

3-21-2017

# Supernode Transformation On Parallel Systems With Distributed Memory – An Analytical Approach

Yong Chen  
*Santa Clara University*

Follow this and additional works at: [http://scholarcommons.scu.edu/eng\\_phd\\_theses](http://scholarcommons.scu.edu/eng_phd_theses)



Part of the [Computer Engineering Commons](#)

---

## Recommended Citation

Chen, Yong, "Supernode Transformation On Parallel Systems With Distributed Memory – An Analytical Approach" (2017).  
*Engineering Ph.D. Theses*. 8.  
[http://scholarcommons.scu.edu/eng\\_phd\\_theses/8](http://scholarcommons.scu.edu/eng_phd_theses/8)

This Dissertation is brought to you for free and open access by the Student Scholarship at Scholar Commons. It has been accepted for inclusion in Engineering Ph.D. Theses by an authorized administrator of Scholar Commons. For more information, please contact [rscroggin@scu.edu](mailto:rscroggin@scu.edu).

# **Santa Clara University**

Department of Computer Engineering

Date: March 21, 2017

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY

**Yong Chen**

ENTITLED


## **Supernode Transformation On Parallel Systems With Distributed Memory**

### **– An Analytical Approach**


BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE


OF


**DOCTOR OF PHILOSOPHY IN COMPUTER ENGINEERING**

  
Thesis Advisor

  
Chairman of Department

  
Thesis Reader

  
Thesis Reader

  
Thesis Reader

  
Thesis Reader

**Supernode Transformation On Parallel Systems With  
Distributed Memory  
– An Analytical Approach**

**By**

**Yong Chen**

**Dissertation**

Submitted in Partial Fulfillment of the Requirements  
for the Degree of Doctor of Philosophy  
in Computer Engineering  
in the School of Engineering at  
Santa Clara University, 2017

Santa Clara, California

## **Acknowledgements**

This dissertation, and my entire Ph.D. study, could have not been completed without the tremendous support from my family. The continued encouragement from my wife and my parents made the completion of my study possible.

Dr. Shang, my advisor, has been patiently providing suggestions, direction and guidance for my study. Countless discussions, reviews ensured the progress of my study and the publication of the papers.

My doctoral committee members, Dr. Amer, Dr. Fang, Dr. Figueira, Dr. Tran, provided invaluable feedback that helped improve my study greatly.

# **Supernode Transformation On Parallel Systems With Distributed Memory**

## **– An Analytical Approach**

Yong Chen

Department of Computer Engineering  
Santa Clara University  
Santa Clara, California  
2017

### **ABSTRACT**

Supernode transformation, or tiling, is a technique that partitions algorithms to improve data locality and parallelism by balancing computation and inter-processor communication costs to achieve shortest execution or running time. It groups multiple iterations of nested loops into supernodes to be assigned to processors for processing in parallel. A supernode transformation can be described by supernode size and shape. This research focuses on supernode transformation on multi-processor architectures with distributed memory, including computer cluster systems and General Purpose Graphic Processing Units (GPGPUs). The research involves supernode scheduling, supernode mapping to processors, and the finding of the optimal supernode size, for achieving the

shortest total running time. The algorithms considered are two nested loops with regular data dependencies. The Longest Common Subsequence problem is used as an illustration. A novel mathematical model for the total running time is established as a function of the supernode size, algorithm parameters such as the problem size and the data dependence, the computation time of each loop iteration, architecture parameters such as the number of processors, and the communication cost. The optimal supernode size is derived from this closed form model. The model and the optimal supernode size provide better results than previous researches and are verified by simulations on multi-processor systems including computer cluster systems and GPGPUs.

# TABLE OF CONTENTS

<b>1. INTRODUCTION.....</b>	<b>4</b>
<b>2. RELATED WORK.....</b>	<b>7</b>
2.1 SUPERNODE SCHEDULING, SIZE AND SHAPE .....	8
2.2 THE POLYHEDRAL MODEL AND AFFINE TRANSFORMATION .....	10
2.3 SCHEDULING WITH FIXED PROCESSORS.....	11
2.4 FOCUS OF THIS RESEARCH.....	14
<b>3. MODELS AND TERMINOLOGY .....</b>	<b>15</b>
3.1 ALGORITHM MODEL.....	15
3.2 ARCHITECTURE MODELS.....	17
3.2.1 <i>Computer Cluster Systems</i> .....	18
3.2.2 <i>GPGPUs</i> .....	18
3.3 LINEAR SCHEDULING .....	20
3.3.1 <i>Example of the LCS problem scheduling on GPGPU</i> .....	22
3.4 SUPERNODE TRANSFORMATION ON MULTI-PROCESSOR SYSTEM WITH $P$ PROCESSORS .....	22
<b>4. TOTAL EXECUTION TIME WITH SUPERNODE TRANSFORMATION.....</b>	<b>27</b>
<b>5. SUPERNODE TRANSFORMATION ON COMPUTER CLUSTERS.....</b>	<b>32</b>
5.1 LEMMAS AND THEOREM.....	33
5.1.1 <i>Lemma A1</i> .....	33
5.1.2 <i>Lemma A2</i> .....	34
5.1.3 <i>Lemma A3</i> .....	34
5.1.4 <i>Theorem</i> .....	35
5.2 SIMULATION .....	35
<b>6. SUPERNODE TRANSFORMATION ON GPGPUS.....</b>	<b>40</b>
6.1 ANALYTICAL RESULTS .....	40
6.2 LEMMAS AND THEOREM.....	44
6.2.1 <i>Lemma B1</i> .....	45
6.2.2 <i>Lemma B2</i> .....	46
6.2.3 <i>Lemma B3</i> .....	46
6.2.4 <i>Theorem</i> .....	46
6.3 SIMULATION RESULTS.....	47
6.4 RESULT ANALYSIS .....	51
6.5 THE SELECTION OF GPGUP ARCHITECTURE MODEL PARAMETER $P$ .....	52
<b>7. CONCLUSION.....</b>	<b>54</b>
<b>8. FUTURE WORK .....</b>	<b>55</b>
<b>9. APPENDIX.....</b>	<b>56</b>
9.1 APPENDIX A: LEMMAS FOR SUPERNODE TRANSFORMATION ON CLUSTER SYSTEMS.....	56
9.1.1 <i>Lemma A1</i> .....	56
9.1.2 <i>Lemma A2</i> .....	60
9.1.3 <i>Lemma A3</i> .....	60
9.2 APPENDIX B: LEMMAS FOR SUPERNODE TRANSFORMATION ON GPGPUS .....	62

9.2.1	<i>Lemma B1</i> .....	62
9.2.2	<i>Lemma B2</i> .....	65
9.2.3	<i>Lemma B3</i> .....	66
9.3	APPENDIX C: SOURCE CODE OF SUPERNODE TRANSFORMATION ON COMPUTER CLUSTER SYSTEMS .....	68
9.3.1	<i>Code Running on the Master Computer</i> .....	68
9.3.2	<i>Code Running on the Computing Nodes</i> .....	89
9.3.3	<i>Include File</i> .....	94
9.4	APPENDIX D: CODE ON GPGPUS .....	99
<b>10.</b>	<b>REFERENCES .....</b>	<b>124</b>



## List of Figures

Figure 1: A Uniform Dependence Algorithm.	9
Figure 2: Column-wise cyclic distribution of rectangular tiles on limited processors.	13
Figure 3: A typical GPGPU memory hierarchy.	20
Figure 4: The ten wavefronts of the linear schedule vector $[1,1]$ .	21
Figure 5: Two dimensional uniform dependence algorithm iteration space.	24
Figure 6: Iteration space after supernode transformation.	25
Figure 7: Total execution times for different values of $(m,n)$ on cluster.	38
Figure 8: A LCS Supernode of size $w \times h$ and its dependent nodes.	42
Figure 9: Total execution times of the LCS problem on GPGPU.	49
Figure 10: On a cluster system, the total execution time for a two-dimensional uniform dependence algorithm $T$ is convex.	58

# 1. Introduction

Supernode partitioning, or tiling, is a transformation technique that groups a number of iterations in a nested loop in order to improve data locality and parallelism, thus ultimately improving execution performance on multi-processor systems. This paper addresses the problem of applying supernode transformation on multi-processor systems with distributed memory, including computer clustering system and the General Purpose Graphic Processing Units (GPGPUs), especially on finding the optimal supernode size to minimize the total running time.

In a parallel system with multiple processors, the total running time consists of two parts: the computation time and the communication time. An algorithm can be partitioned into supernodes or tiles where each supernode is assigned to one processor for parallel execution. If the supernode is too small (or too large), the communication time (or computation time) will dominate and the total running time is not minimized due to non-optimal data locality and parallelism. Finding the optimal supernode size to achieve optimized locality to minimize the total running time is critical in supernode transformation.

