การลดการพึ่งพิงกันของข้อมูลในเมทริกซ์กำหนดการพลวัต

นายกิลเลอร์มอร์ เดลกาโด

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิทยาศาสตรมหาบัณฑิต
สาขาวิชาวิทยาการคอมพิวเตอร์และเทคโนโลยีสารสนเทศ
ภาควิชาคณิตศาสตร์และวิทยาการคอมพิวเตอร์
คณะวิทยาศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย
ปีการศึกษา 2554
ลิขสิทธิ์ของจุฬาลงกรณ์มหาวิทยาลัย

บทคัดย่อและแฟ้มข้อมูลฉบับเต็มของวิทยานิพนธ์ตั้งแต่ปีการศึกษา 2554 ที่ให้บริการในคลังปัญญาจุฬาฯ (CUIR)
เป็นแฟ้มข้อมูลของนิสิตเจ้าของวิทยานิพนธ์ที่ส่งผ่านทางบัณฑิตวิทยาลัย
The abstract and full text of theses from the academic year 2011 in Chulalongkorn University Intellectual Repository(CUIR)
are the thesis authors' files submitted through the Graduate School.

DATA DEPENDENCY REDUCTION IN DYNAMIC PROGRAMMING MATRIX

Mr. Guillermo Delgado

A Thesis Submitted in Partial Fulfillment of the Requirements

for the Degree of Master of Science Program in

Computer Science and Information Technology

Department of Mathematics and Computer Science

Faculty of Science

Chulalongkorn University

Academic Year 2011

Copyright of Chulalongkorn University

Thesis Title	Data Dependency Reduction in Dynamic Programming Matrix
By	Mr. Guillermo Delgado
Field of Study	Computer Science and Information Technology
Thesis Advisor	Assistant Professor Chatchawit Aporntewan, Ph.D.

Accepted by the Faculty of Science, Chulalongkorn University in Partial Fulfillment of the Requirements for the Master's Degree

……………………………………………….. Dean of the Faculty of Science

(Professor Supot Hannongbua, Ph.D.)

THESIS COMMITTEE

……………………………………………….. Chairman

(Assistant Professor Saranya Maneeroj, Ph.D.)

……………………………………………….. Thesis Advisor

(Assistant Professor Chatchawit Aporntewan, Ph.D.)

……………………………………………….. Examiner

(Assistant Professor Jaruloj Chongstitvatana, Ph.D.)

……………………………………………….. External Examiner

(Associate Professor Nachol Chaiyaratana, Ph.D.)

กิลเลอร์มอร์ เดลกาโด : การลดการพึ่งพิงกันของข้อมูลในเมทริกซ์กำหนดการพลวัต. (DATA DEPENDENCY REDUCTION IN DYNAMIC PROGRAMMING MATRIX) อ. ที่ปรึกษาวิทยานิพนธ์หลัก : ผศ.ดร.ชัชวิทย์ อาภรณ์เทวัญ, 50 หน้า.

กำหนดการพลวัตมีบทบาทสำคัญในการแก้ปัญหาเชิงคำนวณจำนวนมาก ในขณะที่จำนวนคอร์ต่อโปรเซสเซอร์กำลังเพิ่มขึ้นอย่างรวดเร็ว ซอฟต์แวร์ใหม่ๆ ต้องสามารถใช้ประโยชน์จากข้อดีของสถาปัตยกรรมแบบมัลติคอร์ได้ โดยปกติกำหนดการพลวัตเริ่มจากการสร้างเมทริกซ์และคำนวณค่าในเมทริกซ์ไปทีละค่า การทำงานแบบขนานจะสร้างเธรดหนึ่งเธรดต่อหนึ่งแถว และรักษาการพึ่งพิงกันของข้อมูลด้วยการประสานเวลาของเธรด อย่างไรก็ตามเมื่อจำนวนเธรดเพิ่มขึ้นสมรรถนะจะลดลงเนื่องจากการพึ่งพิงกันของข้อมูล ในที่นี้เราเสนอวิธีใหม่สำหรับการคำนวณเมทริกซ์แบบขนาน ซึ่งตรงข้ามกับวิธีมาตรฐานที่คำนวณจากบนลงล่างและจากซ้ายไปขวาเท่านั้น วิธีที่เราเสนอนั้นทำในหลายทิศทาง ดังนั้นจะลดเวลาที่ใช้รอการพึ่งพิงกันของข้อมูลได้มาก เพื่อสาธิตการทำงานของวิธีที่เรานำเสนอ เราเลือกขั้นตอนวิธีการปรับแนวลำดับเฉพาะที่ แบบที่เรียกว่า สมิธ-วอเตอร์แมน ซึ่งเป็นปัญหากำหนดการพลวัตแบบหนึ่ง อย่างไรก็ตามวิธีของเราไม่ได้จำกัดแค่ขั้นตอนวิธีสมิธวอเตอร์แมน แต่ใช้กับปัญหากำหนดการพลวัตอื่นๆ ที่มีรูปแบบคล้ายกันได้ด้วย การเปรียบเทียบกับวิธีมาตรฐานบนสถานีงาน HP Z800 ที่มี 8 คอร์แสดงให้เห็นว่าวิธีที่เรานำเสนอทำงานได้เร็วกว่าอย่างมีนัยยะสำคัญ

ภาควิชา ..คณิตศาสตร์และวิทยาการคอมพิวเตอร์..	ลายมือชื่อนิสิต ....
สาขาวิชา ..วิทยาการคอมพิวเตอร์..	ลายมือชื่อ อ.ที่ปรึกษาวิทยานิพนธ์
..และเทคโนโลยีสารสนเทศ..	หลัก………………………………….....
ปีการศึกษา ..2554..	

5273619123 : MAJOR COMPUTER SCIENCE AND INFORMATION TECHNOLOGY
KEYWORDS : DYNAMIC PROGRAMMING / DATA DEPENDENCY

GUILLERMO DELGAGO : DATA DEPENDENCY REDUCTION IN DYNAMIC PROGRAMMING MATRIX. ADVISOR : ASST. PROF. CHATCHAWIT APORNTEWAN, Ph.D., 50 pp.

Dynamic Programming (DP) plays an important role in solving a large number of computational problems. As the number of cores per processor is increasing rapidly, new software must be capable of exploiting the advantages of multi-core architectures. A typical DP begins with constructing a matrix, and then calculating each element one by one. The standard parallelization spawns multiple threads, one for each row, while maintains the data dependency via thread synchronization. However, as the number of threads increase, the performance degrades due to data dependency. Herein, we proposed a novel method for calculating a DP matrix in parallel. In contrast to the standard method that always calculates from up to down and left to right, our method performs the calculation in multiple directions. Therefore, the wait time for data dependency is remarkably reduced. To demonstrate our method, a local sequence alignment algorithm called Smith-Waterman (SW) was chosen as an instance of DP. However, our method is not only limited to SW algorithm, but it is applicable to other DP problems that have similar patterns of data dependency. A comparison with the standard method was conducted on a HP Z800 workstation with a total of eight cores. The results show that our method performs significantly faster.

Department : ..Mathematics and Computer Science..
Field of Study: ..Computer Science and..
..Information Technology..
Academic Year : ..2011..

Student's Signature……………………
Advisor's Signature ……………………

Acknowledgement

I would like to acknowledge my advisor, Professor Chatchatwit Aporntewan, at the department of Mathematics in Chulalongkorn University, for all his patience, help and support. Without his ideas and hard work this thesis would have never been possible. He has always been there whenever I needed him for the thesis or any other aspect of my life. Today, I am happy to consider him a friend. I would also like to thank my family, friends and specially my girlfriend; who encouraged me to get involved in this rewarding project.

May I dedicate this work to all the people mentioned above. Without them, this work would never be done. I have encountered many problems while working in this thesis, but they were so little compared to all the support given by these people.

Contents

	Page
Abstract (Thai)........	iv
Abstract (English)........	v
Acknowledgement........	vi
Contents........	vii
List of Tables........	ix
List of Figures........	x
Chapter	
I Introduction........	1
1.1 Objectives........	1
1.2 Scope of the Work........	2
1.3 Problem Formulation........	2
1.4 Expected Outcomes........	3
II Theoretical Background........	4
2.1 Dynamic Programming........	4
2.2 Smith-Waterman Algorithm........	7
2.3 Multi-core Architecture........	7
2.4 Parallelization........	8
2.5 Data Dependency in Parallel Computing........	9
2.6 Non-Uniform Memory Access........	10
III Data Structure and Algorithmic Design........	12
3.1 Uni-Directional Filling (UDF)	12
3.2 Bi-Directional Filling (BDF)	13
3.3 A Theoretical Comparison between UDF and BDF........	13
3.4 Data Structures........	14
3.5 Code Explanation........	16
3.5.1 UDF........	16

Chapter Page

3.5.2 BDF........ 20

3.6 NUMA Optimization........ 26

IV Experimental Results and Discussion........ 27

V Conclusion........ 33

References........ 34

Appendices........ 35

Appendix A Publications........ 36

Appendix B UDF source code........ 40

Appendix C BDF source code........ 44

Biography........ 50

List of Tables

Table		Page
1	A table used by DP to calculate F(6)………………………………………..	5
2	A comparison of access time between local and remote memory….......	32

List of Figures

Figure		Page
1	An execution tree generated to obtain F(6)………………………………..	4
2	Data dependency of sequence alignment..	6
3	Single-core and multi-core architectures..	7
4	An NUMA architecture with two nodes...	11
5	A graphical representation of UDF…………..	12
6	A graphical representation of BDF...	13
7	Critical points in UDF and BDF..	14
8	A graphical representation of data structures for BDF…….....................	15
9	A comparison of delays generated by UDF and BDF............................	28
10	A comparison of elapsed times between UDF and BDF.........................	29
11	BDF's speedup compared with UDF……………………........................	30
12	Core utilization of UDF and BDF..	31
13	A comparison of elapsed times between the best and the worst cases..	32

CHAPTER I

Introduction

Dynamic Programming (DP) has been used to solve a wide range of computational problems [1]. DP is based on problem decomposition, recursively breaking down a problem into smaller sub-problems. The identical sub-problems are computed only once, hence avoiding redundant computation. We focus on the bottom-up approach where the solutions are iteratively generated from small to large sub-problems. In this process, a table or an array called "DP matrix" is needed [2]. A sequential algorithm fills the matrix element by element. In contrast to parallel computing, where multiple matrix elements are calculated simultaneously while maintaining the data dependencies, e.g. the calculation of an element depends on the others.

