LOAD BALANCING
FOR PARALLEL LOOP
SCHEDULING

A thesis submitted to The University of Manchester
for the degree of Master of Philosophy
in the Faculty of Science and Engineering

2021

Xingran Ruan

Department of Computer Science
# Contents

Abstract 7

Declaration 9

Copyright 10

Acknowledgements 11

1 Introduction 12
   1.1 The Advent of Parallel Computing 12
   1.2 The Loop Schedule Problem 13
   1.3 Solution and Contribution 16
   1.4 Basic Concepts 17
      1.4.1 Loops 17
      1.4.2 Canonical Loop Nest 21
      1.4.3 General Loop Nest 22
      1.4.4 Performance Measurement 23
   1.5 Thesis Overview 27

2 The Load Balancing Problem 29
   2.1 Introduction 29
   2.2 The Problems of Loop Partitioning and Scheduling 31
      2.2.1 Static Approach for Appeasing Problems of Loop Partitioning and Scheduling 34
      2.2.2 Dynamic Approach for Appeasing Problems of Loop Partitioning and Scheduling 43
      2.2.3 Discussion 47
   2.3 Loop Partitioning and Scheduling Schemes 47
List of Tables

2.1 The assignment of iterations on 4 processors by cyclic round-robin approach ........................................... 41
2.2 The assignment of iterations on 4 processors by static block scheduling ..................................................... 41
2.3 The assignment of iterations on 4 processors by canonical loop scheduling ................................................. 41
3.1 Achieved load imbalance and relative load imbalance for vector addition, rectangular loop nest. ..................... 71
3.2 Achieved load imbalance and relative load imbalance for upper triangular matrix addition, rectangular loop nest. 72
3.3 Achieved load imbalance and relative load imbalance for vector addition, triangular loop nest. ..................... 72
3.4 Achieved load imbalance and relative load imbalance for adjoint convolution, triangular loop nest. ............... 72
3.5 Achieved load imbalance and relative load imbalance for upper triangular matrix addition, triangular loop nest. 72
3.6 Achieved load imbalance and relative load imbalance for upper triangular matrix multiplication, triangular loop nest. 72
3.7 Achieved load imbalance and relative load imbalance for upper triangular matrix addition, trapezoid loop nest. 73
3.8 Achieved load imbalance and relative load imbalance for adjoint convolution, trapezoid loop nest. ............... 73
3.9 Achieved load imbalance and relative load imbalance for vector addition, trapezoid loop nest. ..................... 73
3.10 Achieved load imbalance and relative load imbalance for vector addition, Gaussian loop nest. .................... 73
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Map from task set to processor set</td>
<td>15</td>
</tr>
<tr>
<td>1.2</td>
<td>The examples of loop nests</td>
<td>18</td>
</tr>
<tr>
<td>1.3</td>
<td>Geometrical representation of the rectangular loop</td>
<td>19</td>
</tr>
<tr>
<td>1.4</td>
<td>Geometrical representation of non-rectangular loop</td>
<td>20</td>
</tr>
<tr>
<td>1.5</td>
<td>Geometrical representation of Gaussian type loop</td>
<td>21</td>
</tr>
<tr>
<td>1.6</td>
<td>The general form of a Triangular/Trapezoid/Gaussian loop nest</td>
<td>22</td>
</tr>
<tr>
<td>1.7</td>
<td>The general form of a Triangular/Trapezoid/Gaussian loop nest</td>
<td>23</td>
</tr>
<tr>
<td>2.1</td>
<td>The abstract view of partitioning</td>
<td>33</td>
</tr>
<tr>
<td>2.2</td>
<td>The abstract view of scheduling</td>
<td>33</td>
</tr>
<tr>
<td>2.3</td>
<td>OpenMP clause</td>
<td>35</td>
</tr>
<tr>
<td>2.4</td>
<td>The efficiency of Cyclic Round-Robin</td>
<td>36</td>
</tr>
<tr>
<td>2.5</td>
<td>Execution time for $N = 1000$, Cyclic Round-Robin</td>
<td>36</td>
</tr>
<tr>
<td>2.6</td>
<td>Execution time for $N = 200000$, Static Block Scheduling</td>
<td>37</td>
</tr>
<tr>
<td>2.7</td>
<td>The load imbalance of Static Block Scheduling</td>
<td>38</td>
</tr>
<tr>
<td>2.8</td>
<td>False sharing influence on Cyclic Round Robin and Static Block Scheduling</td>
<td>39</td>
</tr>
<tr>
<td>2.9</td>
<td>Performance of three schemes on triangle loop nest</td>
<td>42</td>
</tr>
<tr>
<td>2.10</td>
<td>Total number of times of executing loop body on each processor</td>
<td>42</td>
</tr>
<tr>
<td>2.11</td>
<td>Execution time for $N = 200000$, Pure Dynamic Scheduling</td>
<td>44</td>
</tr>
<tr>
<td>2.12</td>
<td>Performance of Dynamic Chunk Scheduling with various chunk size</td>
<td>45</td>
</tr>
<tr>
<td>2.13</td>
<td>Iteration’s distribution by Dynamic Chunk Scheduling with various chunk size</td>
<td>46</td>
</tr>
<tr>
<td>2.14</td>
<td>Relative load imbalance</td>
<td>46</td>
</tr>
<tr>
<td>2.15</td>
<td>Sample code for Automatic collapsing loop partitioning</td>
<td>49</td>
</tr>
<tr>
<td>2.16</td>
<td>Sample code for performance comparison</td>
<td>52</td>
</tr>
<tr>
<td>2.17</td>
<td>Execution time for $N = 10000$ rectangular matrix addition</td>
<td>53</td>
</tr>
</tbody>
</table>
2.18 Total number of times of executing loop body on each processor, rectangular matrix addition ........................................ 53
2.19 Execution time for \( N = 10000 \) triangular adjoint convolution . ...... 54
2.20 Total number of times of executing loop body on each processor, triangular adjoint convolution .................................................. 54
2.21 Execution time for \( N = 4000 \) triangular upper triangle matrix multiplication ................................................................. 55
2.22 Total number of times of executing loop body on each processor, triangular upper triangle matrix multiplication .............. 56
2.23 Example of Gaussian loop nest .............................................................. 57
2.24 Dividing trapezoid into small trapezoids .................................................. 58
2.25 The general form of a double Gaussian loop nest .................................. 59
2.26 Simplify the code structure ................................................................. 60
2.27 The example form of a double Gaussian loop nest ................................ 62

3.1 Rectangular benchmark applications .................................................. 68
3.2 Triangular benchmark applications .................................................... 68
3.3 Trapezoid benchmark applications .................................................... 69
3.4 Gaussian benchmark applications ..................................................... 69
3.5 Execution time for \( N = 10000 \) rectangular vector addition, rectangular loop nest ................................................................. 74
3.6 Execution time for \( N = 10000 \) upper triangular matrix addition, rectangular loop nest ................................................................. 75
3.7 Execution time for \( N = 10000 \) vector addition, triangular loop nest ................................................................. 75
3.8 Execution time for \( N = 10000 \) upper triangular matrix addition, triangular loop nest ................................................................. 76
3.9 Execution time for \( N = 10000 \) vector adjoint convolution, triangular loop nest ................................................................. 76
3.10 Execution time for \( N = 1000 \) upper triangular matrix multiplication, triangular loop nest ................................................................. 77
3.11 Execution time for \( N = 10000 \) vector addition, trapezoid loop nest ................................................................. 77
3.12 Execution time for \( N = 10000 \) upper triangular matrix addition, trapezoid loop nest ................................................................. 78
3.13 Execution time for \( N = 10000 \) vector adjoint convolution, trapezoid loop nest ................................................................. 78
3.14 Execution time for \( N = 10000 \) vector addition, Gaussian loop nest .... 79
Parallel computing often needs to parallelise loop structures, which are a rich source of computing power. To do this, the partitioning phase splits loop iterations into independent tasks and the scheduling phase decides how they will be assigned to processors. The aim of loop partitioning and scheduling schemes is to transform a sequential program into segments of equal workloads. More specifically, compilers partition a number of parallel jobs into several blocks and map these blocks on processors for which the workloads on processors are as even as possible. Ideally, the amount of work on each processor is the same, in which case perfect load balance is achieved. If the distribution of workload among processors is not the same, load imbalance usually exists among processors.

This thesis considers the load imbalance achieved by different loop partitioning and scheduling approaches with various classical benchmark applications. In addition, it proposes an extension with respect to the canonical loop scheduling scheme, called combined canonical loop scheduling scheme to reduce load imbalance when dealing with Gaussian type loop nests.

First, a description of the loop scheduling problem and several important basic concepts are provided in the introduction. Then the load balancing problem is defined and static and dynamic loop scheduling schemes are analysed. Focusing on the poor performance of loop scheduling schemes when dealing with Gaussian loop nests, we propose a new loop scheduling strategy, called combined canonical loop scheduling, and provide a theoretical analysis about its load balance properties.

Finally, a series of experiments is simulated to evaluate combined canonical
loop scheduling and four other selected loop scheduling schemes by five different benchmark programs. The results show that the new proposed strategy achieves perfect load balance with respect to depth 2 rectangular, triangular, trapezoid loop nests and nearly perfect load balance with respect to Gaussian loop nests.
Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.
Copyright

i. The author of this thesis (including any appendices and/or schedules to this thesis) owns certain copyright or related rights in it (the “Copyright”) and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

ii. Copies of this thesis, either in full or in extracts and whether in hard or electronic copy, may be made only in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has from time to time. This page must form part of any such copies made.

iii. The ownership of certain Copyright, patents, designs, trade marks and other intellectual property (the “Intellectual Property”) and any reproductions of copyright works in the thesis, for example graphs and tables (“Reproductions”), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and commercialisation of this thesis, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see [http://documents.manchester.ac.uk/DocuInfo.aspx?DocID=24420](http://documents.manchester.ac.uk/DocuInfo.aspx?DocID=24420)), in any relevant Thesis restriction declarations deposited in the University Library, The University Library’s regulations (see [http://www.library.manchester.ac.uk/about/regulations/](http://www.library.manchester.ac.uk/about/regulations/)) and in The University’s policy on presentation of Theses.
Acknowledgements

I would like to explicit my sincere gratitude to my supervisor Professor Rizos Sakellariou, who has the belief and the substance of a genius: he continually and adequately bring his spirit of adventure in regard to research study and academic, and an excitement in regard to teaching. If I did not get the help from his guidance, my final thesis would not possible to be completed.
Chapter 1

Introduction

1.1 The Advent of Parallel Computing

It has been proved remarkably astute that the law first proposed by Gordon Moore (an Intel founder) that the number of transistors on microprocessors will become double every one to two years. It claims that the central processing unit’s performance (CPU) would also approximately double every one to two years. In 1955 the Project 704 introduced by IBM with Gene Amdahl architect was the first commercial machine with floating-point hardware and was capable of approximately five kFLOPS. The principal advanced feature of the 704 was its high-speed magnetic core storage or memory. In a 704, thousands of cores were strung on a complex of wires in such a fashion that several wires passed through the centre of each core. In the following year, Project 7030 (STRETCH) was started in IBM; it was the largest, fastest, operating a general computer in which as many as four instructions can be processed simultaneously. In the years that followed, the main efforts on parallelism were placed on improving the performance of the hardware level of machine architecture.

Simultaneously, the analysis of parallel computing and mining down the theoretical principles for parallel computing was leaving alongside approaches to design the computer’s architecture for supporting high-level parallel programming. However, the starting point for the explosion of interest in parallel computing can be placed around the mid-1970’s. Two factors lead this explosion, the development of Very large-scale integration (VLSI), and Physical limitation. The VLSI is the process of creating an integrated circuit (IC) by combining millions of MOS transistors onto a single chip. It allows building the parallel computer
architecture on a wider scale than before; The performance of the given developed technologies on machines has approached the physical limitation. The research community noticed that parallel computing provides the promise and reachable key to improving computing performance.

Nowadays, it has been proved that parallel computing makes an outstanding contribution to a number of computing-intensive problems or applications. Its potential capacities have revealed it one of the primary vehicles in the current quest for high-performance computing, which promises to open exciting new possibilities in science and engineering. However, despite its undoubtedly significant impact in several scientific disciplines, there is comparatively little use of parallelism in the mainstream software production industry. The most common reason for this slow improvement of parallelism is the lack of sufficient methodology level support. What contributes to an efficient parallel programming methodology will vary from machine to machine. Directly, parallel computing users have to spend much time deploying programs from one to another new machine.

A solution, which has been pushed forward since the 1990s, has been referred to as the development of parallelizing compilers. In a general sense, this term describes that the compilers aim to transform a sequential program into some fragments that are equivalent to each other and distribute them to a specific architecture. This transformation’s motivation is evident that a compiler’s capacity that performs this transformation can free the machine from a time-consuming task, thus achieving a faster, more efficient production of software. Problems extended from this general field of research are studied in this thesis.

1.2 The Loop Schedule Problem

Given the current advanced parallelizing compiler technologies, developing one parallel compiler which can achieve ‘perfect equal’ on a broad class of application or problems, and a comprehensive type of architectures becomes a challenging mission. The lack of a ‘one-to-everything’ parallel compiler model, as well as the divergent running environment and performance requirements, constitute the significant barrier. Thus, existing parallelizing compilers usually fix the specific predefined requirements. While their internal implemented parallelizing strategies may differ and achieve various wide objectives, in some cases, critical decisions still need to be made by the programmer.
CHAPTER 1. INTRODUCTION

However, it is helpful to consider two procedures: the key points of designing a parallelizing compiler. The first procedure detects the pattern to deal with the available parallelism and applies a transformation program that can enable this; the second procedure is mapping the detected available pattern on the given parallelism architecture. In both procedures, the parallelizing compiler has several options to achieve its objectives; for instance, in the first procedure, the compiler can transform the original sequence of available parallel pattern into several different sequences, while, in the second procedure, the parallelizing compiler can map by round sequence, by random, even by combining round sequence and random. The option that the compiler decided to employ must obey maximizing performance, generally, faster running efficiency. This implies that a performance evaluator should assess an effective parallelizing compiler and that any proposed optimization should be applied subject to their forecast results on an evaluated rule.

Traditionally, these two procedures were referred to as two independent parts. The reason why they concluded such jointly independent status can be attributed to historical factors. The second procedure, mapping, being the procedure which straightly related to the utilization of parallelism on the machine level, has been cumulated with much attention since the first parallel machine was built. On the other hand, after the mapping procedure achieved many developments and detected the specific parallel pattern, the transformed pattern’s sequence was referred to as optimization of the mapping procedure. However, both procedures are interdependent and improve the final performance together. It is unacceptable to evaluate them under different rules. They should be examined under a common framework used to achieve high performance or optimizing the model that describes performance in the context of a parallelizing compiler.

