Universidade Estadual de Campinas
Instituto de Computação

Luís Felipe Souza de Mattos

DOACROSS Parallelization using Component Annotation and Loop-carried Probability

Paralelização de Laços DOACROSS Usando Anotações de Componentes e Probabilidade de Loop-Carried

CAMPINAS
2018

Luís Felipe Souza de Mattos

DOACROSS Parallelization using Component Annotation and Loop-carried Probability

Paralelização de Laços DOACROSS Usando Anotações de Componentes e Probabilidade de Loop-Carried

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Thesis presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Computer Science.

Supervisor/Orientador: Prof. Dr. Guido Costa Souza de Araújo
Co-supervisor/Coorientador: Prof. Dr. Márcio Machado Pereira

Este exemplar corresponde à versão final da Dissertação defendida por Luís Felipe Souza de Mattos e orientada pelo Prof. Dr. Guido Costa Souza de Araújo.

CAMPINAS
2018

Agência(s) de fomento e nº(s) de processo(s): FUNCAMP
ORCID: https://orcid.org/0000-0002-8802-6891

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

M436d

Mattos, Luís Felipe Souza de, 1990-
Doacross parallelization using component annotation and loop-carried probability / Luís Felipe Souza de Mattos. – Campinas, SP : [s.n.], 2018.

Orientador: Guido Costa Souza de Araújo.
Coorientador: Márcio Machado Pereira.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. OpenMP (Programação paralela). 2. Programação paralela (Computação). 3. Computação de alto desempenho. I. Araújo, Guido Costa Souza de, 1962-. II. Pereira, Márcio Machado, 1959-. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

Informações para Biblioteca Digital

Título em outro idioma: Paralelização de laços doacross usando anotações de componentes e probabilidade de loop-carried
Palavras-chave em inglês:
OpenMP (Parallel programming)
Parallel programming (Computer science)
High performance computing
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Guido Costa Souza de Araújo [Orientador]
Sandro Rigo
Fernando Magno Quintão Pereira
Data de defesa: 16-05-2018
Programa de Pós-Graduação: Ciência da Computação

Universidade Estadual de Campinas
Instituto de Computação

Luís Felipe Souza de Mattos

DOACROSS Parallelization using Component Annotation and Loop-carried Probability

Paralelização de Laços DOACROSS Usando Anotações de Componentes e Probabilidade de Loop-Carried

Banca Examinadora:

- Prof. Dr. Guido Costa Souza de Araújo
  Institute of Computing - UNICAMP
- Prof. Dr. Fernando Magno Quintão Pereira
  Universidade Federal de Minas Gerais (UFMG)
- Prof. Dr. Sandro Rigo
  Institute of Computing - UNICAMP

A ata da defesa, assinada pelos membros da Comissão Examinadora, consta no SIGA/Sistema de Fluxo de Dissertação/Tese e na Secretaria do Programa da Unidade.

Campinas, 16 de maio de 2018

Agradecimentos

Primeiramente gostaria de agradecer ao meu orientador, Dr. Guido Araújo, que sempre me guiou e me ajudou bastante com a minha pesquisa. Sem ele, esta pesquisa não teria sido possível.

Gostaria de agradecer também ao meu coorientador, Dr. Márcio Pereira que além de revisar todo meu trabalho, também me ajudou bastante durante a pesquisa.

Agradeço também aos meus colegas de projeto, Divino César Lucas, Juan Salamanca e João Paulo Carvalho, que me ajudaram durante a pesquisa e nos experimentos.

Gostaria de agradecer também à FUNCAMP, com parceria do projeto da Samsung, pela oportunidade de poder trabalhar nesse projeto que me ajudou a trabalhar em equipe e conhecer outras ideias.

Resumo

A paralelização de laços é usada para se obter melhor desempenho em algoritmos intensivos, entretando, não são todos os laços que podem ser facilmente paralelizados.

Os laços chamados de *DOACROSS* possuem dependências entre iterações, i.e. uma iteração calcula um dado que é usado por outra iteração futura. Este tipo de dependência é chamada de *loop-carried* e não pode ser paralelizada trivialmente porque a ordem de execução das iterações deve ser respeitada.

Algumas técnicas podem ser usadas para paralelizar este tipo de laço, porém o programador deve entender como funciona o algoritmo e deve escolher quais instruções podem ser executadas em paralelo e quais instruções devem ser executadas sequencialmente. Estas componentes sequenciais e paralelas precisam ser separadas manualmente pelo programador e a comunicação entre as componentes deve ser incluída, a fim de respeitar as dependências entre componentes e as dependências entre iterações.

Implementar essas técnicas é um trabalho laborioso que requer uma certa experiência do programador para separar as componentes e encontrar as dependências para implementar a comunicação entre as componentes/threads. Esta comunicação pode ser feita através de filas ou buffers, dependendo do algoritmo de paralelização escolhido.

Uma das técnicas de paralelização é o algoritmo mais tradicional, chamado de *DOACROSS* [9] que foi implementado no OpenMP 4.5 através da cláusula *depend* da diretiva *ordered*. Este pragma deve ser usado dentro da região de um laço paralelo do OpenMP a fim de separar as componentes que devem ser sequenciais. A comunicação e a sincronização são implementadas automaticamente utilizando a biblioteca de runtime do OpenMP. Este método remove do programador o trabalho de programação, entretando, ainda é necessário delimitar explicitamente as componentes sequenciais.

Outro algoritmo de paralelização estudado foi o Batched DoAcross (BDX) [1]. Este algoritmo pode ser usado para reduzir o overhead da comunicação entre componentes, entretanto, a implementação deve ser feita manualmente pelo programador e requer que o programador separe as componentes sequenciais e paralelas, crie barreiras de sincronização para as componentes sequenciais, crie buffers para a comunicação entre componentes e crie variáveis compartilhadas para a comunicação entre as threads (dependências entre iterações).

Nos experimentos, foi percebido que a escolha do algoritmo de paralelização depende de alguns fatores, i.e. a estrutura do algoritmo, a proporção das dependências entre iterações, o número de iterações do laço e o tamanho do laço.

Foi criada então uma nova cláusula para o OpenMP que, quando usada juntamente com a diretiva *ordered*, consegue separar as componentes sequenciais e paralelas e implementar essas técnicas de forma automática. Esta cláusula, chamada de *use*, deve receber um parâmetro que especifica qual técnica o programador quer utilizar para paralelizar o laço.

Abstract

Loop parallelization can be used to achieve better performance on intensive algorithms, however, not all loops can be easily parallelized.

The called 'DOACROSS' loops have dependences between different iterations, i.e. some iteration computes a data which is used in a later iteration. This kind of dependence is called loop-carried dependence and cannot be simply parallelized because iterations execution order must be respected.

Some techniques can be used to parallelize this kind of loop, however, the programmer must understand how the algorithm works and choose which instructions can be executed in parallel and which instructions need to be serialized. These serial and parallel components need to be manually separated by programmer and communication between components must be included to respect dependences inside loop body and between threads to respect loop-carried dependences.

Implementing these techniques is a laborious work that requires a certain expertise from programmer to separate loop components and find dependences to implement communication between components/threads. This communication can be done by using a queue or a buffer, depending on the algorithm used to parallelize.

One of these parallelization techniques is the traditional DOACROSS [9], which was implemented by using *depend* clause for the *ordered* directive in OpenMP 4.5. This OpenMP construct is used within OpenMP loop region to separate serial and parallel components, then, communication and synchronization are automatically implemented by OpenMP Runtime. This method removes most of the programming work from the programmer, however still requires to explicitly delimit serial region.

Another studied parallelization technique is the Batched DoAcross (BDX) [1]. This algorithm can be used to reduce the communication overhead of synchronization between components, however, the implementation must be done manually by programmer, which requires for the programmer to separate serial and parallel components, create barriers to synchronization in serial components, create buffers for communication between components and create the shared variables for communication between threads (loop-carried dependences).

In our experiments, we noticed that some factors must be taken for the choice of parallelization technique, i.e. algorithm structure, loop-carried ratio, number of loop iterations and loop size.

We created a new OpenMP clause that, used together with the *ordered* directive, can separate these components and implement these techniques automatically. This clause, is called *use*, receive a parameter for specifying which parallelization technique the programmer want to be implemented.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

1.1 Parallel Computation

Parallelism is a great technique used to improve algorithms performance, especially algorithms that require a great number of operations. These intensive algorithms usually have loops that are responsible for most of the program's execution time. Despite the growth of multicore architectures, programmers still struggle to extract the optimal parallelism of algorithms.

Automatic parallelization is already a feature present in some compilers, which analyze the source code during the compilation process, select which loops can be parallelized and inserts appropriate code for the parallelization of these loops. However, there is some kind of loops that are not easily parallelized, and for these loops, the compilers do not attempt to do parallelization automatically. These kind of loops must be manually parallelized by the programmer, and generally require modifications in the algorithm to achieve performance gains when parallelized.

Even with the modifications, it is not guaranteed that all loops will have a performance improvement. These loops have some instructions in the body of the loop that can not be parallelized. Because of this, the other instructions in the body of the loop will be responsible for the performance gain, when parallelized. However, there is an overhead to be able to control the parallelization and synchronization of this loop, and to achieve a performance gain, the parallelism of the parallelizable instructions must compensate for this overhead, in addition to the execution time of the sequential instructions.

This dissertation is organized as follows.

Chapter 1 presents an introduction to this work and explains researched concepts. Section 1.1 gives a brief introduction to parallel computation. Section 1.2 describes the types of studied loops and how to obtain parallelism from them. Section 1.3 details the parallelization algorithms. Section 1.4 explains the concept of loop-carried ratio.

Chapter 2 details the work of implementing proposed clause and is organized in sections as follows. Section 2.1 details code transformations of *use(doax)* clause. Section 2.2 details code transformations of *use(bdx, ...)* clause and is separated into two subsections: 2.2.1 details AST level transformations and 2.2.2 details LLVM IR level transformations.

Chapter 3 presents some related work.

Chapter 4 presents experiments results on well-known benchmark suites and results

analysis.

Chapter 5 presents some conclusions made based on experimental results.

1.2 Loops Classification and Parallelism

Based on data dependency analysis, the loops are divided into two categories: DOALL loops and DOACROSS loops.

Present in most scientific applications, DOALL loops have no dependency between iterations and can be parallelized using various techniques, for example: PThreads, OpenMP, MPI, OpenCL and CUDA. Because there are no dependencies between iterations, they can be executed in any order, and in this case, the output data will have the same output expected from a serial version. The parallelization of this type of loop is very simple, usually distributing the iterations between the available threads or cores.

Unlike DOALL loops, DOACROSS loops have dependencies between iterations. These dependencies, called loop-carried dependencies, occur when an iteration uses a calculated data in a previous iteration. Because of this dependence, the loop needs to be executed in the correct order of iterations so that the output data matches the expected output when the loop is executed serially.