The patterns of data dependency are varied among DP problems. However, most of them including sequence alignment can be formulated as filling a two-dimensional table, and they share the similar pattern where the matrix element at row i and column j, denoted by $m[i,j]$, depends on $m[i,j-1]$, $m[i-1,j-1]$, and $m[i-1,j]$. In other words, $m[i,j]$ cannot be calculated if the other three elements have not been computed yet. The classical parallel algorithm partitions the data in rows and fills multiple rows simultaneously [3]. To maintain the data dependency, a row cannot precede the row above. As a result, the degree of parallelism is bounded by the data dependency.

In this thesis, we propose a novel data partitioning that reduces the data dependency and elevates the degree of parallelism. Instead of filling the matrix in only row-wise direction, we fill the matrix in both row-wise and column-wise directions, and name it "Bi-Directional Filling (BDF)" as oppose to the classical uni-Directional Filling (UDF).

1.1 Objectives

- To design an algorithm that reduces data dependencies in parallel dynamic programming.

- To create a program of the proposed algorithm, in order to execute and measure performance in real hardware.
- To compare the performance of both the classical and the proposed algorithms.

1.2 Scope of the Work

- We aim to parallelization only on multi-core architectures.
- Our programming is based on Microsoft .NET and C#.

1.3 Problem Formulation

Basically the goal of this project is creating a parallel algorithm to fill a matrix *m*, where the calculation of a matrix element depends on other elements that have been previously calculated. To be more specific, the element *m[i,j]* needs to wait the calculations of *m[i,j-1]*, *m[i-1,j-1]*, and *m[i-1,j]* where *m[i,j]* denotes the matrix element at row *i* and column *j*. The classical parallelization approach assigns a thread to each row, and synchronizes threads to maintain data dependency. However, as the number of cores increases, the overall performance drops dramatically. The major bottleneck is due to the data dependency we have mentioned. As a result, we need to invent a new algorithm that will reduce the dependencies, and will make the performance scalable with the number of cores.

This work aims to address the following issues.

- Theoretically, how to reduce the data dependencies in bottom-up dynamic programming algorithms such as the local sequence alignment [4].
- How to go from a theoretical model to a real software implementation
- Performance gain due to data dependency reduction.

1.4 Expected Outcomes

- A novel method for parallel construction of DP matrix and the reduction in data dependencies.
- An implementation of parallel SW algorithm with our method and the faster execution time compared to the classical method.
- A demonstration of real-world applications, for instance, local sequence alignment.

CHAPTER II

Theoretical Background

This chapter is dedicated to explain some fundamentals in computer science and current technologies that are important to understand the rest of thesis.

2.1 Dynamic Programming

Dynamic Programming (DP) is a widely used problem solving method characterized by problem decomposition [1]. The basic idea is to break down the initial problem into smaller sub-problems in order to reach a final solution. When compared with other methodologies, Dynamic Programming's advantage is that each sub-problem will be executed only once, avoiding redundant computation.

A classic example where DP provides great benefits is the Fibonacci Problem. This problems receives an integer n as the input, and calculates the Fibonacci value F(n) as the output.

Figure 1 shows the execution tree produced by the classic recursive approach to obtain F(6). Several sub-problems are executed more than once. For instance F(2) is executed up to 5 times for a relatively small problem size. That is a waste of computational power and time.

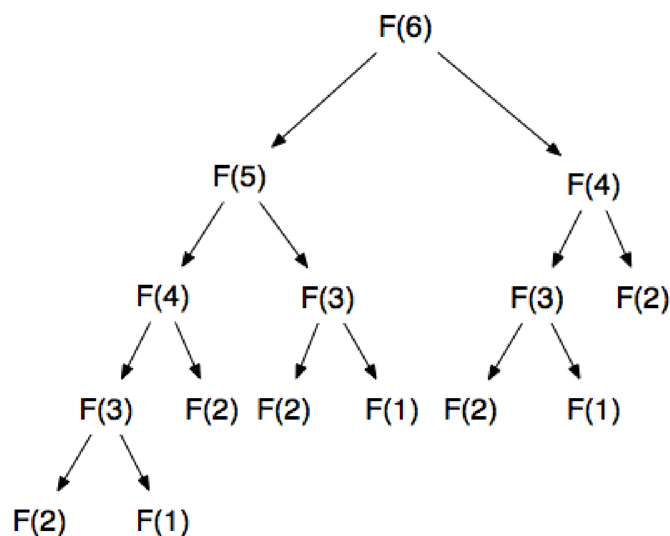

Figure 1: An execution tree generated to obtain F(6).

Dynamic Programming solves the redundant computation by creating a table where the values of the sub-problems will be stored. Then, when the solution to a previously solved sub-problem is needed, it can be taken directly from the table instead of calculating it again.

In the previous example we know that, in order to calculate F(6), we are going to need F(5), F(4),F(3),F(2) and F(1). DP takes an advantage of this characteristic and starts the execution from the basic case, making it bigger until reaching the final solution. As a basic case, we know that both F(1) and F(2) have the value 1. Table 1 shows the table used by the DP algorithm to store all the previous values until reaching the final solution. To calculate the value of a sub-problem, it is only necessary to add the two previous values in the table.

n	F(n)
1	1
2	1
3	2
4	3
5	5
6	8

Table 1: A table used by DP to calculate F(6).

There are two different approaches to solve a DP algorithm. Top-down approach and Bottom-up approach.

a) Top-down approach: top-down DP follows the classic recursive execution. The difference is that when a sub-problem is solved, the solution will be stored in memory. That way, when the same solution

is needed again, the sub-problem does not need to be calculated again. This approach goes form bigger to smaller sub-problems to find a solution.

b) Bottom-up approach: bottom-up DP goes one step further. It modifies the execution so that it will start by calculating first the smallest sub-problem (basic case). Then, the execution will go from smaller to bigger sub-problems until reaching the final solution. Both the Fibonacci example and the case studied in this thesis follow the bottom-up approach.

A classic example of the bottom-up approach uses a 2D table or a dynamic programming matrix to store the solutions. Filling the matrix starts from small to large sub-problems. A sequential algorithm fills the matrix element by element, but a parallel algorithm calculates multiple matrix elements simultaneously while maintaining the data dependencies, e.g. the calculation of an element depends on the others.

The data dependency of parallel sequence alignment is shown in Figure 2. The calculation of *m[i,j]* needs the calculated values of *m[i,j-1]*, *m[i-1,j-1]*, and *m[i-1,j]*. This pattern is common found in many applications.

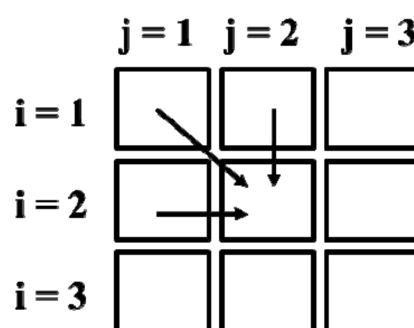

Figure 2. Data dependency of sequence alignment.

2.2 Smith-Waterman Algorithm

We choose a local sequence alignment called Smith-Waterman algorithm [4,5] as an example for our illustration. Sequence alignments have been used extensively in bioinformatics and demanded a great computational power due to large problem size, e.g. human chromosome 1 is about 247 million nucleotides long. Local sequence alignment aims to identify the local similarity between two sequences and provides the optimal alignment score.

A simplified version of Smith-Waterman [6] is defined as

$$M(i,0) = M(0,j) = 0 \; for \; all \; i,j$$

$$M[i,j] = \begin{cases} M[i-1,j-1] & if \; A[i] = B[i] \\ \max\left(M[i-1,j-1], M[i-1,j], M[i,j-1]\right) & otherwise. \end{cases}$$

Note that *A[i]* and *B[j]* are the *i*th and *j*th letters of sequences *A* and *B* respectively. The optimal alignment score is defined as $argmax_{i,j} \; M[i,j]$. Therefore, to obtain the best alignment score all matrix elements must be calculated.

2.3 Multi-core Architecture

The current trend of microprocessors is moving towards multi-core (see Figure 3). Single core cannot be improved further due to the fundamental limits in terms of electrical power and transistor density. A strategy to exploit multi-core processors is to spawn threads. Each thread will be executed in parallel. Now desktop computers with 4 to 12 cores are commercially available, and everyone can afford. Consequently, we opt to parallelization on multi-core architectures.

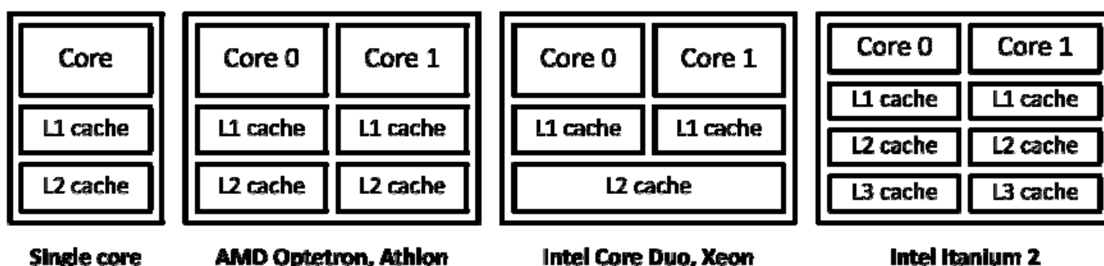

Figure 3. Single-core and multi-core architectures.

2.4 Parallelization

After choosing to create a parallel application to solve a given problem, the programmer needs to decide which parallel approach is going to be utilized. The problem has to be divided into different tasks with dependencies between them. This process is called decomposition. In high-level programming, there are mainly three kinds of decomposition:

a) Task decomposition: this approach divides the execution by the functions performed. If two or more functions can be executed at the same time, they are scheduled together. This is the simplest way of decomposition, because it takes advantage of function independency and requires little modification of the source code. For example, if we use the analogy of building a house, task decomposition would schedule installing the doors and installing the windows at the same time.

b) Data decomposition: this kind of decompositions allows tasks to execute the same kind of work in different parts of the data. When calculations in one part of the data do not depend on results obtained in other areas, this kind of decomposition tends to be the best approach. Following the previous example, data decomposition would assign several tasks (or operators) to paint the house at the same time. A possible approach would be to assign each operator one of the rooms, or one of the exterior walls.

c) Data flow decomposition: the key aspect to divide and schedule the work is how the data flows between tasks. A classic example is the producer-consumer problem, where the output of a task becomes the input of another task. Data flow is the most complex and delicate kind

of decomposition. Synchronization is a key aspect to keep the parallelization efficient. In the house building problem, data flow decomposition would be used for installing and painting the floor. The painting operator would need to wait until the installing operator completes at least part of his job to continue painting. The case studied in this thesis falls into the data flow category.

2.5 Data Dependency in Parallel Computing

Already mentioned in previous sections, data dependencies are the main concern in this thesis. We say there is a data dependency in parallel computing, when a thread needs to wait until another thread processes data in order to continue its execution.