The works achieved in this thesis address the problem of mapping loop structure, which has been claimed that it consumes a significant part of resources when dealing with parallel tasks. This means that most of the execution time of a parallel program is spent on processing loop structures; thus, we expect to run those loop structures efficiently in order to process the parallelization in high performance. Especially in scientific programs, parallel loops account for the most significant percentage of parallelism. As a result, a large amount of research power has focused on mapping those parallel for loop structures onto a parallel computer; this problem is called loop schedule problem, and a lot of schemes with
various structure have been proposed to deal with it. The proposed schemes are different from detecting to mapping. They can be distinguished by compile-time schemes and run-time schemes. This is caused by not being available to collect, hard to determine the parallel pattern’s information at compile time. We can infer that compile-time schemes are armed with potential advantages to achieve better workload balance among processors in a parallel computer since it takes into account the parallel pattern to map. Also, in the context of run-time schemes, not knowing the map relation (i.e. the map from tasks to processors, see Fig. 1.1) at the compile-time limit the adaptability of the evaluation method and abstract the final strategy regarding an optimal compiler.

Figure 1.1: Map from task set to processor set.

The motivation of applying run-time loop mapping schemes on loop structure is that, often, a loop structure does not have a uniform structure or regular structure; in other words, not every loop iteration consumes the same number of instructions. However, even in this case, the amount of work consumed by each iteration can follow a regular pattern; for example, the consumption follows a linear model in which the increments of the next iteration are the same, or the consumption of each iteration applies a specific distribution. Thus, it raises the question of whether we can propose an efficient compile-time scheme for this loops class. The main objective of this thesis is focused on answering this question.
CHAPTER 1. INTRODUCTION

1.3 Solution and Contribution

This thesis applies symbolic analysis techniques to analyze unknown variables in compile-time; in this way, we improve the 'specification' of the compiler when dealing with a parallel program. The derived information saying execution time, the number of iterations, etc., can be used to assess program execution performance and, thereby, evaluate the improvement of a proposed compile-time parallel compiler.

The performance measurements model and the overhead model, which are described in Section 1.4.4, are adopted for parallel compiler evaluation. From the illustrated overhead’s factors, the load imbalance which is the overhead caused by poor distribution of the computation load among processors is the main target of our research; thus, the core objective of the loop schedule schemes analyzed in this thesis is to minimize the load imbalance among processors. The most idealized condition is achieving perfect load balance, where the distributed work on each processor is absolutely equal. Meanwhile, our proposed work avoids increasing overhead caused by other factors.

The loop scheduling schemes proposed in this thesis are discussed under generalized loop structure, whose loop bounds are denoted as variables, and those variables are invisible at compile time. Based on the general condition, we can apply the proposed schemes to other applications widely in which the iteration space is reasonably smooth that can be approximated by a collection of trapezoid spaces. What’s more, we can add appropriate statements that let loop iterations’ structure be known at compile time while transforming a parallel work. Throughout the thesis, all program fragments are realized in C language. However, there is nothing inherent in our approach which limits us to use C language, which means that all fragments discussed in this thesis can be tested on any other computer language.

In particular, this thesis makes the following contributions:

- A methodology called Combined Canonical Loop Scheduling, is presented to partitioning generalized loop nests in which the iteration space is smooth and can be approximated by a collection of trapezoid; then schedule the partitions of the target loop nest into parallel processors evenly.

- In the end, an experimental study of the performance of different loop schedule schemes is completed.
1.4 Basic Concepts

Before introducing the central core of this thesis, we provide the instruction of fundamental concepts of parallel computing and provide a brief description of later sections’ ideas in this section.

1.4.1 Loops

The syntax of a For loop (usually referred to as a single loop) is illustrated below:

\[
\text{for } \text{index} = \text{lower\_bound} \text{ to upper\_bound}, \text{(stride)} \text{ do} \\
\text{loop\_body} \\
\text{end for}
\]

This permits repeated execution of the statement (referred to as the loop body) between for and end for a number of times, which depends on three parameters: the lower bound, the upper bound, and the stride. Each execution of the loop body is known as an iteration. For each iteration, the index value is different. At first, the index is assigned with the value of lower bound. For subsequent iterations, the index is increased by stride in each subsequent iteration, i.e.,

\[\text{index}_{i+1} = \text{index}_i + \text{stride}, i = 0, 1, 2, \ldots\]

The loop terminates once the index satisfies the terminal condition (for example, index exceeds lower bound, the stride is negative or exceeds upper bound, the stride is positive). Please note that, in the former case, the lower bound smaller than the upper bound, or in the latter case, the lower bound larger than the upper bound, the loop body is never executed. In this case, such a loop is referred to as empty loop. Although it is not an error and is allowed in C language, this thesis does not study this case.

In general, the loop body may contain some internal loops; that is to say, an internal loop is surrounded by external loops. This case is known as loop nest, and the number of loops surrounding a loop plus 1 indicates its depth. Henceforth, each loop is considered as a loop nest with depth \( d \geq 1 \), where \( d \) is an integer value. The pseudocode shown in Fig. 1.2a below is an example of a

\footnote{These three parameters, lower bound, upper bound and stride can be constants, variables, or arithmetic expression. Herein these three parameters are constants. If the stride is omitted, its value is 1 in default.}
loop nest with depth = 3; herein we denote abbreviation $L_i$ and $U_i$ as the lower bound and upper bound of loop nest within depth $i$; in addition, we denote $L_{i,j}$ and $U_{i,j}$ as the lower and upper bound of $j$th loop nest, if depth $i$ contains more than one loop nests. Any individual loop in the loop nest is categorized by its index. The loop A in the loop nest of Fig. 1.2a is denoted as the outermost loop (the one, whose loop contains all other loops). What’s more, two loops can be contained in the same loop and may use the same variable (the index of the second one will be overwritten); for example, in Fig.1.2a, loop B contains two C loops. And we claim those two C loops are in the same level. The level of a loop is calculated by the number of its surrounding loops plus one. So in Fig.1.2a, the level of A loop is 1; the level of B loop is 2, and the level of C loops is 3. It can be noticed that the max level of a loop nest equals the depth of the loop nest. If a loop with depth $d$ where $d \geq 2$, at which only one loop be contained at each level $l$, where $1 \leq l \leq d$. Then it is called perfect loop nest. For example, the loop nest shows in Fig.1.2b is a perfect loop nest (we removed one of the C loops in the previous one). The perfect loop nest with depth 2 is denoted as double loops see Fig.1.2c.

(a) A for loop nest of depth 3.  
(b) A perfect for loop nest of depth 3.

(c) A double loop nest

Figure 1.2: The examples of loop nests

Given a particular loop nest, the execution of the statement is decided by its surrounding ‘vectors’. We denote each values of index of surrounding loops as an
iteration vector or iteration point. The set of all the iteration points of the loop nest is defined as iteration space. In the general case, the n-dimensional iteration space corresponds to the loop nest with depth n, whose loop body is surrounded by n loops. Now we have introduced all characteristics of a loop nest, and then we will move on to the shape of iteration space of loop nests.

A parallel loop is a kind of loops in which there are no dependencies among its iterations, i.e. all iterations can be executed simultaneously and in any sequence. Whenever a loop’s bound, upper bound and lower bound, do not depend on the loop’s index, then the loop space is regarded as an n-dimensional rectangle. As a result, a rectangular loop is a kind of parallel loops in which bounds are independent of the index, commonly, the execution time of each iteration is uniformly distributed. There is one example here showing rectangular parallel loops with uniform execution time distributions. For more intuitive sight, see Fig. 1.3 for the corresponding geometrical representation of the rectangular loop.

```plaintext
/* Rectangular parallel loop*/
for A = 1 to 10 do
  for B = 1 to 10 do
    loop_body
  end for
end for
```

![Figure 1.3: Geometrical representation of the rectangular loop.](image)

In other cases, the value of the loop’s bound, upper bound and lower bound, depends on the loop’s index, then the loop space is regarded as n-dimensional.

\footnote{Without a specific claim, all loops discussed in this thesis are parallel loops.}
non-rectangle. As a result, a **non-rectangular loop** is a kind of parallel loop in which bounds are dependent on the index, commonly, the execution times of a statement are different by the index value. There are two examples here showing a triangle parallel loop and a trapezoid parallel loop. For more intuitive sight, see Fig. 1.4 for the corresponding geometrical representation of a triangle loop and a trapezoid loop.

```c
/* Triangle parallel loop*/
for A = 1 to 10 do
    for B = 1 to A do
        loop_body;
    end for
end for
```

```c
/* Trapezoid parallel loop*/
for A = 1 to 10 do
    for B = 2 to A + 2 do
        loop_body;
    end for
end for
```

(a) Geometrical representation of the triangle loop.

(b) Geometrical representation of the trapezoid loop.

Figure 1.4: Geometrical representation of non-rectangular loop.
As shown in the above examples, the bound of inner loop is linearly increasing by the surrounding index. Also, this relation is linearly decreasing in some cases decided by the programmer. We can tell that the iteration space would be a triangle in a linearly increasing case, and it would be transformed in a linearly decreasing case. However, the iteration vectors are not always represented in a linear model, and it could be more complex, even random. Thus the parallel loops are not always 'friendly' like this. In a more complex general case, parallel loops can display an unregular shape, saying the bound of inner loops is applying to Gaussian distribution with parameter surrounding index. We denote this special iteration space as Gaussian-type loop. For example, Fig. 1.5 shows a Gaussian-type double loop, the bound of the inner loop is to apply Normal distribution. We will discuss more Gaussian-type loops in Section 2.4, here we provide a short description.

![Figure 1.5: Geometrical representation of Gaussian type loop.](image)

### 1.4.2 Canonical Loop Nest

We have introduced all loop nests’ characteristics in the last section and listed two common loop nest types, rectangle loop nest and non-rectangle loop nest. An even more specific type, the Gaussian-type loop, has been displayed. However, the classification of loop nests has not been told, and in other words, we will introduce a generalized form to classify loop nests. Here we claim the general form of a loop nest in Fig. 1.6 in which $l_2$ is a constant; $L_2A + l_2$ allows the lower bound of index $B$ represent a linear form of index $A$. Please note that we denote the upper bound as $\text{fun}(A)$, the reason to do so is that the upper bound of Gaussian loop can not be described by a linear form $U_2 + u_2$ where $u_2$ is a constant. In the case
of Gaussian loop nest, \( f_{\text{un}}(A) = M f(A) \) where \( M \) is a constant represents a scale; \( f(A) \) is Eq. 2.6.

\[
\text{For } A = L_1 \text{ to } U_1, (\text{stride })
\]
\[
\text{loopbody.1;}
\]
\[
\text{For } B = L_2 A + l_2 \text{ to } \text{fun}(A), (\text{stride})
\]
\[
\text{loopbody.2;}
\]
\[
\text{End For}
\]
\[
\text{loopbody.3;}
\]
\[
\text{End For}
\]

Figure 1.6: The general form of a Triangular/Trapezoid/Gaussian loop nest.

Please note that the loop nests claimed in this section have the general form like Fig.1.6. To make our discussion simple, we assume that the loop bodies labelled with \text{loopbody.1}, \text{loopbody.2}, \text{loopbody.3} do not contain any other loop body, which means there are no variables claimed in these three loop bodies depending on the value of the surrounded index (A, as well as B and C for the second and third level). In this way, the executed contexts of the loop body corresponding to each iteration are the same with each other. What’s more, we assume that \text{loopbody.2} contains at least one computing operation, i.e. it is not empty, making the loop nest effective. Since if it were empty, then the loop nest would degenerate into a single loop, and each iteration of the outermost loop would execute the same; On the contrary, \text{loopbody.1} and \text{loopbody.3} could be empty, in which case, the loop nest would be a perfect loop. Thus we call a loop nest is \textbf{canonical} if and only if all upper bound larger than the corresponding lower bound \[Sak96\]. Compared with the definition given in \[Sak96\], we extend the type of loop nest into a wider range. A more specific definition will be claimed in Section 2.4.

### 1.4.3 General Loop Nest

We have claimed the description of the canonical loop simply. The term canonical loop is essential in this thesis since the rest of the analysis, including detecting and partitioning, is based on canonical loops. Now we bring a new sight that
extends the canonical loop in order to make our study more generalized to satisfy other applications. Refer to a general loop form in Fig. 1.6, for a canonical loop, we have assumed that the loop bodies do not contain any other variables whose value depends on the surrounding index and the outer loop and its inner loops are not empty (the upper bound is larger than corresponding lower bound). Under such assumptions, the canonical loop is defined. We provide the loop nest form in Fig. 1.7.

![Loop Nest](image)

Figure 1.7: The general form of a Triangular/Trapezoid/Gaussian loop nest.

In Fig. 1.7, we assume that the loopbody1 contains another loop whose bound depends on the surrounding index A; the loopbody3 contains a loop whose bound depends on its outer loop’s index A as well. The general loop nest have the general form in Fig. 1.6, but without the restrictions associated with the description of canonical loops.

### 1.4.4 Performance Measurement

As already mentioned in the above subsections, the main motivation for developing a parallel compiler is to reach higher performance on processing parallel programs. It is inevitable to propose a measurement of the performance which meets the requirements of a specific program. A common approach to performance measurement is estimating the overhead associated with parallel computing. The $t_o$ is denoted as the time spent on overheads; $t_s$ represents the time used to process sequential version of a parallel program; then the running time of a
parallel program $t_p$ on $p$ processors is described as:

$$t_p = t_o + t_s/p.$$ 

A more detailed classification of overheads is listed in Crovella’s research:\cite{Cro94}:

- **Insufficient parallelism**: it refers that the overhead is caused by sequential section.
- **Load imbalance**: it refers that the overhead is caused by unequal computation load distributed on processors.
- **Resource contention**: it refers that the overhead occurs when high requirements of system capacity, such as memory, communication channel, or interconnection, and the reserved capacity is limited/unsustainable to support the requirements.
- **Synchronization**: it refers that the overhead is caused when a processor is waiting to acquire a lock or at a barrier.

These four categories of overhead, insufficient parallelism, load imbalance, resource contention, and synchronization, are independent and can be added together. Thus, $t_o$, the total time spends on overhead, can be represented by the sum of these four factors. The definition of the performance estimation model using this property requires four variables that present the illustration of each specific overhead category. These four variables should represent the time spent on overheads that occur while executing a program on a parallel computer. We use $C$ for representing these four variables. Then we denote $C_{IP}$ for the time spent on overhead due to insufficient parallelism; $C_{LI}$ for the time spent on overhead due to load imbalance; $C_{RC}$ for the time spent on overhead due to resource contention; $C_S$ for the time spent on overhead due to synchronization. The estimation for total overhead $t_o$ can be computed by:

$$t_o = C_{IP} + C_{LI} + C_{RC} + C_S. \quad (1.1)$$

The final objective of the parallel compiler is to minimize $t_o$ to improve performance. However, in practice, the performance measurement may not be practical to analyze all variables mentioned in Eq. (1.1). In some cases, performance measurements can evaluate the performance by progressively computing the source of
overhead or by characterized approaches to computing some options of overhead. However, a widespread situation may arise where reducing one source of overhead may increase another. So the performance measurements aim to find the trade-off between these overhead factors as well, attempt to reduce one source of overhead, and avoid another source increase or slightly increase.

As a performance measurement, it is required to provide a large volume of information for the parallel compiler’s developer so that the developer can understand the application precisely enough to form a rough evaluation. In order to achieve such a requirement, the performance measurement tools are not allowed to bring too much information to the user but transform rich information for users by organizing and collecting data. Thus the measurement can assist the compiler’s developer rather than overwhelms them. There are three basic approaches used for measurement, the toolkit approach, the specialized approach, and the characterization approach.