The algorithms considered in this paper are nested loops with regular data dependencies or uniform dependencies [5]. Such an algorithm can be described by its iteration index space consisting of all iteration index vectors of the loop nest, and a dependence matrix, consisting of all uniform dependence vectors as its columns. The Longest Common Subsequence Problem, the LCS problem [2], which has found wide applications, such as in bioinformatics or in computer science, is used to illustrate how to use supernode transformations to minimize the total running time.

The multi-processor architectures considered in this paper are computer cluster systems and the GPGPUs. A computer cluster consists of a set of loosely or tightly connected computers that work together to form a single computational unit. The computers of a cluster are usually connected to each other through fast local area networks. The clusters are formed to improve performance and availability over that of a single computer while still being much more cost-effective than single computers.

Another important computer architecture considered is the GPGPU system. Inside each GPGPU, there are multiple streaming multi-processors (SMs), each SM contains multiple cores for concurrent operations and these cores share a cache on the same chip. Then the SMs are connected at high level and share a global memory of a much larger size but with a much slower access speed.

The basic approach in this paper is as follows. Given an algorithm with two nested loops and a multi-processor distributed memory architecture with a fixed number of processors or GPGPU SMs, model for the total running time is established. This total running time is expressed as a function of the supernode size, algorithm parameters, such as the problem size and data dependence, the computation time of each loop iteration, architecture parameters, such as the number of processors/GPGPU SMs, and the communication cost. This estimated expression of the total running time is a convex function with two variables. By working on the derivatives, the optimal supernode size can be estimated.

The contributions of this research are as follow. For algorithms with two nested loops and regular dependences, a novel mathematical model for the total running time on multi-processor distributed memory systems is established, the model is closed form by

dividing solution space into three sub spaces. The total running time is expressed as a function of the supernode size, algorithm parameters such as the problem size and data dependence, the computation time of each loop iteration, architecture parameters such as the number of the computing nodes in cluster system, or the number of GPGPU blocks, and the communication cost. The optimal supernode size is obtained based on this model. This optimal supernode size leads to significant performance improvement than without using optimal supernode size, due to optimized locality, it leads to much better results than previous research.

The rest of this research report is organized as following. Section 2 summarizes the related work that has been done in the area of supernode transformation. Section 3 presents the algorithm and architecture models. Section 4 shows how the mathematical models of total running time for supernode transformation are established. Section 5 discusses how to obtain the optimal supernode size for the cluster system. Section 6 presents analytical and simulation results for the GPGPU architecture. Section 8 provides future work direction and section 9 contains additional information as appendix.

## 2. Related Work

Irigoin and Troilet [4] proposed the supernode partitioning technique for multiprocessors in 1988 as a new restructuring method. The idea was to combine multiple loop iterations of perfectly nested DO loops in order to provide vector statements, parallel tasks, and data reference locality. They first defined hyperplane partitioning, then generalized it to partitioning with multiple hyperplanes. They gave conditions for valid partitioning, and other reasonable constraints on supernodes to ensure the supernodes were: 1) atomic, each tile is a unit of computation, all synchronization points are beginnings and ends of tiles; 2) identical, this is to allow for automatic code generation; 3) bounded, the number of points inside a tile to be bounded by a constant independent of the domain size. They only briefly discussed the choice of parameters of supernode partitioning. They noted that supercomputer architectures were too intricate to derive analytical expression for the partitioned program execution time. They listed a number of possible optimization goals, noting that these goals were often conflicting.

Since then, researchers have studied supernode transformation in different contexts. The research has been mostly focused on the model of the supernode transformation and how to use the model to construct the optimal supernode transformation. In general, supernode transformations can be described by the size, the shape and the relative ratio of the sides of each supernode. The communication cost can be modeled either as a constant where the start-up time dominates or as a linear function of the message size. To construct an optimal supernode transformation for a general algorithm with any convex index set, any dependence structure, and a general architecture is difficult or sometimes impossible. So researchers tried special cases. Also, the optimal solution is quite different

for the case where the number of processor cores is unlimited and the case where that number is given and fixed. Following paragraphs summarize the research development in supernode transformation.

## 2.1 Supernode Scheduling, Size and Shape

Scheduling is one of the challenging problems in the parallel computing, hence lots of research on it. But even without supernode transformation, finding the optimal scheduling is hard. Sinharoy and Szymanski in [9] presented efficient algorithms for finding the optimum wavefront and for partitioning the optimum wavefront into sections to be assigned to arbitrary large array of processors. Their algorithms can be used for one or higher dimensional processor arrays. But these algorithms are complex even for a two-dimension array computation.

Shang and Fortes in [5] addressed the problem of identifying optimal linear schedules for uniform dependence algorithms to minimize their execution time. An algorithm can be thought of as a set of indexed computations. A uniform dependence algorithm is defined as an algorithm whose dependence vectors are uniform, where the data dependence vector is the difference of indexes where a variable is used and where that variable is generated. [Figure 1](#) shows an example of a two-dimensional uniform dependence algorithm where each node is an indexed computation, the dependence

vector is  $D = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix}$ , the dashed lines are wavefronts on which computations are

independent of each other. They proposed procedures to find optimal linear schedules based on the mathematical solution of a nonlinear optimization problem.

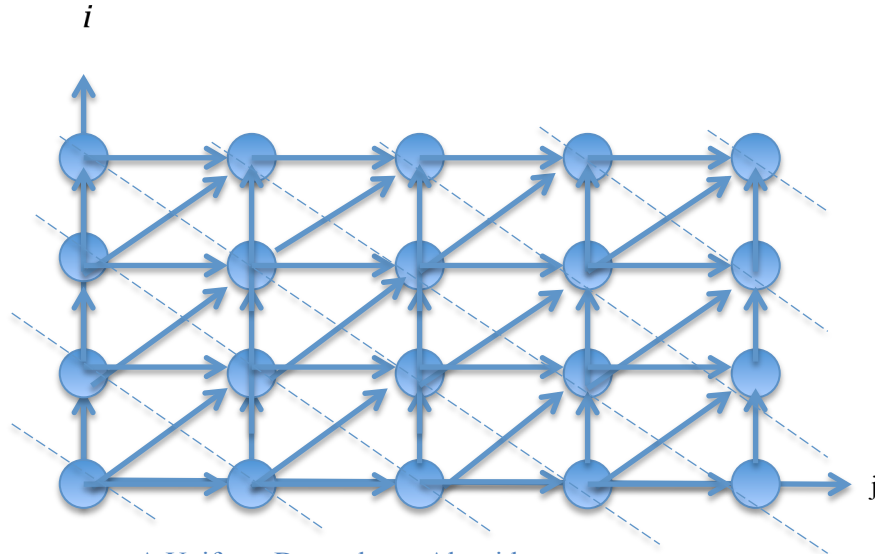


Figure 1: A Uniform Dependence Algorithm

Hodzic and Shang in [1,11] presented an execution of tiles for uniform dependence algorithms using linear scheduling under the condition of largely enough processors available, where each processor executes all tiles along a specific dimension, with non-overlapping communication and computation phases. They discussed optimal supernode size and shape with a few interesting findings. First, they found the optimal supernode size is the ratio of the communication cost over the computation cost of one iteration of the original loop. Secondly, the supernode shape is a function of the cone spanned by all dependence vectors. Thirdly, the ratio of the lengths of supernode sides should be such that the index set after the supernode transformation should have equal side lengths. Based on their method, the optimal supernode transformation can be found for  $n$  dimensional algorithms with at most  $n$  dependence vectors and two-dimensional algorithms with any number of dependence vectors.

In [27] Goumas et al. tried to improve the overall execution time of nested FOR-loops by using a modified linear scheduling, and by mitigating communication overhead by efficiently overlapping the communication and computation phases. They used Direct Memory Access (DMA) engines and network interfaces (NICs) that can work in parallel with the CPUs.

While Goumas's method applied to a cluster of single CPUs, Athanasaki et al. in [14] extended it to a cluster of symmetric multiprocessors (SMP nodes). They grouped together neighboring tiles along a hyperplane and these tiles are concurrently executed by the CPUs of the same SMP node, taking advantage of the fact that there is no need for tile synchronization and communication between intra-node CPUs.