One of the most widely known examples of data dependency is the Producer-Consumer problem mentioned in the previous section. This problem is based in two threads or tasks. The first thread, the producer, is in charge of creating data and locating it in a common buffer. The consumer has to remove the data from the buffer. This process is repeated again and again until the execution is terminated. The tricky part of this problem is that both threads depend on each other. The consumer cannot consume data if the buffer is empty and the producer cannot produce data if the buffer is full. That means each thread has data dependency with the other.

In this paper, we use an algorithm with a more complex data dependency pattern than the Producer-Consumer problem. This is the Smith-Waterman Algorithm presented previous sections. However, it is important to mention that the data dependency pattern presented by SW is a very common one. There are several widely used problems with exactly the same or very similar pattern. Some examples are:

- Error Diffusion Problem [2]: variations of this algorithm are used by many computer graphics and image processing programs. It can

efficiently transform a gray scale image into a black and white one while keeping the resemblance to the maximum.

- Levenshtein Distance [6]: given two strings, it calculates the number of changes (deletions, insertions or substitutions) needed to transform one string into the other.
- Longest Common Subsequence [14]: its aim is to identify a subsequence of two given strings with the maximum length.

As seen in [2], the most common approach to parallelize algorithms with the same or similar data dependency patterns to the case studied is to assign tasks with rows of the matrix from top to bottom. The tasks perform calculations in the rows from left to right.

2.6 Non-Uniform Memory Access (NUMA)

The demand of memory grows as the number of cores increases. However, sharing single memory causes several problems, e.g. cache coherence, memory latency, and bottleneck. To avoid performance penalty, a solution is to distribute the memory [7]. Each core is connected with its local memory via a dedicated bus. Accessing remote memory can be done through a shared communication network which is typically slower. For the time being, a number of multi-core workstations for instance HP Z800 have already become NUMA (see Figure 4). Each node consists of a quad-core processor and a local memory. HP Z800 workstation is of this configuration. NUMA-aware applications can put a thread and its frequently-used memory on the same side to maximize the performance. We will consider code optimization for NUMA in the very last chapter.

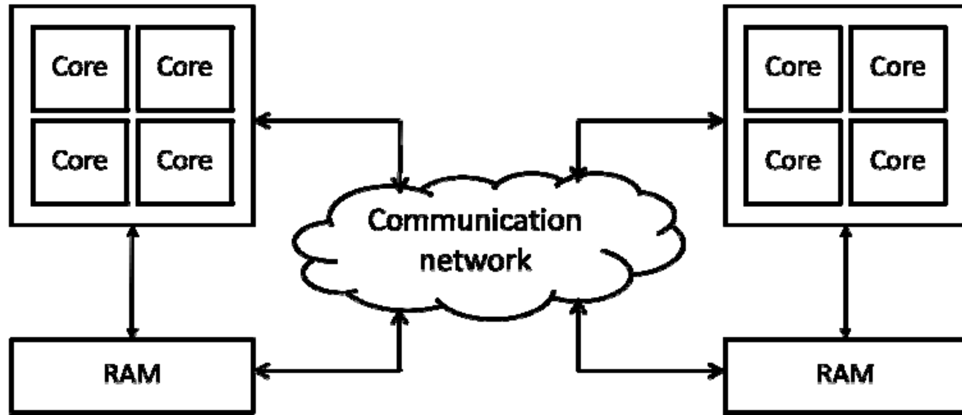

Figure 4. An NUMA architecture with two nodes.

CHAPTER III

Data Structure and Algorithmic Design

In this chapter, we will show our algorithmic design step by step plus the data structures that are needed.

3.1 Uni-Directional Filling (UDF)

We name the classical parallelization of dynamic programming [3] as Uni-Directional Filling (UDF). Figure 5 shows UDF with two threads, T1 and T2. Both threads fill the matrix simultaneously from top to bottom and from left to right. Note that the data dependency is maintained. A row has never preceded the row above.

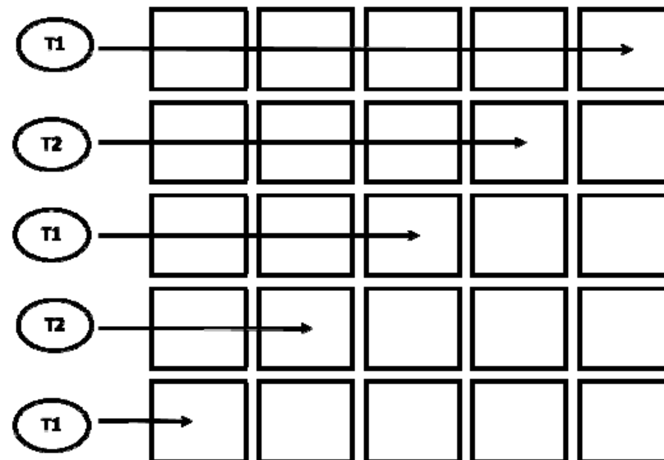

Figure 5. A graphical representation of UDF.

UDF reduces the execution time of sequential algorithm, but the performance gain is not scalable with the number of cores. UDF is seriously affected by the data dependencies mentioned in the previous chapter. As the number of cores increases, UDF cannot utilize them effectively. Most time, processing cores are idle because of waiting for data dependencies. Data dependency is a fundamental limit of parallel dynamic programming. A reduction in data dependency will improve all computational problems in dynamic programming class.

We have observed that UDF processes only in a row-wise direction. It might be better if we process in both row-wise and column-wise directions.

3.2 Bi-Directional Filling (BDF)

We propose a novel method called “Bi-Directional Filling” or BDF. Basically, BDF fills the matrix in both row-wise and column-wise directions simultaneously (see Figure 6). A half of threads take the row-wise direction, and another half takes the column-wise direction.

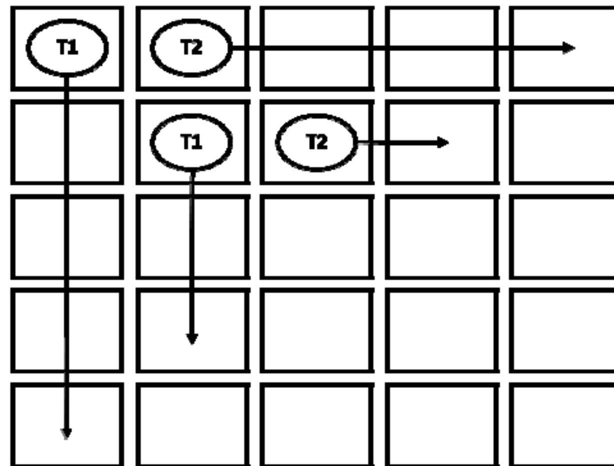

Figure 6. A graphical representation of BDF.

BDF splits threads into two groups, row threads and column threads. Roughly speaking, BDF reduces data dependencies because there are no needs to synchronize between row and column threads. Only synchronization in the same group is needed. The next section will elucidate and quantify the data dependency reduction.

3.3 A Theoretical Comparison between UDF and BDF

It happens to be nearly impossible to predict how UDF and BDF are going to perform once they are transformed in a real world application. However, it is possible to make an approximation of their performance by identifying cells in the matrix with the highest risk of generating delays in the execution. This section shows a comparison based on this kind of cells, which we call critical points.

We say there is a critical point when a thread needs a result calculated by a different thread in the previous time step. The critical points generated by UDF and BDF are shown in Figure 7 which follows this parameter setting.

- A 5x5 square matrix.
- Two threads, representing a dual-core computer.
- The dash line represents the first thread, T1.
- The solid line represents the second thread, T2.
- A number is the ideal time unit at which the matrix element will be calculated.
- A star denotes a critical point raised by two consecutive elements.

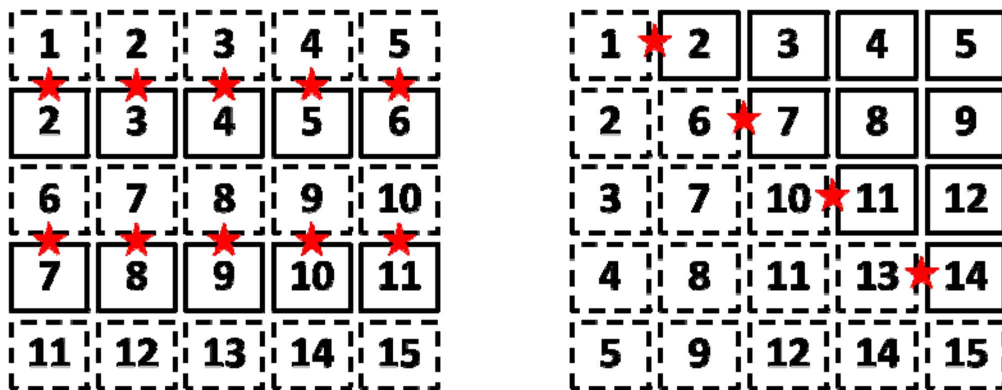

Figure 7. Critical points in UDF (left) and BDF(right).

UDF produces 10 critical points while BDF produces only 4 critical points. It is clear that BDF outperforms UDF in terms of critical point. But if the matrix size and number of threads grow, the gap will increase sharply. For instance, a 100x100 matrix UDF and BDF produce 5,000 and 99 critical points respectively.

3.4 Data Structures

A typical implementation of a matrix is a 2D array. However, BDF accesses the array in both row-wise and column-wise directions. We found that the column-wise access severely deteriorated performance. This is due to the fact that physical memory is one dimensional, and 2D arrays in C# are row-major [8]. A row is contiguous in physical memory, but a column is not. Therefore, row-wise access is greatly benefited from cache. Cache is designated to exploit the “locality of reference,” the next memory access tends to be a location nearby the previous access. This behavior is common in most programs. Column-wise access obviously violates the

locality of reference, and may trigger "cache miss" and "page fault" which deteriorate performance. In summary, we need a data structure that is effective for both row-wise and column-wise access.

An alternative is to use jagged arrays. Jagged arrays allow having rows with different lengths. Here is an example of a jagged array in C# language.

```
int[][] jaggedArray = new int[3][]; // allocate 3 rows
jaggedArray[0] = new int[3];        // allocate the 1st row (length = 3)
jaggedArray[1] = new int[2];        // allocate the 2nd row (length = 2)
jaggedArray[2] = new int[1];        // allocate the 3rd row (length = 1)
```

Our proposed data structure for DP matrix is composed of 1D array and two jagged arrays.

- D : a 1D array that represents the main diagonal of the matrix.
- R : a jagged array representing the elements that will be calculated row-wise.
- C : a jagged array representing the elements that will be calculated column-wise.

A graphical representation is shown in Figure 8. This assures that processing the matrix in column direction is exactly as fast as row direction.