The toolkit approaches such as PPUTTS toolkit, IPS-2 system, and ChaosMON system, implements a series of measurements and presentation methods into one general structure. Two other kinds of toolkits, PIE and ParaGraph, focus on providing multiple presentations for performance. Those toolkits permit the users to analyze performance data from the various value of variables. In the toolkit approach, the reasons which cause the performance degeneration or performance improvement will not be provided usually. It is up to the user to discover the inherent reasons to form a conclusion of performance changing from multiple viewpoints.

On the opposite of the toolkit approach, the specialized approach mainly focuses on one specific overhead category and collects data, then presenting a performance instruction for the user. Those tools are designed that focus on each of the overhead categories, including memory system effects, insufficient parallelism, and load imbalance, synchronization costs, and resource contention. The specialized approaches build for memory system effects aim to take a trade-off between minimizing tool execution costs and acquiring more complete measurements such as Mtool, SHMAP tool, MemSpy, etc. For instance, a number of tools, MaxPar and SPAN, are developed to reveal the parallelism in an application and support user to deal with the code fragments which consist program’s serial fraction. The serial fraction has resulted from insufficient parallelism in which the number of the parallel task is smaller
than the number of available processors or caused by load imbalance, in which a number of tasks are created by application but are not equipped with the same load. A specialized approach to study synchronization cost caused by the different arriving times of tasks is proposed by Ambati and Pradeep et al. Finally, regarding the resource contention which occurs when high demand is generated on a system with limited capacities, such as memory, interconnection network. Sakellariou proposes a scheme in , which studies compiler’s performance when dealing with memory contention by simulating experiments on KSR series multiprocessors.

From the above description, the specialized tools are very useful in performance measurement. However, they usually can not deliver a complete report since they do not measure all overhead factors in one simulation. As a result, specialized tools are restricted to a condition in which the main overhead factors are given initially. Regarding this drawback, the characterization approach focuses on all overhead factors, and meanwhile, assesses them all in every simulation. After evaluating all factors, a description of performance in terms of overhead will be sent to users. The difference of characterization approaches is how to collect information about factors and how to display this information to users.

During the study of parallel compiler, a number of performance measurements have been used to evaluate experiment results, the most common being elapsed time, speed-up, and efficiency.

The elapsed time, which is either called wall clock time, is usually referred to as the program’s total running time period. Elapsed time is the most effective method to describe the performance of the parallel program. Under the same condition, such as the deployed architecture, input program, same interconnection environment, the program with shorter elapsed time achieves better performance. However, the illustration of performance results has been commonly based on measurements derived from the elapsed time. For example, speed-up and efficiency.

The speed-up of a tested parallel program which runs on an architecture with \( p \) processors is denoted as \( S_p \), where

\[
S_p = \frac{t_s}{t_p}.
\]

\( t_s \) is the time spent executing a sequential version of the program; \( t_p \) is the time spent executing the program on \( p \) processors. The value of speed-up normally
relates to the number of processors $p$. If the value of speed-up $S_p$ equals to $p$, then we refer this as linear speed-up. Sometimes the running program results from a value of speed-up greater than $p$ on $p$ processors, in this case, we refer to it as superlinear speed-up.

Finally, the efficiency $E_p$ of a program is defined based on its speed-up:

$$E_p = \frac{S_p}{p}.$$

where $S_p$ is the speed-up of the program; $p$ is the number of processors. From the equation, we can tell that the efficiency $E_p$ is a measurement of the computation’s cost-effectiveness. Not only these three measurements are used to measure the performance of a program, but also their derivatives form \cite{Sak96, CAK17, Bul98}. It is clear that the motivation for performance measurements is analyzing the program deeply and presenting the parallel program’s properties explicitly. However, all these measurements may be useless if the correct way that they have been proposed is unclear. In this thesis, the main objective of any experiment established is to compare the performance, so, we will adopt elapsed time to complete the evaluation.

### 1.5 Thesis Overview

This section provides the motivation of the study in the thesis and presents the final objective of our research; it has also described a short instruction to some basic concepts of parallel computing. All of these basic concepts are the base knowledge of this thesis. The rest of the thesis is built as follows:

Chapter 2 takes into account the partitioning and mapping strategy that happened during compile-time and run-time. The main factor of overhead discusses in this thesis is load imbalance; thus, the overhead due to load imbalance is minimized. Our attention focuses on mapping rectangular loop nests, mapping non-rectangular loop nests, and mapping the Gaussian-type loop nest, a typical complex class of loop nest. Our works form the fundamental study of stereotype loop nest and achieve good load balance under certain conditions.

Chapter 3 provides a complete report on experiments that proposed comparing the performance of the loop schedule schemes developed in Chapter 2 with others, especially those schedule schemes implemented in OpenMP.
In the end, Chapter 4 provides a summary of the work completed in this thesis, also provides directions for future research.
Chapter 2

The Load Balancing Problem

2.1 Introduction

The previous chapter shows a description for parallel computation and shows the basic concepts of our main target, the loop nests. This section will illustrate the problems of dealing with loop nests and introduce strategies for mapping loop nests on parallel processors while minimizing the load imbalance. We claimed the categories of overheads that occurred in processing parallel programs in Chapter 1, the load imbalance is one of the categories caused by poor distribution of parallelized computational work in processors. Thus it is necessary to distribute computational work as even as possible among processors in order to reduce overhead due to load imbalance.

In a parallel computer with $p$ processors, we assume the total workload sent to $p$ processors is $Load$, and the workload distributed on processor $i$ is $Load_i$, where $0 \leq i < p$. If the following condition is satisfied

$$Load_i = \frac{Load}{p},$$

then the load balance is obtained among $p$ processors. In practice, load balance stage is hard to reach, the workload $Load_i$ in processors are not same. Some of processors received a small workload, but others received relative much more workload. We denote the imbalance as $Im$ which describes the difference between
Load\(_i\) and Load/\(p\):

\[
Im_i = Load_i - \frac{\text{Load}}{p}.
\]

When there is a processor \(i\) in which the difference does not equal zero, then the workload distributed among processors is imbalanced. The imbalance level of processor \(i\) is \(Im_i\). A positive \(Im_i\) indicates that the workload distributed on processor \(i\) is more than average \(\text{Load}/p\) and the condition of processor \(i\) may be considered as overload. On the contrary, a negative value of \(Im_i\) indicates that the workload assigned on processor \(i\) is less than average \(\text{Load}/p\), and the process \(i\) is in under-load status. The highest absolute value of imbalance \(Im_i\) among processors (the processor \(i\) which is the longest distance to average value) represents the overall overhead of the parallel computing due to load imbalance. Now we provide the load imbalance \(Im\) of distributing workload \(\text{Load}\) on \(p\) processors:

\[
Im = \max_{0 \leq i < p} |Im_i| = \max_{0 \leq i < p} |Load_i - \frac{\text{Load}}{p}|.
\]  

(2.1)

We apply absolute values in the equation to avoid the special situation that most processors are slightly overloading, but a few processors are extremely underloaded. Without using absolute value, the load imbalance variable will only take into account overhead, but missing consider the processors with underload. Reducing the overhead problem due to load imbalance has been transformed to finding \(\text{Load}_0, \text{Load}_1, \text{Load}_2, \ldots, \text{Load}_{p-1}\) such that the load imbalance \(Im\) is minimized.

The ratio of the overhead due to load imbalance on the total overhead depends on the machine used. From equation 1.1, the load imbalance \(Im\) is one metric corresponding to \(C_LL\). Obviously, even though the final overhead values \(t_o\) are different on different machines, the value of load imbalance \(Im\) may be the same. As a performance metric to evaluate the impact of load imbalance on performance, it is effective to apply a relative load imbalance which is defined as the following:

\[
ImR = \frac{Im}{Load_{\text{argmax}(Im_i)}}
\]  

(2.2)

In equation 2.2, \(Im\) represents the load imbalance; \(Load_{\text{argmax}(Im_i)}\) represents the \(Load_i\) with the biggest difference to average value \(\text{Load}/p\). For example, the load
vector on 4 processors is \([1.5, 1.6, 1.0, 1.5]\), the average load on it is 1.4 then the \(\text{Load}_{\arg\max}(I_m)\) represents the third processor \(i = 2\) and \(\text{Load}_{\arg\max}(I_m) = 1.0\). It can be proved that \(I_{mR}\) takes value from interval \([0, 1 - \frac{1}{p}]\). It is explicit that the value of relative load imbalance \(I_{mR}\) closing to zero means a better load balance and less influence on overhead. In this case, a good performance on parallel program is expected. Regarding to the value close to \(1 - \frac{1}{p}\), it denotes a poor balanced workload distribution happened on processors and nothing improvement is expected from parallelisation.

However, although the factors of overhead, insufficient parallelism, load imbalance, resource contention, and synchronization are independent, the strategy which optimizes the performance of one of these factors may influence others. Commonly, it is inevitable to increase others while reducing one factor of overhead. In the following subsections, one simple loop mapping scheme called round-robin will be introduced. When applying round-robin, the resource contention overhead may arise. So one intuitive requirement for the loop partitioning strategy discussed in the following sections appears that it takes fewer times to assign workload on processors. Meanwhile, the workload assigned to processors is the same or nearly the same.

In order to realize these requirements, Section 2.4 proposes a loop nest partitioning strategy which bases on a state-of-art loop scheduling scheme to distribute the Gaussian workload in every processor are same. The following sections show different schemes for partitioning and mapping loop nest. The performance of schemes is discussed when taking into account different types of loop nests, such as rectangular loop nests, triangular loop nests, even Gaussian type loop nests. All experiments are simulated under the same architecture as described in Section 3.2.

### 2.2 The Problems of Loop Partitioning and Scheduling

The parallel compiler has been promoted to support parallel machines from executing the time-consuming workload in the previous introduction. It aims to transform the inputted sequential programs into some parallel fragments. In this thesis, we have focused on loop nests. In order to accomplish this transformation, we do not have one standard strategy to apply. However, we can adopt
detecting and mapping procedures mentioned in Chapter 1. The detection procedure attempts to collect loop nest information to find which partition can be executed in parallel and form up sequential iterations into some parallel combination. The mapping procedure maps the output of the detection procedure onto multi-processors of a parallel machine.

When discussing the mapping procedure, we assume that a parallel pattern has been detected. These two procedures should not be separated totally; for example, the compiler can decide how to form parallelism from sequential workload if the mapping method is given initially. The outcome of the detecting procedure is a set of loop fragments that can be executed in parallel. During the mapping procedure, the parallel compiler must adopt one strategy to distribute the available loops onto processors in such a way that the overheads mentioned in equation 1.1 are minimized.

When parallel loops are scheduled on multi-processors in parallel in order to reduce the total completion time, the iterations of a loop is partitioned into several chunks, and chunks are mapped to processors; each chunk contains a number of iterations, and the number of iteration in the chunk expresses its size. How to partition a loop into chunks and map the chunks to processors to achieve a balance computation load is a problem. This is an NP-hard problem and is known as Loop Partitioning and Scheduling Problem.

From the parallel compiler’s point of view, when dealing with loop partitioning and scheduling problems, the total progress can be divided into two-stages, including partitioning and scheduling. In stage one, the partitioning stage, the sequential iterations are formed up into several subsets; see the abstract view of partitioning in Fig. 2.1. In stage two, the scheduling stage, the partitioned chunks are mapped on processors; see the abstract view of scheduling in Fig. 2.2 (The numbers in boxes represent the sequence of the iteration.). Regarding partitioning and scheduling, we derive three possible options to deal with loop partitioning and scheduling problem based on the definition given by [Sar89]: firstly, the partitioning and scheduling procedure are both happened in compile-time; secondly, the iterations are partitioned at compile-time and postpone the scheduling procedure until running time; thirdly, there is nothing to do during compile-time that both procedure, partitioning and scheduling, are postponed until running time. The main reason we delay the starting time point of scheduling to running time is that the available information is collected at different time points. At compile
time, without receiving the outcome of the partitioning procedure, there is no additional information to support schedule iterations. It may produce a better load balance on each processor if postponing the scheduling procedure until receiving the outcome from the partitioning procedure, even though the running time has to be extended.

Figure 2.1: The abstract view of partitioning.

Figure 2.2: The abstract view of scheduling.

The loop partitioning and scheduling problem has received extensive attention in the research community. Many researchers have put their efforts to develop general approaches for reducing load imbalance when partitioning and scheduling loops on a given parallel machine. However, no single loop partitioning and scheduling technique can reduce load imbalance from all types of a parallel program to effectively optimize the parallel application’s performance. Indeed, the
influence of schemes for loop partitioning and scheduling on performance is determined by the characteristics of the loop nests and potential running architecture characteristics. Commonly, the proposed schemes’ performance would degenerate if there is a change to the running environment, such as deployment on another machine or running another type of loop nest.

A short description of several loop partitioning and scheduling schemes is provided below in order to illustrate a direct recognition of problems of loop partitioning and scheduling. One crucial factor is that whether to postpone the scheduling procedure until the running time, based on which, we distinguish the schemes mentioned by two sections, static loop partitioning and scheduling schemes (in which scheduling decisions are made during compile-time); dynamic loop partitioning and scheduling schemes (in which scheduling decisions are made during the running time).

2.2.1 Static Approach for Appeasing Problems of Loop Partitioning and Scheduling

A simple method to distribute loops on parallel processors as evenly as possible is by assigning a fixed number of iterations at every time. Given a loop nest, there exist multiple approaches to distribute iterations on the processors in balance. Here we introduce two simple main approaches to reveal how the proposed approaches managing load balancing in processors and the bottleneck of each approach. The first approach of loop partitioning and scheduling by distributing iterations onto processors in a round-robin rule. For example, given $p$ processors and the total number of iteration is $N$; the first iteration will be executed by processor 0, the second will be executed by processor 1, the $p+1$th iteration will be assigned to processor 1 again, and so on. The final distribution of total iterations is that processor 0 executes 1th, $p+1$th, $2p+1$th, $3p+1$th, ... iterations; processor 1 executes 2th, $p+2$th, $2p+2$th, $3p+2$th, ... iterations; processor $m$ executes $m$th, $p+m$th, $2p+m$th, $3p+m$th, ... iterations, where $0 \leq m < p$; We denote this approach as **Cyclic Round-Robin**. From the example above, we can tell that the load balance performance of Cyclic Round-Robin is acceptable; the difference in the number of iterations (computation workload) on processors is smaller than 1. Even the load on each processor will reach perfect load balance when $N$ is a multiplied number of $p$, i.e. $\text{mod}(N,p) = 0$. To illustrate the efficiency
of Cyclic Round-Robin when dealing with the parallel program, we consider the code fragment in Fig. 2.4a, which executes the same work in each iteration. As we are using OpenMP to parallel the loop nest, here we provide the structure of the OpenMP clause to illustrate how it performs in IDE, see Fig. 2.3. In the rest of experiments, we get output by applying different loop scheduling strategies in the method argument also we try different loop nests in the For loop nest part. The A loop was parallelized by distributing the iterations among processors using Cyclic Round-Robin. The corresponding performance (execution time) result of Cyclic Round-Robin for \( N = 200000 \), and \( p = 1, 2, 4, 8, 16 \) is displayed in Fig. 2.4a. It can be seen that the total execution time of sample code in Fig. 2.4a is reduced while increasing the number of processors \( p \). However, the decrements are different in different processor numbers. We can tell the execution time reduced sharply between 2 processors and 4 processors, and the decrement is very small between 8 processors and 16 processors. The reason causing this we have revealed in Section 1.4.4; the total overheads can be classified into four categories, such as insufficient parallelism, load imbalance, resource contention, synchronization in Eq. 1.1; and reducing one of the factors may affect another. In order to illustrate this problem, we modify the number of iteration to a small number, saying \( N = 1000 \). In this case, the ratio of overhead caused by load imbalance is small, thus the improvements of the parallel compiler which aims to reduce load imbalance among processors, on overheads are relatively weak. Unfortunately, the impact of reducing load imbalance may cause negative effects on running performance such as extending the execution time. The performance of cyclic round-robin for \( N = 1000 \), and \( p = 1, 2, 4, 8, 16 \) is provided in Fig. 2.5.