For this reason, this kind of loop has established an important area of research within the field of loop parallelism. The optimization of these loops is a difficult task, since there must be a synchronization between threads/cores that execute the iterations of the loop. In addition, these loops can not be simply parallelized with the traditional techniques mentioned above and can not simply have distributed iterations in threads/cores. This is because threads do not have a guaranteed execution order.

Figure 1.1 shows an example of a DOACROSS loop. The instruction in line 2 calculates the value of $A[i]$, but the value of $A[i-1]$, calculated on the previous iteration, is needed. Also, at instruction in line 3, j value is updated every iteration, using value of previous iteration too. However, instruction in line 4, do not have a dependence on previous iterations of the loop, thus, can be executed in parallel once the value of $A[i]$ and k are calculated in the same iteration.

Using this information about the loop, we can separate the loop body into two components: a serial component, with instructions from lines 2 and 3 and a parallel component, with instruction from line 4.

```
1  For (i = 1; i < N; i++) {
2    A[i] = A[i - 1] * i;
3    k = f(A, j++);
4    B[i] = A[i] + k;
5  }
6
```

Figure 1.1: Sequential example of DOACROSS loop

The dependence graph on Figure 1.2 shows the two components and its dependences. The component $C1$, with instructions of line 2 and 3, has a self-dependence with previous

iteration, and component $C2$ with instruction of line 4 has an input dependence from component $C1$ (values of $A[i]$ and k).

Component $C1$ must be serialized for correct data output, however, component $C2$ can be executed in parallel after the values of $A[i]$ and k are calculated for that iteration. The DOACROSS parallelization techniques use these components to extract parallelism. Depending on the choice of parallelization technique, while iteration i of component $C1$ is executed, other threads/cores can execute the iteration $i-1$ of component $C2$ or iteration $i+1$ of component $C1$.

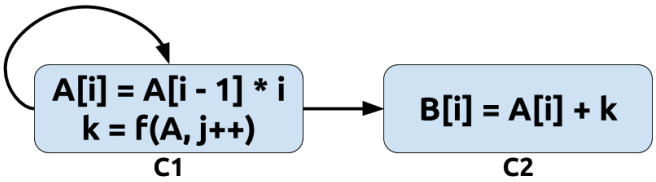

Figure 1.2: Data dependence graph for the example on Figure 1.1

The naive parallelization of this loop has no performance gain when compared to serial version since the whole iteration is serialized. In fact, in most cases, there is a performance loss because of the overhead introduced by synchronization between threads/cores.

This serialization is shown in Figure 1.3 for 2 threads/cores. Each iteration (made of components $C1$ and $C2$) is executed entirely before the other thread execute the next iteration and there is a communication arrow to indicate the end of computation for each iteration. This example can easily be implemented with a barrier or a mutual exclusion with an additional feature to control the iterations order.

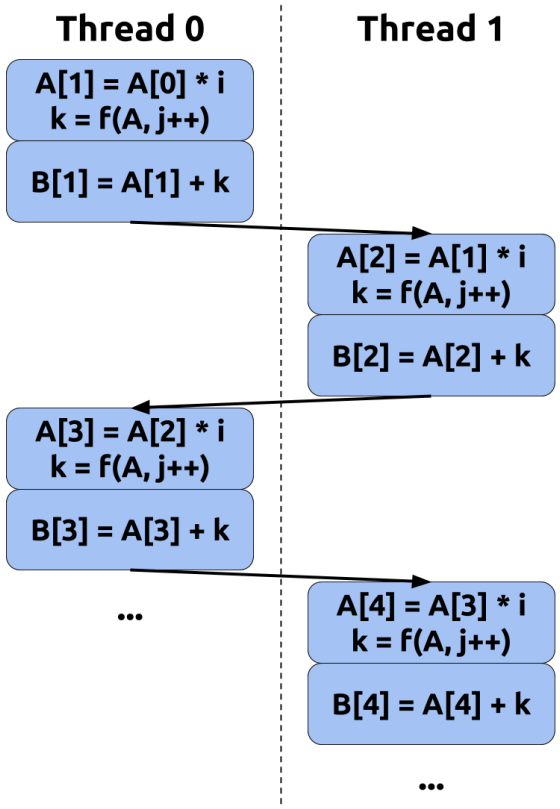

Figure 1.3: DOACROSS loop serialized for execution on 2 threads/cores

1.3 Loop-Carried Ratio

Loop-carried ratio (LCR) represents the percentage of loop iterations that has some dependence from a previous iteration of the loop, because of that, some loops cannot be simply paralelized because of these dependences. However, depending on loop structure, the loop can be classified based on loop-carried ratio type.

Figure 1.4 shows how loops are categorized using the dependence analysis, this analysis can be done during compile time (static) or during execution time (dynamic).

To ease notation understanding, when a loop can be statically analyzed, loop-carried ratio is called loop-carried frequency (LCF), because every execution of loop will materialize the same dependences. Then, we can categorize loops that can be statically analyzed into two categories: DOALL and DOACROSS. DOALL are loops without any loop-carried dependence, this happens when $LCF = 0\%$. Otherwise, when a loop has a loop-carried dependence is called DOACROSS, in other words, a loop is DOACROSS when $LCF \neq 0\%$

On the other hand, when a loop cannot be statically analyzed, compiler cannot assure that dependences will occur or not. In these cases, loop-carried ratio is called loop-carried probability (LCP) because some iterations may have dependences or not, depending on several factors, including program input. We can also categorize these loops into two categories: D-DOALL and D-DOACROSS. D-DOALL or Dynamic DOALL are loops that does not materialize any loop-carried dependence given an execution instance, when

$LCP = 0\%$. However, when a dependence occurs at runtime, loop is categorized as D-DOACROSS or Dynamic DOACROSS. This happens when $LCP \neq 0\%$

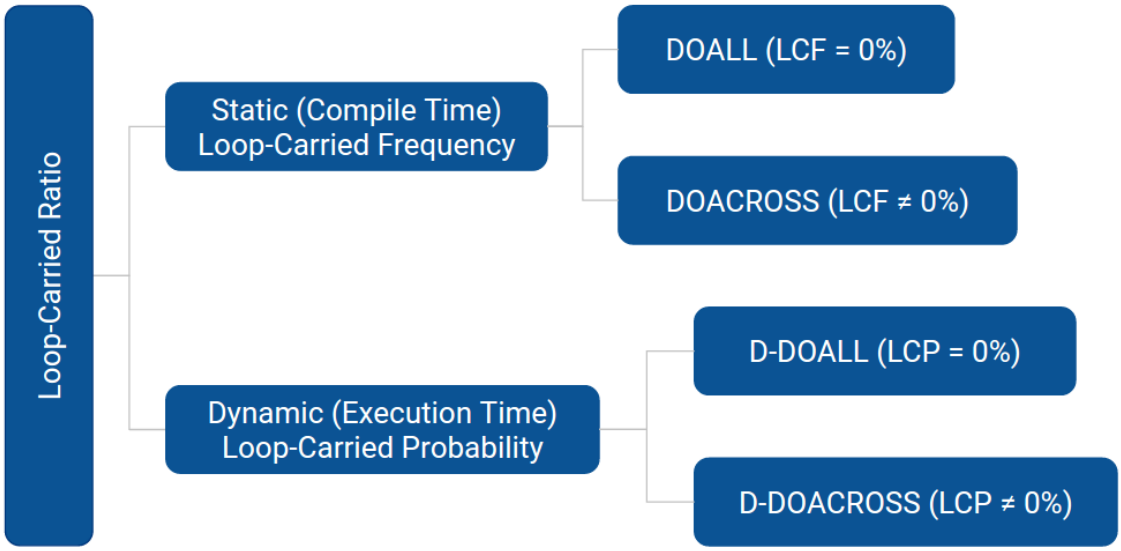

Figure 1.4: BDX Loop Execution

This dissertation will use the following notation:

1. DOALL: Loop can be statically analyzed and $LCF = 0\%$
2. DOACROSS: Loop can be statically analyzed and $LCF \neq 0\%$
3. MAY DOACROSS: Loop cannot be statically analyzed

Case ① is trivial and parallelization is a direct process. Iterations are simply distributed between threads and there is no control or synchronization between them because there is no dependence. Is this case, execution order does not change output generated by loop. For example:

```
1  For (i = 1; i < N; i++) {
2    A[i] = A[i] * i;
3    B[i] = A[i] + k;
4  }
5
```

Figure 1.5: Example of DOALL loop

In case ②, most compilers do not automatically attempt to parallelize the loop because there are dependencies between iterations. In order to parallelize this type of loop, we must use one of the mentioned algorithms (DOAX, BDX or DSWP) to obtain performance and, due to the dependencies between iterations, at least part of the loop must be serialized. So the idea of separating the loop into components, inserting the synchronization only

into serial components, and executing the other components in parallel can be a good way of extracting the parallelism of those loops. For example:

```
1  For (i = 1; i < N; i++) {
2    A[i] = A[i - 1] * i;
3    B[i] = A[i] + k;
4  }
5
```

Figure 1.6: Example of DOACROSS loop

In this case, on line 2, `A[i]` depends on value of `A[i-1]`, which was calculated by previous iteration and by induction, every iteration depends on data calculated by previous iteration. Because of this, 100% of iterations have dependence, making this loop a DOACROSS loop with 100% of loop-carried frequency.

Case ③ is more complicated because, depending on the input instance, this can be a D-DOALL loop or a D-DOACROSS loop. The use of the algorithms mentioned in this type of loop may not be able to extract the best parallelism of the loop, since the synchronization will be done for each iteration, even if this iteration does not have a loop-carried dependency. For this case, the best solution is to speculate the iterations of the loop using transactions. An algorithm called TLS [2] can be used in these loops because the iterations are executed in parallel and, if there is a dependency violation caused by an iteration, the transaction manager causes that transaction to be aborted and, after some time, this iteration is executed again. For example:

```
1  For (i = 1; i < N; i++) {
2    if(condition)
3      A[i] = A[i - 1] * i;
4    else
5      A[i] = A[i] * i;
6    B[i] = A[i] + k;
7  }
8
```

Figure 1.7: Example of MAY DOACROSS loop

In this case, on line 3, `A[i]` depends on value of `A[i-1]`, which was calculated by previous iteration but on line 5 only depends on `A[i]` and there is an `if` surrounding these instructions, which means that sometimes line 3 will be executed otherwise line 5 will be executed, depending on condition. Because of this, some iterations have loop-carried dependence and others do not, the proportion of iterations with dependence is the loop-carried probability for that input instance.

1.4 Parallelization Techniques

1.4.1 DOAX

The DOAX parallelization proposed by R. Cytron [9] distribute the loop iterations into threads and each thread need to communicate with the other threads to respect the data dependences order. Each thread executes the whole iteration instead of breaking the loop into serial and parallel components. The idea is trying to execute the maximum number of iterations in different threads at the same time. This method is very similar to a pipeline. Figure 1.8 shows the iterations distributed to the threads and the communication between them. This example shows the dependence of the $A[i]$ variable, which is updated every iteration of the loop in the component $C1$ and is read in the component $C1$ of the next iteration.