## 2.2 The Polyhedral model and Affine Transformation

The Polyhedral Model is a mathematical framework for affine loop nest analysis and optimization [30,31,32]. It treats an instance of a statement in the loop as an integer point or lattice point in the space called polyhedron. The affine transformations on polytope, based on Linear Algebra and Integer Linear Programming, cause a sequence of complex loop transformations aiming for the improvements such as parallelism and data locality. Supernode transformation or tiling, as one of the key transformations, fits in this model well: it improves data locality by grouping points in the iteration space into supernodes that can be loaded in cache of processors for easy and fast reuse. It also improves parallelism by partitioning the iteration space into independent supernodes that are executed concurrently and atomically on processors, thus reducing communication. In



[37,38], Lim et al. proposed an algorithm to find the optimal affine partition that maximizes the degree of parallelism with the minimum communication in programs with arbitrary loop nests and affine data accesses, and used this algorithm for blocking to improve data locality. In [39] Ahmed et al. presented an approach for synthesizing transformations to enhance locality in imperfectly nested loop via affine embedding functions. In [33] Bondhugula et al. presented an end-to-end automatic integer linear optimization framework that finds good ways of supernode transformation in polyhedral model for parallelism and locality using affine transformations. The key part is to create an affine form cost function that represents the number of hyperplanes the dependence traverses along the hyperplane normal. This cost function is a measurement of reuse distance and also the communication cost if the hyperplane is used to generate supernodes for parallelization and used as a processor space dimension. By minimizing this cost function, they found supernode (or tiling) hyperplanes that not only minimize reuse distances and improve data locality, but also minimize communication volume thus improving parallelism.

## 2.3 Scheduling with Fixed Processors

While many researches assumed unlimited number of processors or SMP nodes available [1,11,14], this assumption does not hold true in practice. The servers nowadays have fixed or limited number of cores, GPGPUs have limited number of SMs. For this reason, researchers studied supernode transformation with fixed processors.

Ohta et al. in [26] discussed the tile scheduling with limited number of physical

processors. When there are  $P$  processors available and interconnected as a ring, a computation domain of two-dimensional rectangle of size  $M \times N$  can be partitioned into rectangular  $w \times h$  tiles with tiles' edges parallel to the axes, when the dependence vector after the tile transformation is  $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ , the best mapping of tiles is as follows: tiles are assigned via column-wise cyclic distribution, that is, tile  $(i, j)$  is allocated to processor  $j \bmod P$ . The execution starts with processor  $P_0$  at tile  $(0,0)$ , after the computation, the result is sent to the adjacent processor  $P_1$ , then concurrently  $P_0$  computes tile  $(0,1)$  and  $P_1$  computes tile  $(1,0)$ , and this process continues, until  $P_0$  finishes all the tiles on column 0, then it moves to column  $P$  and continues computation. The column-wise processor assignment is shown in Figure 2.

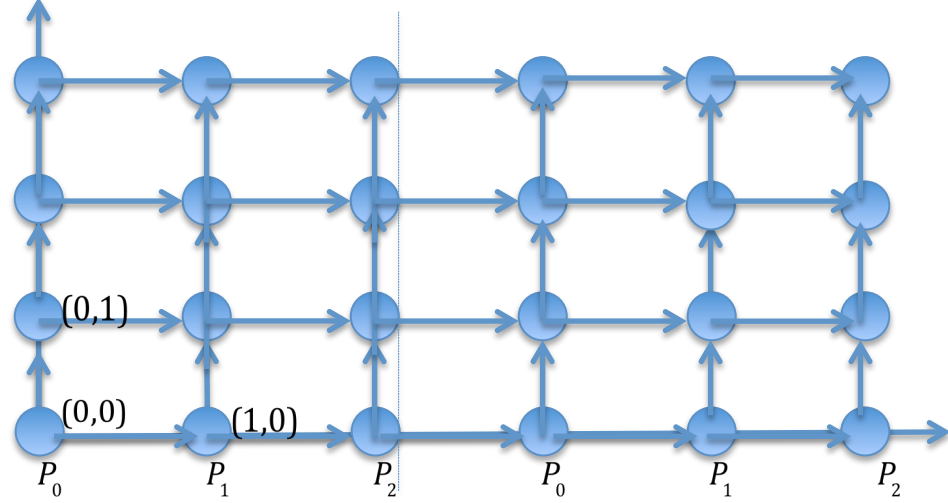
Based on this scheduling and mapping, Ohta et al. further derived the optimal tile size as follows, assuming non-overlapping computation and communication phases:

$$(w, h) = \left( \frac{M}{P}, \sqrt{\frac{Na}{Mt}} \right), T_{opt} = \left( \sqrt{N \left( \frac{Mt}{P} + b \right)} + \sqrt{aP} \right)^2 \quad (1)$$

where  $P$  denotes number of processors,  $t$  denotes the computation time per iteration, and  $a$  is the communication startup time and  $b$  being the coefficient of message size, linear to  $h$ .

Apparently, the cyclic column-wise assignment makes sense due to its load-balancing tile distribution. In [15], Calland et al. demonstrated cyclic column-wise assignment is the best solutions among all possible distributions of tiles to physical processors for a two-dimensional computation domain, with the condition that the computation cost of a tile is greater than its communication cost. They further improved the scheduling and

execution time by overlapping communication and computation phases.



**Figure 2: Column-wise cyclic distribution of rectangular tiles on limited physical processors. Assuming  $P=3$ . Processor  $P_0$  finishes column 0, then moves to column 3.**

But the cyclic column-wise assignment not only has the restriction that the computation cost of a tile needs be greater than its communication cost, it also does not provide the best solution in case of heterogeneous computing platforms. Boulet et al. in [17] handled this problem by aiming at load-balancing the work while not introducing idle time. They presented efficient scheduling and mapping strategies that are asymptotically optimal.

In [16] Athanasaki et al. further proposed four different methods for scheduling tiled iteration spaces onto a clustered system with a fixed number of SMP nodes, namely the cyclic, the mirror, the cluster and the retiling scheduling.

## 2.4 Focus of This Research

The cyclic column-wise allocation of processors is regarded as the best scheduling solution when there are limited processors, due to its load-balancing nature. But as mentioned earlier, it has one restriction: the computation cost of a supernode has to be greater than its communication cost. This may not always be true since CPU performance has been increasing dramatically, especially with the advent of GPGPUs, which has very powerful computing capability, the computation and communication ratio may become very small. Let's take a look at a hypothetical case: assume a computation domain of  $M \times N = 60 \times 60$ , available processors  $P=6$ . Let  $a=400$ ,  $b=0$  and  $t=1$  in (1). Then

based on (1), the optimal  $(w,h) = (\frac{M}{P}, \sqrt{\frac{Na}{Mt}}) = (10, 20)$ , and  $T_{opt} = (\sqrt{N(\frac{Mt}{P} + b)} + \sqrt{aP})^2 = 5400$ .

But if the scheduling is wavefront-wise, and if  $(w,h) = (12,20)$ , the domain size becomes  $5 \times 3$  after transformation. Giving that the longest wavefront is 3, each wavefront can be processed within one  $T_{tile}$  since there are more processors than supernodes on any wavefront, note  $T_{tile} = T_{comp} + T_{comm}$ ,  $T_{comp} = wht$ ,  $T_{comm} = a + bh$ . The total execution time  $T = \text{total\_num\_of\_wavefronts} \times T_{tile} = (5+3-1) \times (12 \times 20 \times 1 + 400) = 4480$ . This result is less and better than the  $T_{opt} = 5400$  obtained via cyclic column-wise scheduling. The reason for this better result is the enhanced data locality.

The goal of this research is to apply supernode transformation and linear scheduling to multi-processor system architectures, including computer cluster systems and GPGPUs, to find the optimal solution that is suitable no matter what the communication and computation ratio is. The research work involves following areas: linear scheduling, tiles to computing nodes or GPGPU blocks mapping, total running time model, optimal supernode size, applied algorithms, cluster system and GPGPU architectures and simulations. The objective of this research is to minimize the total running time. The total running time consists of communication time between supernodes and computation time within supernodes, representing parallelism and data locality respectively. By obtaining the time optimal solution, we not only improve the data locality and parallelism, but also optimize locality for an optimal balance between data locality and parallelism thus achieving optimal result.

### **3. Models and Terminology**

Models for applications or algorithms, parallel computer systems and the mapping from the application to the target parallel system are presented in this section. Some concepts and terms that are necessary to understand this paper are introduced. An example is presented to illustrate different concepts and ideas throughout the paper.

#### **3.1 Algorithm Model**

An algorithm is modeled as a set of indexed computations, and a set of data dependence. An indexed computation corresponds to a loop iteration in the algorithm. The *dimension of the algorithm*  $s$  corresponds to the number of the nested loops in the algorithm. A data dependence is established if one computation uses data generated by another computation, and is represented by a dependence vector that is the difference of two indexes of the computations. In this paper, only algorithms with regular dependence are considered, and such algorithms are called *uniform dependence algorithms* where the dependence vectors are constant. Uniform dependence algorithms can be described by two parameters  $(J, D)$ :  $J$  is the set of all iteration index vectors,  $D$  is a matrix of  $s \times q$  for a  $s$  dimensional algorithm with  $q$  dependences, and each column is a dependence vector. Detailed description of uniform dependence algorithms can be found in [5].