The second approach is to map consecutive iterations onto processors, in which processor 0 executes iterations from 1th to \( N/p \)th, processor 1 executes iteration from \( N/p + 1 \)th to \( 2N/p \)th, processor \( m \) will execute from \( mN/p + 1 \)th to \( (m + 1)N/p \)th, where \( 0 \leq m < p \). This approach is referred to as Static Block Scheduling. Compared with cyclic round-robin, static block scheduling is simple to realize.

Figure 2.3: OpenMP clause.
(a) Sample code for Cyclic Round-Robin:

\[
\text{For } A = 1 \text{ to } N \\
\text{For } B = 1 \text{ to } N \\
\text{Const } += 1 \\
\text{End For} \\
\text{End For}
\]

(b) Execution time for \( N = 200000 \), Cyclic Round-Robin.

Figure 2.4: The efficiency of Cyclic Round-Robin.

Figure 2.5: Execution time for \( N = 1000 \), Cyclic Round-Robin.
and implement in parallel programs. However, the improvement provided by static block scheduling on load imbalance overhead is affected by the system's architecture; it will generate a poor load balance level in some extremely harsh conditions. We will provide evidence for this argument later. First, we will show the efficiency of static block scheduling on parallel computing. Again, we consider the code fragment in Fig. 2.4a which executes the same work in each iteration; loop A was parallelized by distributing the iterations among processors using static block scheduling. The corresponding execution time for \( N = 200000 \), and \( p = 1, 2, 4, 8, 16 \) is illustrated in Fig 2.6. It is the same with the cyclic round-robin approach that the program’s execution time is reduced while increasing the number of processors, and the decrements are different at different number points in Fig 2.6.

![Figure 2.6: Execution time for \( N = 200000 \), Static Block Scheduling.](image)

In the previous paragraph, we mentioned a problem that the characteristics of running architecture limit the improvements provided by static block scheduling on load imbalance overhead. Regarding this, the performance on load balance is similar to the cyclic round-robin approach that a perfect load balance would be obtained when the total number of iterations \( N \) is a multiplied number of processor number \( p \) and a load imbalance would occur when \( N \) is not. The static block scheduling approach assigns a block of consecutive iterations to each processor. It will cause \( N/p \) imbalance at max (recall definition in Section 2.1, the imbalance is the difference between \( Load_i \) and \( Load/p \)). For example, we deliver one single loop with iteration number \( N = 23 \) in one parallel computer with \( p = 4 \).
processors, and the chunk size $D = 5$; the fragment of code we refer Fig. 2.7a. The corresponding performance on load balance is displayed in Fig. 2.7b. During the usage of static block scheduling, the parallel compiler derives the scheduling-block size for loop $A$ that first 5 iterations are distributed on processor 0; iteration 6th through 10th are distributed on processor 1; iteration 11th through 15th are distributed on processor 2; iteration 16th through 20th are distributed on processor 3. However, there are still 3 iterations, 21th through 23th, waiting to be distributed. Based on the static block scheduling approach, all 3 left iterations are assigned on processor 0 (under chunk size 5). Regarding this, cyclic round-robin assigns iterations one by one; it will assign the last three iterations on processor 0, processor 1, and processor 2. This problem is one inherent barrier inside the parallel compiler; we will discuss the approach to solve this kind of barrier in Section 2.3.

(a) Sample code for Static Block Scheduling.

(b) Load distribution for $N = 23$, Static Block Scheduling.

Figure 2.7: The load imbalance of Static Block Scheduling.

Till now, the cyclic round-robin performs better than static block scheduling regarding total execution time and load imbalance. However, in many parallel machines, the former approach, the cyclic round-robin, suffers performance degradation due to false sharing [Sak96]. False sharing is a situation that can occur during a computer running two applications simultaneously, and both applications are accessing the same data in a logical memory region (one of them
is writing). When the data is modified by one application, the copied data in another CPU cache is missing and it has to be reloaded again. To reveal the performance degenerated by false sharing we consider to run the code fragment in Fig. 2.8 with $N = 20000$ on $p = 1, 2, 4, 8, 16$ processors. The code is implemented with add operator between two matrixes with size $N$. When thread access the address of $\text{Matrix1}$, a block which is a sequence of consecutive addresses is loaded in cache. After modifying the value in the block, another thread’s cache which loads the corresponding address occurs missing. Then the thread has to reload the memory to update. The loop $A$ was parallelized by distributing the iterations among processors using cyclic round-robin and static block scheduling approach. The final result are displayed in Fig. 2.8. It shows that static block scheduling obtains a better performance by assigning consecutive iterations on processors compared with cyclic round robin.

![Figure 2.8: False sharing influence on Cyclic Round Robin and Static Block Scheduling.](image)

Readers may notice that all code fragments showed in Fig. 2.4a, Fig. 2.7a, and Fig. 2.8a are rectangular loop nests, in which each iteration of parallel loop implemented with the same amount of computation work, in which case partitioning and scheduling can be performed by a parallel loop with an invariant workload for each iteration. The distribution of these loop nests on processors is nearly the same. However, in the real world, loop nests are not always created like this. For example, a canonical loop in which the upper bound of each depth is larger or equal than lower bound, and the inner loop within outermost loop performs
a different amount of work due to the index of its outermost loops, such as the triangle loop nest and trapezoid loop nest mentioned in Section 1.4.1. In the case of a loop nest, which consists of an outer loop and an inner loop which running depends on the index value of the outer loop, Rizos [Sak96] suggests a creative balanced partitioning and scheduling scheme which called Canonical Loop Scheduling scheme which partitions iterations into $p$ equal size partitions by the outermost loop, see Theorem 2.4.1. Same with cyclic round-robin and static block scheduling, canonical loop scheduling generates the scheduling strategy at the compile time. This scheme aims to distribute the total number of iterations on processors as even as possible. The innovative point of this approach is that it does not balance the number of iterations distributed on each processor but balance the workload distributed on each processor, which means the number of iteration distributed on processors are various while the amount of workload on each processor are same.

Now we will discuss the performance of load balance overhead of three different static schemes, cyclic round-robin, static block scheduling, and canonical loop scheduling, on dealing with various loop types. To illustrate this, we consider the code fragment shown in Fig. 2.9a, with $N = 20000$ on $p = 1, 2, 4, 8, 16$ processors. Here we are using the distribution of iterations by 4 processors as an example. By cyclic round-robin approach, the assignment of iterations on processors is illustrated in Table 2.1 in which the assignment of iterations on 0 is $A = 1, 5, 9, 13, 17, \ldots, 19997$; the assignment of iterations on processor 1 is $A = 2, 6, 10, 14, 18, \ldots, 19998$; the assignment of iterations on processor 2 is $A = 3, 7, 11, 15, 19, \ldots, 19999$; the assignment of iterations on processor 3 is $A = 4, 8, 12, 16, 20, \ldots, 20000$; in addition, the total number of times of executing loop body on processor 0 is $9.9 \times 10^7$; on the processor 1 is $10^8$; on processor 2 is $1.0001 \times 10^8$; on processor 3 is $1.0002 \times 10^8$. The corresponding iterations’ distribution on processors assigned by static block scheduling is illustrated in Table 2.2 that processor 0 executes $A = 1$ through $A = 5000$; processor 1 executes $A = 5001$ through $A = 10000$; processor 2 executes $A = 10001$ through $A = 15000$; processor 3 executes $A = 15001$ through $A = 20000$; The times of executing a loop body is $2.5005 \times 10^7$ on processor 0; is $7.5005 \times 10^8$ on processor 1; is $1.25005 \times 10^9$ on processor 2; is $1.75005 \times 10^9$ on processor 3. One amazing distribution of workload is showed in Table 2.3 which is achieved by canonical loop scheduling, that processor 0 executes $A = 1, 5, 9, 13, 17, \ldots, 19996, 20000$ with total number $1.0005 \times$
CHAPTER 2. THE LOAD BALANCING PROBLEM

Table 2.1: The assignment of iterations on 4 processors by cyclic round-robin approach.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Iterations</th>
<th>Total number of execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor 0</td>
<td>$A = 1, 5, 9, 13, 17, \ldots, 19997$</td>
<td>$9.9 \times 10^6$</td>
</tr>
<tr>
<td>Processor 1</td>
<td>$A = 2, 6, 10, 14, 18, \ldots, 19998$</td>
<td>$10^8$</td>
</tr>
<tr>
<td>Processor 2</td>
<td>$A = 3, 7, 11, 15, 19, \ldots, 19999$</td>
<td>$1.0001 \times 10^8$</td>
</tr>
<tr>
<td>Processor 3</td>
<td>$A = 4, 8, 12, 16, 20, \ldots, 20000$</td>
<td>$1.0002 \times 10^8$</td>
</tr>
</tbody>
</table>

Table 2.2: The assignment of iterations on 4 processors by static block scheduling.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Iterations</th>
<th>Total number of execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor 0</td>
<td>$A = 1, 2, 3, 4, 5, \ldots, 5000$</td>
<td>$2.5005 \times 10^8$</td>
</tr>
<tr>
<td>Processor 1</td>
<td>$A = 5001, 5002, 5003, 5004, 5005, \ldots, 10000$</td>
<td>$7.5005 \times 10^8$</td>
</tr>
<tr>
<td>Processor 2</td>
<td>$A = 10001, 10002, 10003, 10004, 10005, \ldots, 15000$</td>
<td>$1.25005 \times 10^9$</td>
</tr>
<tr>
<td>Processor 3</td>
<td>$A = 15001, 15002, 15003, 15004, 15005, \ldots, 20000$</td>
<td>$1.75005 \times 10^9$</td>
</tr>
</tbody>
</table>

Table 2.3: The assignment of iterations on 4 processors by canonical loop scheduling.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Iterations</th>
<th>Total number of execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor 0</td>
<td>$A = 1, 3, 9, 13, 17, \ldots, 19996, 20000$</td>
<td>$1.0005 \times 10^8$</td>
</tr>
<tr>
<td>Processor 1</td>
<td>$A = 2, 6, 10, 14, 18, \ldots, 19995, 19999$</td>
<td>$1.0005 \times 10^8$</td>
</tr>
<tr>
<td>Processor 2</td>
<td>$A = 3, 7, 11, 15, 19, \ldots, 19994, 19998$</td>
<td>$1.0005 \times 10^8$</td>
</tr>
<tr>
<td>Processor 3</td>
<td>$A = 4, 8, 12, 16, 20, \ldots, 19993, 19997$</td>
<td>$1.0005 \times 10^8$</td>
</tr>
</tbody>
</table>

10^8 of times of execution; processor 1 executes $A = 2, 6, 10, 14, 18, \ldots, 19995, 19999$ with total number $1.0005 \times 10^8$ of times of execution; processor 2 executes $A = 3, 7, 11, 15, 19, \ldots, 19994, 19998$ with total number $1.0005 \times 10^8$ of times of execution; processor 3 executes $A = 4, 8, 12, 16, 20, \ldots, 19993, 19997$ with total number $1.0005 \times 10^8$ of times of execution. To the more explicit presentation of the load imbalance performance of these three approaches, we provide the load distribution graph, in Fig. 2.10, based on the total number of times of executing a loop body in processors.

In the case of more general loop nests, such as triangle/trapezoid loop nest, traditional parallel compilers may not be able to generate a distributing strategy during compile-time. Apart from variances of the inner loop nest’s upper bound, additional overheads may affect the compiler’s performance in multiple ways. Thus researchers have moved on to studying distributing iterations after compile-time.
(a) Sample code for triangle loop nest.

\begin{verbatim}
  For A = 1 to N 
    For B = 1 to 2A 
      Const++;
    End For
  End For
\end{verbatim}

(b) Performance of schemes on triangle loop nest.

Figure 2.9: Performance of three schemes on triangle loop nest.

Figure 2.10: Total number of times of executing loop body on each processor.
2.2.2 Dynamic Approach for Appeasing Problems of Loop Partitioning and Scheduling

Unlike static approaches, which generate a strategy for distributing iterations during compile-time, the dynamic approach postpones this procedure, distributing, until run-time. In this way, the parallel compiler can collect more information about the parallel program and parallel machine to support building distributing strategy. A simple dynamic approach to achieve load balance among processors during the running program is to schedule the next waiting iteration to the idle processor at a time and cycle this procedure until all iterations are distributed; this approach is known as Pure Dynamic Scheduling. For example, given $p$ processors and the total number of iteration is $N$; the first $p$ iterations will be assigned to $p$ processors separately; then the next iteration $p+1$ will be assigned to processor 2 if processor 2 firstly became idle; the iteration $p+2$ will be assigned to the next idle processor. From the example above, it is clear that pure dynamic scheduling can reduce load imbalance since it avoids a processor be idle for a long time and keeps all processors are working, even though the number of iterations assigned on each processor is not the same. This property is similar to canonical loop scheduling so that dynamic approaches can perform well even in general loop nests such as triangular/trapezoid loop nests. To illustrate the efficiency of pure dynamic scheduling when dealing with the parallel program, we consider the code fragment in Fig.2.4a which executes the same work in each iteration. The A loop was parallelized by distributing the iterations among processors using pure dynamic scheduling. The corresponding performance (execution time) result of pure dynamic scheduling for $N = 200000$, and $p = 1, 2, 4, 8, 16$ is displayed in Fig.2.11. It can be seen that the total execution time of sample code in Fig.2.4a is reduced while increasing the number of processors $p$. However, the decrements are different in different processor numbers. We can tell the execution time reduced sharply between 2 processors and 4 processors, and then the decrement becomes very small between 8 processors and 16 processors. The reason causing this we have revealed in Section 1.4.1. Similar to the relation between static block scheduling and cyclic round-robin, another classic dynamic approach manages load imbalance overhead by assigning a number of iterations to an idle processor during run-time. This approach is called Dynamic Chunk Scheduling, it maps consecutive iterations onto processors, normally $D$ number of iterations, where $D$ is the chunk size; in which processor 0 executes iterations from
CHAPTER 2. THE LOAD BALANCING PROBLEM

Figure 2.11: Execution time for $N = 200000$, Pure Dynamic Scheduling.