This method has some limitations and introduces communication overhead for each iteration. Each arrow in Figure 1.8 is a communication between threads. As mentioned before, these communications have a high latency and there is an overhead for managing this synchronization between the threads.

There is a DOACROSS based technique called Post/Wait, proposed by P. Unnikrishnan and J. Shirako [22] which uses two functions to specify the dependences for a serial component. These functions act as a barrier to synchronize the threads execution order.

The *Wait* function, located at the beginning of the serial component, receives an argument with the iteration indexes of the loop-carried dependence in the serial component. This function has a busy waiting loop and waits for the iteration dependences to be fulfilled before the execution of the program continues.

The *Post* function, located at the end of the serial component, receives the current iteration indexes as an argument and updates the dependences array to indicate that the current iteration has finished the serial component.

The execution is similar to that shown in Figure 1.8, where each thread executes a whole iteration. However, each thread waits only for the component $C1$ of the previous iteration to be completed to start the computation of the component $C1$ of the current iteration. Moreover, as soon as thread finishes the execution of component $C1$, it continues the execution of the component $C2$ of the same iteration.

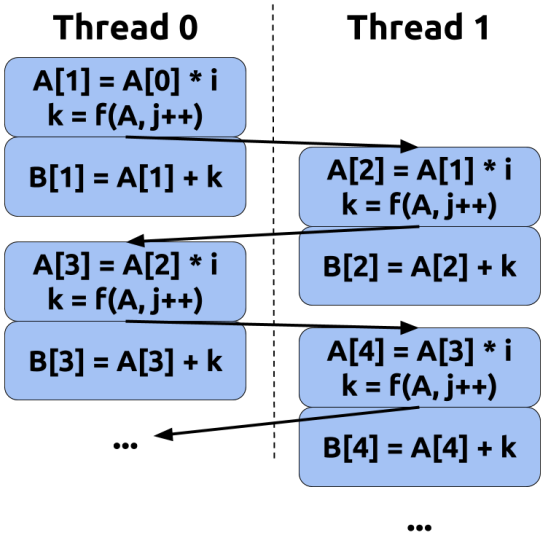

Figure 1.8: DOAX Loop Execution

The Post/Wait technique was also proposed as an OpenMP extension [23] by the same authors and was incorporated in OpenMP 4.1 [3] as the *ordered* directive with the *depend* clause to specify the loop-carried dependences. Figure 1.9 shows an example of this new OpenMP construct. The serial component are the instructions of lines 4 and 5, delimited by the *ordered* directives on lines 3 and 6. The parallel component is the instruction on line 7 since it is not delimited by any *ordered* directive.

```
1  #pragma omp parallel for ordered(1)
2  For (i = 1; i < N; i++) {
3    #pragma omp ordered depend(sink: i-1)
4    A[i] = A[i - 1] * i;
5    k = f(A, j++);
6    #pragma omp ordered depend(source)
7    B[i] = A[i] + k;
8  }
9
```

Figure 1.9: Example of DOACROSS loop with the *ordered* directive

In this example, the *depend* clause with the *sink* dependence type indicates an input dependence, so the threads can only execute the instruction on line 4 only if the iteration $i - 1$ has already been executed. The *depend* clause with the *source* dependence type indicates the end of the serialized region and is used to update the dependence vector, which is used to control the execution between the threads. The *ordered* clause in the *parallel for* construct must have a parameter. This parameter is the number of nested loops associated with the *parallel for*, this information is needed to create the dependence vectors and knows how many iteration indexes are expected on the *depend* clause with the *sink* dependence type.

1.4.2 DSWP

Proposed by Ram Rangan [21], the DSWP (Decoupled Software Pipeline) breaks the sequential and the parallel components of the loop and distribute the components between the threads instead of distributing the iterations. Each thread executes all the iterations of each component, removing the need for synchronizing the loop-carried dependences. However, there must be a synchronization for the dependences between the components. Figure 1.10 shows the same example in Figure 1.1 using the DSWP method.

In DSWP, the parallelism gain comes from the pipelined execution of the components and the fact that each thread executes only one component of the loop, which causes a better caching and less false sharing problems. However, there is a communication queue used for every iteration of the loop and this causes a great overhead.

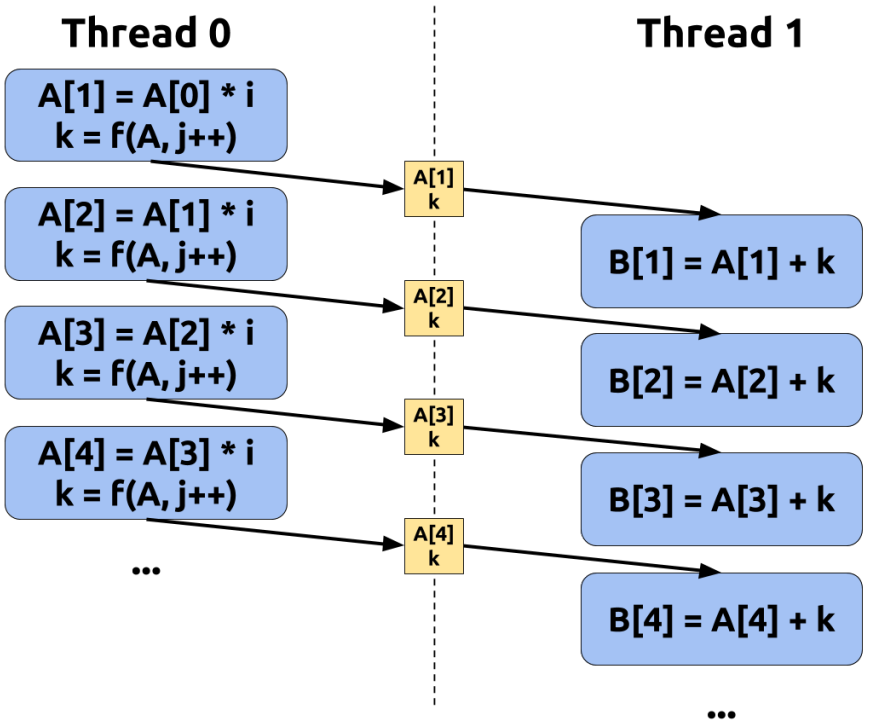

Figure 1.10: DSWP Loop Execution

In this example, each iteration has the component $C1$ generating the data for $A[i]$ and then using the queue between the threads for communicating this data to the other threads. When the other threads detect that the queue has a job finished, get this data from the queue and execute the component $C2$.

There is another version of the DSWP called PS-DSWP. This version tries to spread the parallel regions between the threads/cores, to have even more parallelism than the simple pipeline idea from the original DSWP.

1.4.3 BDX

The BDX, proposed by Divino César S. Lucas [1], is a generalization of the DOACROSS method. It uses a buffer to execute several iterations of the loop at once before doing the inter-thread communication. This buffering process reduces the number of communications between the threads and consequently reduces the number of communications between the iterations. The example on Figure 1.11 shows the same loop of Figure 1.1, however, when compared with the DOACROSS and DSWP methods, there are fewer arrows of inter-thread communication.

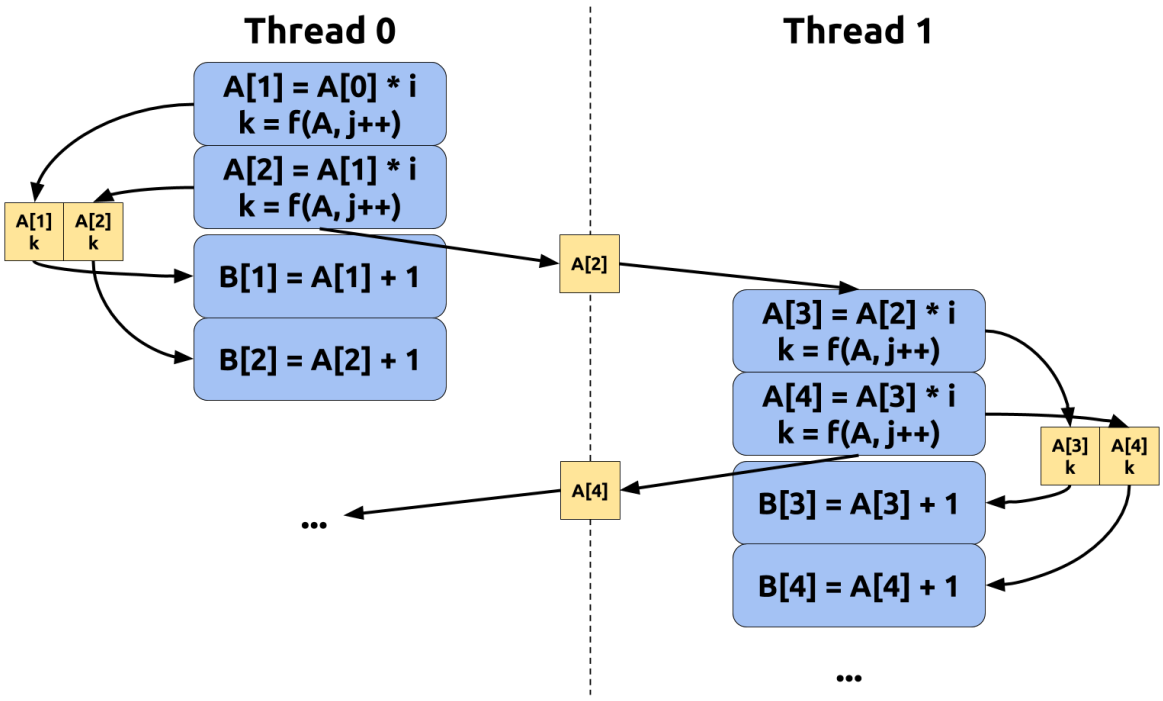

Figure 1.11: BDX Loop Execution

The BDX algorithm creates a buffer for storing the local results. In the example, the values of $A[1]$ and $A[2]$ are buffered by the component $C1$ and used by the component $C2$. For the inter-thread communication, there is a queue between the threads, in the example, the values of $A[2]$ and $A[4]$ are passed by this queue because different threads are executing different iterations of the same component.

Just like the DSWP, there is the parallel version of the BDX, called PS-BDX. This parallel version tries to spread the parallel regions between the threads/cores, to have even more parallelism than the original BDX.

1.4.4 TLS

The technique that we call TLS for easier identification, was proposed by Salamanca *et al.* [2] and uses coarse-grained thread-level speculation (TLS) in order to try to obtain parallelism from MAY DOACROSS loops. This technique creates hardware transactional memory (HTM) transactions with the whole loop body and try to execute them in parallel.

