dividing the iteration space between the threads and synchronizing them at the end of the parallel region.

This thesis designs and implements the strategy to use OpenMP as an alternative parallelization strategy to parallel the SAC program. The OpenMP strategy would not only provide the programmers with another solution to parallel the SAC program, but more importantly would provide a possibility to execute the SAC program in parallel on the platform that does not support PTHREAD.

To summarize, the work described in this thesis contributes the following to the state of the art:

- It provides a concise and efficient solution to split the common multithread facilities from the existing PTHREAD module into a brand new module. And with a small trick, two different run time libraries could be generated. One is

- for PTHREAD solution and the other is for OpenMP solution. The work here also benefits other potential multithread solutions in the future.

- It reuses the first two sub phases from the multithread compilation phase in the compiler framework. The first sub phase is to analyze whether it is worthwhile to execute the WITH-loop in parallel or not. And the second sub phase is to determine the environment of the function execution is in sequential environment or in concurrent environment.

- It uses the OpenMP **parallel** region to specify the code to be executed by multiple threads which are generated by OpenMP compiler. One OpenMP parallel region corresponds to one WITH-loop in SAC.

- It uses the **private** clause in OpenMP to specify the attributes of the variables so that the correctness of the OpenMP multithread program can be guaranteed.

- It uses the **for** directive in OpenMP to provide the programmers with versatile methods to win the utmost speedup from OpenMP.

- It uses the **reduction** clause in OpenMP to parallel the execution of fold WITH-loop which uses the simple mathematical operators supported by OpenMP reduction clause as the fold operator.

- It uses the **critical** directive in OpenMP to parallel the execution of fold WITH-loop which uses the user-defined function as the fold operator.

- It inserts appropriate segment of code to avoid the false sharing problem which has extremely negative impact on the performance of the multithread program generated.

- It uses the nest level to generate the nested OpenMP parallel region to accelerate the execution of the program.

In the experiments, micro and synthetic benchmarks are first carried on to test the performance of genarray WITH-loop and two variants of fold WITH-loop. The figures in Section 5.1 show that for both genarray WITH-loop and fold WITH-loop which uses the simple operator, the OpenMP parallelization strategy achieves good performance. For genarray WITH-loop, the performance of OpenMP strategy is a little worse than Pthread

strategy; and for the fold WITH-loop using the simple operator, OpenMP strategy sometimes achieve better performance than Pthread strategy. But for the fold WITH-loop which uses the user-defined function in the operator, the performance of OpenMP parallelization strategy is quite bad and is even worse than the sequential version especially when the problem size scales to a large number. This is because the cost of using OpenMP critical construct is quite expensive.

In the experiments of two typical numerical benchmarks, the Relax benchmark and the NAS MG benchmarks are also used to evaluate the performance of the OpenMP parallelization strategy. From the results the Relax benchmark, we know that the performance of OpenMP solution scales linearly, but still is a little worse than the Pthread solution. For the NAS MG benchmark, the scalability for both OpenMP and Pthread strategy is not good due to some problems inside the compiler. But we can expect the scalability for both strategies to be improved greatly after we fix the problems in the future.

OpenMP parallelization strategy provides the possibility to run SAC code on some architecture which currently does not support Pthread. Due to time constraints, no experiment is done on these architectures. But theoretically, it is possible.

Bibliography

[1] Clemens Grelck and Sven-Bodo Scholz. SAC FROM HIGH-LEVEL PROGRAMMING WITH ARRAYS TO EFFICIENT PARALLEL EXECUTION. *In Parallel Processing Letters 13(3), pp.401-412, ©World Scientific Publishing, 2003.*

[2] H.P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*, vol. 103 of Studies in Logics and the Foundations of Mathematics. North Holland, Amsterdam, The Netherlands, 1981.

[3] Clemens Grelck and Sven-Bodo Scholz. SAC: A Functional Array Language for Efficient Multithreaded Execution. *In International Journal of Parallel Programming 34(4), pp. 383-427 Springer-Verlag, Dordrecht, Netherlands, 2006.*

[4] Clemens Grelck. A Multithreaded Compiler Backend for High-Level Array Programming. In M.H. Hamza, ed., *Proceedings of the 21st International Multi-Conference on Applied Informatics (AI'03), Part II: International Conference on Parallel and Distributed Computing and Networks (PDCN'03),* pp. 478-484. ACTA Press, 2003.