1th to $D$th, processor 1 executes iteration from $D + 1$th to $2D$th iteration, processor $m$ will execute from $mD + 1$th to $(m + 1)D$th, where $0 \leq m < p$. Compared with pure dynamic scheduling, dynamic chunk scheduling reduces the number of times required to access the centre queue; the larger the chunk size $D$, the fewer times required. We knew that dynamic chunk scheduling would be transformed into pure dynamic scheduling when $D = 1$; and it can achieve good load balance by keeping all processors with equal working time. However, it may achieve poor performance when $D = 1$, due to running overheads. Thus determining an appropriate value of chunk size $D$ is a trade-off between parallel compiler’s performance and load balance performance. To illustrate this performance tradeoff between execution time and load balance, we consider to run the code fragment in Fig. 2.12a with $N = 200000$, $p = 1, 2, 4, 8, 16$, in which the upper bound of inner loop is a multiple by the index of outer loop, $U_2 = 3A$. Besides this, we set up five different chunk size $D$ to reveal the performance variation with a different value of chunk size. Taking $D = \frac{N}{5p}$ as an example, the first iteration through $\frac{N}{5p}$th will be assigned to processor 0; the iteration $\frac{N}{5p} + 1$th through $2\frac{N}{5p}$th will be assigned to processor 1; the iteration $m\frac{N}{5p} + 1$th through $(m + 1)\frac{N}{5p}$th will be assigned to processor $m$, where $0 \leq m < p$. After finishing the first round assignment, the iteration $\frac{N}{5} + 1$th through $(p + 1)\frac{N}{5p}$ will be sent to the first idle processor, instead of the processor 0. The corresponding execution time at the different number of processors with different chunk size $D$ is provided in Fig. 2.12b. From the performance figure, one important variable of program performance, execution time, is
decreasing while the dynamic chunk scheduling approach applies a smaller chunk size of $D$. The reason to explain this result is that loop $A$ is evenly or nearly evenly distributed on processors, and a smaller value of chunk size can perform a more balanced distribution.

(a) Sample code for triangle loop nest.

(b) Execution time for $N = 200000$, Dynamic Chunk Scheduling.

Figure 2.12: Performance of Dynamic Chunk Scheduling with various chunk size.

In order to confirm our explanation, we set up another experiment to study the load distribution among processors when apply dynamic chunk scheduling with different value of chunk sizes $D$. To illustrate the load difference in each processor, we consider to run the code fragment in Fig. 2.13a with $N = 200000$, $p = 4$, and $D = 1, \frac{N}{20p}, \frac{N}{10p}, \frac{N}{5p}, \frac{N}{p}$. The parameter $m$ in the code fragment represents the current thread number running loop body. For example, assuming iteration $A = 1, B = 2$ are assigned to processor 0, then $m = 0$ while running the iteration $A = 1, B = 2$. The corresponding workload distribution of various chunk sizes $D$ is displayed in Fig. 2.13b; recall the relative load imbalance which we designed to access load imbalance among processors:

$$ Im_R = \frac{Im}{Load_{\arg\max}(Im)}.$$ 

To evaluate the load imbalance among processors, we show the relative load imbalance resulted by different chunk sizes $D$ in Fig. 2.14. It can be seen that the largest chunk size $D = \frac{N}{p}$ achieves the worst load balance; the smallest chunk sizes $D = 1$ achieves the best load balance. An evenly or nearly evenly distribution of workload can reach low load imbalance overhead, the total execution time can be
CHAPTER 2. THE LOAD BALANCING PROBLEM

reduced. However, the improvements of dynamic chunk scheduling’s performance vary at different chunk sizes $D$; even though $D = 1$ achieves the best load balance, its improvement is relatively limited compared with $D = \frac{N}{5p}$.

(a) Sample code for triangle loop (b) Total number of times of executing loop body on each processor.

Figure 2.13: Iteration’s distribution by Dynamic Chunk Scheduling with various chunk size.

Figure 2.14: Relative load imbalance.

The advantage of the dynamic approach is that each processor’s idle duration is minimized, resulting in a shorter execution time. Its disadvantage is that it incurs a high running overhead since the processors have to access the centre queue, which stores iterations a considerable number of times while the total number of iterations $N$ is enormous. Thus determining the value of chunk size
means finding the balance point between performance on execution time and load imbalance; during this phase, the programmer needs to be armed with much more weight on the efficiency of dynamic scheduling approaches.

2.2.3 Discussion

Dynamic loop schedule approaches focus more on distributing an evenly working time for each processor at run-time; however, achieving this objective has to consume additional overheads, such as resource contention overhead. In the parallel compiler context, where strategies are based on an economic model, the dynamic approaches can not provide appropriate supports. Thus, when taking into account modelling communication costs, programmers have preferred to apply static approaches for managing parallel programs. However, a static loop schedule approach may not distribute work among processors evenly that the running time of each processor is unequal, but static approaches always bring less resource contention overhead; and it will achieve better performance if taking into account architecture information and program information. That is to say, as a static loop partitioning and scheduling scheme, the less volume of information about loop nests and architecture of running machine result two challenges: (i) an inconsistent computation load of chunks; (ii) an inconsistent computation load assigned to processors. In other words, when the workload of each chunk partitioned during compile-time is partially unequal, unexpected load imbalance may happen, which dramatically degrades the efficiency of scheduling. In order to mitigate the impact of the mentioned challenges in loop partitioning and scheduling, a proper algorithm must be proposed for both partitioning and scheduling steps. A fine-grained decomposition in combination with an appropriate scheduling strategy can realize this goal.

2.3 Loop Partitioning and Scheduling Schemes

Based on the simulation results in Section 2.2, it is explicit that reducing load imbalance overhead is a critical approach to improve the performance of parallel program; one effective parallel compiler, or be more specific, the loop partitioning and scheduling approach takes responsibility for reducing load imbalance. Also, the performance of static and dynamic loop partitioning and scheduling approaches shows that postponing the scheduling procedure until run-time can achieve better
load balance since the greater volume of information about the loop is available, the easier it is to schedule the iterations with a greater level of load balance; but the improved performance is at the cost of a considerable number of times to access centre queue which dramatically increasing the resource contention overhead. We have introduced a part of static and dynamic loop partitioning and scheduling approaches in Section 2.2; in the following subsections, we classify the proposed approaches by run-time schemes and compile-time scheme, and introduce the details of each loop partitioning and scheduling approach. As the bridge connecting problems of loop partition and schedule mentioned in Section 2.2, and our new proposed scheme, combined canonical loop scheduling scheme in Section 2.4, this section aims to compare the load imbalance achieved by various partitioning schemes with a different type of loop nest.

Loop scheduling algorithms boil down to the following two categories: *Static scheme*, in which iterations are partitioned and scheduled to processors during compile time; and *Dynamic scheme*, in which iterations’ scheduling strategy is postponed until run-time. Here, some classical static and dynamic loop scheduling approaches are discussed in the following subsections.

### 2.3.1 Static schemes

- **Cyclic Round Robin (CRR)** [BMSB11]: It is the simplest approach to partition and schedule loop nest among processors as even as possible. Assuming the upper bound of outer loop nest is $N$; the total number of processors is $p$; then $m$th processor, $0 \leq m < p$, will be assigned with iteration $m + 1 + kp$, $k = 0, 1, 2, \ldots, N/p - 1$.

- **Static Block Scheduling (SBS)** [LTSS93]: Its working algorithm is similar to cyclic round-robin; during compile time, a number of consecutive iterations are partitioned in one block which with the size of $N/p$ normally, where $n$ and $p$ indicates the total number of iterations and the number of processors, respectively. Then the $p$ generated blocks are distributed on $p$ processors.

- **Canonical Loop Scheduling (CLS)** [SG97]: Unlike cyclic round robin and static block scheduling which aims to distribute same number of iterations on processors. The canonical loop scheduling approach aims to distribute the workload evenly among processors. Assuming the upper bound
of outermost loop is $N$; total number of processors is $p$; then the processor 0 will be assigned with iteration $\{A < \frac{N}{2}|0, p, 2p, \ldots\} \cup \{\frac{N}{2} \leq A \leq N|N, N - p, N - 2p, \ldots\}$; the processor 1 will be assigned with $\{A < \frac{N}{2}|1, p + 1, 2p + 1, \ldots\} \cup \{\frac{N}{2} \leq A \leq N|N - 1, N - 1 - p, N - 1 - 2p, \ldots\}$; the processor $m$, $0 \leq m < N$ will be assigned with $\{A < \frac{N}{2}|m, p + m, 2p + m, \ldots\} \cup \{\frac{N}{2} \leq A \leq N|N - m, N - m - p, N - m - 2p, \ldots\}$.

- **Automatic Collapsing Loop Partitioning (ACLP)** \cite{CAK17}: Automatic collapsing loop partitioning is referred to as a supported algorithm to improve the performance of proposed loop partitioning and scheduling approaches by combining some loops that are perfectly nested into one single loop. This approach performs well when dealing with high dimensional loop nests. Taking the loop nest in Fig. 2.15 as an example, automatic collapsing loop partitioning will collapse both $A$ and $B$ loops into one single loop running \((N-1)N/2\) iterations. The original index value $A$ and $B$ will be presented by single loop index $u$ by calculating the following expressions:

$$A = \left\lfloor -\frac{\sqrt{4N^2 - 4N - 8u + 9} - 2N + 1}{2} \right\rfloor$$

$$B = \left\lfloor -\frac{2AN - 2u - A^2 - 3A}{2} \right\rfloor.$$

After collapsing the original loop nest, the proposed approaches such as cyclic round robin or static block scheduling can be applied on new loop nest to distribute iterations on processors.

![Figure 2.15: Sample code for Automatic collapsing loop partitioning.](image)

### 2.3.2 Dynamic schemes

- **Pure Dynamic Scheduling (PDS)** \cite{PAGC19}: In this method each iteration is scheduled during running time. Assuming the total number of iterations...
is $N$; the number of processors is $p$; the first $p$ iterations will be assigned to processors and the $p+1$th iteration will wait to be distributed until one of processor become idle. That is to say, whenever a processor becomes idle, the next waiting iteration is assigned to it. Although, this strategy achieves good load balancing, it suffers from high run-time overhead. We have discussed this disadvantage in the last section.

- **Dynamic Chunk Scheduling (DCS)** \cite{PAGC+19}: This method first partitions iterations into chunks with equal size $D$, then the first $p$ generated chunks will be assigned to $p$ processors; similar with pure dynamic scheduling, the $p+1$th chunk will wait to be distributed until one of processor become idle. The chunk size is one important factor for the performance of dynamic chunk scheduling. Partitioning loops into smaller chunks will reduce the load imbalance overhead and guarantee a good load balance but at the price of increasing resource contention overheads. In contrast, partitioning loops into larger chunks can obviate this problem but can not reduce load imbalance overhead effective. Based on the DCS’s performance in Fig.2.12b and Fig.2.14, we fix the chunk size $D = N/(5p)$ to test the performance of DCS on the rest paper.\footnote{DCS is a generalization of PDS in which chunk size equals one.}

- **Guided Self-Scheduling (GSS)** \cite{PK87}: Regarding the tradeoff problem in dynamic chunk scheduling, guided self-scheduling tries to propose a tradeoff between reducing load imbalance and increasing resource contention overhead by decreasing each chunk’s size during the run-time. In this approach, iterations are dynamically partitioned into chunks during run-time, and chunks are assigned to processors on demand. Each chunk’s size is calculated as $r/p$, where $p$ and $r$ are the number of processors and the number of remained iterations, respectively. For example, given $N$ iterations and $p$ processors; the first processor will be distributed with a chunk which contains $\lceil N/p \rceil$ iterations, the remaining number of iterations is $r = N - \lfloor N/p \rfloor$; then the second processor will be distributed a chunk which contains $\lceil r/p \rceil$ iterations, and so on. In this way, the chunk size will be reduced to 1. However, the initial chunks’ size is significantly large in this approach.

Now we have displayed all the classical loop partitioning and scheduling approaches; most of them have been implemented in an open-source library to
CHAPTER 2. THE LOAD BALANCING PROBLEM

support dealing with parallel programs. Regarding the performance of these approaches, there is no ‘one-for-all’ scheme, which means a specific approach leads to the best performance on reducing load imbalance in all kinds of situations, including loop nest shape and the parallel machine’s architecture. However, regarding the specific loop nest and the parallel machine’s specific architecture, we can find the best performed one on reducing load imbalance from these well-known classical loop partitioning and scheduling schemes. In order to evaluate these different approaches’ performance on various workloads, we implement the performance comparison in the next section. Also, this performance comparison section acts a good role in deriving our new extended loop partitioning and scheduling scheme.

2.3.3 Performance Comparison

The content presented in Section 2.3 provides a detailed description about each static and dynamic approaches; same with the preview of loop partitioning and scheduling approaches in Section 2.2, these static and dynamic approaches will expose the same problem, such as the increment of performance is decreasing (in other words, the increments are not significant like increasing the number of processor form one to two) while adding more parallel processors in a machine. All approaches are inevitable to reduce load imbalance among processors at the price of rising the number of accessing centre queue. This shows lack of scalability and the performance of reducing load imbalance degenerates while dealing with other particular types of loop nests. Even though all approaches will expose these problems, but the effects on load imbalance and execution time are various; some can keep performing acceptable good performance, some can not. What’s more, while managing the same loop nest with the same parallel architecture, we can find one that has the best performance among them.

In order to illustrate this, consider the code fragment shown in Fig. 2.16, which implemented two different loop nest types including rectangle, triangle; three different loop body including vector addition in Fig. 2.16a, adjoint convolution in Fig. 2.16b, and upper triangular matrix multiplication in Fig. 2.16c. Assuming that the upper bound of the outermost loop nest in vector addition and adjoint convolution are \( N = 10000 \); the upper bound of the outermost loop nest in the upper triangular matrix is \( N = 4000 \); the options of the total number of processors are \( p = 1, 2, 4, 8, 16 \). We will run these three provided code fragments by applying...
Figure 2.16: Sample code for performance comparison.

seven different schemes on the parallel machine with varying numbers of processors \( p \). In order to study the performance on execution time and increment of performance, the execution time of the parallel program achieved by each scheme at a different number of processors will be recorded. Besides this, we will fix the number of a total number of processors, saying \( p = 4 \), to study the load distribution achieved by different schemes on four processors, to illustrate each scheme’s performance on reducing load imbalance.

A rectangle loop nest, which executes the same amount of workload at each iteration, simplifies the complexity of loop partitioning and scheduling schemes when distributing iterations; it acts as a very common benchmark program to test the performance of the loop partitioning and scheduling schemes. Based on the Fig.2.17 all schemes achieve similar execution time on executing loop nest; the corresponding load distribution among 4 processors is displayed in Fig.2.18. It is
clear that all static approaches perform well in terms of load balance regarding distributing rectangular loop nests. Triangle loop nest, which executes a different amount of workload at each iteration, raises the complexity for loop partitioning and scheduling schemes when distributing iterations; it also acts as a very common benchmark program to test the performance of loop partitioning and scheduling schemes. As a benchmark program to test the scalability of loop schedule schemes, it can reveal each scheme’s performance not only on execution time but also the load imbalance level. Based on Fig. 2.19, pure dynamic scheduling schemes achieve
the worst execution time on executing loop nest even though the corresponding load imbalance relatively low in Fig. 2.20. Among these loop scheduling schemes, canonical loop scheduling achieves good performance on execution time while keeping an evenly load distribution among processors. Even though the increment of performance decreases while increasing the number of processors, it shows a piece of evidence that it is possible to achieve high performance on executing loop nests while reducing the load imbalance. In order to test the scalability on a more general form, saying a depth 3 triangle loop nest, we apply all schemes on processing triangular upper triangle matrix multiplication.