Loop-carried dependences in this case will cause to transactions to conflict with each other and the iteration which such dependence happens has a transaction abort and is re-executed.

Intuition tells that this technique must be better when the number of iterations with loop-carried dependences is low, because transaction aborts have a high overhead.

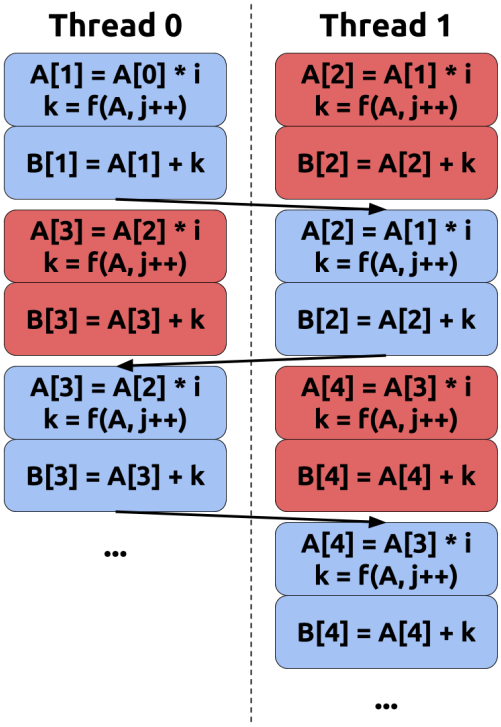

Figure 1.12: TLS Loop Execution

Figure 1.12 shows execution flow for loop of Figure 1.1, which is a DOACROSS loop with $LCF = 100\%$, which means that this technique will try to execute several iterations at same time but there is a dependence for every iteration. Because of this dependence, lots of conflicts will occur, causing transactions to abort at every iteration.

In Figure 1.12, these aborts are represented by red components blocks, while thread 0 executes statement with $A[1]$, thread 1 tries to execute statement with $A[2]$. However, a conflict occurs because thread 1 needs the value of $A[1]$ before and thread 0 writes its value after, causing a transaction abort of whole iteration for thread 1. Then, transaction has a rollback and try to execute again, but this time, thread 0 finished executing its iteration and dependence is completed, allowing for thread 1 to continue executing.

Chapter 2

Hypotheses

The idea of implementing this new clause *use* is to have an easy way for the programmer to simply annotate loop components using *ordered* directive and transparently transform the original source code into a new one using the selected parallelization technique. This new clause is responsible for these transformations during code generation compilation step.

2.1 The *'use'* clause

The proposed clause must be used along with *ordered* clause to separate sequential and parallel components of the loop body. The idea of using this clause is to improve *ordered* funcionality to expand parallelization techniques possibilities for the programmer to choose. The implemented clause *use* receives two arguments: parallelization technique and strip size.

Parallelization technique argument can either be DOAX, BDX or PSBDX and strip size is used only by BDX or PSBDX to adjust batch size. If DOAX is chosen as parallelization technique, then strip size value is ignored.

Another clause was expanded to implement a needed feature in OpenMP, parameter *'var'* in *depend* clause. This implementation is needed to programmer be able to mark which scalar variables are loop-carried. This clause will be explained in Section 2.5.

```
1  #pragma omp parallel for ordered(1) use(doax|bdx|psbdx, strip)
2  For (i = 1; i < N; i++) {
3    #pragma omp ordered depend(sink: i-1) depend(var: j)
4    A[i] = A[i - 1] * i;
5    k = f(A, j++);
6    #pragma omp ordered depend(source)
7    B[i] = A[i] + k;
8  }
9
```

Figure 2.1: Example of DOACROSS loop with the *use* clause

2.2 The *'depend(var:...)'* clause

2.3 Clause Limitations

The *use* clause is very versatile and programmers can parallelize loops using specified techniques easily than implementing them manually. However, this clause have some limitations that prevent from using it to parallelize all types of loops by simply adding the clause in OpenMP pragma. Most limitations are because OpenMP *ordered* is used to separate loop components, since OpenMP *ordered* has its limitations by itself. Limitations for using this clause are show in list below:

- Loop Exit Point: limitation imposed by OpenMP, any statement that is able to jump to another statement outside the loop violates OpenMP rules and does not even compile correctly. For example when there is a *break* statement inside loop with OpenMP pragma. Even using *cancel* OpenMP pragma does not compile, because it cannot be used along with *ordered* clause

- Conditional Dependence: another limitation from *ordered* clause, example of Listing 1.7 demonstrates this case. Line 3 has a dependence of a previous iteration while Line 5 does not, however, *ordered* directives with *depend* clauses must be surrounding the whole *if/else*, because surrounding only statement of Line 3, will cause to *ordered* synchronization to fail if *else* condition is taken. In this case, a deadlock occurs because dependence of Line 3 might be not synchronized at Line 5, however, Line 3 will still busy wait for its dependence.

- DOAX Components: because of how *ordered* it is implemented in OpenMP Runtime, it can have only one sequential component in loop body. This happens because of a internally allocated dependence array, which keeps record of completed iterations from sequential component. This array is not reset after sequential component finishes its iterations, which causes incorrect execution order if there is at least another sequential component. However, this limitation applies only to DOAX technique. Loops marked with more than one sequential component can still be parallelized using BDX or PSBDX.

- Batch Size: for now, there is a limitation for batch size value, it must be a divisible value of the number of iterations of the loop. This occurs because if there is a remainder, loop control gets lost and executes more iterations than original loop. This is easy to solve by spliting the loop into two other loops, the first one with a number of iterations that is a multiple value of batch size and the second one with a number of iterations from remainder of division. However, this is not implemented in this clause yet.

2.4 DOAX Transformations

Since DOAX algorithm is already implemented in OpenMP by using *ordered* construct, *use* clause handle code generation by simply removing *use* from *parallel for* pragma and

letting OpenMP code generation flow normally.

OpenMP uses POST/WAIT runtime function calls to implement DOAX in OpenMP using *ordered* directive and *depend* clause to insert these function calls. Clause *depend(sink: ...)* is replaced by a function call to the proposed *wait* function, which is basically a busy waiting that compares current loop index with specified dependence index in clause argument. Clause *depend(source)* is replaced by a function call to the proposed *post* function, which is responsible for updating dependences array to indicate that current iteration has been completed and dependant iterations can be executed. Figure 2.2 shows an example of the transformed source of example in figure 1.9.

```
1  #pragma omp parallel for ordered(1)
2  For (i = 1; i < N; i++) {
3    @WAIT(i-1);@
4    A[i] = A[i - 1] * i;
5    k = f(A, j++);
6    @POST(i);@
7    B[i] = A[i] + k;
8  }
9
```

Figure 2.2: *ordered* directive replaced by POST/WAIT function calls

2.5 BDX Transformations

Given an annotated loop with *ordered* directives for the serial component, we can separate all components by analyzing the pragma annotations. Whenever we find an *ordered* directive with *depend(sink:...)* clause, it is the start of a serial component and whenever we find an *ordered* directive with *depend(source)* clause, it is the end of a serial component. All statements between these two clauses, in that order, are part of a serial component, otherwise they are part of a parallel component. An example is shown in figure 2.1.

With these components, it is possible to start the loop body transformation. However, this transformation must be done in two steps: at the AST level and in the LLVM IR level, because some information needed for transformation are present only in AST structure or in LLVM IR.

For example, at the AST level we can find information about pragmas and loop structure and in the LLVM IR level we can find dependences between loop components.

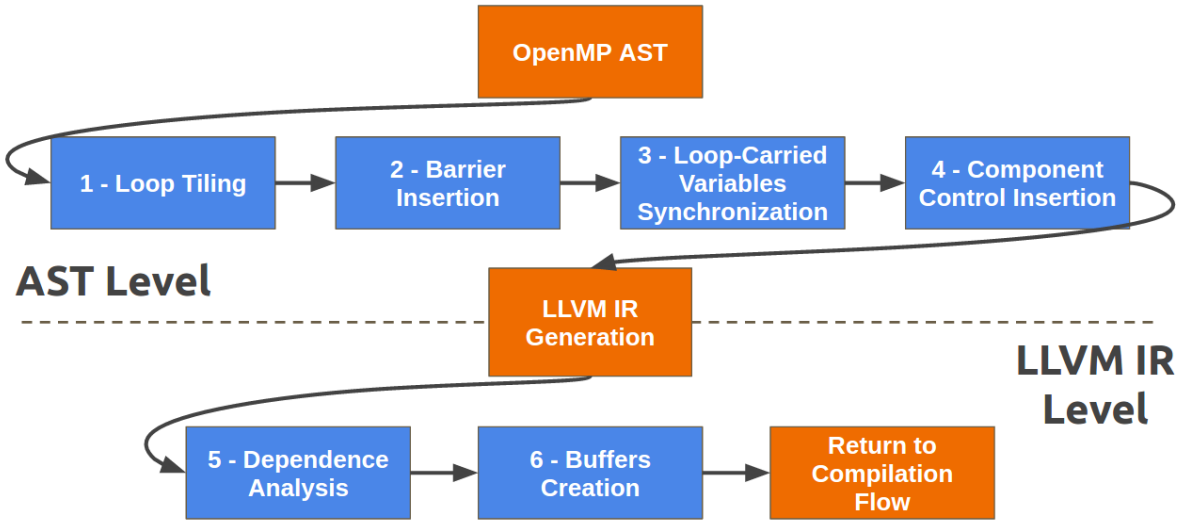

Figure 2.3: BDX Loop transformation flow

These two steps were incorporated into Clang's execution flow. The first step is done before OpenMP code generation, which is responsible for receiving an AST and generating the LLVM IR. The second step is done as a first step of optimization, receiving a raw LLVM IR (that is, before any LLVM optimization) and generating a transformed LLVM IR. Figure 2.3 shows a diagram of loop transformations for AST Level (steps 1, 2, 3 and 4) and LLVM IR Level (steps 5 and 6).

```
1  #pragma omp parallel for ordered(1) use(psbdx, 10)
2  For (i = 1; i < N; i++) {
3    #pragma omp ordered depend(sink: i-1) depend(var: j)
4    A[i] = A[i - 1] * i;
5    k = f(A, j++);
6    #pragma omp ordered depend(source)
7    B[i] = A[i] + k;
8  }
9
```

Figure 2.4: Example of DOACROSS loop with the *use* clause

For this example, statements at lines 3 to 6, which represent a sequential component, will be called $C1$ and statement at line 7, which is parallel, will be called $C2$.