The two-sequence LCS problem is used to illustrate the algorithm model and is defined as following [2]: given two input sequences:  $X = (x_1, x_2, \dots, x_M)$  and  $Y = (y_1, y_2, \dots, y_N)$  with  $M$  and  $N$  being the sizes of each sequence, the LCS problem is to find the length of the longest common sequence, denoted as  $LCS(X_M, Y_N)$ . For example,  $LCS(\underline{A}BCBD\underline{A}B, \underline{B}DC\underline{A}BA) = 4$  where the longest common subsequence is underlined. A LCS recursion is presented below where  $LCS(X_i, Y_j)$  is the length of the longest common sub sequence of two sub sequences  $X_1, \dots, X_i$  and  $Y_1, \dots, Y_j$ .

$$LCS(X_i, Y_j) = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ LCS(X_{i-1}, Y_{j-1}) + 1 & \text{if } x_i = y_j \\ \max(LCS(X_i, Y_{j-1}), LCS(X_{i-1}, Y_j)) & \text{if } x_i \neq y_j \end{cases}$$

A dynamic programming algorithm calculating LCS from [10] is shown in (2). This algorithm has two nested loops. Therefore, the LCS algorithm is two dimensional or  $s=2$ . Each index vector is a two dimensional vector  $\begin{bmatrix} i \\ j \end{bmatrix}$  corresponding to an iteration  $(i, j)$  and  $J = \left\{ \begin{bmatrix} i \\ j \end{bmatrix}, 1 \leq i \leq M, 1 \leq j \leq N \right\}$ . There are three data dependence vectors in the LCS and  $D = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix}$  where each column represents one dependence vector. The dependence graph is shown in [Figure 1](#) where each point represents an iteration and an arrow represents a data dependence between two iterations of the algorithm in (2). Because the LCS has uniform dependence, the dependence graph is regular. The execution time to process each iteration  $(i, j)$  is denoted as  $t_c$ . Thus the LCS is modeled by parameters  $(J, D, M, N, t_c)$ .

```

for (i=0; i≤M; i++)
  for (j=0; j≤N; j++){
    if (i==0||j==0)
      c[i][j]=0;
    else if (x[i]==y[j])
      c[i][j]=c[i-1][j-1]+1;
    else
      c[i][j]=max(c[i][j-1],c[i-1][j]);
  }

```

(2)

### 3.2 Architecture Models

In this research, two important multi-processor distributed memory computer architectures are considered, they are cluster systems and the GPGPUs.

### 3.2.1 Computer Cluster Systems

First, computer cluster architecture is considered. In a computer cluster system, there are two types of computers: a master computer and computing computers (also called computing nodes). The master computer breaks the algorithm into small tasks and sends them to computing nodes to process in parallel. The master computer manages the cluster and coordinates computing nodes such as synchronization due to dependence considerations.

The number of the computing nodes is modeled as the number of processors, denoted as  $P$ , the communication time a computing node takes to receive dependent data from other computing nodes is denoted as  $t_r$ , and the time a computing node takes to send resultant data to other nodes is denoted as  $t_s$ . Therefore, a computer cluster system is modeled by parameters  $(P, t_r, t_s)$ .

### 3.2.2 GPGPUs

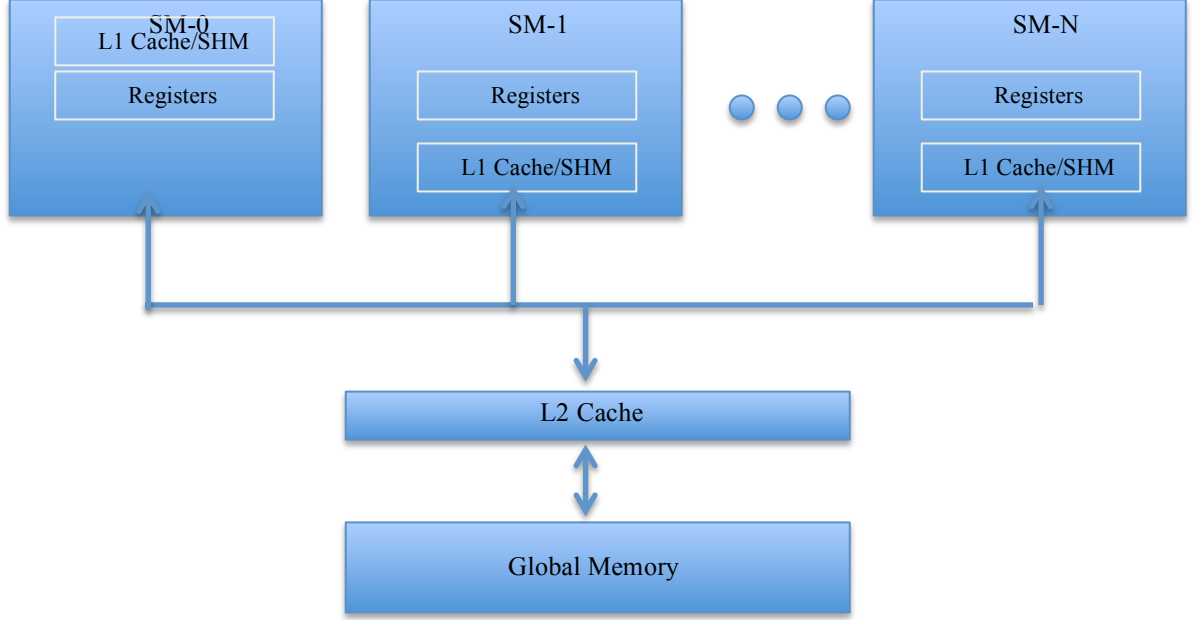
Another architecture considered is GPGPUs. GPGPUs employ Single Instruction, Multiple Thread (SIMT) parallel execution model, where multiple independent threads execute concurrently using a single instruction, this provides excellent concurrent processing capability. In 2006, Nvidia developed CUDA programming model to promote



the use of its GPGPUs. At the core of CUDA programming model are three key abstractions: a hierarchy of thread groups, a memory hierarchy, and barrier synchronization. A CUDA program launches a grid of blocks, the blocks reside and run on GPU's streaming multiprocessors (SMs), and each block contains a group of concurrent threads. A block can be thought of as a cluster of threads that run cooperatively while still independently. All threads run the same code with different data, differentiated by block id and thread id. GPGPUs have a tiered memory hierarchy shown in [Figure 3](#). Each SM has a large and unified register file and a L1 cache, to be used by threads privately, and a local memory of low latency called shared memory that is accessible and shared by all threads within the block running on this SM. This fast shared memory provides great data locality improvement opportunity for algorithms. A larger L2 cache is provided and shared among all SMs to service all load and store from/to global memory. The global memory is a very large memory accessible by all threads but has high latency. The goal of this research is to obtain optimal execution time for algorithms by optimizing data locality via optimal use of GPGPU parallel processing capability and fast on-chip shared memory.

Moving data between threads in a block on a SM and the global memory incurs significant cost, due to the high latency of global memory. To improve performance, in addition to the L2 cache, GPGPUs use coalescing technique for memory access to the global memory. The global memory accesses by threads of a block are coalesced into a single memory transaction when the words accessed by threads lie in the same segment, i.e., within a certain memory space with contiguous addresses. The memory segment size

$s$  is 32 bytes if threads access 1-byte words, or 64 bytes when accessing 2-byte words and 128 bytes when accessing 4-byte or 8-byte words.



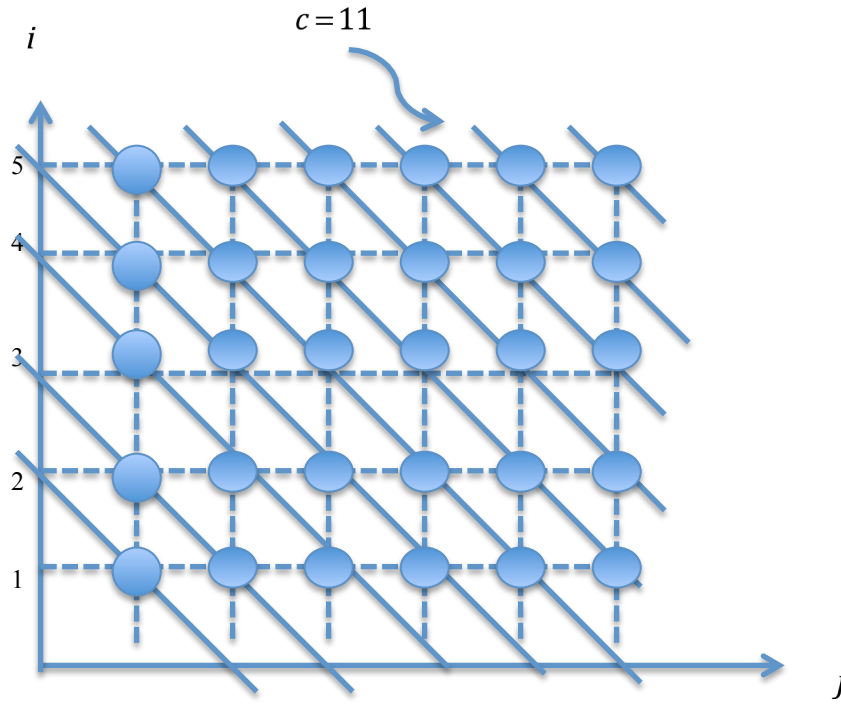
**Figure 3: A typical GPGPU memory hierarchy.** It contains a global memory and L2 cache shared by all blocks, a L1 cache private to threads, and a shared memory SHM shared by threads within the blocks.