![Figure 2.19: Execution time for $N = 10000$ triangular adjoint convolution.](image)

![Figure 2.20: Total number of times of executing loop body on each processor, triangular adjoint convolution.](image)
Chapter 2. The Load Balancing Problem

Triangular upper triangle matrix multiplication which executing different amount of workload at each iteration where there are two inner loops whose number of the index depends on its outer loop nest rises the complexity for loop partitioning and scheduling schemes; it implemented with the multiplication of two upper triangular $n \times n$ matrices; it also acts as a very common benchmark program to test the performance of loop partitioning and scheduling schemes. As a benchmark program to test the scalability of loop schedule schemes, it can reveal each scheme’s performance not only on execution time but also the load imbalance level. Based on Fig. 2.19, pure dynamic scheduling schemes achieve the best execution time on executing loop nest. Among these loop scheduling schemes, canonical loop scheduling also achieves good performance on execution time while keeping an evenly load distribution among processors.

Figure 2.21: Execution time for $N = 4000$ triangular upper triangle matrix multiplication.

Canonical loop scheduling is limited in that it can not be applied on more general loop nests, saying the upper bound of the inner loop is not a simple function of the index of the outer loop. In the case of more general loop nests, the loop scheduling schemes should be extended to reduce the load imbalance among processors. We will discuss a more general loop nest type, called Gaussian loop type, in the following sections.
CHAPTER 2. THE LOAD BALANCING PROBLEM

Figure 2.22: Total number of times of executing loop body on each processor, triangular upper triangle matrix multiplication.

2.4 Partitioning Gaussian Type Loop Nests

The partitioning and scheduling schemes described in Section 2.3 all achieve good performance on execution time and a good evenly load distribution or nearly evenly load distribution when the input parallel program is rectangular loop nest; however, the performance of some schemes, such as cyclic round-robin, static block scheduling, pure dynamic scheduling, dynamic chunk scheduling degenerate while processing non-rectangular loop nest. Only canonical loop scheduling keeps good performance on processing this kind of non-rectangular loop nest which the upper bound of its inner loop nest is a simple linear function of corresponding outer loop nest’s index. This implies that the formation of the index of the outer loop on the inner loop’s bound is constrained. For those loop nests in which the upper bound of the inner loop is not a linear function of the index of the outer loop nest, the performance of canonical loop scheduling on load imbalance will degenerate. A simple example where the former constraint is not satisfied is that of a Gaussian type loop nest shown in Fig. 2.23. Let’s think about the loop nest shown in Fig. 2.23 in which a double loop nest with the index of outer loop A, \(0 \leq A < N\) and the index of the inner loop B, \(0 \leq B < f(A)\) where \(f(x), x > 0\) is the Gaussian density function. It is obvious that the execution times of different index of outer loop in a Gaussian type parallel loop are non-linear. Thus, it is apparent that the proposed loop scheduling and partitioning schemes may not be sufficient to deal with Gaussian type loop nests, and a new approach
should be proposed.

(a) Sample code with Gaussian distribution.

(b) Gaussian distribution iteration space.

Figure 2.23: Example of Gaussian loop nest.

To propose a new approach, recall the principles of canonical loop scheduling scheme that the triangle/trapezoid iteration space is divided into $p$ part of equal area. Thus similarly, our problem can be represented as that of finding a method to divide the area of Gaussian loop space into several likelihood trapezoid areas. We claim a ‘likelihood trapezoid’ because the Gaussian density function’s increment is not linear; any three consecutive points cannot be connected by one straight line. If so, we can apply Theorem 3.1 in [Sak97]:

**Theorem 2.4.1.** Consider a canonical triangle/trapezoid loop nest; if the index of the outer loop can be partitioned into $2p$ equal partitions, then the loop nest can be partitioned into $p$ partitions of equal workload, where the $k'$-th partition of the loop nest, $0 \leq k' < p$, consists of the $k$-th and $(2p-k-1)$-th partition along with the index of the outermost loop.

Then, each newly generated trapezoid can be equally distributed on $p$ processors. This draft solution turns out that a possible approach exists to manage the load imbalance problem for distributing Gaussian type loop nests. Assuming $N$ is multiple numbers of $2p$, then we can divide the given iteration space with length $N$ into $N/2p$ fragments of approximate trapezoids with length $2p$. In the ideal case, the hypotenuse is a linear function, saying $f(A) = kA + b$ where $A$ is the index of the outer loop nest. Then we provide the following theorem:

**Theorem 2.4.2.** Dividing the trapezoid of area $\gamma$ by the height line into $2p$ unequal sized small trapezoids, it is possible to form the new generated $2p$ trapezoids into $p$ parts of equal areas $\gamma/p$. 
Figure 2.24: Dividing trapezoid into small trapezoids.

Proof. Consider the trapezoid $A_1 A_{2p+1} B_{2p+1} B_1$ in Fig. 2.24, the side $A_1 A_{2p+1}$ is equally divided into $2p$ equal parts, which satisfies

$$A_1 A_2 = A_2 A_3 = A_3 A_4 = \ldots A_{2p-1} A_{2p} = A_{2p} A_{2p+1} = \frac{A_1 A_{2p+1}}{2p}.$$  

Then it is obvious that the side $B_1 B_{2p+1}$ is divided into $2p$ segments of equal length as well, i.e.,

$$B_1 B_2 = B_2 B_3 = B_3 B_4 = \ldots B_{2p-1} B_{2p} = B_{2p} B_{2p+1} = \frac{B_1 B_{2p+1}}{2p}.$$  

We extend side $A_1 A_{2p+1}$ and side $B_1 B_{2p+1}$, and cross as point $A_0$. To make the proof simple, the trapezoid we chose is special that satisfies $A_0 A_1 = A_1 A_2$. (Due to similar triangular, $\frac{A_0 A_1}{A_0 A_2} = k$ where $k$ might not be equal to $\frac{1}{2}$ in the general case.)

Because the triangles $A_0 A_1 B_1$, $A_0 A_2 B_2$, $A_0 A_3 B_3$ are similar, then if we assume the area of triangle $A_0 A_1 B_1 = \varepsilon$ we get

$$\text{Area}(A_1 A_2 B_2 B_1) = 3\varepsilon, \text{Area}(A_2 A_3 B_3 B_2) = 5\varepsilon, \text{Area}(A_3 A_4 B_4 B_3) = 7\varepsilon, \text{etc.}$$  

Therefore $\text{Area}(A_i A_{i+1} B_{i+1} B_i) = (2i+1)\varepsilon$, where $1 \leq i \leq 2p$, $\text{Area}(A_{2p} A_{2p+1} B_{2p+1} B_{2p}) =$
4p + 1\epsilon. One simple partitioning approach to combine all these small trapezoids into \( p \) equal area is to combine \( k \)th smallest trapezoid with \( k \)th largest. The area of these two trapezoid is \((4p + 4)\epsilon\).

Theorem 2.4.2 can be extended into a more general trapezoid, saying \( A_0A_1 \neq A_1A_2 \), that is to say the area ratio of to triangles, \( A_0A_1B_1 \) and \( A_0A_2B_2 \) is not 1/4 but the square of ratio \( A_0A_1/A_0A_2 \). In this general case, following the proof steps of Theorem 2.4.2, the sum of area of \( k \)th smallest trapezoid and \( k \)th largest trapezoid is the same.

However, the discussion of geometric form of loop nest is not adequate for a real one. Thus the following sections transfer the work into exact loop nest.

2.4.1 Canonical Loop Nests

The loop nest discussed in this section has the general structure as shown in Fig. 2.25, where \( M \) is the coefficient of the max number of iteration. We assume that the loop body does not contain any code whose execution is dependent on the value of the index of its outer loop nest \( A, B \). Then, each loop body’s computing workload remains the same at each iteration no matter the value of the corresponding index. What’s more, in order to make sure that the loop nest is not empty, the loop body should contain at least one executable code.

The definition of canonical loop nest was first introduced in [Sak96], but here, we are discussing a different case, the Gaussian loop nest, which is different to the triangle and trapezoid loop nest, an extended canonical Gaussian loop nest is claimed as follows:
**Definition 2.4.1.** Consider the Gaussian loop nest shown in Fig. 2.25; this Gaussian loop nest is *canonical* if and only if the following constraints are satisfied:

\[
U_1 > L_1, \\
M \neq 0, \\
L_2 \leq M f(A), \forall A \in [L_1, U_1]
\]

Based on the Definition 2.4.1, we can extend the Theorem 2.4.2 from geometry to real Gaussian loop nest.

**Theorem 2.4.3.** Given a canonical loop nest; if the total number of outer loop’s iteration can be partitioned into \(2p\) segments, then the Gaussian loop nest can be partitioned into \(p\) parts of nearly equal workload.

(a) Omit the arc area.  
(b) Likelihood trapezoid loop nest code fragment.

Figure 2.26: Simplify the code structure.

**Proof.** We denote \(N = U_1 - L_1 + 1\) be the total number of outer loop’s iteration. Due to our assumption that \(N\) can be divided into \(N/2p\) segments; it is obvious that all segments are likelihood trapezoids, see in Fig. 2.26, where

\[
U_2 = M f(j2p), \\
q = \frac{M f((j+1)2p-1) - M f(j2p)}{2p}(A - j2p).
\]
If the total iterations in each segment can be combined into \( p \) nearly equal partitions then all \( N/2p \) segments can be combined into \( p \) nearly equal partitions. Assuming that the computing workload corresponding to the loop body is \( \text{Load}_B \), then the \( a \)th segment of the outer loop will execute \( \text{Load}_a \), where

\[
\text{Load}_a = T_a \text{Load}_B,
\]

where \( T_a \) is the number of times the loop body inside inner loop are executed in \( a \)th segment of outer loop. In the \( a \)th segment, we need to find out the number \( a_k \) and \( a_x \) which makes the \( \text{Load}_{a_k} + \text{Load}_{a_x} \) are nearly equal, for all \( a_k \) and \( a_x \) in \( a \)th segment. Moreover, given a loop nest in Fig. 2.26, the \( \text{Load}_{a_k} + \text{Load}_{a_x} \) equals to \( \text{Load}_a/p \) in ideal. Therefore,

\[
\text{Load}_{a_k} + \text{Load}_{a_x} = \frac{1}{p} \sum_{i=0}^{2p-1} \text{Load}_{a_i}, \iff \quad (2.3)
\]

\[
T_{a_k} \text{Load}_B + T_{a_x} \text{Load}_B = \frac{1}{p} \sum_{i=0}^{2p-1} (T_{a_i} \text{Load}_B), \iff \quad (2.4)
\]

\[
T_{a_k} + T_{a_x} = \frac{1}{p} \sum_{i=0}^{2p-1} T_{a_i}. \quad (2.5)
\]

the problem has been transformed into making \( T_{a_k} + T_{a_x} \) constant. The \( a_k \)th partition means the \( k \)th iteration of \( a \)th segment regarding to the outer loop nest, it will execute the loop body in the inner loop nest \( T_{a_k} \) number of times:

\[
T_{a_k} = \sum_{i=L_k}^{q(A-j2p)+U_2} 1
\]

Since we assume that the loop nest is canonical, and it satisfies \( L_2 \leq Mf(A), \forall A \in [L_1,U_1] \). Thus we can express the \( T_{a_k} \):

\[
T_{a_k} = qk - (qj2p - U_2) - L_2 + 1
\]

which has linear equation form \( ax + b \). So we rewrite the \( T_{a_k} \)

\[
T_{a_k} = qk + C,
\]

where \( C = -(qj2p - U_2) - L_2 + 1 \) is one constant. By replacing \( T_{a_k} \) and \( T_{a_x} \) in
CHAPTER 2. THE LOAD BALANCING PROBLEM

Eq. 2.3, we get

\[ q(k + x) - 2C = q(2p - 1) - 2C, \Leftrightarrow \]

\[ x = 2p - 1 - k. \]

Thus we find the partitioning strategy which can make the likelihood trapezoid loop nest nearly evenly partitioned into \( p \) parts, that combine the \( k \)th smallest number of times to execute inner loop body with the \( k \)th biggest number of times to execute inner loop body. Here we can provide a partitioned set as an example,

\[ p(a_0, a_{2p-1}); p(a_1, a_{2p-2}); p(a_2, a_{2p-3}); \ldots; p(a_{p-1}, a_p) \]

where \( p(a_0, a_{2p-1}) \) means partitions 0th index of \( a \)th segment with 2\( p \)th index of \( a \)th segment.

\[ \text{Figure 2.27: The example form of a double Gaussian loop nest.} \]

In order to illustrate the partitioning approach studied in Theorem 2.4.3, consider the Gaussian loop nest shown in Fig. 2.27. As described in Fig. 2.25, this example sets the parameters \( L_1 = 1, U_1 = 15000, L_2 = 0, M = 1000 \); we apply the discrete form of Gaussian density function which is used in \([Bul98]\) as well;

\[ f(A) = \exp\left\{-\left(\frac{A - (M + N\sin(2\pi/\tau)/4)}{N/8}\right)^2\right\}; \quad (2.6) \]

these values satisfy the constrains in Definition 2.4.1. Thus, assuming that the number of processors is 2, \( \text{mod}(15000, 8) = 0 \) the partitioning approach described in Theorem 2.4.3 leads to a perfect load balance and the difference of total number of iterations distributed on two processors is 8, i.e. \( \text{Im} = 8 \); the load imbalance \( \text{Im}_R \) achieved is \( 4.1416 \times 10^{-7} \).
2.5 Concluding Remarks

Some classical static and dynamic loop partitioning and scheduling approaches have been discussed, and the corresponding performance on rectangle/triangle/trapezoid are tested in this section; also in the last subsection, a new extended loop partitioning and scheduling strategy for distributing Gaussian loop nest on parallel processors at compile time was provided. The objective of this new strategy is to minimize the load imbalance overhead when dealing with a Gaussian type loop nest. At the end of the last subsection, an example of distributing the Gaussian loop nest has been presented. The result shows that a new extended approach performs well regarding load imbalance overhead. In order to be clear, we name this new approach as Combined Canonical Loop Scheduling.

Based on this new approach, from the point of view of an algorithm, the first step is to split the outer loop nest into segments with length $2p$ before partitioning the given loop nest; The further actions are based on each segment, since the real segment space is not a standard trapezoid, one of its sides is an arc, so we refer each segment as a likelihood trapezoid and the partitioning approach will partition the likelihood trapezoid space into $p$ equal size parts by following Theorem 2.4.3. In this way, we can achieve $p$ nearly equal size partitions regarding the real segment space.

In the next section, a series of experiments will be set up, and the performance of all discussed schemes on executing program and on reducing load imbalance will be evaluated.
Chapter 3

Experimental Results

3.1 Introduction

To evaluate the performance achieved from loop partitioning and scheduling schemes, we design a series of experiments and simulations in this section. Two aspects of analysis will be followed in this section: the first one main focus on the performance on reducing load imbalance when applying different schemes on various workload; the second aspect of analysis mainly focuses on the performance of executing workload which reveals by the variable, execution time. Our objective has been to evaluate the discussed schemes and show the scalability of the new proposed approach, a combined canonical loop scheduling scheme on various types of parallel workload.