2.5.1 AST Transformation

With these components, it is possible to start the transformation of the loop body. The outer loop transformation is shown in Listing 2.5. A loop-carried variable must be synchronized between threads, then, a global variable *loop_carried_j* is created for this synchronization and is initialized before loop starts. Another global variable is created to adjust batch size in LLVM IR, then, variable *__bdx_buffer_size* is initialized with *use* strip size value before loop starts.

```
1  loop_carried_j = j;
2  __bdx_buffer_size = 10;
3  #pragma omp parallel for schedule(static, 1) private(j)
4  for (i = 1; i < N; i = i + 10) {
5    C1
6    C2
7  }
8
```

Figure 2.5: Example of DOACROSS loop transformed by clause *use*

Then, each loop component is also transformed separately.

First step is the insertion of two function calls surrounding each component's body: $\_\_bdx\_stage\_begin\_\_$ and $\_\_bdx\_stage\_end\_\_$. These functions are needed to pass some information from AST level into LLVM IR level, because these information are needed for dependence analysis and buffer creation but are not explicitly available at LLVM IR level. Function $\_\_bdx\_stage\_begin\_\_$ is inserted before first statement of each component and is used to delimit component start, because this information is needed for posterior analysis and is not available at LLVM IR level. Also, this function has an argument which is the inner loop iteration variable, again, in order to ease posterior analysis. Function $\_\_bdx\_stage\_end\_\_$ is inserted after last statement of loop component to simply delimit it from other components at LLVM IR level. Listings 2.6 and 2.7 show these functions call insertion for components C1 and C2.

```
1  __bdx_stage_begin__(i);
2  A[i] = A[i - 1] * i;
3  k = f(A, j++);
4  __bdx_stage_end__();
5
```

Figure 2.6: C1 with function calls delimiters

```
1  __bdx_stage_begin__(i);
2  B[i] = A[i] + k;
3  __bdx_stage_end__();
4
```

Figure 2.7: C2 with function calls delimiters

The next step is to apply a loop tiling because each component must run a number of iterations before synchronizing and fill buffers to communicate dependences between components. To do this, a new *for* loop is created containing the component instructions as loop body, as shown in Listings 2.8 and 2.9. Also, all uses of iteration variable inside component must be replaced by this newly created loop iteration variable and the increment of original *for* loop must be updated to represent the size of tiling, as shown in Line 4 of Listing 2.5. This tiling transformation is represented by Step 1 of Figure 2.3.

```
1  for (int ibdx = i; ibdx < i + 10; ibdx++) {
2    __bdx_stage_begin__(ibdx);
3    A[ibdx] = A[ibdx - 1] * ibdx;
4    k = f(A, j++);
5    __bdx_stage_end__();
6  }
7
```

Figure 2.8: Loop tiling of C1

```
1  for (int ibdx = i; ibdx < i + 10; ibdx++) {
2    __bdx_stage_begin__(ibdx);
3    B[ibdx] = A[ibdx] + k;
4    __bdx_stage_end__();
5  }
6
```

Figure 2.9: Loop tiling of C2

Another transformation needed at this level is to synchronize scalar loop-carried variables if component is sequential, this is done by *depend(var:...)* clause and this transformation includes synchronization of loop-carried scalar variables before and after inner loop. This is done by creating a new global variable for each loop-carried variable and this global variable is is responsible for carrying the value between threads, as shown in Listing 2.10. Line 1 shows input synchronization, where global variable value is copied into a private copy of loop-carried variable and line 8 shows local variable value being updated into global variable. The loop-carried variables synchronization is represented by Step 3 of Figure 2.3.

```
1  j = __bdx_loop_carried_j;
2  for (int ibdx = i; ibdx < i + 10; ibdx++) {
3    __bdx_stage_begin__(ibdx);
4    A[ibdx] = A[ibdx - 1] * ibdx;
5    k = f(A, j++);
6    __bdx_stage_end__();
7  }
8  __bdx_loop_carried_j = j;
9
```

Figure 2.10: Loop-carried variable synchronization of C1

Then, also for sequential components, there must be included a barrier before the tiling loop, in order to ensure the synchronization between threads. This synchronization is needed for sequential components to execute in the correct iterations order. The barrier is just a simple busy waiting loop that checks the value of a shared variable, when this variable has the same value as thread ID, that thread can execute that component, as shown in line 1 of Listing 2.11. Also, synchronization after inner loop is needed to update

the value of the shared variable used for control synchronization, as shown in line 10 of Listing 2.11. This barrier insertion is represented by Step 2 of Figure 2.3.

```
1  while (__bdx_flags != omp_get_thread_num());
2  j = __bdx_loop_carried_j;
3  for (int ibdx = i; ibdx < i + 10; ibdx++) {
4    __bdx_stage_begin__(ibdx);
5    A[ibdx] = A[ibdx - 1] * ibdx;
6    k = f(A, j++);
7    __bdx_stage_end__();
8  }
9  __bdx_loop_carried_j = j;
10 __bdx_flags = (__bdx_flags + 1) % omp_get_num_threads();
11
```

Figure 2.11: Control synchronization of C1

In order to make *use* implementation more flexible for future implementation of other parallelization techniques, an *if* is included surrounding each component, as shown in Line 1 of Listing 2.12 and Line 1 of Listing 2.13. This decision had in mind that in some parallelization techniques not all threads execute all components of the loop, i.e. DSWP, and this function *__bdx_cond__* can be used by each thread choose which components must be executed.

```
1  if (__bdx_cond__(0)) {
2    while (__bdx_flags != omp_get_thread_num());
3    j = __bdx_loop_carried_j;
4    for (int ibdx = i; ibdx < i + 10; ibdx++) {
5      __bdx_stage_begin__(ibdx);
6      A[ibdx] = A[ibdx - 1] * ibdx;
7      k = f(A, j++);
8      __bdx_stage_end__();
9    }
10   __bdx_loop_carried_j = j;
11   __bdx_flags = (__bdx_flags + 1) % omp_get_num_threads();
12 }
13
```

Figure 2.12: C1 transformed by clause *use*

```
1  if (__bdx_cond__(1)) {
2    for (int ibdx = i; ibdx < i + 10; ibdx++) {
3      __bdx_stage_begin__(ibdx);
4      B[ibdx] = A[ibdx] + k;
5      __bdx_stage_end__();
6    }
7  }
8
```

Figure 2.13: C2 transformed by clause *use*

In this example, the buffers have not yet been created because all transformations up to this step are done at the AST level and the buffers are created at the IR level.

All of these transformations are done at the Abstract Syntax Tree (AST) level because some necessary information (ie, OpenMP pragmas, loop iterator) are available only, or are much easier to obtain, at this stage of the compilation process.

With these modifications made at the AST level, code generation is ready to create the LLVM IR. To do this, our method simply generates all modifications in AST and allows OpenMP to handle code generation. This is much easier because there is no need to deal with variable scopes since OpenMP already does it.

2.5.2 LLVM IR Transformation

After the AST transformations, the buffers have yet to be created for the algorithm to work. This is because the dependencies between the components in the same thread need to be communicated. For this, one should have a simple dependency analysis that decides whether a buffer for a given variable should be created or not.

For this analysis, we faced with some questions regarding code optimization. Some references to the variables were optimized so that we lost the references between the components and, with this, the buffers were not created correctly. The solution was to encapsulate our analysis step within Clang's optimization pass flow, making dependence analysis pass to run as early as possible during the optimization process.

With generated LLVM IR, as shown in Listings 2.15 and 2.16, a dependence analysis is needed in order to detect which variables must be buffered for the BDX to work. This analysis just checks when a variable is written in a component and read in another component, if this happens, there must be a buffer to communicate this variable between those components. Then, all uses of that variable are replaced by buffer access indexed by batch iterator.

This dependence analysis is done over LLVM IR because is much easier to be done on this level due to OpenMP code generation and variable privatizations.

```
1  For each component c
2    For each instruction i on component c
3      If instruction i is a store or a temporary attribution
4        If i is a store
5          val <- store address
6        Else
7          val <- i
8        Find all uses of val
9        For each use u of val
10         If u and i are in different components
11           val needs a buffer
12
```

Figure 2.14: Buffer Detection Algorithm

Dependence analysis algortihm, as shown in figure 2.14 is very simple and checks if there is a value assignment in a loop component that is used in another loop component, if that is the case, a buffer must be created in order to communicate such dependence. This pass has two steps: detection and creation. Detection step (Step 5 of Figure 2.3) analyze entirely the LLVM IR and decides which variables need a buffer, however, does not create buffers. Once all buffers have been detected, creation step (Step 6 of Figure 2.3) receives information from detection step and buffers are created by allocating arrays and all uses of the variables are replaced by buffer access indexed with batch loop index.

```
1  ...
2  %121 = call i32 @f(i32* %36, i32 %119)
3  store i32 %121, i32* %37, align 4, !tbaa !1
4  call void @__bdx_stage_end__()
5    ...
6
```

Figure 2.15: Intermediary representation of first loop component

```
1  ...
2  %149 = load i32, i32* %148, align 4, !tbaa !1
3  %150 = load i32, i32* %37, align 4, !tbaa !1
4  %151 = add nsw i32 %149, %150
5  ...
6
```

Figure 2.16: Intermediary representation of second loop component

For example, at line 3 of figure 2.15, there is a store into an address stored in *%37*, which represents assignment of variable k at line 5 of figure 2.4. Then, *%37* is used to load variable value at line 3 of figure 2.16. Since these two uses of *%37* are in different components and one of them modifies the variable value, there must be a communication between these components, so, a buffer must be created to do this communication.

These buffers are created only once for the loop and are in local memory for each thread, that is, buffers are created by each thread, before starts executing loop body and are deallocated after loop body finished its execution.

Abstracting buffer creation to a higher level, Listing 2.17 shows buffer allocation in line 3 and buffer deallocation at line 9. Listings 2.18 and 2.19 show how this buffer is used, replacing all uses of original variable that need to be communicated by a buffer indexed with the inner loop iteration variable.

```
1  loop_carried_j = j;
2  __bdx_buffer_size = 10;
3  buffer_k = (int *)malloc(__bdx_buffer_size * sizeof(int));
4  #pragma omp parallel for schedule(static, 1) private(j)
5  for (i = 1; i < N; i = i + 10) {
6    C1
7    C2
8  }
9  free(buffer_k);
10
```

Figure 2.17: Example of DOACROSS loop transformed by clause *use*

```
1  if (__bdx_cond__(0)) {
2    while (__bdx_flags != omp_get_thread_num());
3    j = __bdx_loop_carried_j;
4    for (int ibdx = i; ibdx < i + 10; ibdx++) {
5      __bdx_stage_begin__(ibdx);
6      A[ibdx] = A[ibdx - 1] * ibdx;
7      buffer_k[ibdx] = f(A, j++);
8      __bdx_stage_end__();
9    }
10   __bdx_loop_carried_j = j;
11   __bdx_flags = (__bdx_flags + 1) % omp_get_num_threads();
12 }
13
```

Figure 2.18: C1 transformed by clause *use*

```
1  if (__bdx_cond__(1)) {
2    for (int ibdx = i; ibdx < i + 10; ibdx++) {
3      __bdx_stage_begin__(ibdx);
4      B[ibdx] = A[ibdx] + buffer_k[ibdx];
5      __bdx_stage_end__();
6    }
7  }
8
```

Figure 2.19: C2 transformed by clause *use*

After these transformations, BDX implementation is completed, with all needed control and synchronization statements, so this loop is transformed into a parallel version that uses BDX technique to parallelize it.