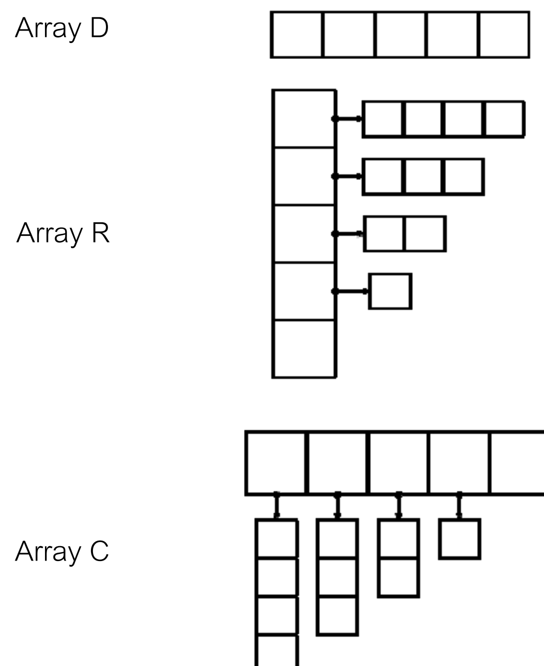

Figure 8. A graphical representation of data structures for BDF.

Accessing the matrix at row *i* and column *j* , *M[i,j]*, can be transformed as follows.

$$M[i,j] = \begin{cases} D[i] & if\ i = j \\ C[j, i-j-1] & if\ i > j \\ R[i, j-i-1] & if\ i < j \end{cases}$$

With the creation of our own data structure, we allow UDF and BDF to be compared in the same conditions. Each algorithm uses the data structure that best suits its characteristics. For UDF that would be a 2D array, and for BDF the combination of data structures mentioned above.

3.5 Code Explanation

This section will present the code part by part with a brief explanation of each part. Full versions of both UDF and BDF can be found in the appendices B and C respectively.

3.5.1 UDF

a) Global variable

```
// the first string of numbers to compare
public static int[] S1;

// the second string of numbers to compare
public static int[] S2;

// dynamic programming matrix
public static int[,] M;

// table dimension
public static int Dim = 10000;

// current row
public static int CurrentRow = 1;

// time spent waiting
public static int NumWaits = 0;
```

b) ProcessRow calculates all the values of a single row of the matrix. It also takes care of the synchronization, warranting that a cell will not be processed if the previous needed values are not calculated yet.

```
public static void ProcessRow(int row)
    {
        bool waited = false;
```

```
        for (int i = 1; i <= Dim; i++)
        {
            while (M[row - 1, i] == -1)
            {
                if (waited == false)
                {
                    waited = true;
                    /* wait for data dependency */
                    lock (typeof(UDF))
                        NumWaits++;
                }
            }
            if (S1[row] == S2[i])
                M[row, i] = M[row - 1, i - 1] + 1;
            else
                M[row, i] = GetMax(new int[] { M[row - 1, i - 1] - 1,
M[row, i - 1] - 1, M[row - 1, i] - 1 });
        }
    }
```

c) GetMax is a simple function that, given an array of integers, returns the maximum value among them.

```
public static int GetMax(int[] values)
    {
        int max = 0;
        foreach (int i in values)
        {
            if (i > max)
                max = i;
        }
        return max;
    }
```

d) InitM initializes all the values in the dynamic programming matrix M. Basically, it sets the first row and column all to zeroes, while setting any other value to -1.

```
public static void InitM()
    {
        M = new int[Dim + 1, Dim + 1];
        for (int i = 0; i <= Dim; i++)
        {
            for (int j = 0; j <= Dim; j++)
            {
                M[i, j] = (i == 0 || j == 0 ? 0 : -1);
            }
        }
    }
```

e) InitS creates the two strings of random integers that will be used for the algorithm.

```
public static void InitS()
    {
        Random random = new Random();
        S1 = new int[Dim + 1];
        S2 = new int[Dim + 1];
        // first character in both strings is not used for the
calculations
        for (int i = 0; i <= Dim; i++)
        {
            S1[i] = random.Next(4);
            S2[i] = random.Next(4);
        }
    }
```

f) DoIt represents the behavior of a given thread in the program. Each thread will keep processing rows until reaching the final result, while maintaining synchronization with other threads.

```
public static void DoIt()
    {
        int Row;
        while (true)
        {
            lock (typeof(UDF))
            {
                if (CurrentRow > Dim) return;
                Row = CurrentRow;
                CurrentRow++;
            }
            ProcessRow(Row);
        }
    }
```

g) The Main function of the program will take care of several assignments and the general management of the program. The top functionalities are:

- Initialize all the variables.
- Control the number of threads that will be used.
- Set how many times the algorithm will be executed. By doing this, we are able to see several runs of the program at the same time and compare them to get an idea of the average behavior of our algorithm.
- Create and initialize the tasks or threads.
- Output the results of the execution.

```
static void Main(string[] args)
    {
        // number of program runs. it can be changed to see the
different results
        int NumRuns = 4;

        // array to store the time spent for each run
        TimeSpan[] times = new TimeSpan[NumRuns];

        // array to store how many times the threads have to wait
        int[] waits = new int[NumRuns];

        // number of tasks that will be created to solve the problem
        int NumTasks = 8;

        // create the two random strings
        InitS();

        // create a stopwatch to monitor the time spent
        Stopwatch watch = Stopwatch.StartNew();

        for (int run = 0; run < NumRuns; run++)
        {
            // restart every variable before each run
            CurrentRow = 1;
            NumWaits = 0;
            InitM();
            Task[] tasks = new Task[NumTasks];
            watch.Reset();
            watch.Start();

            for (int i = 0; i < NumTasks; i++)
            {
                tasks[i] = new Task(() =>
                {
                    DoIt();
                });
                tasks[i].Start();
            }

            Task.WaitAll(tasks);

            watch.Stop();
            times[run] = (watch.Elapsed);
            waits[run] = NumWaits;

        }
        Console.WriteLine("Classic algorithm performance for " +
NumTasks + " treads in " + NumRuns + " program runs");
        Console.WriteLine("\t Run \t Time \t\t\t Wait");
        for (int i = 0; i < NumRuns; i++)
        {
            Console.WriteLine("\t " + (i + 1) + " \t " + times[i] + " \t
" + waits[i]);
        }

        Console.ReadLine();
    }
```

3.5.2 BDF

a) Global variables

```
// the first string of numbers to compare
public static int[] S1;

// the second string of numbers to compare
public static int[] S2;

// data structure for the row threads
public static int[][] R;

// data structure for the column threads
public static int[][] C;

// data structure for the main diagonal
public static int[] D;

// table dimension
public static int Dim = 5000;

// current row
public static int CurrentRow = 1;

// current column
public static int CurrentCol = 1;

// current direction (0 = column, 1 = row)
public static int CurrentDir = 0;

// times a thread had to wait
public static int NumWaits = 0;
```

b) ProcessDiagonal processes one cell (x) in the main diagonal of the dynamic programming matrix. The calculation of the mentioned cell has to be synchronized with both the columns and the rows data structure. That is because in order to calculate the value of the cell three previous values will be needed: the previous one in the diagonal, one in the rows structure and another in the columns structure.

```
public static void ProcessDiagonal(int x)
    {
        bool waited = false;
        while ((C[x - 1][0] == -1) || (R[x - 1][0] == -1) || (D[x - 1]
== -1))
        {
            if (waited == false)
            {
                waited = true;
                /* wait for data dependency */
                lock (typeof(MultiDirectional))
```

```
                    NumWaits++;
                }
            }
            if (S1[x] == S2[x])
                D[x] = D[x - 1] + 1;
            else
                D[x] = GetMax(new int[] { D[x - 1] - 1, R[x - 1][0] - 1, C[x
- 1][0] - 1 });
        }
```

c) ProcessRow will calculate and entire row in the R data structure. The execution will have to be synchronized with the diagonal data structure (D) and with previous rows in R.

```
public static void ProcessRow(int x)
    {
        if (x < Dim)
        {
            if (S1[x] == S2[x + 1])
                R[x][0] = R[x - 1][0] + 1;
            else
                R[x][0] = GetMax(new int[] { R[x - 1][0] - 1, D[x] - 1,
R[x - 1][1] - 1 });
            bool waited = false;
            for (int i = 1; i < R[x].Length; i++)
            {
                while (R[x - 1][i + 1] == -1)
                {
                    if (waited == false)
                    {
                        waited = true;
                        /* wait for data dependency */
                        lock (typeof(MultiDirectional))
                            NumWaits++;
                    }
                }
                if (S1[x] == S2[i + x + 1])
                    R[x][i] = R[x - 1][i] + 1;
                else
                    R[x][i] = GetMax(new int[] { R[x - 1][i] - 1, R[x][i
- 1] - 1, R[x - 1][i + 1] - 1 });
            }
        }
    }
```

d) ProcessCol will calculate and entire column in the columns data structure (C). The execution will have to be synchronized with D and the previous column in C.

```
public static void ProcessCol(int x)
    {
        if (x < Dim)
        {
            if (S1[x + 1] == S2[x])
```

```
                C[x][0] = C[x - 1][0] + 1;
            else
                C[x][0] = GetMax(new int[] { C[x - 1][0] - 1, D[x] - 1,
C[x - 1][1] - 1 });
            bool waited = false;
            for (int i = 1; i < C[x].Length; i++)
            {
                while (C[x - 1][i + 1] == -1)
                {
                    if (waited == false)
                    {
                        waited = true;
                        /* wait for data dependency */
                        lock (typeof(MultiDirectional))
                            NumWaits++;
                    }
                }
                if (S1[i + x + 1] == S2[x])
                    C[x][i] = C[x - 1][i] + 1;
                else
                    C[x][i] = GetMax(new int[] { C[x - 1][i] - 1, C[x][i
- 1] - 1, C[x - 1][i + 1] - 1 });
            }
        }
    }
```

e) GetMax is a simple function that, given an array of integers, returns the maximum value among them.

```
public static int GetMax(int[] values)
    {
        int max = 0;
        foreach (int i in values)
        {
            if (i > max)
                max = i;
        }
        return max;
    }
```

f) InitM will create and initialize the three data structures that will represent the dynamic programming matrix. This includes two Jagged arrays (one for the rows and other for the columns) and a 1D array representing the main diagonal of the virtual matrix. Using them together, they will behave as whole data structure that will be time efficient when memory accesses are required in both column and row directions. The first row and the first column of the virtual matrix will be set to zeroes, while any other element will be set to -1.

```
public static void InitM()
    {
        D = new int[Dim + 1];
        D[0] = 0;
        for (int i = 1; i < Dim + 1; i++)
        {
            D[i] = -1;
        }
        R = new int[Dim][];
        C = new int[Dim][];
        for (int i = 0; i < Dim; i++)
        {
            R[i] = new int[Dim - i];
            C[i] = new int[Dim - i];
            for (int j = 0; j < Dim - i; j++)
            {
                R[i][j] = (i == 0 ? 0 : -1);
                C[i][j] = (i == 0 ? 0 : -1);
            }
        }

    }
```