[5] Steffen Kuthe. A Hybrid Shared Memory Execution Model for SAC. *Diploma thesis, Institute for Software Technology and Programming Languages, University of Lübeck, 2005.*

[6] Kai Trojahner. Implicit Memory Management for a Functional Array Processing Language. *Diploma thesis, Institute for Software Technology and Programming Languages, University of Lübeck, 2005.*

[7] Sven-Bodo Scholz. *Single Assignment C — Efficient Support for High-Level Array Operations in a Functional Setting*. Journal of Functional Programming, vol. , 2003. Accepted for publication.

[8] Brian W. Kernighan and Dennis M. Ritchie. *The C programming Language.* Prentice-Hall, 1988. 2nd edition.

[9] http://www.sac-home.org

[10] Clemens Grelck. *Implicit Shared Memory Multiprocessor Support for the Functional Programming Language SAC --- Single Assignment C. PhD thesis, Institute of Computer Science and Applied Mathematics, University of Kiel, ©Logos Verlag, 2001.*

[11] John L.Henessy and David A.Patterson. *Computer Architecture – A Quantitative Approach Fourth Edition*.

[12] Alfred V．Aho、Ravi Sethi and Jeffrey D.Ullman.
Compilers:Principles,Techniques,and Tools. 3rd Edition.

[13] N.P. Jouppi and D. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. In *Proceedings of ASPLOS III*, pages 272–282, 1989.

[14] Barbara Chapman, Gabriele Jost and Ruud van der Pas. *Using OpenMP Portable Shared Memory Parallel Programming*. The MIT Press Cambridge, Massachusetts, London, England, 2008

[15] Eduard Ayguadé, Nawal Copty, Member, IEEE Computer Society, Alejandro Duran, Jay Hoeflinger,Yuan Lin, Federico Massaioli, Member, IEEE, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. *The Design of OpenMP Tasks.* In *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, VOL. 20, NO. 3, MARCH 2009*

[16] Eduard Ayguadé, Duran, A., Hoeinger, J., Massaioli, F., Teruel, X.: An experimental evaluation of the new OpenMP tasking model. In *Adve, V.S., Garzar_an, M.J., Petersen, P., eds.: LCPC. Volume 5234 of LNCS., Springer (2007) 63-77*

[17] Eduard Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, P. Unnikrishnan, and G. Zhang. A Proposal for Task Parallelism in OpenMP. In 3rd *International Workshop on OpenMP (IWOMP'07)*, 2007.

[18] Kevin. O'Brien, Kathryn. O'Brien, Z. Sura, T. Chen, and T. Zhang. Supporting OpenMP on Cell. *International Journel of Parallel Programming (IJPP)*, 36(3):289–311, June 2008.

[19] http://www.research.ibm.com/cell/

[20] http://gpgpu.org/

[21] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. OpenMP to GPGPU: A Compiler Framework for AutomaticTranslation and Optimization. In *PPoPP'09, 2009.*

[22] https://computing.llnl.gov/tutorials/pthreads/

[23] OpenMP Architecture Review Board. OpenMP Application Program Interface, May 2005.

[24] W. J. Bolosky and M. L.Scott. False sharing and its effect on shared memory performance. In *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, pages 57–71, San Diego, CA, September 1993.

[25] Aram Visser. Function parallelization in SAC. *Master thesis, Faculty of Science, University of Amsterdam, 2011.*

[26] Clemens Grelck. Shared memory multiprocessor support for functional array processing in SAC. *Journal of Functional Programming*, 15 (2005) 353 – 401.

[27] Clemens Grelck., Trojahner, K. Implicit Memory Management for SAC. In Grelck, C., Huch, F., eds. Proceedings of the 16th International Workshop on Implementation and Application of Functional Languages, IFL 2004, Lubeck, Germany, September 8-10, 2004. Volume 0408 of Technical Report, University of Kiel (2004) 335 - 348

[28] Clemens Grelck, Sven-Bodo Scholz. Efficient Heap Management for Declarative Data Parallel Programming on Multicores. *Annual Symposium on Principles of Programming Languages, 3rd Workshop on Declarative Aspects of Multicore Programming (DAMP'08).*

[29] D. Bailey, J. Barton, T. Lasinski, and H.Simon(Eds.). *The NAS Parallel Benchmarks.* NAS Technical Report RNR-91-002, NASA Ames Research Center, Moffett Field, CA, 1991, http://www.nas.nasa.gov/Research/.

[30] Clemens Grelck. Implementing the NAS Benchmark MG in SAC. In Viktor K. Prasanna and George Westrom, editors, *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS'02), Fort Lauderdale, Florida, USA*. IEEE Computer Society Press, 2002.