Chapter 3

Related Work

HELIX is a compiler that has previously delivered good speed-ups for irregular programs on a six-core Intel i7 machine [7]. HELIX parallelizes loops in sequential programs, distributing the iterations to available cores in a round-robin fashion. To preserve dependences between iterations or (may) loop-carried dependences, HELIX creates *sequential segments* that are subsets of iterations whose execution on cores must respect the loop-iteration order of the sequential program. These sequential segments correspond to SCCs in a *Data-Dependence Graph (DDG)* that have at least one loop-carried dependence. An SCC formed by a single node with no loop-carried dependences is considered a *parallel segment* that does not need synchronization. In contrast, this paper proposes a new OpenMP clause that enables programmers to annotate sequential segments that should synchronize or speculate their iterations.

Decoupled Software Pipeline (DSWP) [17, 21] is a Pipelined Multithreading algorithm for parallelizing loops with loop-carried dependences. It transforms the loop body in a way that it becomes a computational pipeline where each thread executes a different stage of the loop and data dependencies flow only in one direction among the stages. DSWP proposes the use of inter-core/thread queues to communicate loop-independent dependences between stages and decouple their execution. By using queues as a communication mechanism between stages, DSWP becomes quite resilient to communication latency variations. However, the complexity of managing the queues makes difficult to achieve speedups with DSWP in many cases [1, 19].

Batched DOACROSS [1] is a generalization of the idea behind DOAX [9], which separates loop body into sequential and parallel components and executes these components as a pipeline, respecting dependences between iterations and components. However, this technique uses buffers to do a loop tiling and transform loop components into batches, a batch is executed by a thread entirely before next batch can be executed by another thread. This batching process results in less synchronizations between threads and better memory locality in some cases. DOAX is a particular case of BDX when batch size is only of one iteration.

Salamanca *et al.* use coarse-grained TLS to speculate a (strip-mined) whole iteration and perform conflict detection and resolution at the end of the iteration to detect RAW dependence violations [2]. They describe how speculation support designed for HTM can also be used to implement TLS [2]. They focused their work on the impact of false sharing

and the importance of judicious strip mining and privatization to achieve performance. They provide a detailed description of the additional software support that is necessary for both the Intel Core and the IBM POWER8 architectures to enable TLS. Moreover, in [6] they carefully evaluate the performance of TLS on Intel Core and POWER8 using 22 loops from `cBench` focusing on the characterization of the loops. This paper extends [6] by providing programmers with an OpenMP based approach to automatically run TLS on DOACROSS loops.

Post/Wait [22] technique is a simple implementation of DOAX [9] technique, which uses function calls and a runtime library to create DOAX pipeline. In this technique, sequential components are surrounded by two functions responsible for synchronizing loop execution. A `Wait` function is a simple barrier which checks iterations dependences and only allow one thread to execute sequential components at same time to assure correct iterations execution order. A `Post` function is simply the end of the barrier and is responsible for updating the variable being used to control the barrier. Since it's implementation is very simple, Post/Wait was incorporated in OpenMP 4.1 [23] as `ordered` directive specified with `depend` clause.

For this dissertation, the concept of SCCs used by HELIX and DSWP is essential for selecting loop components correctly, Batched DOACROSS is used within our implementation as a doacross loop parallelization technique, Salamanca *et al.* implementation and Post/Wait are used as comparison techniques to validate performance in experimental results. Also, OpenMP Post/Wait implementation structure is used to ease programmer's annotation of loops.

Chapter 4

Results

4.1 Methodology

The clause *use* has been implemented using OpenMP infrastructure of LLVM/Clang 4.0 and has been evaluated using well-known SPEC CPU 2006, StarBench, and cBench benchmark suites as shown in Table 4.1. This clause has been tested in a quad-core Intel Haswell TSX machine and Intel OpenMP Library Version 20160808, also each benchmark has been executed ten times and average values were calculated.

All benchmarks were compiled using -O3 optimization flag and to guarantee that each software thread is bound to one hardware thread (core) and to decrease the number of conflict aborts, the environment variable KMP_AFFINITY is set to granularity=fine,scatter for TLS.

4.2 Experimental Results

Parallelized loops from each benchmark are detailed in Table 4.1, this table shows some informations about the loops: source location (benchmark, file, line number and function), loop coverage and number of times the loop is executed during program execution.

Table 4.2 shows detailed information about execution parameters and obtained results for all benchmarks, i.e. loop execution time and speedup. Loops were split into two classes: (a) those with low LCP (Loops A, E, H, I, J and, V from `cBench`) and those with high LCP (Bzip, Rota, Rgb, RotC and, Ray).

Table 4.2 also shows, in the Components column, the loop components: number of components and type (S-Sequential, P-Parallel). Information for Bzip, parallelized with DOAX, was omitted because the OpenMP ordered parallelization scheme does not support the parallelization of a loop with two consecutive sequential stages.

Notice that the listed low LCP loops (LCP = 0%) are MAY DOACROSS, i.e. the compiler could not prove, at compile time, that they are free of loop-carried dependences. However, these loops have no materialized loop-carried dependence at runtime.

Figure 4.1 shows loops wall-time speed-ups normalized to the sequential execution time, when the `use` clause is set to work with the TLS (orange bar), DOAX (purple bar) and, BDX (blue bar) algorithms. A dotted vertical line was added to separate the loop

Table 4.1: Loops extracted from SPEC CPU 2006, StarBench, and `cBench` applications

Loop ID	Benchmark	Location	Function/Method	%*Cov*	Invocations
A	`automotive_bitcount`	`bitcnts.c`,65	`main1`	100%	560
E	`automotive_susan_s`	`susan.c`,725	`susan_smoothing`	100%	22050
H	`automotive_susan_e`	`susan.c`,1117	`susan_edges`	18%	374
I	`automotive_susan_e`	`susan.c`,1056	`susan_edges`	56%	374
J	`automotive_susan_s`	`susan.c`,723	`susan_smoothing`	100%	49
V	`automotive_susan_c`	`susan.c`,1614	`susan_corners`	7%	782
Bzip	`Bzip2`	`bzip2.c`,460	`compressStream`	90%	3
Rota	`Rotate`	`program.cpp`,89	`main`	100%	1
Rgb	`RGB-YUV`	`bmark.c`,280	`main`	100%	1
RotC	`Rot-CC`	`program.cpp`,91	`main`	100%	1
Ray	`Ray-Rot`	`program.cpp`,101	`main`	100%	1

Table 4.2: Characterization and Execution of Loops

Loop ID	Loop Characterization		Serial Execution	TLS Execution			DOAX Execution		BDX Execution		
	# Components	LCP (%)	Loop Exec. Time (s)	Strip Size	Loop Exec. Time (s)	Loop Speedup	Loop Exec. Time (s)	Loop Speedup	Batch Size	Loop Exec. Time (s)	Loop Speedup
A	P-S	0%	0.0084396	502	0.00308929	2.73	0.03108403	0.27	1000	0.017491	0.48
E	P-S	0%	0.000145	15	0.00007256	2.00	0.00015865	0.91	60	0.000112	1.29
H	S	0%	0.002515	1	0.00069519	3.62	0.00160649	1.57	34	0.001521	1.65
I	S	0%	0.0033662	2	0.00157754	2.13	0.00496690	0.68	37	0.004409	0.76
J	S	0%	0.0569932	1	0.02979592	1.91	0.06440706	0.88	45	0.057311	0.99
V	S	34%	0.00016	1	0.00014067	1.14	0.00043272	0.37	40	0.000236	0.68
Bzip	S-S	100%	10.52	80	10.74	0.98	-	-	80	10.08	1.04
Rota	S-P	100%	23.22	2	23.96	0.97	16.89	1.37	1	8.07	2.88
Rgb	P-S-P	100%	17.42	2	18.10	0.96	13.70	1.27	1	5.39	3.23
RotC	S-P	100%	24.86	2	25.57	0.97	13.27	1.87	1	8.83	2.82
Ray	S-P	100%	2.82	6	2.91	0.97	2.57	1.10	1	0.79	3.58

classes with low (left) and high LCP (right).

Figure 4.2 shows the abort/commit ratios for the coarse-grained TLS of the evaluated loops. Aborts may be caused by: memory conflicts, capacity issues, explicit instructions (`xabort`) and OS or microarchitecture events (e.g. system calls, interrupts or traps) [6]. An order-inversion abort rolls back a transaction that completes execution out of order using an explicit abort instruction. As shown in Figure 4.2 the class of low LCP loops (left) exhibits a larger share of commits (blue bar) when compared to high LCP loops (right).

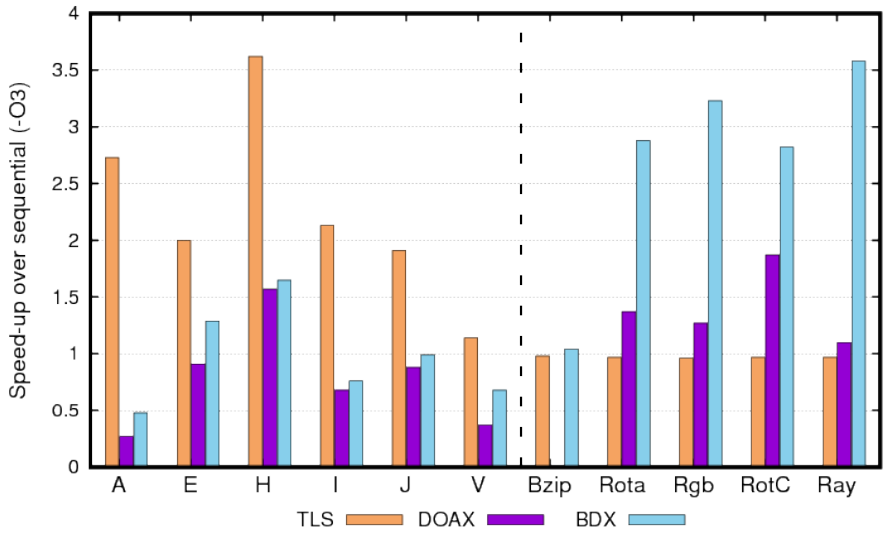

Figure 4.1: Performance of the loops running the three parallelization techniques