g) InitS creates the two strings of random integers that will be used for the algorithm.

```
public static void InitS()
    {
        Random random = new Random();
        S1 = new int[Dim + 1];
        S2 = new int[Dim + 1];
        // first character in both strings is not used for the
calculations
        for (int i = 0; i <= Dim; i++)
        {
            S1[i] = random.Next(4);
            S2[i] = random.Next(4);
        }
    }
```

h) DoIt will represent the behavior of each thread in the execution. It will process either one column or one row in the matrix. The synchronization is done with the use of the diagonal array. If the necessary value in D is not yet calculated, it has to be processed before continuing with the rest of the execution.

```
public static void DoIt()
    {
        int Row, Col, Dir;
        while (true)
        {
            lock (typeof(MultiDirectional))
            {
```

```
                if (CurrentRow > Dim || CurrentCol > Dim) return;

                Row = CurrentRow;
                Col = CurrentCol;
                Dir = CurrentDir;
                if (CurrentDir == 0) // column direction
                {
                    CurrentDir = 1;
                    if (D[CurrentCol] == -1)
                        ProcessDiagonal(CurrentCol);
                    CurrentCol++;
                }
                else // row direction
                {
                    CurrentDir = 0;
                    if (D[CurrentRow] == -1)
                        ProcessDiagonal(CurrentRow);
                    CurrentRow++;
                }
            }
            if (Dir == 0)
            {
                ProcessCol(Col);
            }
            else
            {
                ProcessRow(Row);
            }
        }
    }
```

i) The main function of the program will take care of several assignments and the general management of the program. The top functionalities are:

- Initialize all the variables.
- Control the number of threads that will be used.
- Set how many times the algorithm will be executed. By doing this, we are able to see several runs of the program at the same time and compare them to get an idea of the average behavior of our algorithm.
- Create and initialize the tasks or threads.
- Output the results of the execution.

```
static void Main(string[] args)
    {
        // number of program runs. it can be changed to see the
different results
        int NumRuns = 4;
```

```
        // array to store the time spent for each run
        TimeSpan[] times = new TimeSpan[NumRuns];

        // array to store the times a thread had to wait
        int[] waits = new int[NumRuns];

        // number of tasks that will be created to solve the problem
        int NumTasks = 8;


        InitS();
        Stopwatch watch = Stopwatch.StartNew();

        for (int run = 0; run < NumRuns; run++)
        {
            // restart every variable before each run
            InitM();
            watch.Reset();
            CurrentRow = 1;
            CurrentCol = 1;
            CurrentDir = 0;
            NumWaits = 0;
            Task[] tasks = new Task[NumTasks];

            watch.Start();

            for (int i = 0; i < NumTasks; i++)
            {
                tasks[i] = new Task(() =>
                {
                    DoIt();
                });
                tasks[i].Start();
            }

            Task.WaitAll(tasks);

            watch.Stop();
            times[run] = (watch.Elapsed);
            waits[run] = NumWaits;
        }
        Console.WriteLine("BDF algorithm performance for " + NumTasks +
" treads in " + NumRuns + " program runs");
        Console.WriteLine("\t Run \t Time \t\t\t Wait");
        for (int i = 0; i < NumRuns; i++)
        {
            Console.WriteLine("\t " + (i + 1) + " \t " + times[i] + " \t
" + waits[i]);
        }
        Console.ReadLine();
    }
```

3.6 NUMA Optimization

As the number of cores and memory capacity are increasing, NUMA (Non-Uniform Memory Access) will play an important role in multi-core architectures. The new operating systems such Windows 7 and Linux support NUMA by providing a set of system calls for controlling processor affinity and allocating memory on a specific NUMA node. Note that setting processor affinity allows user programs to be executed on a specific core in a specific NUMA node. For instance, affinity = 0x0F prefers core 0 to 3 (the first NUMA node) and affinity = 0xF0 prefers core 4 to 7 (the second NUMA node). However, applications must be aware of NUMA and use it. Otherwise, NUMA will be managed by operating systems and probably not optimal. The worst case is executing a thread on one node and allocating its memory on another node. In summary, NUMA optimization is to allocate a thread and its frequently-used memory on the same node.

Unfortunately, the current .NET framework 4.0 does not support NUMA. So we make a direct call to Windows API via Platform Invocation Services (P/Invoke). Two important calls are 1) SetThreadAffinityMask and 2) VirtualAllocExNuma. The first call is for setting processor affinity of a thread. The second call is for allocating virtual memory on a specified NUMA node.

CHAPTER IV

Experimental Results and Discussion

The data dependency analysis in the previous chapter shows that BDF theoretically yields less number of critical points than that of UDF. Therefore, BDF is expected to outperform UDF in practice. However, we need to implement UDF and BDF to investigate the actual performance. The implementation involves some parameter settings, hardware and software configurations as follows.

- HP Z800 workstation with two Intel Xeon E5520 processors (a total of 8 cores), 16GB RAM, and Windows 7 Professional 64-bit.
- In BIOS setting, hyper-threading was disabled.
- HP Z800 supported non-uniform memory access (NUMA) with two memory nodes. Each node is comprised of four cores (a quad-core processor). At beginning, we disabled NUMA (enabled interleaved memory) so that the memory became homogeneous and uniform access.
- The DP matrix was set at 20,000x20,000. We added a counter variable for counting critical points. Elapsed time was measured by a stopwatch (System.Diagnostics.Stopwatch).
- Smith-Waterman algorithm is a simplified version [6]. Sequences were random.
- .NET framework 4.0 [9]. All thread programming was carried out via System.Threading.Task. For thread synchronization, we used only the “lock” keyword.

In order to study multiple scenarios, 8 different versions of the programs were executed. Each one representing a number of launched threads from 1 to 8. Each version was executed 10 times to increase reliability. The results shown in this section are the average of the mentioned 10 runs.

The first aspect we wanted to compare was the number of delays produced by the effect of data dependencies.

Section 3.3 showed a theoretical comparison of cells with high risk of generating delays (critical points) produced by UDF and BDF. In this section we want to calculate the actual number of delays produced by both methods in a real word environment. In order to achieve this, we created a counter that would be increased every time a thread has to go into a waiting process (delay).

Figure 9 shows the total number of delays varied with the number of threads. It is clear that BDF produced less number of delays than UDF. At 8 threads, BDF obtains a 55% reduction in the delays number. Moreover, the gap increases with the degree of parallelism (the number of threads).

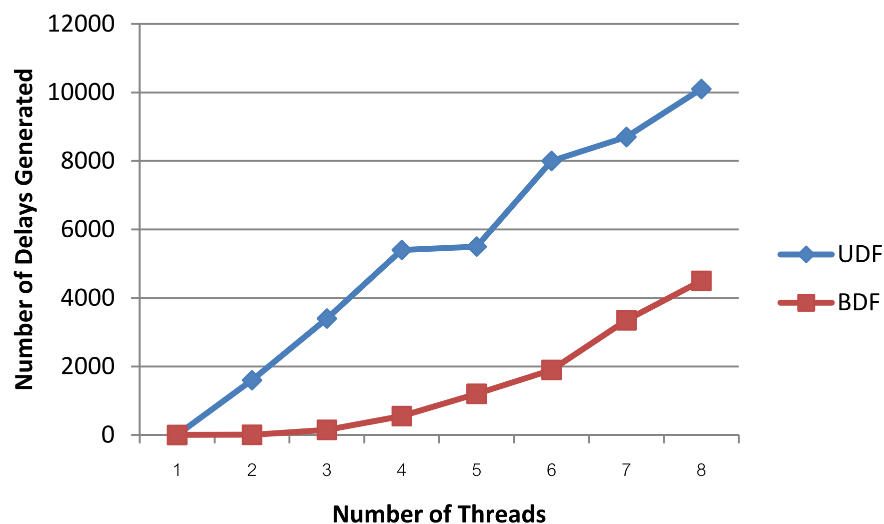

Figure 9. A comparison of delays generated by UDF and BDF.

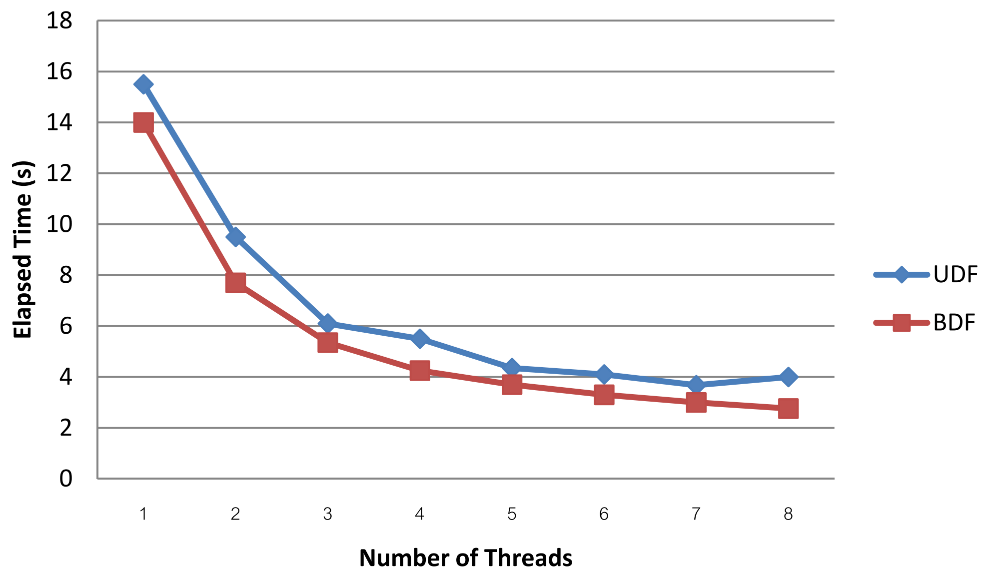

Figure 10. A comparison of elapsed times between UDF and BDF.

Figure 10 shows the elapsed time varied with the number of threads. It is seen that BDF slightly outperforms UDF for all number of threads, although BDF produces very less critical points. This can be explained by Amdahl's law [10,11]. Amdahl's law states that the maximum speed up of a particular system is:

$$\frac{1}{(1-P)+\frac{P}{S}}$$

- *P* is the fraction of the system that can be optimized.
- *S* is the speed up of *P*.

In our system, we can divide the total elapsed time into two parts, the execution time (1 – P) and the time waiting for data dependencies (P). BDF improves only the fraction P. Hence, if P is small, optimizing P will not yield much overall improvement. For instance, optimizing P = 0.1 with 100 times speedup yields 1 / (1 – 0.1 + (0.1/100)) = 1.11 or about 11% improvement. Note that even at one thread BDF still outperforms UDF. This is due to the fact that the data structure of BDF is composed of many 1D arrays as opposed to one big 2D array in UDF.