Let  $P$  be the size of the grid, that is, the number of blocks/clusters, and  $t_s$  and  $t_r$  be the times each block spends on saving data to and retrieving data from the global memory, respectively. Therefore, a GPGPU can be modeled by parameters  $(P, t_s, t_r, s)$ .

### 3.3 Linear Scheduling

A linear schedule is a mapping from the multi-dimensional iteration vectors in the iteration space  $J$  into a one-dimensional execution time space. This mapping is expressed as a linear function  $f$  that involves a multiplication of a row vector  $\Pi$ , called

*linear schedule vector*, by each and every column vector in the iteration space. In other words, an iteration with index vector  $j$  is assigned to execute at time  $\Pi j$ . A linear schedule has to respect data dependences. That is, if an iteration depends on another iteration, a feasible linear schedule should schedule the latter iteration to execute before the former one. As described in [5], a linear schedule vector  $\Pi$  is *feasible* if  $\Pi d_j > 0, j = 1, \dots, q$ . Another concept associated with a linear schedule is its wavefronts in the iteration space. All iterations that are assigned to the same execution time form a *wavefront*. More description of linear schedule can be found in [5].



**Figure 4: The ten wavefronts of the linear schedule vector  $[1,1]$  for the two-dimensional uniform dependency algorithm in Figure 1, with  $N=5$ ,  $M=6$  and  $c=2, \dots, 11$ .**

For the two dimensional iteration space in Figure 1, a feasible linear schedule vector is  $[1,1]$ , and the corresponding feasible linear schedule is  $f([i,j]^t) = [1,1][i,j]^t = i + j$ . The

entire iteration space is partitioned into wavefronts  $\left\{ \begin{bmatrix} i \\ j \end{bmatrix} : i+j=c \right\}$  where  $c=2,\dots,N+M$ . Note all iterations  $(i,j)$  on the same wavefront are independent and can be executed at the same time in parallel, provided there are enough computing nodes available. **Figure 4** shows a two-dimensional iteration space with  $N=5$  and  $M=6$ , it is partitioned by ten wavefronts with  $c=2,\dots,11$ .

### 3.3.1 Example of the LCS problem scheduling on GPGPU

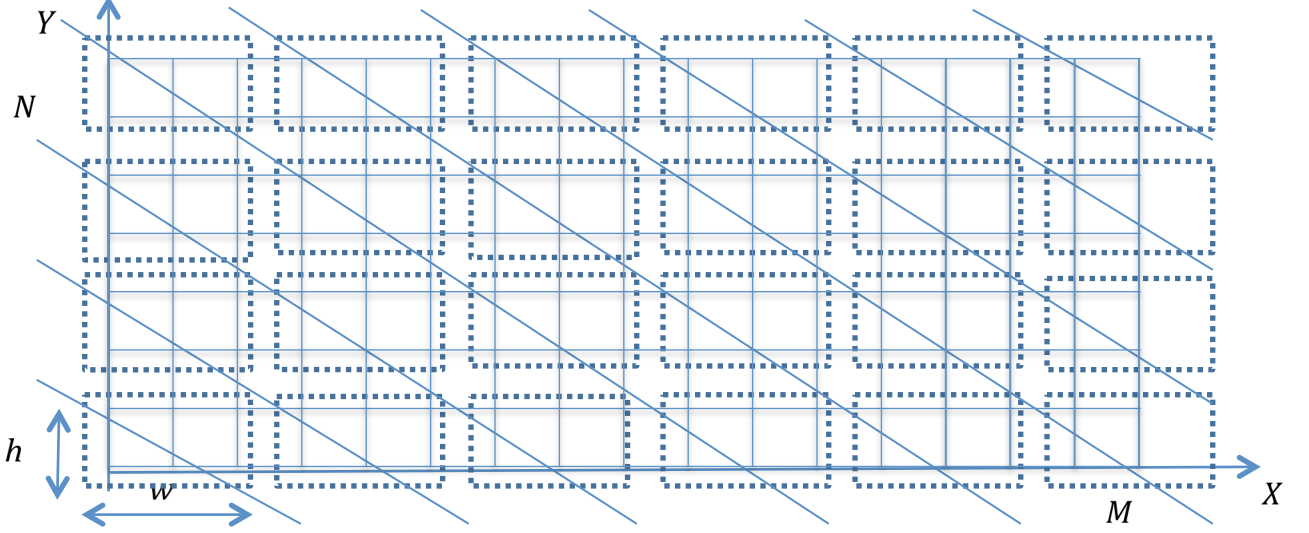
The linear schedule  $f([i,j]^t)=[1,1][i,j]^t=i+j$  can be applied to LCS problem to ensure feasible and maximum parallelism, it is the traditional way of exploiting the parallelism in diagonal direction [36]. In [35] J. Yang et al. proposed an efficient approach to schedule iterations row by row by first changing the data dependency in the score table used by the dynamic programming algorithms for higher degrees of parallelism to take advantage of GPGPU's parallel processing capability. This approach is shown to be three times faster than the traditional way in [36].

## 3.4 Supernode transformation on multi-processor system with $P$ processors

To see the motivation of supernode transformation, let's examine the different executions of the LCS problem. The first is the sequential execution by a single processor and the total execution time is  $MNt_c$ , because there are  $MN$  iterations and each iteration takes  $t_c$  time. In the parallel processing, all the iterations on the same wavefront are executed in parallel assuming there are enough processors. There are two phases: computation phase when the processor calculates an iteration in time  $t_c$  and communication phase when the processor sends and receives data from other processors with communication time  $t_{comm} = t_r + t_s$ . Hence the execution time of each wavefront is  $t_c + t_{comm}$ . There are  $M+N-1$  wavefronts, so the total execution time is  $(t_c + t_{comm})(M+N-1)$ . If  $t_c = 1$ ,  $t_{comm} = 100$ ,  $M = 100$ ,  $N = 1000$ , then the sequential execution time is  $MNt_c = 10^5$  and the parallel execution time is  $(t_c + t_{comm})(M+N-1) = 101 * 1099 > 10^5$ . This means the parallel execution time is even worse than the sequential one.

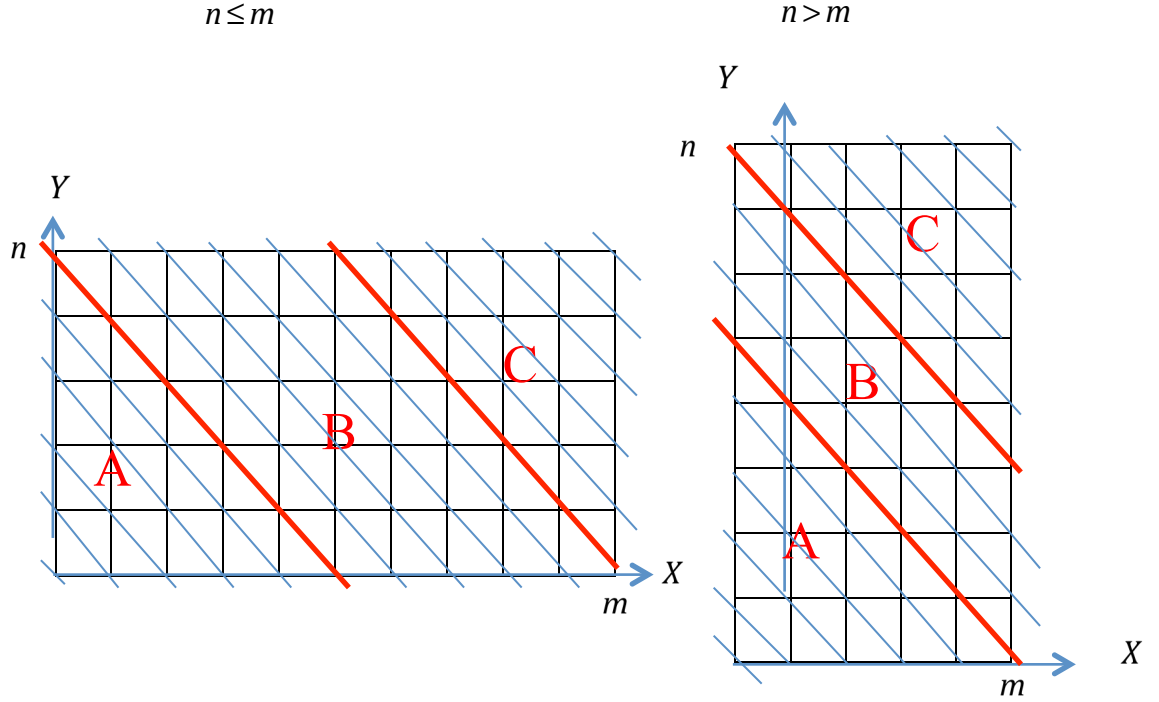
Supernode transformation is to optimize data locality by addressing the problem of unbalanced computation and communication costs. Instead of assigning one iteration to a processor, a set of neighboring iterations are grouped as a supernode and assigned to a processor. This way, the iterations in the same supernode are processed faster on the same processor, and the number of wavefronts is reduced and the system will spend less time on communication so the total execution time is minimized. For example, for the two dimensional iteration space in Figure 5 with each intersection being an iteration, one possible supernode transformation is  $w=3$  and  $h=2$ , thus six iterations are grouped to