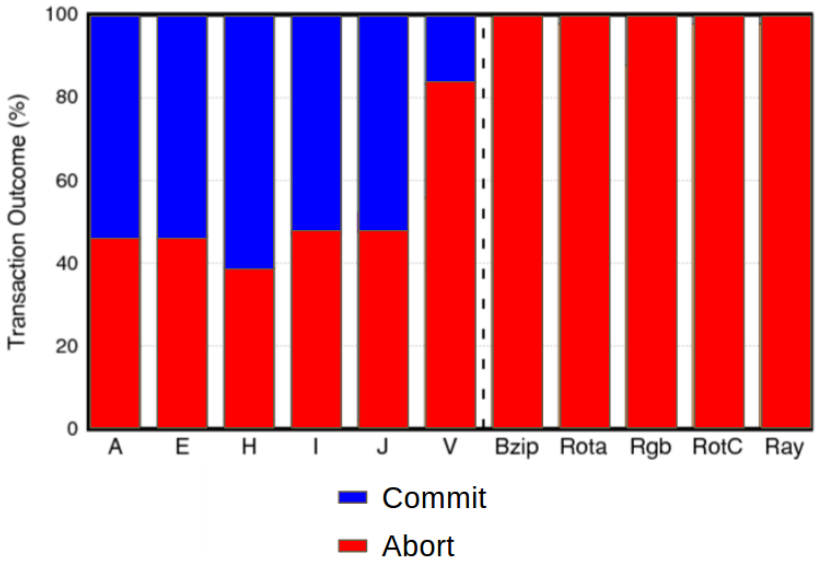

Figure 4.2: Ratio of aborts and commits for coarse-grained TLS execution on Intel Core

4.2.1 Performance Analysis

The hypothesis presented is that loops with low LCP have a better perfomance improvement when parallelized using TLS technique, otherwise, for loops with greater LCP, non-speculative DOACROSS parallelization techniques have a better performance improvement. By analyzing Figure 4.1 and Table 4.2 in all loops with low LCP, TLS parallelization performed better than DOAX and BDX. Furthermore, TLS performance degrades as LCP increases. This becomes clearer in the case of loop V (LCP = 34%) which exhibits the lowest TLS speed-up of those in its class (1.14x).

TLS is better applied on loops that compiler cannot prove that iterations are independent and when compiler may detect a dependence, this dependence do not occur at runtime. Some tested loops, which have a may loop-carried dependence, fit into this situation, because these dependences does not occur at runtime and there are few transaction aborts. Notice that these loops (A, E, H, I, J and, V) have near zero LCP, which indicates that TLS is better applied in loops with low LCP.

The evaluated DOACROSS loops have many actual dependences that materialize at runtime (LCP = 100%) resulting in large conflict-abort ratios, what prevents TLS from delivering performance improvements. `loopV` has a substantial LCP and conflict-abort ratio. However, this loop is a special case because although it has LCP = 34%, TLS can still deliver some performance improvement. As explained in [6], the input of this loop is a sparse image with most of the pixels set to zero; although the image corners create loop-carried dependences they are computed close to each other within the same strip.

Overall, BDX performed better than DOAX, because it is well-known that DOAX is intensive in communication due to its need to forward loop-carried dependences through shared memory at every iteration. BDX, on the other hand, only communicates intra-component dependences after a batch of iterations. Moreover, since iterations are distributed among threads, DOAX suffers from more cache misses and false sharing which

reduce data locality. By executing each stage in batching mode and applying a simple communication mechanisms BDX can be effective to parallelize small and large loops. Nevertheless, notice that for loops of small granularity (A, E, H, I, J and, V) both BDX and DOAX struggle to produce performance improvements because communication overhead is greater than performance gain.

The strip size parameters for TLS and batch size parameters for BDX, shown in Table 4.2, were determined empirically running a number of experiments. However, one can also choose batch sizes based on loop or stage size. In other words, small loops require larger batch sizes and bigger loops require small batch sizes. Experiments revealed that BDX can produce good speedups if the loop has a sufficient number of instructions and supports a well-balanced stage partitioning. Consistent slowdowns were only observed in the loops where the stage sizes were tiny (loops A. I and V of `cBench`).

An interesting observation about the three algorithms (TLS, DOAX, and BDX) is that although all three are applied in a similar manner through the `use` clause, the speed-ups achieved by them can be very different. This reinforces what was claimed that TLS technique results in good speed-ups for low LCP loops and non-speculative algorithms (DOAX or BDX) produce better speed-ups for high LCR (LCF or LCP) loops. Another conclusion that can be drawn from the experiments is that for those loops containing at least one parallel stage, BDX performs better than DOAX.

Overall, the experiments made it clear that enabling the user to drive the parallelization process, by quickly selecting which loop parallelization algorithm to use is paramount to achieve good performance improvements across a wide range of program loops.

A methodology for choosing better algorithm for each type of loop is shown in table 4.3. This table shows DOACROSS loop types (DOACROSS, MAY DOACROSS and UNDEFINED components) and LCP classes (low or high). Combining these two metrics, programmer is able to decide which one is the better algorithm for each case.

Loop Classification	High LCP	Low LCP
(1) UNDEFINED Components	–	TLS
(2) MAY DOACROSS Loops	BDX	TLS
(3) DOACROSS Loops	DOAX or BDX	

Table 4.3: Selecting parallelization technique based on loop-carried probability and knowledge of sequential components

Chapter 5

Conclusions

Analysis have proved that loop-carried probability and parallelization technique choice are correlated. Thus, loop-carried probability can be used as a metric for selecting which technique has a better performance improvement when parallelizing an algorithm with DOACROSS loops. However, the knowledge about loop components must be used as a metric too, because loop categories (DOACROSS, MAY DOACROSS and UNDEFINED) may have different behaviors depending on parallelization technique used to parallelize the loop.

As seen on experimental results, we can categorize studied loops into two categories: loops with low loop-carried probability and loops with high loop-carried probability and each category is better parallelized using a different technique, depending on loop structure. Loops with low loop-carried probability have better performance when speculated using TLS, because there are less transactions aborts and performance improvement is greater than transactions overhead. Loops with high loop-carried probability have better performance when parallelized using BDX or DOAX, because independent regions of the loop can be executed in parallel and only the regions with loop-carried dependences must be executed sequentially.

Generally, when programmer knows loop iterations dependences, that is, loop is a DOACROSS or a MAY DOACROSS, the only factor to decide which technique has a better performance gain is the LCP. Otherwise, when loop is UNDEFINED, using the TLS technique is the only option, however, performance is only improved when LCP is low. When a loop is UNDEFINED but has a high LCP, all techniques presented fail to improve the performance and even worse, introduce an overhead for managing and synchronizing threads.

Bibliography

[1] Divino César S. Lucas and Guido Araujo. The Batched DOACROSS loop parallelization algorithm. In *HPCS 2015*, Amsterdam, the Netherlands, 2015.

[2] J. Salamanca, J. N. Amaral, and G. Araujo. Evaluating and improving thread-level speculation in hardware transactional memories. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2016, pp. 586–595.

[3] OpenMP Specifications http://openmp.org/wp/openmp-specifications

[4] Salamanca, J., Mattos, L., Araujo, G.. Loop-carried dependence verification in OpenMP. In: 10th International Workshop on OpenMP (IWOMP 2014). pp. 87–102. Salvador, Brazil (2014)

[5] Parsec 3.0 http://parsec.cs.princeton.edu/

[6] Salamanca, J., Amaral, J.N., Araujo, G. Performance evaluation of thread-level speculation in off-the-shelf hardware transactional memories. In: Euro-Par 2017: Parallel Processing. pp. 607–621. Santiago de Compostela, Spain (2017)

[7] S. Campanoni, T. Jones, G. Holloway, V. J. Reddi, G.-Y. Wei, and D. Brooks. Helix: automatic parallelization of irregular programs for chip multiprocessing. In *CGO '12*, 2012.

[8] W. R. Chen, W. Yang, and W. C. Hsu. A lock-free cache-friendly software queue buffer for decoupled software pipelining. In *ICS 2010*, 2010.

[9] R. Cytron. Doacross: Beyond vectorization for multiprocessors. In *ICPP*, 1986.

[10] J. E. Fritts, F. W. Steiling, J. A. Tucek, and W. Wolf. Mediabench ii video: Expediting the next generation of video systems research. *Microprocess. Microsyst.*, 33(4), 2009.

[11] J. Giacomoni, T. Moseley, and M. Vachharajani. Fastforward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue. In *PPoPP '13*, 2008.

[12] J. L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, Sept. 2006.

[13] J. Huang, T. B. Jablin, S. R. Beard, N. P. Johnson, and D. I. August. Automatically exploiting cross-invocation parallelism using runtime information. In *CGO '13*, 2013.

[14] J. Huang, A. Raman, T. B. Jablin, Y. Zhang, T.-H. Hung, and D. I. August. Decoupled software pipelining creates parallelization opportunities. In *CGO '8*, 2010.

[15] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach.* Morgan Kaufmann Publishers Inc., 2002.

[16] P. P. Lee, T. Bu, and G. Chandranmenon. A lock-free, cache-efficient shared ring buffer for multi-core architectures. In *ANCS '5*, 2009.

[17] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *MICRO '38*, 2005.

[18] E. Raman, G. Ottoni, A. Raman, M. J. Bridges, and D. I. August. Parallel-stage decoupled software pipelining. In *CGO '6*, 2008.

[19] R. Rangan, N. Vachharajani, G. Ottoni, and D. I. August. Performance scalability of decoupled software pipelining. In *TACO '2008*, 2008.

[20] R. Rangan, N. Vachharajani, A. Stoler, G. Ottoni, D. I. August, and G. Z. N. Cai. Support for high-frequency streaming in cmps. In *MICRO '39*, 2006.

[21] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August. Decoupled software pipelining with the synchronization array. In *PACT '13*, 2004.

[22] P. Unnikrishnan, J. Shirako, K. Barton, S. Chatterjee, R. Silvera, and V. Sarkar. A practical approach to doacross parallelization. In *Euro-Par '18*, 2012.

[23] J. Shirako, P. Unnikrishnan, S. Chatterjee, K. Li, V. Sarkar Expressing DOACROSS Loop Dependences in OpenMP In *IWOMP '9*, 2013

[24] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August. Speculative decoupled software pipelining. In *PACT '16*, 2007.