However, after normalizing the total elapsed time to one, it is possible to appreciate the real performance of BDF when compared with UDF. Figure 11 shows BDF's speedup for each number of threads.

The percentage of improvement increases with the number of threads. This implies that the fraction P also increases with the number of threads. In other words, as we elevate the degree of parallelism by adding more threads, the fraction of time waiting for data dependencies increases. At a certain number of threads, we believe that this fraction will be significantly large (>50%), and data dependency will be a major bottleneck of parallelization. Although P is as large as 50%, Amdahl's law indicates a very small maximum speed up at 2.0 (if P is completely removed). But the speed up is the number of times faster than a parallel algorithm (UDF), not a sequential algorithm. If UDF is 100 times faster than a sequential algorithm, BDF can be 2 x 100 = 200 times faster.

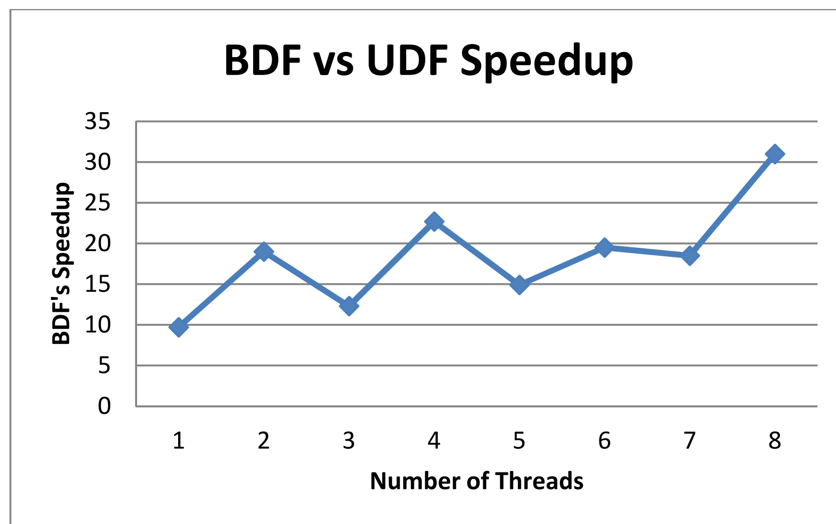

Figure 11. BDF's speedup compared with UDF.

Figure 12 shows a comparison of CPU utilization between UDF and BDF. The data was collected via "Concurrency Visualizer," a tool provided by MS Visual Studio 2010 (Premium and Ultimate versions). At the beginning of execution (on the left), only one core is busy due to sequential initialization. Then the program spawns multiple threads and makes the cores busy. It is very obvious that BDF greatly improves the core utilization. As a result, BDF runs faster.

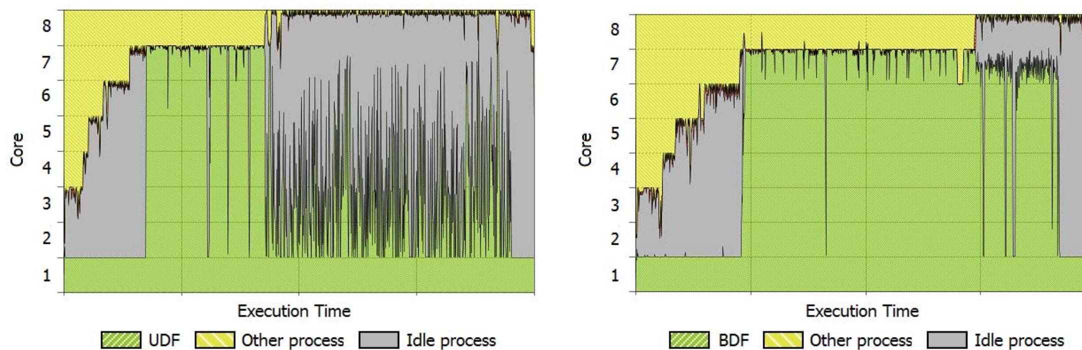

Figure 12. Core utilization of UDF (left) and BDF (right).

Figure 13 compares the best and the worst case of NUMA (Non-Uniform Memory Access). In the best case, threads and their memory are allocated on the same NUMA node. Consequently, threads always use local memory. In the worst case, threads and their memory are separated on different nodes, and threads always use remote memory. Surprisingly, the performance of the best and the worst cases does not differ. We found that this is a cache effect. While a thread is executing a page of memory, cache loads the next page in advance. Subsequently, moving to the next consecutive page has no delays. To prove this, we setup an experiment as follows. A total of 10,000 pages were allocated on local or remote memory. Each page is 4k bytes. We performed both sequential and random reads. Table 2 shows that local memory is as good as remote memory if it is read sequentially. In case of random read, remote memory is about 37% slower. Cache is designed to exploit the locality of reference. Therefore, the sequential read gains the most benefit of cache. Dynamic programming reads the matrix sequentially, hence no difference between local and remote memory.

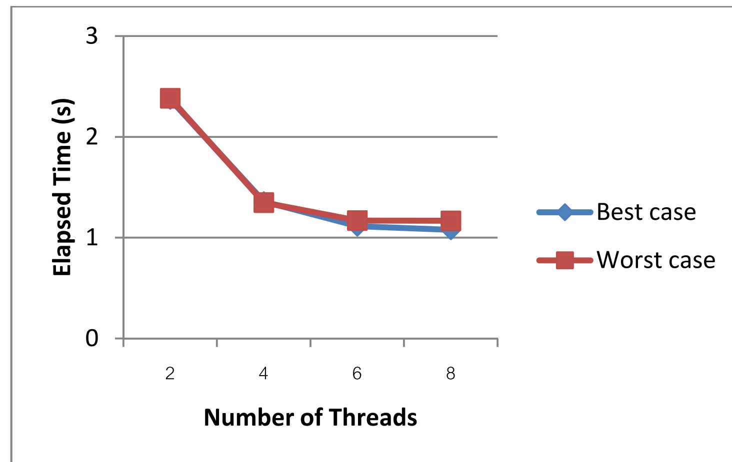

Figure 13. A comparison of elapsed times between the best and the worst cases.

Table 2. A comparison of access time between local and remote memory.

	Local memory	Remote memory
Sequential read	49.40 ms	50.20 ms
Randomly read	1,594.80 ms	2,184.80 ms

Chapter V

Conclusion

BDF has proven as a promising method for reducing data dependencies in dynamic programming tables. The implementation of BDF is not very difficult. It does not require complicated programming, but only a little modification of DP tables using a series of one-dimensional arrays.

The classical data partitioning like UDF has been used for long because it is simple and easy programming. In the past, the number of processing cores in a commercial CPU was very small, typically 2 to 4 cores. But now 4-to-12-cores computers are quite common, and tend to be increased rapidly. Therefore, UDF may not be sufficient to unleash the true power of multi-core architectures. In fact, data dependency is the fundamental limit of all architectures that exploit parallelism. Any reductions in data dependency will elevate the degree of parallelism and make the performance scalable with the number of processing cores.

As clearly seen in the experimental results, BDF outperforms UDF in all aspects. We strongly believe that the performance improvement of BDF will be more significant as the number of cores increase because data dependency will become a major bottleneck at higher level of parallelism. At 8 cores, BDF is about 31% faster than UDF. This is not very impressive at the first glance, but it is important to note that BDF is compared with a standard parallelization, not a sequential algorithm. A small improvement due to BDF multiplies the speed up of standard parallelization.

Finally, we emphasize that BDF does not only improve the performance of Smith-Waterman algorithm. It can be applied to any DP problems that have a similar data dependency pattern. Moreover, BDF can be generalized to multi-directional filling (MDF) for some applications, for example, multiple sequence alignment [12] and dynamic time warping [13]. At higher dimensions, there are more data dependencies, and MDF will play a crucial role in unleashing the true power of parallel computing.

This work has been published in JCSSE 2011 [15].

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, Introduction to Algorithms. 2nd Edition : McGraw-Hill, 2003.

[2] S. Dasgupta, C. H. Papadimitriou, U. V. Vazirani, Algorithms. chapter 6, July 18, 2006.

[3] S. Akhter, J. Roberts. Multi-Core Programming, 1st Edition, Intel Press, 2006.

[4] N. Cristianini, M. W. Hahn, Introduction to Computational Genomics. 1st Edition, Cambridge University Press, 2007.

[5] T. F. Smith, M. S. Waterman, Identification of Common Molecular Subsequences. Journal of Molecular Biology 147 (1981): 195–197.

[6] Z. Su et al., Plagiarism Detection Using the Levenshtein Distance and Smith-Waterman Algorithm. 3rd Int. Conf. on Innovative Computing Information and Control (ICICIC'08) (2008) : 569.

[7] N. Manchanda, K. Anand, Non-Uniform Memory Access (NUMA).

[8] U. Drepper, What Every Programmer Should Know About Memory. Red Hat, Inc., November 2007.

[9] A. Freeman, Pro .NET 4 Parallel Programming in C#. 1st Edition : Apress, 2010.

[10] G. M. Amdahl, Validity of the single processor approach to achieving large scale computing capabilities. AFIPS spring joint computer conf. (1967).

[11] M. D. Hill, M. R. Marty, Amdahl's Law in the Multicore Era. IEEE Computer 41 (2008) : 33-38.

[12] R. C. Edgar, S. Batzoglou. Multiple sequence alignment. Current Opinion in Structural Biology 16 (2006) : 1-6.

[13] M. Müller, Information Retrieval for Music and Motion. Springer (2007).

[14] L. Bergroth, H. Hakonen, T. Raita, A Survey of Longest Common Subsequence Algorithms. 7th Int. Symp. on String Processing and Information Retrieval (SPIRE 2000) (2000) : 39-48.

[15] G. Delgado, C. Aporntewan, Data Dependency Reduction in Dynamic Programming Matrix. 8th Int. Joint Conf. on Computer Science and Software Engineering (JCSSE) (2011) : 234-236.