form a supernode. The iteration space after the supernode transformation is shown in Figure 6. There are two possible cases:  $n \leq m$  and  $n > m$ .



**Figure 5: Two dimensional uniform dependence algorithm iteration space, each intersection is an iteration. Each dotted-line rectangle of  $w \times h$  is a supernode, or a tile.**

A supernode can be described by two parameters: the size and the shape. In [11], it is proven that the rectangular is the best shape for the two dimensional uniform dependence algorithms because it is the minimal parallelogram covering the cone of all dependence vectors. Therefore, for the two dimensional uniform dependence algorithms, the supernode shape and size can be defined by a rectangular of width  $w$  and height  $h$ . Thus their supernode transformation can be modeled by  $(f, w, h)$  where  $f$  is the linear schedule and  $w$  and  $h$  are the supernode sizes in  $X$  and  $Y$  directions.



**Figure 6:** Iteration space after supernode transformation, each intersection is a supernode. There are two cases:  $n < m$  and  $n \geq m$ .

This research is to apply supernode transformation  $(f, w, h)$  to two dimensional uniform dependence algorithms  $(J, D)$  on cluster system  $(P, t_r, t_s)$  or GPGPUS  $(P, t_s, t_r, s)$  for time optimal execution enabled by optimal locality. For this purpose four key parameters are formally introduced here for later modeling use:

$t_{sn}$  - the time used to process one supernode, including both computation and communication time by one computing node. It is a function of  $(w, h, t_c, t_s, t_r)$ .

$T$  - total execution or running time, our goal is to minimize this value.

$m$  - the problem size in  $X$  direction after transformation, so  $m = \frac{M}{w}$ , and

$m \in [1, M]$ .

$n$  - the problem size in  $Y$  direction after transformation, so  $n = \frac{N}{h}$ , and

$$n \in [1, N].$$

Without losing generality, this research focuses on the case where  $n \geq m$ . The iteration space after supernode transformation can be divided into three regions A, B and C, as shown in Figure 6. When  $n \geq m$ , in region A, there are  $m$  wavefronts and the number of supernodes in wavefronts increases from one to  $m$ . In region B, there are  $n - m - 1$  wavefronts and the number of supernodes in each wavefront is a constant  $m$ . In region C, there are  $m$  wavefronts and the number of supernodes in wavefronts decreases from  $m$  to one.



#### 4. Total Execution Time With Supernode Transformation

In this section, an execution model for a given two dimensional uniform dependence algorithm  $(J, D)$  on a multi-processor system such as a cluster system  $(P, t_r, t_s)$  or GPGPU  $(P, t_s, t_r, s)$  is established. Based on this model, a novel closed form expression of the total execution time  $T$  is derived. This expression is used in the later sections to guide the selection of the optimal supernode transformations.

To respect the data dependence, the execution has to start at the first wavefront with only one supernode at the lower left corner in region A in Figure 6, and moves towards the upper right corner in region C, the execution of wavefront  $c$  can not start until all the wavefronts  $1, \dots, c-1$  are executed. The mapping of the supernodes on a wavefront to the processors (computing nodes in cluster, or GPGPU blocks) is as follows: assuming the current wavefront  $c$  has  $\mu$  supernodes, then the entire wavefront  $c$  is divided into  $\left\lceil \frac{\mu}{P} \right\rceil$  contiguous sections, each section has  $P$  supernodes, with the exception that the last section may have fewer than  $P$  supernodes. The sections are executed sequentially and each section is processed by  $P$  processors in parallel, with one processor handling one supernode. So some processors may be idle when processing the last section, since there may be fewer than  $P$  supernodes. When a processor processes a supernode, it first takes  $t_r$  time to retrieve dependent data in earlier wavefronts such as  $c-1$  or  $c-2$  from other computing nodes of the cluster, or from main memory of GPGPU. Then the execution

enters into computation phase by computing  $w \cdot h$  iterations in supernode. Finally it takes  $t_s$  time to send the computation results to other computing nodes in the cluster, or back to the main memory of the GPGPU, for computing wavefronts such as  $c+1$ . Thus the execution time of one wavefront is  $t_{sn} \cdot \left\lceil \frac{\mu}{P} \right\rceil$  where  $t_{sn}$  is the time for a computing node of the cluster, or a block of the GPGPU, to process one supernode. Here in both data retrieving and sending phases, communication startup time is assumed to be dominant, so communication cost is fixed.

Let  $T_A$  be the total execution time for all wavefronts in region A,  $T_B$  for region B and  $T_C$  for region C. So  $T = T_A + T_B + T_C$ , and since  $T_A = T_C$ ,  $T = 2T_A + T_B$ .

For region A, the execution starts from lower left corner where the number of supernodes is one, then it moves towards the upper right direction until it reaches the wavefront with number of supernodes  $m$ . Thus:  $T_A = \sum_{c=1}^m \left\lceil \frac{c}{P} \right\rceil t_{sn} = t_{sn} \sum_{c=1}^m \left\lceil \frac{c}{P} \right\rceil$ .

Let  $m = kP + r$  where  $k \in \mathbb{Z}^+$ ,  $\mathbb{Z}^+$  being the set of all positive integers,  $r$  being the remainder,  $r \in [0, P-1]$ . Let  $v = k+1 = \left\lceil \frac{m}{P} \right\rceil$ . In region A, there are  $m$  wavefronts, for each wavefront  $c$  in the first set of  $P$  wavefronts, where  $c \in [1, P]$ ,  $\left\lceil \frac{c}{P} \right\rceil = 1$ . For each wavefront  $c$  in the second set of  $P$  wavefronts, where  $c \in [P+1, 2P]$ ,  $\left\lceil \frac{c}{P} \right\rceil = 2$ , and so

on. For wavefronts  $c$  where  $c \in [(k-1)P+1, kP]$ ,  $\left\lceil \frac{c}{P} \right\rceil = k$ . For the last  $r$  wavefronts

where  $c \in [kP+1, kP+r]$ ,  $\left\lceil \frac{c}{P} \right\rceil = k+1 = v$ . Thus:

$$\begin{aligned} T_A &= t_{sn} (\underbrace{1+1+\dots+1}_P + \underbrace{2+2+\dots+2}_P + \dots + \underbrace{k+k+\dots+k}_P + \underbrace{v+v+\dots+v}_r) \\ &= t_{sn} \left( \frac{m-r}{P} + 1 \right) \left( \frac{m+r}{2} \right) \end{aligned} \quad (3)$$

For  $T_B$ , the total number of wavefronts in region B is  $n-m-1$ . Each wavefront has  $m$  supernodes and the number of execution sections is  $\left\lceil \frac{m}{P} \right\rceil$ . So:

$$T_B = t_{sn} (n-m-1) \left\lceil \frac{m}{P} \right\rceil \quad (4)$$

therefore:

$$T = 2T_A + T_B = t_{sn} \left( \frac{m-r}{P} + 1 \right) (m+r) + t_{sn} (n-m-1) \left\lceil \frac{m}{P} \right\rceil \quad (5)$$

To get a closed form for  $T$ , let's consider the following three cases:

Case 1:  $m \leq P$ . This happens when  $k=1$  and  $r=0$ , or  $k=0$  and  $r \in [1, P-1]$ . Giving