Appendix A

Attachment 1

Loop A from automotive_bitcount, line 65 from bitcnts.c file:

```
1  int random = rand_r(&rSeed);
2  n = 0;
3  #pragma omp parallel for ordered(1) private(seed) use(psbdx, 1000)
4  for (j = 0; j < iterations; j++) {
5    int temp = 0;
6    seed = 13 * j + random;
7    temp += pBitCntFunc[i](seed);
8
9  #pragma omp ordered depend(sink: j-1) depend(var: n)
10   n += temp;
11 #pragma omp ordered depend(source)
12 }
```

Appendix B

Attachment 2

Loop E from automotive_susan_s, line 725 from susan.c file:

```
1  for (i = mask_size; i < y_size - mask_size; i++) {
2  #pragma omp parallel for ordered(1) private(
                  \
3      area, total, dpt, ip, centre, cp, brightness, tmp, x, y) use(
     psbdx, 60)
4    for (j = mask_size; j < x_size - mask_size; j++) {
5      area = 0;
6      total = 0;
7      dpt = dp;
8      ip = in + ((i - mask_size) * x_size) + j - mask_size;
9      centre = in[i * x_size + j];
10     cp = bp + centre;
11     for (y = -mask_size; y <= mask_size; y++) {
12       for (x = -mask_size; x <= mask_size; x++) {
13         brightness = *ip++;
14         tmp = *dpt++ * *(cp - brightness);
15         area += tmp;
16         total += tmp * brightness;
17       }
18       ip += increment;
19     }
20     tmp = area - 10000;
21
22 #pragma omp ordered depend(sink : j - 1) depend(var : out)
23     if (tmp == 0)
24       *out++ = median(in, i, j, x_size);
25     else
26       *out++ = ((total - (centre * 10000)) / tmp);
27 #pragma omp ordered depend(source)
28   }
29 }
```

Appendix C

Attachment 3

Loop H from automotive_susan_e, line 1117 from susan.c file:

```
1  #pragma omp parallel for ordered(1) private(
                \
2      m, n, cp, p, x, y, c, z, do_symmetry, w, a, b, j) use(psbdx, 34)
3  for (i = 4; i < y_size - 4; i++) {
4  #pragma omp ordered depend(sink : i - 1)
5    for (j = 4; j < x_size - 4; j++) {
6      if (r[i * x_size + j] > 0) {
7        m = r[i * x_size + j];
8        n = max_no - m;
9        cp = bp + in[i * x_size + j];
10
11       if (n > 600) {
12         p = in + (i - 3) * x_size + j - 1;
13         x = 0;
14         y = 0;
15         ...
16       } else
17         do_symmetry = 1;
18
19       if (do_symmetry == 1) {
20         p = in + (i - 3) * x_size + j - 1;
21         x = 0;
22         y = 0;
23         w = 0;
24
25         /*   |       \
26              y  -x-  w
27              |         \   */
28
29         ...
30       }
31     }
32   }
33 #pragma omp ordered depend(source)
34 }
```

Appendix D

Attachment 4

Loop I from automotive_susan_e, line 1056 from susan.c file:

```
1  #pragma omp parallel for ordered(1) private(n, p, cp, j) use(psbdx,
       37)
2  for (i = 3; i < y_size - 3; i++) {
3  #pragma omp ordered depend(sink : i - 1)
4    for (j = 3; j < x_size - 3; j++) {
5      n = 100;
6      p = in + (i - 3) * x_size + j - 1;
7      cp = bp + in[i * x_size + j];
8
9      ...
10
11     if (n <= max_no)
12       r[i * x_size + j] = max_no - n;
13   }
14 #pragma omp ordered depend(source)
15 }
```

Appendix E

Attachment 5

Loop J from automotive_susan_s, line 723 from susan.c file:

```
1  #pragma omp parallel for ordered(1) private(
                  \
2      area, total, dpt, ip, centre, cp, brightness, tmp, x, y, j) use(
     psbdx, 45)
3  for (i = mask_size; i < y_size - mask_size; i++) {
4  #pragma omp ordered depend(sink : i - 1) depend(var : out)
5    for (j = mask_size; j < x_size - mask_size; j++) {
6      area = 0;
7      total = 0;
8      dpt = dp;
9      ip = in + ((i - mask_size) * x_size) + j - mask_size;
10     centre = in[i * x_size + j];
11     cp = bp + centre;
12     for (y = -mask_size; y <= mask_size; y++) {
13       for (x = -mask_size; x <= mask_size; x++) {
14         brightness = *ip++;
15         tmp = *dpt++ * *(cp - brightness);
16         area += tmp;
17         total += tmp * brightness;
18       }
19       ip += increment;
20     }
21     tmp = area - 10000;
22
23     if (tmp == 0)
24       *out++ = median(in, i, j, x_size);
25     else
26       *out++ = ((total - (centre * 10000)) / tmp);
27   }
28 #pragma omp ordered depend(source)
29 }
```

Appendix F

Attachment 6

Loop V from automotive_susan_c, line 1614 from susan.c file:

```
1  #pragma omp parallel for ordered(1) private(x, flag, j) use(psbdx,
       40)
2  for (i = 5; i < y_size - 5; i++) {
3  #pragma omp ordered depend(sink : i - 1) depend(var : n)
4    for (j = 5; j < x_size - 5; j++) {
5      x = r[i * x_size + j];
6      if (x > 0) {
7        /* 5x5 mask */
8  #ifdef FIVE_SUPP
9        flag = (x > r[(i - 1) * x_size + j + 2]) &&
10         (x > r[(i)*x_size + j + 1]) && (x > r[(i)*x_size + j + 2])
       &&
11           (x > r[(i + 1) * x_size + j - 1]) &&
12           (x > r[(i + 1) * x_size + j]) &&
13           (x > r[(i + 1) * x_size + j + 1]) &&
14           (x > r[(i + 1) * x_size + j + 2]) &&
15           (x > r[(i + 2) * x_size + j - 2]) &&
16           (x > r[(i + 2) * x_size + j - 1]) &&
17           (x > r[(i + 2) * x_size + j]) &&
18           (x > r[(i + 2) * x_size + j + 1]) &&
19           (x > r[(i + 2) * x_size + j + 2]) &&
20           (x >= r[(i - 2) * x_size + j - 2]) &&
21           (x >= r[(i - 2) * x_size + j - 1]) &&
22           (x >= r[(i - 2) * x_size + j]) &&
23           (x >= r[(i - 2) * x_size + j + 1]) &&
24           (x >= r[(i - 2) * x_size + j + 2]) &&
25           (x >= r[(i - 1) * x_size + j - 2]) &&
26           (x >= r[(i - 1) * x_size + j - 1]) &&
27           (x >= r[(i - 1) * x_size + j]) &&
28           (x >= r[(i - 1) * x_size + j + 1]) &&
29           (x >= r[(i)*x_size + j - 2]) && (x >= r[(i)*x_size + j - 1])
       &&
30           (x >= r[(i + 1) * x_size + j - 2]);
31 #endif
32 #ifdef SEVEN_SUPP
33       ...
34     }
```

```
35    }
36 #pragma omp ordered depend(source)
37 }
```

Appendix G

Attachment 7

Loop Bzip from Bzip2, line 460 from bzip2.c file:

```
1  UChar *ibuf2;
2
3  #pragma omp parallel for private(nIbuf, ibuf2) firstprivate(bzerr)
       ordered(1) use(psbdx, 80)
4  for (cnt=0; cnt<MaxIterations; cnt++) {
5  #pragma omp ordered depend(sink: cnt-1)
6    ibuf2 = (UChar *)malloc(5001*sizeof(UChar));
7    nIbuf = fread ( ibuf2, sizeof(UChar), 5000, stream );
8  #pragma omp ordered depend(source)
9
10 #pragma omp ordered depend(sink: cnt-1)
11   BZ2_bzWrite ( &bzerr, bzf, (void*)ibuf2, nIbuf );
12   free(ibuf2);
13 #pragma omp ordered depend(source)
14 }
```

Appendix H

Attachment 8

Loop Rota from Rotate, line 89 from program.cpp file:

```
1  int exec = 1;
2  #pragma omp parallel for ordered(1) use(psbdx)
3  for (i = 0; i < qtdFiles; i++) {
4  #pragma omp ordered depend(sink : i - 1)
5    RotateEngine *re = new RotateEngine;
6    if (exec) {
7      if (!re->init(srcfiles[i], destfiles[i], angle)) {
8        exec = 0;
9        continue;
10     }
11   }
12
13 #pragma omp ordered depend(source)
14   RotateEngine *re2 = re;
15
16   if (exec) {
17     re2->run();
18     re2->finish();
19   }
20 }
21 TIME(loop_time_end);
22 double t = timevaldiff(&loop_time_start, &loop_time_end);
23 fprintf(stderr, "Total execution time: %lf (s)\n", t/1.0e3);
24
25 if (exec == 0) return BAD_EXIT;
```

Appendix I

Attachment 9

Loop Rgb from RGB-YUV, line 280 from bmark.c file:

```
1  int exec = 1;
2  #pragma omp parallel for ordered(1) use(psbdx)
3  for (i = 0; i < qtdFiles; i++) {
4    rgbyuv_args_t args;
5    rgbyuv_args_t *argsP = &args;
6  #pragma omp ordered depend(sink : i - 1)
7    if (exec) {
8      if (initialize(argsP, srcfiles[i])) {
9        fprintf(stderr, "Could Not Initialize Kernel Data\n");
10       exec = 0;
11       continue;
12     }
13   }
14 #pragma omp ordered depend(source)
15   if (exec) {
16     processImage(argsP);
17
18     writeComponents(argsP);
19
20     if (finalize(argsP)) {
21       fprintf(stderr, "Could Not Free Allocated Memory\n");
22       exec = 0;
23       continue;
24     }
25   }
26 }
27
28 if (!exec) return BAD_EXIT;
```

Appendix J

Attachment 10

Loop RotC from Rot-CC, line 91 from program.cpp file:

```
1  int exec = 1;
2
3  #pragma omp parallel for ordered(1) use(psbdx)
4  for (i = 0; i < qtdFiles; i++) {
5  #pragma omp ordered depend(sink : i - 1)
6    BenchmarkEngine *be = new BenchmarkEngine;
7    if (exec) {
8      if (!be->init(srcfiles[i], destfiles[i], angle)) {
9        exec = 0;
10       continue;
11     }
12   }
13 #pragma omp ordered depend(source)
14   if (exec) {
15     be->run();
16     be->finish();
17   }
18 }
19
20 if (!exec) return BAD_EXIT;
```

Appendix K

Attachment 11

Loop Ray from Ray-Rot, line 101 from program.cpp file:

```
1  int exec = 1;
2  #pragma omp parallel for ordered(1) use(psbdx)
3  for (i = 0; i < qtdFiles; i++) {
4  #pragma omp ordered depend(sink : i - 1)
5    RotateEngine *re = new RotateEngine;
6    RayEngine *ra = new RayEngine;
7
8    if (exec) {
9      if (!ra->init(srcfiles[i], xres, yres, rpp)) {
10       cerr << "Raytracing Kernel Init failed!" << endl;
11       exec = 0;
12       continue;
13     }
14
15     if (!re->init(ra->getOutputImage(), angle, destfiles[i])) {
16       cerr << "Rotation Kernel Init failed!" << endl;
17       exec = 0;
18       continue;
19     }
20   }
21 #pragma omp ordered depend(source)
22   if (exec) {
23     ra->printRaytracingState();
24     re->printRotationState();
25
26     ra->run();
27     re->run();
28
29     ra->finish();
30     re->finish();
31   }
32 }
33
34 if (!exec) return BAD_EXIT;
```