To sum up, the previous section had discussed eight different loop partitioning and scheduling schemes. Here in this section, we choose two static approaches, two dynamic approaches to compare the performance with the Combined Canonical Loop Scheduling approach. The selected approaches are as following (for simplify the description in graphs, we denote each loop partitioning and scheduling scheme with an abbreviation):

- Static block scheduling (SBS), based on composing blocks of consecutive loop iterations, maps the partitioned blocks onto processors, in which processor 0 executes iterations from 1th to $N/p$th, processor 1 executes iteration from $N/p + 1$th to $2N/p$th, processor $m$ will execute from $mN/p + 1$th to $(m+1)N/p$th, where $0 \leq m < p$. 
• Canonical loop scheduling (CLS), based on Theorem 4.1 in [Sak96], partitions $k$th iteration of outer loop nest with $2p - k + 1$th iteration of outer loop nest. Then scheduling $p$ equal partitions on $p$ processors.

• Dynamic chunk scheduling (DCS) first partitions iterations into chunks with equal size $D$, then assigning the first $p$ generated chunks to $p$ processors; similar with pure dynamic scheduling, the $p + 1$th chunk will wait to be distributed until one of processor become idle. The chunk size is one important factor for the performance of dynamic chunk scheduling. Partitioning loops into smaller chunks will reduce the load imbalance overhead and guarantee a good load balance but at the price of increasing resource contention overheads. In contrast, partitioning loops into larger chunks can obviate this problem but can not reduce load imbalance overhead effectively.

• Guided self-scheduling (GSS), based on the trade-off between load imbalance and running overheads, decreases the size of each partitioned chunk during the run-time. In this approach, iterations are dynamically partitioned into chunks during run-time, and chunks are assigned to processors on demand. Each chunk’s size is calculated as $r/p$, where $p$ and $r$ are the number of processors and the number of remained iterations, respectively.

• Combined canonical loop scheduling (CCLS), based on Theorem 2.4.3 in Section 2.4.1. It divides the iteration space of the outer loop nest into segments with length $2p$, then applying canonical loop scheduling on each segment. This scheme will be discussed in Section 3.3.1.

3.2 Simulator

OpenMP is a decades-old application program interface for parallel computing on shared memory architecture. A number of applications of science, engineering, and industry are programmed on the OpenMP platform because of its compatibility and simplicity. Here in all experiments are simulated by OpenMP using C language on IDE Visual Studio 2019 community. The underlying system is 64-bit operating system, x64-based processor implemented with processor: Intel(R) Core(TM) i5-10600KF (6 cores) @ 4.10Ghz with 12MB cache; installed memory (RAM): 16.0 GB. In the experiment phase, we use OpenMP
to create 2, 4, 8, 16 threads. Please note the OpenMP created threads are OS ab-
stractions which is no relation with real threads in hardware. As total 16 abstract
threads exceed the threads implemented on i5-10600KF, the 12 real threads will
work to those abstract threads in cycle.

3.3 Benchmark Programs

The selected loop partitioning and scheduling schemes are tested on a suite of four
benchmark programs that have been chosen to reveal different features. What’s
more, we implement those four different benchmark programs with rectangular
loop nest, triangular loop nest, trapezoid loop nest, and Gaussian loop nest re-
spectively, generating a total 10 interesting benchmark for loops for evaluating
five selected loop partitioning and scheduling schemes.

In essence, these four benchmark programs are as following:

- **Vector addition**: This application is a classical simple program used to test
  the performance on load imbalance, in which each iteration of the outer
  loop nest sums up two vector together; the corresponding rectangular form
  is shown in Fig. 3.1a; the corresponding triangular form is shown in Fig. 3.2a;
  the corresponding trapezoid form is shown in Fig. 3.3a.

- **Upper Triangular Matrix Addition**: This application is a typical example
  of a loop nest that performs the same or different amount of workload at
each iteration along with the outer loop nest depending on the surrounding
loop nest’s form; the corresponding rectangular shape is shown in Fig. 3.1b;
the corresponding triangular form is shown in Fig. 3.2c; the corresponding
trapezoid form is shown in Fig. 3.3c.

- **Adjoint Convolution**: This application is used to evaluate the performance
  of dynamic schemes [Sak96]. Comparing with upper triangle matrix addi-
tion, this application is considerably smaller; the corresponding triangular
form is shown in Fig. 3.2b; the corresponding trapezoid form is shown in
Fig. 3.3b.

- **Upper Triangular Matrix Multiplication**: The multiplication of two upper-
  triangular matrices with size $N \times N$. Compared with the first three appli-
cations, upper triangular matrix multiplication consumes more amount of
workload at each iteration; the corresponding triangular form is shown in Fig. 3.2d.

- **Gaussian Vector Addition**: We reference the Gaussian workload which used as benchmark program in \cite{Bul98}. The corresponding Gaussian form is shown in Fig. 3.4, in which the upper bound of the inner loop nest applies normal distribution. The parameter $\tau$ controls how rapidly the load distribution changes from one execution of the loop to the next, and a large value gives a nearly static load, a small value gives a rapidly varying one. Herein, we fix $\tau = 1000$. Inside the Gaussian loop nest, the ‘vector addition’ operation is executed at each iteration.

All these programs are tested on SBS, CLS, DCS, GSS, and CCLS. We did not implement adjoint convolution into a rectangular loop nest since adjoint convolution is not feasible for rectangular form.

The implementation details of selected benchmark programs deserve further explanations. The rectangular loop nests corresponding to the first two benchmark programs, the vector addition and upper triangular matrix addition, with two depths and the parameters of loop structure satisfy the definition of canonical loop nest, given in Definition 2.4.1; what’s more, the workload of each iteration performs same; thus the combined canonical loop scheduling can achieve a perfect load balance according to Theorem 2.4.3.

The triangular loop nests corresponding to all four benchmark programs, the vector addition, upper triangular matrix addition, adjoint convolution, upper triangular matrix multiplication are canonical loop nest as well; the upper bound of the inner loop is a linear function with respect to the index of outer loop nest; thus the decomposed shapes are trapezoid which can lead to a perfect load balance based on combined canonical loop scheduling.

The trapezoid loop nest corresponding to vector addition, adjoint convolution, and upper triangular matrix addition is a canonical loop nest, the same as the triangular loop nest. The proposed new scheme, combined canonical loop scheduling, can achieve a perfect load balance.

Finally, the Gaussian vector addition benchmark program belongs to the canonical loop nest by definition; but the bound of the inner loop is not a simple function of the corresponding index of the outer loop nest. Regarding this difficulty, combined canonical loop scheduling can achieve a nearly perfect load balance according to Theorem 2.4.3.
CHAPTER 3. EXPERIMENTAL RESULTS

(a) Rectangular vector addition, depth 2.

(b) Rectangular upper triangular matrix addition, depth 2.

Figure 3.1: Rectangular benchmark applications.

(a) Triangular vector addition, depth 2.

(b) Triangular adjoint convolution, depth 2.

(c) Triangular upper triangular matrix addition, depth 2.

(d) Triangular upper triangular matrix multiplication, depth 3.

Figure 3.2: Triangular benchmark applications.
CHAPTER 3. EXPERIMENTAL RESULTS

(a) Trapezoid vector addition, depth 2.

(b) Trapezoid adjoint convolution, depth 2.

(c) Trapezoid upper triangular matrix addition, depth 2.

Figure 3.3: Trapezoid benchmark applications.

Figure 3.4: Gaussian benchmark applications.
CHAPTER 3. EXPERIMENTAL RESULTS

3.3.1 Evaluating the Load Imbalance

Recall the measurement of load imbalance, $Im$ and $Im_R$; each loop partitioning and scheduling scheme will be evaluated by these two measurements:

- Load imbalance ($Im$):

$$ Im = \max_{0 \leq i < p} |Im_i| = \max_{0 \leq i < p} |Load_i - \frac{Load}{p}|. $$  \hfill (3.1)

- Relative load imbalance ($Im_R$):

$$ Im_R = \frac{Im}{Load_{\text{argmax}_{i}(Im_i)}}. $$  \hfill (3.2)

Firstly, we compare the load imbalance, $Im$, and relative load imbalance $Im_R$, resulting from five different schemes; the load imbalance is computed in terms of the number of times executing the loop body.

The rectangular benchmark applications are used to evaluate the performance of combined canonical loop scheduling when dealing with uniform workload along with the index of the outer loop nest. In terms of non-rectangular benchmarks, triangular and trapezoid benchmark applications, the non-uniform workload but applying to a linear function along with the index of outer loop nest is generated. The performance is evaluated to check whether combined canonical loop scheduling inherited all features from canonical loop scheduling. To illustrate the performance of combined canonical loop scheduling on a new loop nest form, the Gaussian type loop nest, we import the vector addition implemented in the Gaussian loop nest that used to act benchmark in $[\text{Bul98}]$.

As a general conclusion, CCLS, SBS, CLS, DCS perform perfect load balance in dealing with all rectangular and triangular benchmark applications; Only CCLS and GSS perform a small value of load imbalance in dealing with Gaussian type benchmark, in which CCLS performs nearly perfect load balance.

In Table 3.1 and Table 3.2 all static loop partitioning and scheduling approaches perform perfect load balance, which confirmed our description about static approaches in Section 2.1; the dynamic approaches cause a load imbalance distribution among processors due to infeasible chunk size generating strategy.

In Table 3.3, Table 3.4, Table 3.5, Table 3.7, Table 3.8 and Table 3.9, CCLS
Table 3.1: Achieved load imbalance and relative load imbalance for vector addition, rectangular loop nest.

<table>
<thead>
<tr>
<th>N</th>
<th>Mapping scheme</th>
<th>Number of processors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Im</td>
<td>Im</td>
</tr>
<tr>
<td>$10^4$</td>
<td>CCLS</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>SBS</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>CLS</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>DCS</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>GSS</td>
<td>0.0</td>
</tr>
</tbody>
</table>

...and CLS perform perfect load balance in $p = 2, 4, 8$; which is analysed in Theorem 2.4.3 and Theorem 4.1 in [Sak96]; the reason why slightly load imbalance caused in $p = 16$ is that $N$ is not a multiple of $2p$. However, compared with SBS, DCS, and GSS, CCLS and CLS achieve better load balance distribution. In Table 3.6, in which a 3 depth loop nest is chosen and the structure of loop nest is more general; but CCLS exhibits a smallest value of load imbalance and relative load imbalance; the reason may be attributed mainly to the scalability of CCLS which is built for loop nest whose bound of inner loop is a non-linear function of index of outer loop nest.

To evaluate the CCLS’s performance on processing Gaussian workload, we collect the load imbalance and relative load imbalance of a different number of processors. Also, to illustrate the performance of CCLS at various iterations $N$, we test the CCLS and other 4 schemes on Gaussian vector addition with $N = 128; N = 1024; N = 10000$. Given a small number of iterations $N$, the curve of generated Gaussian distribution by Eq. 2.6 is not as smooth as the curve generated for a significant $N$ value. In which case, the performance would degenerate because we approximate the fragment with a trapezoid in Fig 2.26a. However, the new proposed scheme, CCLS, achieves a slight relative load imbalance in all cases, $N = 128, N = 1024, and N = 10000$; a nearly load balance distribution is achieved by CCLS in Table 3.10, which confirmed the analysis in Theorem 2.4.3; the rest of the schemes, SBS, CLS, DCS, GSS failed to balance the Gaussian workload among processors. What’s more, the relative load imbalance is decreasing while the number of iterations $N$ increases.

Apart from evaluating the load imbalance of these five schemes, the benchmark application’s execution time’s performance is critical for the newly proposed method. To illustrate the performance of CCLS on execution time, we compare different benchmark applications’ execution time when applying various schemes.
Table 3.2: Achieved load imbalance and relative load imbalance for upper triangular matrix addition, rectangular loop nest.

<table>
<thead>
<tr>
<th>N</th>
<th>Scheme</th>
<th>Number of processors</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Im</td>
<td>Im&lt;sub&gt;g&lt;/sub&gt;</td>
<td>Im</td>
<td>Im&lt;sub&gt;g&lt;/sub&gt;</td>
<td>Im</td>
</tr>
<tr>
<td>10&lt;sup&gt;4&lt;/sup&gt;</td>
<td>CCLS</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>SBS</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>CLS</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>DCS</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>1.5 x 10&lt;sup&gt;6&lt;/sup&gt;</td>
</tr>
<tr>
<td></td>
<td>GSS</td>
<td>0.0</td>
<td>0.0</td>
<td>4.88 x 10&lt;sup&gt;6&lt;/sup&gt;</td>
<td>0.1633</td>
<td>5.08 x 10&lt;sup&gt;6&lt;/sup&gt;</td>
</tr>
</tbody>
</table>

Table 3.3: Achieved load imbalance and relative load imbalance for vector addition, triangular loop nest.

<table>
<thead>
<tr>
<th>N</th>
<th>Scheme</th>
<th>Number of processors</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Im</td>
<td>Im&lt;sub&gt;g&lt;/sub&gt;</td>
<td>Im</td>
<td>Im&lt;sub&gt;g&lt;/sub&gt;</td>
<td>Im</td>
</tr>
<tr>
<td>10&lt;sup&gt;4&lt;/sup&gt;</td>
<td>CCLS</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>SBS</td>
<td>1.25 x 10&lt;sup&gt;7&lt;/sup&gt;</td>
<td>1.0002</td>
<td>9.375 x 10&lt;sup&gt;6&lt;/sup&gt;</td>
<td>3.0012</td>
<td>5.468750.0</td>
</tr>
<tr>
<td></td>
<td>CLS</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>DCS</td>
<td>6.25 x 10&lt;sup&gt;6&lt;/sup&gt;</td>
<td>0.3334</td>
<td>4687500.0</td>
<td>0.6001</td>
<td>3.32 x 10&lt;sup&gt;6&lt;/sup&gt;</td>
</tr>
<tr>
<td></td>
<td>GSS</td>
<td>94003.0</td>
<td>0.0038</td>
<td>743914.0</td>
<td>0.0562</td>
<td>810181.0</td>
</tr>
</tbody>
</table>

Table 3.4: Achieved load imbalance and relative load imbalance for adjoint convolution, triangular loop nest.

<table>
<thead>
<tr>
<th>N</th>
<th>Scheme</th>
<th>Number of processors</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Im</td>
<td>Im&lt;sub&gt;g&lt;/sub&gt;</td>
<td>Im</td>
<td>Im&lt;sub&gt;g&lt;/sub&gt;</td>
<td>Im</td>
</tr>
<tr>
<td>10&lt;sup&gt;4&lt;/sup&gt;</td>
<td>CCLS</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>SBS</td>
<td>1.25 x 10&lt;sup&gt;7&lt;/sup&gt;</td>
<td>0.3333</td>
<td>9.375 x 10&lt;sup&gt;6&lt;/sup&gt;</td>
<td>0.4286</td>
<td>5.468750.0</td>
</tr>
<tr>
<td></td>
<td>CLS</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>DCS</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>781250.0</td>
</tr>
<tr>
<td></td>
<td>GSS</td>
<td>1.25 x 10&lt;sup&gt;7&lt;/sup&gt;</td>
<td>0.3333</td>
<td>9375000.0</td>
<td>0.4285</td>
<td>5.468750.0</td>
</tr>
</tbody>
</table>

Table 3.5: Achieved load imbalance and relative load imbalance for upper triangular matrix addition, triangular loop nest.