$\left\lceil \frac{m}{P} \right\rceil = 1$ , the total execution time based on (5) is:

$$T_{m \leq P} = t_{sn} (n+m-1) \quad (6)$$

Case 2:  $m \geq P$ ,  $m = kP$ ,  $r=0$ . Giving  $\left\lceil \frac{m}{P} \right\rceil = k$ , equation (5) becomes:

$$T_{m=kP} = t_{sn}(n+P-1)\frac{m}{P} \quad (7)$$

Case 3:  $m \geq P$ ,  $m = kP + r$ ,  $r \in [1, P-1]$ . In this case,  $\left\lceil \frac{m}{P} \right\rceil = \frac{m-r}{P} + 1$ , so equation (4)

becomes:

$$T_B = t_{sn}(n-m-1)\left(\frac{m-r}{P} + 1\right) \quad (8)$$

Therefore, the total execution time in (5) becomes:

$$T_{m=kP+r} = t_{sn}\left(\frac{m-r}{P} + 1\right)(n+r-1) \quad (9)$$

Equations (3)-(9) form the mathematical foundation for the optimal solution of supernode transformation, and are summarized as following:

$$T = \begin{cases} t_{sn}(n+m-1) & 1 \leq m \leq P, m \leq n \leq N \\ t_{sn}(n+P-1)\frac{m}{P} & m = kP, k \geq 1, m \leq M, m \leq n \leq N \\ t_{sn}\left(\frac{m-r}{P} + 1\right)(n+r-1) & m = kP+r, k \geq 1, 1 \leq m \leq M, m \leq n \leq N, 1 \leq r \leq P-1 \end{cases} \quad (10)$$

The goal of this research is to find the optimal solution  $(m_0, n_0)$  that minimizes the total execution time  $T$ . In the following sections, supernode transformation is applied to two multi-processor distributed memory systems: computer cluster systems and GPGPUs. Equations (3)-(10) above are used to find the optimal solution that minimizes the total execution time  $T$ . Note the model applies to two dimensional uniform

dependence algorithms with rectangular iteration space, with non-overlapping communication and computation phases.

## 5. Supernode Transformation On Computer Clusters

This section discusses supernode transformation on computer cluster system  $(P, t_r, t_s)$ , especially the finding of the optimal solution  $(m_0, n_0)$  of two dimensional uniform dependence algorithm. The optimal solution minimizes the total execution time expressed in (10).

In (10),  $t_{sn}$  is the execution time of one supernode on one computing node of the cluster. As mentioned in section 3, the execution of one supernode has three phases: data reading phase that takes time  $t_r$ , computing phase, and data saving phase that takes time  $t_s$ . For a supernode with a rectangular shape  $w \times h$  and processed by one computing node, the computing time is  $w \cdot h \cdot t_c$  where  $t_c$  is the computation time of one iteration. Let  $A = MNt_c$  and  $B = t_s + t_r$ , then:

$$t_{sn} = t_r + wht_c + t_s = \frac{A}{mn} + B \quad (2)$$

The basic idea of how to find the optimal solution of  $T$  in (10) is as follows. The solution space of  $m \in [1, M]$  is divided into three subspaces  $S_1$ ,  $S_2$  and  $S_3$ , where:

$$S_1 = \{m : m \in [1, P]\}, \quad S_2 = \{m : m = kP, k \in \mathbb{Z}^+, P \leq m \leq M\}, \quad \text{and}$$

$S_3 = \{m : m = kP + r, k \in \mathbb{Z}^+, P \leq m \leq M, 1 \leq r \leq P-1\}$ . The best solution in each subspace is identified, and then the optimal solution is obtained by comparing these best solutions of the three subspaces.

## 5.1 Lemmas and Theorem

In the following, three lemmas are introduced followed by the main theorem presenting the optimal solution. The theorem is proved by these three lemmas, while the proof of these lemmas can be found in Appendix A in section 9. Equations numbered (A x) are from Appendix A.

For subspace  $S_1$ , according to (10) and (11), the total execution time  $T$  is:

$$T_{m \leq P}(m, n) = \left( \frac{A}{mn} + B \right) (n + m - 1) \quad (\text{A0})$$

As discussed in the Appendix A,  $T_{m \leq P}(m, n)$  is convex in  $[2, P]$ , and has at most one minimum point, or stationary point. If there exists such a stationary point, it is denoted as

$(m_e, n_e)$ . Let  $(2, n_{b2})$  and  $(P, n_{bP})$  be two points in  $S_1$  such that  $\frac{\partial T}{\partial n}(2, n_{b2}) = 0$ ,

$\frac{\partial T}{\partial n}(P, n_{bP}) = 0$ . Let  $(m_1, n_1) \in S_1$  be the local minimum point, that is,

$T(m_1, n_1) = \min\{T(m, n) : m \in S_1\}$ , then  $(m_1, n_1)$  can be found by Lemma A1.

### 5.1.1 Lemma A1

In the solution subspace  $S_1 = \{m : m \in [1, P]\}$ , if the stationary point  $(m_e, n_e)$  exists,

$(m_1, n_1) \in \{(m_e, n_e), (1, 1)\}$ . Otherwise,  $(m_1, n_1) \in \{(2, n_{b2}), (P, n_{bP}), (1, 1)\}$ , where

$$n_e = \sqrt{\frac{(m_e - 1)A}{m_e B}}, \quad n_{b2} = \sqrt{\frac{A}{2B}}, \quad n_{bP} = \sqrt{\frac{(P - 1)A}{PB}}$$

Lemma A1 provides a candidate set for  $(m_1, n_1)$ . The candidate with the lowest  $T$  value is the local minimum point. Equation (A5) from Appendix A is used to obtain  $m_e$  in  $[2, P]$ :

$$B^2m^5 - B^2m^4 - 2ABm^3 + 2ABm^2 + (A^2 - AB)m - A^2 = 0 \quad (\text{A5})$$

Equation (A5) is a polynomial with one variable, real coefficients, and odd degree. So (A5) must have at least one real root [12]. (A5) may have more than one real root because some extraneous roots may be generated during processing. However, as discussed in Appendix A, (A5) has at most one valid real root. For a root  $m_{root}$  and its corresponding  $n_{root}$ , if they satisfy  $\frac{\partial T}{\partial m} = 0$  and  $\frac{\partial T}{\partial n} = 0$  for (A0), then  $m_e = m_{root}$ . Otherwise,  $(m_e, n_e)$  does not exist.

### 5.1.2 Lemma A2

In the solution subspace  $S_2 = \{m : m = kP, 1 \leq k, m \leq M\}$ ,  $T$  defined in (10) takes minimum at  $(m_2, n_2) = (P, \sqrt{\frac{A(P-1)}{PB}})$ . Note  $m = P$  is also in solution subspace  $S_1$ .

### 5.1.3 Lemma A3

For any  $m$  in the solution subspace  $S_3 = \{m : m = kP + r, 1 \leq k, m \leq M, 1 \leq r \leq P-1\}$ ,  $T_{m=kP+r}$  defined in (10) is always greater than  $T_{m=P}$  in solution subspace  $S_1$ .



#### 5.1.4 Theorem

In a cluster system with fixed  $P$  computing nodes, the total execution time is minimized with supernode size  $(m_0, n_0)$  that equals  $(m_1, n_1)$  defined in Lemma A1.

Proof: According to Lemma A3, the optimal solution is not in  $S_3$ . Lemma A2 tells that the local minimum solution of  $S_2$  is at  $m_2 = P$  which is also in solution subspace  $S_1$ . So the optimal solution is in solution subspace  $S_1$ . Thus  $(m_0, n_0) = (m_1, n_1)$ .

In the above discussion,  $m_e, n_e, n_{b_2}$  and  $n_{b_P}$  might be real numbers instead of integers. Then the nearest integer value will be used as the best integer solution for the convex function  $T_{m \leq P}$ , denoted by symbol  $\lceil \cdot \rceil$ . For example, when  $m_e$  is not an integer, then we get  $m_e = \lceil m_e \rceil$ .

## 5.2 Simulation

Simulations are conducted to find the optimal supernode size with the shortest running time for the LCS problem of size (600,1200). A multi-core system is used to simulate a cluster system. The system used is a X86\_64 8 CPU server with 8 cores, the kernel release is 3.13.0-55-generic. The operating system is Ubuntu 14.04.2 LTS. One core is designated as the master computer node and six cores are used as computing nodes, hence  $P=6$ . The master computer and computing nodes reside on their dedicated cores via Linux *sched\_setaffinity()* call. To facilitate more efficient communication and provide synchronization between computing nodes, the computing nodes communicate with the

master computer only. The master computer drives the entire workflow, starts and initializes all computing nodes. For each execution of a section of a wavefront, the master computer sends dependent data to each and every computing node via Unix sockets in time  $t_r$ , receives results from these computing nodes in time  $t_s$  after the computations are completed, and provides synchronization between computing nodes. The time each computing node spends on computation of one supernode is  $w \cdot h \cdot t_c$ .