Appendices

Appendix A

2011 Eighth International Joint Conference
on Computer Science and Software Engineering (JCSSE)

Data Dependency Reduction in Dynamic Programming Matrix

Guillermo Delgado
Master in Computer Science and Information Technology
Department of Mathematics, Faculty of Science
Chulalongkorn University, Bangkok, 10330, Thailand
GuillermoDelga@gmail.com

Chatchawit Aporntewan
Department of Mathematics, Faculty of Science
Chulalongkorn University, Bangkok, 10330, Thailand
Chatchawit.A@chula.ac.th

***Abstract*—Dynamic Programming (DP) plays an important role in solving a large number of computational problems. As the number of cores per processor is increasing rapidly, new software must be capable of exploiting the advantages of multi-core architectures. A typical DP begins with constructing a matrix, and then calculating each element one by one. The standard parallelization spawns multiple threads, one for each row, while maintains the data dependency via thread synchronization. However, as the number of cores increase, the performance degrades due to data dependency.**

Herein, we propose a novel method for calculating a DP matrix in parallel. In contrast to the standard method that always calculates from up to down and left to right, our method performs the calculation in multiple directions. Therefore, the wait time for data dependency is remarkably reduced. To demonstrate our method, a local sequence alignment algorithm called Smith-Waterman (SW) was chosen as an instance of DP. However, our method is not only limited to SW algorithm, but it is applicable to other DP problems that have the similar patterns of data dependency.

A comparison with the standard method was conducted on HP Z800 Workstation equipped with two quad-core processors. The results show that our method performs significantly faster than the classical approach.

Keywords-component; parallel dynamic programming; data dependency reduction; multi-core; multi-threading

I. Introduction

Dynamic Programming (DP) has been used to solve a wide range of computational problems [1]. DP is based on problem decomposition, recursively breaking down a problem into smaller sub-problems. The identical sub-problems are computed only once, hence avoiding redundant computation.

Our scope is limited to the bottom-up approach where the solutions are iteratively generated from small to large sub-problems. In this process, a table or an array called "DP matrix" is needed. A sequential algorithm fills the matrix element by element. In contrast, with parallel computing multiple matrix elements are calculated simultaneously while maintaining the data dependency, e.g. the calculation of an element depends on the others.

The patterns of data dependency are varied among DP problems. However, most of them including sequence alignment can be formulated as filling a two-dimensional table, and they share the similar pattern where *m[i][j]* depends on *m[i][j-1]*, *m[i-1][j-1]*, and *m[i-1][j]*. In other words, *m[i][j]* cannot be calculated if the other three elements have not been computed yet. The classical parallel algorithm partitions the data in rows and fills multiple rows simultaneously [2]. To maintain the data dependency, a row cannot precede the row above. As a result, the degree of parallelism is bounded by the data dependency.

In this paper, we propose a novel data partitioning that reduces the data dependency and elevates the degree of parallelism. Instead of filling the matrix in only row-wise direction, we fill the matrix in both row-wise and column-wise directions, and name it "Bi-Directional Filling (BDF)" as oppose to the classical Uni-Directional Filling (UDF). The remaining sections are organized as follows. Section 2 illustrates the BDF and analyzes the reduction of data dependency compared to that of UDF. Section 3 explains the multi-threading implementation and makes a performance comparison between UDF and BDF. Sections 4 discusses the paper.

II. Filling the Matrix in Parallel

A. Data Dependency

As mentioned earlier, the patterns of data dependency vary among DP problems. However, most of them share the same pattern as shown in Fig. 1. The arrows indicate the data flow, where the element *m[i][j]* needs to wait the calculations of *m[i][j-1]*, *m[i-1][j-1]*, and *m[i-1][j]* where *m[i][j]* denotes the matrix element at row *i* and column *j*.

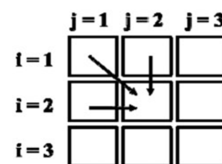

Figure 1. A typical pattern of data dependency in DP.

B. Uni-Directional Filling (UDF)

UDF enables the parallelism by processing multiple rows simultaneously as shown in Fig. 2. The rows are computed from left to right and from top to bottom.

978-1-4577-0687-5/11/$26.00 ©2011 IEEE

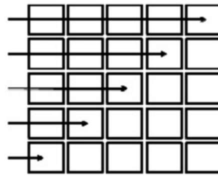

Figure 2. A snapshot of a DP matrix processed by UDF.

C. *Bi-Directional Filling (BDF)*

BDF fills the matrix in both row-wise and column-wise directions as shown in Fig. 3. The numbers indicate the order in which the rows and the columns are processed.

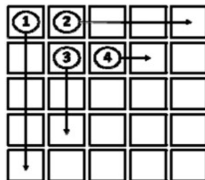

Figure 3. A snapshot of a DP matrix processed by BDF.

D. *Data Dependency Reduction*

Fig. 4 and 5 demonstrate the ideal executions of UDF and BDF with two threads (one represented by dash lines and another represented by solid lines). The numbers inside the boxes are time steps where cells were updated. The stars indicate the critical data dependency where one thread uses the result of another thread in the previous time step. Given an *NxN* matrix, the ideal executions of UDF and BDF contain $N\lfloor N/2 \rfloor$ and *N-1* critical points respectively.

Even with the small example of a 5x5 matrix shown in Fig. 4 and 5, there is a big difference (10 dependencies for UDF and 4 for BDF). But as the matrix becomes bigger, the gap dramatically increases. For example, in a 100x100 matrix the dependencies would be 5,000 and 99 for UDF and BDF respectively.

Figure 4. An ideal execution of UDF with two threads.

Figure 5. An ideal execution of BDF with two threads.

III. A Performance Comparison, UDF vs. BDF

The data dependency analysis in the previous section shows that BDF yields less critical dependency than that of UDF. Therefore, BDF is expected to outperform UDF in practice. However, we need to implement UDF and BDF to investigate the real performance. This implementation involves some parameter settings, hardware and software configurations.

A. *Smith-Waterman (SW) Algorithm*

We chose sequence alignment to make a performance comparison between UDF and BDF. There are many applications of sequence alignment in life science. As the human genome project is advancing, more computational power is needed to cope with larger problem size. Smith-Waterman (SW) is a local-alignment algorithm [3]. We employed a simplified SW [4], which is defined as follows:

$$M[i,0] = M[0,j] = 0 \text{ for all } i, j$$

$$M[i,j] = \begin{cases} M[i-1, j-1] \text{ if } A[i] = B[j] \\ max(0, M[i-1,j-1], M[i-1,j], M[i,j-1]) \text{ otherwise} \end{cases}$$

A[i] and *B[j]* are the ith and j^{th} letter of sequences *A* and *B* respectively. The objective of SW algorithm is to calculate the optimal alignment score, *M[length(A), length(B)]*. In order to do this it is necessary to compute every cell in the matrix *M*.

B. *Data Structures for BDF*

Accessing a matrix in row-wise is faster than that in a column-wise fashion. This happens because the row elements are consecutive in physical memory and are packed in a cache line. To improve the column-wise access, we cannot simply use a two-dimensional array. A solution is to adopt a new data structure that gives the best performance on both row-wise and column-wise access. The new data structure that stores a DP matrix is a collection of one-dimensional arrays. Each of them is either a partial row or a partial column that will be accessed by BDF (see Fig. 3).

C. *.NET Parallel Programming in C#*

We employed C# and .NET Framework 4.0 to test our UDF and BDF implementations. All programming was done using System.Threading.Task [5]. A task (or a thread) corresponds to a row or a column. No calls to Windows APIs were required. Only a "lock" keyword was used for synchronization. The data dependency was handled by checking the condition `while (m[i][j-1] < 0 && m[i-1][j-1] < 0 && m[i-1][j] < 0) { /* wait */ }`. Note that the matrix was initialized with negative numbers in every cell.

D. Hardware and Software Configurations

We employed HP Z800 with two Intel Xeon E5520 processors (a total of 8 cores), 16GB RAM, and Windows 7 Professional 64-bit. In the BIOS, hyperthreading was disabled. HP Z800 supported non-uniform memory access (NUMA) with two memory nodes. Each processor was connected to a local memory and other non-local memory nodes. Accessing local memory was faster. We disabled NUMA so that the access time of the entire memory became homogeneous.

E. Performance Evaluation

We used SW algorithm as a benchmark. The DP matrix was set at 20,000x20,000. Fig. 6 shows that BDF always spends less time than UDF for any number of threads launched. Moreover, at 8 cores BDF reduces the UDF computational time by 31%.

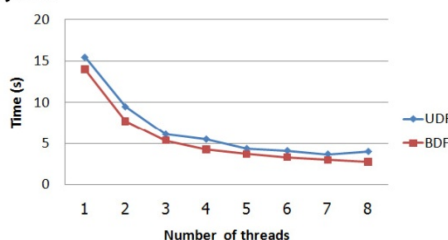

Figure 6. Time performance comparison of UDF and BDF.

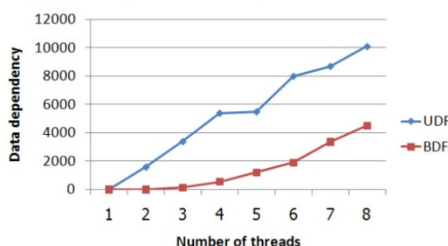

Figure 7. Data dependency comparison of UDF and BDF.

Fig. 7 shows the data dependency which is defined as the number of times that any threads have to wait for other threads in order to continue the execution. BDF dramatically reduces the data dependency in a range from 55% (8 threads) to 99% (2 threads).

F. Concurrency Visualizer

Fig. 8 and 9 show the core utilization for UDF and BDF respectively. Note that the parts where only one core is utilized involve matrix initialization which is done sequentially. It is very clear that BDF can make more utilization of processing cores than UDF. The plots were drawn from "Concurrency Visualizer," a new tool in MS Visual Studio 2010 (only available in Premium and Ultimate versions) [5].

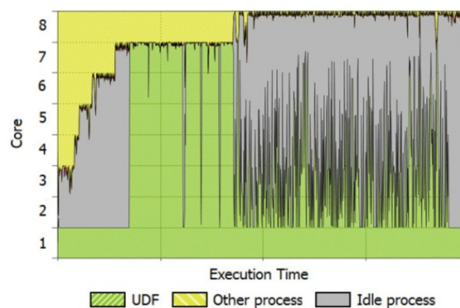

Figure 8. Core utilization for UDF.

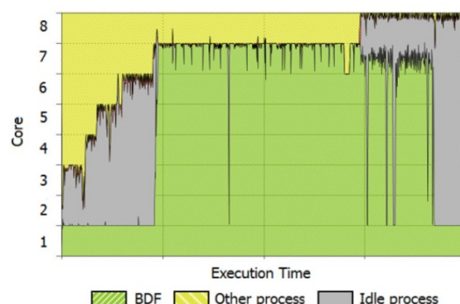

Figure 9. Core utilization for BDF.

IV. Discussion

The classical data partitioning like UDF has been used for long because it is simple and easy programming. In the past, the number of processing cores in a commercial CPU was very small, typically 2 to 4 cores. But now 4-to-12-cores computers are quite common, and tend to be increased rapidly. Therefore, UDF may not be sufficient to unleash the true power of multi-core architectures. In fact, data dependency is the fundamental limit of all architectures that exploit parallelism. Any reductions in data dependency will elevate the degree of parallelism and make the performance scalable with the number of processing cores.