<table>
<thead>
<tr>
<th>N</th>
<th>Scheme</th>
<th>Number of processors</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Im</td>
<td>Im&lt;sub&gt;g&lt;/sub&gt;</td>
<td>Im</td>
<td>Im&lt;sub&gt;g&lt;/sub&gt;</td>
<td>Im</td>
</tr>
<tr>
<td>10&lt;sup&gt;4&lt;/sup&gt;</td>
<td>CCLS</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>SBS</td>
<td>1.25 x 10&lt;sup&gt;7&lt;/sup&gt;</td>
<td>1.0002</td>
<td>9.375 x 10&lt;sup&gt;6&lt;/sup&gt;</td>
<td>3.0012</td>
<td>5.468750.0</td>
</tr>
<tr>
<td></td>
<td>CLS</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>DCS</td>
<td>6.25 x 10&lt;sup&gt;6&lt;/sup&gt;</td>
<td>0.3334</td>
<td>4687500.0</td>
<td>0.6001</td>
<td>2734375.0</td>
</tr>
<tr>
<td></td>
<td>GSS</td>
<td>437432.0</td>
<td>0.0172</td>
<td>144103.0</td>
<td>0.0114</td>
<td>224507.0</td>
</tr>
</tbody>
</table>

Table 3.6: Achieved load imbalance and relative load imbalance for upper triangular matrix multiplication, triangular loop nest.

<table>
<thead>
<tr>
<th>N</th>
<th>Scheme</th>
<th>Number of processors</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Im</td>
<td>Im&lt;sub&gt;g&lt;/sub&gt;</td>
<td>Im</td>
<td>Im&lt;sub&gt;g&lt;/sub&gt;</td>
<td>Im</td>
</tr>
<tr>
<td>10&lt;sup&gt;3&lt;/sup&gt;</td>
<td>CCLS</td>
<td>250.0</td>
<td>3 x 10&lt;sup&gt;-8&lt;/sup&gt;</td>
<td>875.0</td>
<td>2.1 x 10&lt;sup&gt;-5&lt;/sup&gt;</td>
<td>2187.5</td>
</tr>
<tr>
<td></td>
<td>SBS</td>
<td>6.25 x 10&lt;sup&gt;7&lt;/sup&gt;</td>
<td>0.4286</td>
<td>5.468757 x 10&lt;sup&gt;7&lt;/sup&gt;</td>
<td>0.5676</td>
<td>3.418 x 10&lt;sup&gt;7&lt;/sup&gt;</td>
</tr>
<tr>
<td></td>
<td>CLS</td>
<td>62500.0</td>
<td>0.7494 x 10&lt;sup&gt;-3&lt;/sup&gt;</td>
<td>93875.0</td>
<td>0.0022</td>
<td>234810.0</td>
</tr>
<tr>
<td></td>
<td>DCS</td>
<td>13020875.0</td>
<td>0.1351</td>
<td>13346375.0</td>
<td>0.2426</td>
<td>8284000.0</td>
</tr>
<tr>
<td></td>
<td>GSS</td>
<td>6.25 x 10&lt;sup&gt;7&lt;/sup&gt;</td>
<td>0.4286</td>
<td>54687500.0</td>
<td>0.5676</td>
<td>3.418 x 10&lt;sup&gt;7&lt;/sup&gt;</td>
</tr>
</tbody>
</table>
Table 3.7: Achieved load imbalance and relative load imbalance for upper triangular matrix addition, trapezoid loop nest.

<table>
<thead>
<tr>
<th>N</th>
<th>Scheme</th>
<th>Number of processors</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Im</td>
<td>Img</td>
<td>Im</td>
<td>Img</td>
</tr>
<tr>
<td>10⁴</td>
<td>CCLS</td>
<td>1.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>SBS</td>
<td>1.25 × 10⁷</td>
<td>0.2632</td>
<td>93750000.0</td>
<td>0.3489</td>
<td>5468750.0</td>
</tr>
<tr>
<td></td>
<td>CLS</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>DCS</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>GSS</td>
<td>1.25 × 10⁷</td>
<td>0.2632</td>
<td>93750000.0</td>
<td>0.3489</td>
<td>5468750.0</td>
</tr>
</tbody>
</table>

Table 3.8: Achieved load imbalance and relative load imbalance for adjoint convolution, trapezoid loop nest.

<table>
<thead>
<tr>
<th>N</th>
<th>Scheme</th>
<th>Number of processors</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Im</td>
<td>Img</td>
<td>Im</td>
<td>Img</td>
</tr>
<tr>
<td>10⁴</td>
<td>CCLS</td>
<td>1.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>SBS</td>
<td>1.25 × 10⁷</td>
<td>0.2632</td>
<td>93750000.0</td>
<td>0.3489</td>
<td>5468750.0</td>
</tr>
<tr>
<td></td>
<td>CLS</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>DCS</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>GSS</td>
<td>1.25 × 10⁷</td>
<td>0.2632</td>
<td>93750000.0</td>
<td>0.3489</td>
<td>5468750.0</td>
</tr>
</tbody>
</table>

Table 3.9: Achieved load imbalance and relative load imbalance for vector addition, trapezoid loop nest.

<table>
<thead>
<tr>
<th>N</th>
<th>Scheme</th>
<th>Number of processors</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Im</td>
<td>Img</td>
<td>Im</td>
<td>Img</td>
</tr>
<tr>
<td>10⁴</td>
<td>CCLS</td>
<td>1.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>SBS</td>
<td>1.25 × 10⁷</td>
<td>0.2632</td>
<td>93750000.0</td>
<td>0.3489</td>
<td>5468750.0</td>
</tr>
<tr>
<td></td>
<td>CLS</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>DCS</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>GSS</td>
<td>1.25 × 10⁷</td>
<td>0.2632</td>
<td>93750000.0</td>
<td>0.3489</td>
<td>5468750.0</td>
</tr>
</tbody>
</table>

Table 3.10: Achieved load imbalance and relative load imbalance for vector addition, Gaussian loop nest.

<table>
<thead>
<tr>
<th>N</th>
<th>Scheme</th>
<th>Number of processors</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>10²</td>
<td>CCLS</td>
<td>1.0</td>
<td>5.2612 × 10⁻¹⁶</td>
<td>0.5</td>
<td>0.0626</td>
<td>6.25</td>
</tr>
<tr>
<td></td>
<td>SBS</td>
<td>0.0021</td>
<td>47.75</td>
<td>0.4992</td>
<td>55.4375</td>
<td>0.75</td>
</tr>
<tr>
<td></td>
<td>CLS</td>
<td>0.0343</td>
<td>482.5</td>
<td>0.1130</td>
<td>556.25</td>
<td>0.3056</td>
</tr>
<tr>
<td></td>
<td>DCS</td>
<td>0.0321</td>
<td>567.5</td>
<td>0.5443</td>
<td>356.75</td>
<td>0.6</td>
</tr>
<tr>
<td></td>
<td>GSS</td>
<td>35.50</td>
<td>0.0211</td>
<td>567.5</td>
<td>0.5443</td>
<td>356.75</td>
</tr>
<tr>
<td>10⁴</td>
<td>CCLS</td>
<td>4.0</td>
<td>6.8840 × 10⁻¹⁵</td>
<td>145.0</td>
<td>1.5407 × 10⁻¹⁵</td>
<td>22.5</td>
</tr>
<tr>
<td></td>
<td>SBS</td>
<td>248.00</td>
<td>0.0042</td>
<td>2897250.0</td>
<td>0.4980</td>
<td>345592.5</td>
</tr>
<tr>
<td></td>
<td>CLS</td>
<td>256.00</td>
<td>0.0044</td>
<td>3855.0</td>
<td>0.0130</td>
<td>4467.5</td>
</tr>
<tr>
<td></td>
<td>DCS</td>
<td>23390.00</td>
<td>0.0041</td>
<td>2018850.0</td>
<td>2.2949</td>
<td>157112.5</td>
</tr>
<tr>
<td></td>
<td>GSS</td>
<td>2480.00</td>
<td>0.0042</td>
<td>372655.0</td>
<td>0.5406</td>
<td>177751.25</td>
</tr>
<tr>
<td>10¹⁴</td>
<td>CCLS</td>
<td>35.0</td>
<td>2.0975 × 10⁻¹⁷</td>
<td>137.5</td>
<td>1.8833 × 10⁻¹⁶</td>
<td>231.75</td>
</tr>
<tr>
<td></td>
<td>SBS</td>
<td>452950.00</td>
<td>3101.8 × 10⁻⁷</td>
<td>683993000.0</td>
<td>0.4837</td>
<td>681660000.0</td>
</tr>
<tr>
<td></td>
<td>CLS</td>
<td>499050.00</td>
<td>3419.7 × 10⁻⁷</td>
<td>74873.0</td>
<td>0.0010</td>
<td>8774.0</td>
</tr>
<tr>
<td></td>
<td>DCS</td>
<td>361650.00</td>
<td>2478 × 10⁻⁷</td>
<td>318460000.0</td>
<td>0.7740</td>
<td>2314200.0</td>
</tr>
<tr>
<td></td>
<td>GSS</td>
<td>452950.00</td>
<td>3101.8 × 10⁻⁷</td>
<td>593296000.0</td>
<td>0.4484</td>
<td>296010000.0</td>
</tr>
</tbody>
</table>
3.4 Experimental Results

Experiments consist of nine fragments, each fragment running on a given problem size \(N\); one typical benchmark application; a number of processors. As far as possible, all experiments are run when no other jobs are processing at the computer; whenever the execution time of each program exists significant variation we repeat the completed simulations.

The result shows that all schemes perform nearly the same when dealing with rectangular loop nest, see Fig. 3.5, Fig. 3.6. Regarding the rectangular loop nest, each index of the outer loop nest carries the same workload; the load distribution would be perfect balancing if the outer loop can be equally divided into \(p\) segments by schemes.

![Figure 3.5: Execution time for \(N = 10000\) rectangular vector addition, rectangular loop nest.](image)

The execution time of the triangular benchmark applications for vector addition, upper triangular matrix addition, adjoint convolution, and upper triangular matrix multiplication are shown in Fig. 3.7 \((N = 10000)\), Fig. 3.8 \((N = 10000)\), Fig. 3.9 \((N = 10000)\), and Fig. 3.10 \((N = 1000)\) respectively. The performance on execution time have been revealed in Table 3.3, Table 3.4, Table 3.5, and Table 3.6 that CCLS and CLS achieved perfect load balance regarding \(p = 2, 4, 8\), a small load imbalance caused in \(p = 16\) due to \(\text{mod}(N, 2p) \neq 0\). Thus CCLS and CLS perform best.

In the experiments of trapezoid loop nest, the vector addition, upper triangular matrix, and vector adjoint convolution, the execution time results are shown in
CHAPTER 3. EXPERIMENTAL RESULTS

Figure 3.6: Execution time for $N = 10000$ upper triangular matrix addition, rectangular loop nest.

Figure 3.7: Execution time for $N = 10000$ vector addition, triangular loop nest.
CHAPTER 3. EXPERIMENTAL RESULTS

Figure 3.8: Execution time for $N = 10000$ upper triangular matrix addition, triangular loop nest.

Figure 3.9: Execution time for $N = 10000$ vector adjoint convolution, triangular loop nest.
CHAPTER 3. EXPERIMENTAL RESULTS

Figure 3.10: Execution time for $N = 1000$ upper triangular matrix multiplication, triangular loop nest.

Fig. 3.11 ($N = 10000$), Fig. 3.12 ($N = 10000$), and Fig. 3.13 ($N = 10000$) respectively. Similar with triangular loop nest, CCLS and CLS perform the best performance due to a evenly workload distribution achieved among processors, thus CCLS and CLS run significantly faster than SBS, DCS and GSS.

Figure 3.11: Execution time for $N = 10000$ vector addition, trapezoid loop nest.

The execution time of the Gaussian loop nest for vector addition is shown in Fig. 3.14 ($N = 10000$). In the graph, CCLS performs the best of all, while SBS, CLS, DCS, and GSS show poor performance. The result is anticipated from the load imbalance value shown in Table 3.10; an evenly distributed workload reduces the execution time among processors.
Figure 3.12: Execution time for $N = 10000$ upper triangular matrix addition, trapezoid loop nest.

Figure 3.13: Execution time for $N = 10000$ vector adjoint convolution, trapezoid loop nest.
3.5 Concluding Remarks

The experimental results illustrate that selecting an appropriate loop partitioning and scheduling scheme based on the loop nest structure can dramatically optimize running parallel programs. It is worth noting that a dynamic approach will consume computing sources on accessing the centre queue. Static approaches can balance the trade-off between executing performance and load imbalance if one appropriate partitioning and scheduling strategy is selected, such as combined canonical loop scheduling.

The performances of different schemes on execution time correspond to the result of load imbalance. We notice that in a small size problem, when we are decreasing the number $N$, the destructive effects from other running overheads can diminish the positive impact caused by a balanced load distribution.
Chapter 4

Conclusion

4.1 Summary of Results

This thesis is motivated by the load balancing problem of distributing loops nest on parallel processors. This problem is a well-studied topic, and many schemes have been proposed to deal with it. However, as discussed in Section 1.4.4, the proposed schemes are subject to how well they manage the trade-off between load imbalance and performance gain. Regarding this, we evaluate static and dynamic approaches by dealing with the same loop nest sample code. Dynamic loop scheduling approaches focus more on distributing an evenly working time for each processor at run-time; however, achieving this objective has to consume additional overheads, such as resource contention overhead. The static loop schedule approach may not distribute work among processors evenly; in other words, each processor’s running time is unequal; but static approaches always bring less resource contention overhead, and it will achieve better performance if taken into account architecture information and program information.

Following this result, we proposed another new static loop partitioning and scheduling scheme, the combined canonical loop scheduling (CCLS), based on canonical loop scheduling. In Theorem 2.4.3, we proved that CCLS could achieve a nearly perfect load balance in dealing with Gaussian loop nest and achieve perfect load balance in coping with loop nest which bound of inner loop satisfies linear function with respect to the index of its outer loop nest, under pre-defined constraints.

In Chapter 3, we evaluate the new loop scheduling approach with four other classical loop scheduling schemes. The results of load imbalance and execution
time show that combined canonical loop scheduling perfectly inherits advantages of canonical loop scheduling and performs better performance in a more general case, the Gaussian loop nest.

4.2 Further Work

The research described in this thesis leaves two paths for further investigation. Firstly, using probability theory to pre-estimate the load imbalance before applying combined canonical loop scheduling; secondly, extend the combined canonical loop scheduling into a more general case.

What’s more because the overhead caused by resource contention will badly reduce the impact of balanced load on performance, so the data parallelizing is another critical factor to be discussed together with parallel loop nest.

Finally, this thesis is motivated by a practical problem. It is impossible to build a model that can cover all features in the real world. However, the proposed model should reflect the actual case with details as much as possible. Thus new challenges may arise when considering a broad spectrum of practical problems.
Bibliography


[CKPK90] George Cybenko, Lyle Kipp, Lynn Pointer, and David Kuck. Supercomputer performance evaluation and the perfect benchmarks. In


[Kle18] Michael Klemm. Twenty years of the openmp api: Michael klemm, cco, and matthijs van waveren, marketing coordinator for the openmp arb, along with jim crowie, principal engineer, intel corporation, consider the development of openmp over the last 20 years.(high performance computing). Scientific computing world, (159):16–, 2018.