References

[1] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, Introduction to Algorithms, 2nd ed., McGraw-Hill, 2003.

[2] S. Akhter and J. Roberts, Multi-Core Programming, 1st ed., Intel Press, 2006.

[3] N. Cristianini and M. W. Hahn, Introduction to Computational Genomics, 1st ed., Cambridge University Press, 2007.

[4] Z. Su et al., Plagiarism Detection Using the Levenshtein Distance and Smith-Waterman Algorithm, 3rd Int. Conf. on Innovative Computing Information and Control (ICICIC'08), 2008, pp. 569.

[5] A. Freeman, Pro .NET 4 Parallel Programming in C#, 1st ed., Apress, 2010.

Appendix B

Uni-Directional Filling (UDF) Source Code

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Diagnostics;

class UDF
{
    // the first string of numbers to compare
    public static int[] S1;

    // the second string of numbers to compare
    public static int[] S2;

    // dynamic programming matrix
    public static int[,] M;

    // table dimension
    public static int Dim = 10000;

    // current row
    public static int CurrentRow = 1;

    // time spent waiting
    public static int NumWaits = 0;


    public static void ProcessRow(int row)
    {
        bool waited = false;
        for (int i = 1; i <= Dim; i++)
        {
            while (M[row - 1, i] == -1)
            {
                if (waited == false)
                {
                    waited = true;
                    /* wait for data dependency */
                    lock (typeof(UDF))
                        NumWaits++;
                }
            }
            if (S1[row] == S2[i])
                M[row, i] = M[row - 1, i - 1] + 1;
            else
                M[row, i] = GetMax(new int[] { M[row - 1, i - 1] - 1,
M[row, i - 1] - 1, M[row - 1, i] - 1 });
        }
    }

    public static int GetMax(int[] values)
    {
        int max = 0;
```

```
    foreach (int i in values)
    {
        if (i > max)
            max = i;
    }
    return max;
}

public static void InitM()
{
    M = new int[Dim + 1, Dim + 1];
    for (int i = 0; i <= Dim; i++)
    {
        for (int j = 0; j <= Dim; j++)
        {
            M[i, j] = (i == 0 || j == 0 ? 0 : -1);
        }
    }
}

public static void InitS()
{
    Random random = new Random();
    S1 = new int[Dim + 1];
    S2 = new int[Dim + 1];
    // the first character in both strings is not used for calculation
    for (int i = 0; i <= Dim; i++)
    {
        S1[i] = random.Next(4);
        S2[i] = random.Next(4);
    }
}

public static void DoIt()
{
    int Row;
    while (true)
    {
        lock (typeof(UDF))
        {
            if (CurrentRow > Dim) return;
            Row = CurrentRow;
            CurrentRow++;
        }
        ProcessRow(Row);
    }
}

static void Main(string[] args)
{
    // number of time the algorithm will be run
    int NumRuns = 4;

    // array to store the time spent for each run
    TimeSpan[] times = new TimeSpan[NumRuns];

    // array to store how many times the threads have to wait
    int[] waits = new int[NumRuns];
```

```
        // number of tasks that will be created to solve the problem
        int NumTasks = 4;

        // create the two random strings
        InitS();

        // create a stopwatch to monitor the time spent
        Stopwatch watch = Stopwatch.StartNew();

        for (int run = 0; run < NumRuns; run++)
        {
            // restart every variable before each run
            CurrentRow = 1;
            NumWaits = 0;
            InitM();
            Task[] tasks = new Task[NumTasks];
            watch.Reset();
            watch.Start();

            for (int i = 0; i < NumTasks; i++)
            {
                tasks[i] = new Task(() =>
                {
                    DoIt();
                });
                tasks[i].Start();
            }

            Task.WaitAll(tasks);

            watch.Stop();
            times[run] = (watch.Elapsed);
            waits[run] = NumWaits;

        }
        Console.WriteLine("Classic algorithm performance for " + NumTasks +
" treads in " + NumRuns + " program runs");
        Console.WriteLine("\t Run \t Time \t\t\t Wait");
        for (int i = 0; i < NumRuns; i++)
        {
            Console.WriteLine("\t " + (i + 1) + " \t " + times[i] + " \t "
+ waits[i]);
        }

        Console.ReadLine();
    }
}
```

Appendix C

Bi-Directional Filling (UDF) Source Code

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Diagnostics;

class MultiDirectional
{
    // the first string of numbers to compare
    public static int[] S1;

    // the second string of numbers to compare
    public static int[] S2;

    // data structure for the row threads
    public static int[][] R;

    // data structure for the column threads
    public static int[][] C;

    // data structure for the main diagonal
    public static int[] D;

    // table dimension
    public static int Dim = 5000;

    // current row
    public static int CurrentRow = 1;

    // current column
    public static int CurrentCol = 1;

    // current direction (0 = column, 1 = row)
    public static int CurrentDir = 0;

    // times a thread had to wait
    public static int NumWaits = 0;


    public static void ProcessDiagonal(int x)
    {
        bool waited = false;
        while ((C[x - 1][0] == -1) || (R[x - 1][0] == -1) || (D[x - 1] == -
1))
        {
            if (waited == false)
            {
                waited = true;
                /* wait for data dependency */
                lock (typeof(MultiDirectional))
                    NumWaits++;
            }
        }
```

```
        if (S1[x] == S2[x])
            D[x] = D[x - 1] + 1;
        else
            D[x] = GetMax(new int[] { D[x - 1] - 1, R[x - 1][0] - 1, C[x -
1][0] - 1 });
    }

    public static void ProcessRow(int x)
    {
        if (x < Dim)
        {
            if (S1[x] == S2[x + 1])
                R[x][0] = R[x - 1][0] + 1;
            else
                R[x][0] = GetMax(new int[] { R[x - 1][0] - 1, D[x] - 1, R[x
- 1][1] - 1 });
            bool waited = false;
            for (int i = 1; i < R[x].Length; i++)
            {
                while (R[x - 1][i + 1] == -1)
                {
                    if (waited == false)
                    {
                        waited = true;
                        /* wait for data dependency */
                        lock (typeof(MultiDirectional))
                            NumWaits++;
                    }
                }
                if (S1[x] == S2[i + x + 1])
                    R[x][i] = R[x - 1][i] + 1;
                else
                    R[x][i] = GetMax(new int[] { R[x - 1][i] - 1, R[x][i -
1] - 1, R[x - 1][i + 1] - 1 });
            }
        }
    }

    public static void ProcessCol(int x)
    {
        if (x < Dim)
        {
            if (S1[x + 1] == S2[x])
                C[x][0] = C[x - 1][0] + 1;
            else
                C[x][0] = GetMax(new int[] { C[x - 1][0] - 1, D[x] - 1, C[x
- 1][1] - 1 });
            bool waited = false;
            for (int i = 1; i < C[x].Length; i++)
            {
                while (C[x - 1][i + 1] == -1)
                {
                    if (waited == false)
                    {
                        waited = true;
                        /* wait for data dependency */
                        lock (typeof(MultiDirectional))
                            NumWaits++;
```

```
                }
            }
            if (S1[i + x + 1] == S2[x])
                C[x][i] = C[x - 1][i] + 1;
            else
                C[x][i] = GetMax(new int[] { C[x - 1][i] - 1, C[x][i -
1] - 1, C[x - 1][i + 1] - 1 });
        }
    }
}

public static int GetMax(int[] values)
{
    int max = 0;
    foreach (int i in values)
    {
        if (i > max)
            max = i;
    }
    return max;
}

public static void InitM()
{
    D = new int[Dim + 1];
    D[0] = 0;
    for (int i = 1; i < Dim + 1; i++)
    {
        D[i] = -1;
    }
    R = new int[Dim][];
    C = new int[Dim][];
    for (int i = 0; i < Dim; i++)
    {
        R[i] = new int[Dim - i];
        C[i] = new int[Dim - i];
        for (int j = 0; j < Dim - i; j++)
        {
            R[i][j] = (i == 0 ? 0 : -1);
            C[i][j] = (i == 0 ? 0 : -1);
        }
    }

}

public static void InitS()
{
    Random random = new Random();
    S1 = new int[Dim + 1];
    S2 = new int[Dim + 1];
    // the first character in both strings will not be used for the
calculation
    for (int i = 0; i <= Dim; i++)
    {
        S1[i] = random.Next(4);
        S2[i] = random.Next(4);
    }
}
```

```
public static void DoIt()
{
    int Row, Col, Dir;
    while (true)
    {
        lock (typeof(MultiDirectional))
        {
            if (CurrentRow > Dim || CurrentCol > Dim) return;

            Row = CurrentRow;
            Col = CurrentCol;
            Dir = CurrentDir;
            if (CurrentDir == 0) // column direction
            {
                CurrentDir = 1;
                if (D[CurrentCol] == -1)
                    ProcessDiagonal(CurrentCol);
                CurrentCol++;
            }
            else // row direction
            {
                CurrentDir = 0;
                if (D[CurrentRow] == -1)
                    ProcessDiagonal(CurrentRow);
                CurrentRow++;
            }
        }
        if (Dir == 0)
        {
            ProcessCol(Col);
        }
        else
        {
            ProcessRow(Row);
        }
    }
}

static void Main(string[] args)
{
    // number of times the program will run
    int NumRuns = 4;

    // array to store the time spent for each run
    TimeSpan[] times = new TimeSpan[NumRuns];

    // array to store the times a thread had to wait
    int[] waits = new int[NumRuns];

    // number of tasks that will be created to solve the problem
    int NumTasks = 8;


    InitS();
    Stopwatch watch = Stopwatch.StartNew();

    for (int run = 0; run < NumRuns; run++)
```

```
        {
            // restart every variable before each run
            InitM();
            watch.Reset();
            CurrentRow = 1;
            CurrentCol = 1;
            CurrentDir = 0;
            NumWaits = 0;
            Task[] tasks = new Task[NumTasks];

            watch.Start();

            for (int i = 0; i < NumTasks; i++)
            {
                tasks[i] = new Task(() =>
                {
                    DoIt();
                });
                tasks[i].Start();
            }

            Task.WaitAll(tasks);

            watch.Stop();
            times[run] = (watch.Elapsed);
            waits[run] = NumWaits;
        }
        Console.WriteLine("BDF algorithm performance for " + NumTasks + "
treads in " + NumRuns + " program runs");
        Console.WriteLine("\t Run \t Time \t\t\t Wait");
        for (int i = 0; i < NumRuns; i++)
        {
            Console.WriteLine("\t " + (i + 1) + " \t " + times[i] + " \t "
+ waits[i]);
        }
        Console.ReadLine();
    }
}
```

Biography

Mr. Guillermo Delgago was born in 1983. He obtained his degree in Computer Science Engineering from the Antonio de Nebrija University, Madrid, Spain, in 2007.