All possible pairs of  $(m,n)$  are exhaustively tested. In the simulation,  $M=600, N=1200$ , so there are  $600 \times 1200$  possible pairs of  $(m,n)$ . The results are partially shown in the table below. In the table, a row corresponds to a particular value of  $m$  ranging from 1 to  $n$ . The six columns correspond to particular values of  $n=30, 20, 15, 6, 5$  and  $1$ . For example, the total execution time for  $(m,n)=(2,30)$  is  $11103 \mu s$ . The table shows that the total execution time at  $(m,n)=(6,6)$  is the shortest. The test results are also curved in Figure 7.

$m(\leq n)$	$n=30$	$n=20$	$n=15$	$n=6$	$n=5$	$n=1$
1	14474	12084	11662	9719	9752	8502
2	11103	9277	8439	7557	7658	
3	9412	7570	6699	5551	5556	
4	8837	7021	6169	5108	5140	
5	8583	6779	5936	4923	4970	
6	8489	6688	5850	4867		
7	14141	10286	8360			
8	14294	10470	8577			
9	14529	10726	8857			
10	14788	10996	9146			
11	15101	11319	9483			
12	15404	11628	9802			

13	20313	14485	11588			
14	20197	14358	11455			
15	20728	14910	12031			
16	21294	15496				
17	22451	16692				
18	25553	17696				
19	25501	17640				
20	26241	18406				
21	27031					
22	28602					
23	29383					
24	29323					
25	31355					
26	33376					
27	34347					
28	35799					
29	36283					
30	36383					

**Table 1: Total execution times for different values of (m,n) on a cluster system P=6 for the LCS problem of size 600x1200. For example, the total execution time for (m,n)=(2,30) is 11103.**

So  $(m_0, n_0) = (6, 6)$ , or  $(w, h) = (100, 200)$ , is the optimal supernode size, and the shortest total execution time is  $4867\mu s$ , out of which, the total communication time recorded from simulation is  $2118\mu s$ , and the total computation time is  $2749\mu s$ .

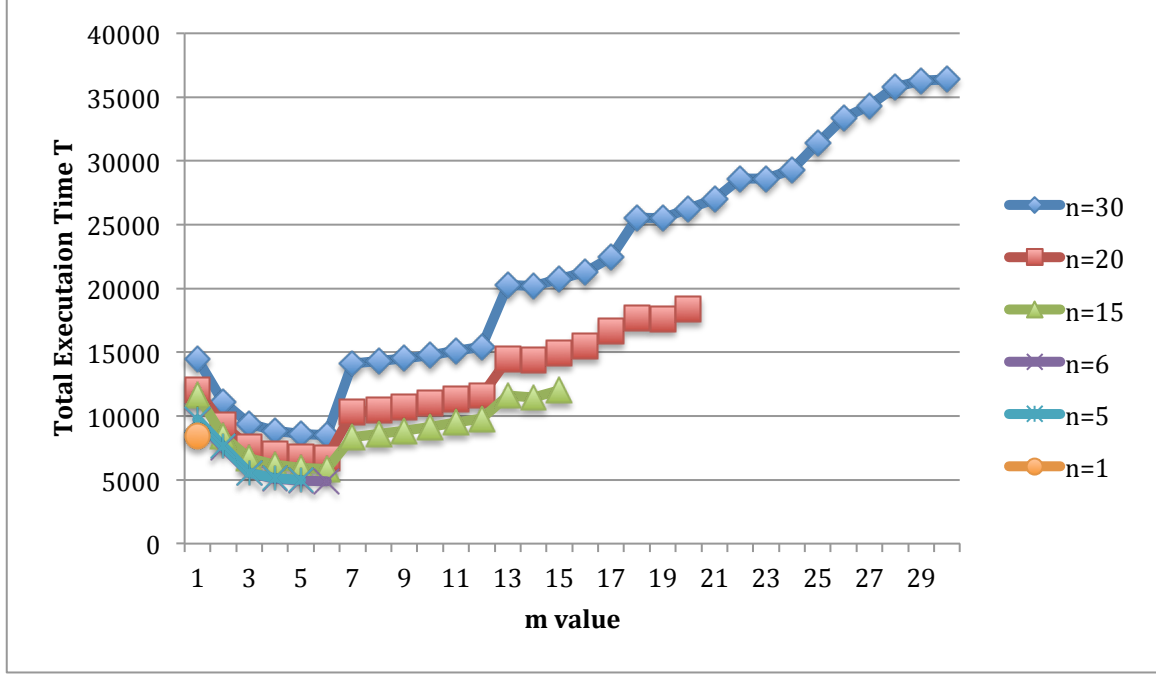


Figure 7: Total execution times for different values of (m,n) on cluster with P=6 for the LCS problem of size 600x1200. Six n values are shown. The chart indicates when (m,n)=(6,6), as indicated in the purple curve, T is the smallest.

To find the analytical value for  $(m_0, n_0)$  from theorem, parameters  $t_r + t_s$  and  $t_c$  are needed. Giving that the total communication cost is  $2118us$ , and there are  $m+n-1$  wavefronts, the communication cost per supernode is calculated as following:

$$t_s + t_r = 2118 / (m+n-1) = 193 us$$

For computation cost  $t_c$ , because the total computation cost is  $2749us$ , there are  $m+n-1$  wavefronts, and each wavefront needs one execution section which costs  $w \cdot h \cdot t_c$ :

$$t_c = total\_comp\_time / ((m+n-1) \cdot w \cdot h) = 2749 / (11 \cdot 100 \cdot 200) = 0.012 us$$

So  $A = MNt_c = 8640$ , and  $B = t_s + t_r = 193$ . Then equation (A5) produces 5 roots: -7, -6, 1, 6 and 7. Root -7, -6, 1 and 7 are not valid since they are not in the valid range of  $[2, P]$  which is  $[2, 6]$ . So only  $m=6$  is the valid root.

$$\text{When } m=6, n = \left\lceil \sqrt{\frac{(m-1)A}{mB}} \right\rceil = 6, \quad \frac{\partial T}{\partial m}(m_{root=6}, n_{root=6}) = 0 \quad \text{and} \quad \frac{\partial T}{\partial n}(m_{root=6}, n_{root=6}) = 0.$$

So  $(m_e, n_e) = (6, 6)$  is the stationary point. Thus according to (A0):

$$T(6, 6) = \left( \frac{A}{mn} + B \right) (n + m - 1) = \left( \frac{8640}{6 \cdot 6} + 193 \right) (6 + 6 - 1) = 4763$$

Then based on Lemma A1, there is a special point (1,1) that needs to be checked:

$$T(1, 1) = \left( \frac{A}{mn} + B \right) (n + m - 1) = \left( \frac{8640}{1 \cdot 1} + 193 \right) (1 + 1 - 1) = 8833$$

According to Theorem, point (6,6) yields smaller  $T$  than the point (1,1), hence it is the optimal point. The analytical optimal solution (6,6) exactly matches the simulation results, and its total running time 4763  $us$  is very close to the simulation minimum result of 4867  $us$ .

If the cyclic column-wise assignment in section 2.3 were used,  $T_{opt}$  would be 5181.

Note  $a = t_r + t_s$  and  $b = 0$ . So this research provides better result, without any restriction.

For the hypothetical case in section 2.3, per the theorem,  $(m_0, n_0) = (2, 2)$ , and  $T_{opt} = 3900$  which is much better than 5400 obtained using cyclic column-wise assignment.

## 6. Supernode Transformation on GPGPUs

This section discusses the supernode transformation on GPGPUs. The two-sequence LCS problem is used as an example to show how to use supernode transformation to map applications to a GPGPUs with the total execution time minimized. Equations of total execution times developed in section 4 apply to GPGPU architectures as well. However, the execution time of one supernode  $t_{sn}$  is different from the one in section 5 and is derived in section 6.1.

### 6.1 Analytical Results

The execution of a two dimensional uniform dependence algorithm such as LCS on a GPGPU is modeled as follows. A supernode with size  $w \times h$  is assigned to a block running on a GPGPU SM. When a block is launched, it first reads in the dependent data for this supernode from the global memory in time  $t_r$ , then it processes the iterations inside the supernode. Note the dependence graph of this supernode is similar to the one in [Figure 1](#), so the iterations are processed by wavefronts from lower-left corner to upper-right corner, and there are  $w+h-1$  wavefronts inside a supernode. The results of the iterations on each wavefront are stored in the shared memory of the block on the SM, and are used by the subsequent dependent wavefronts. After all wavefronts in the supernode are processed, the results for all iterations of the supernode in the shared memory are saved back to the global memory in time  $t_s$ .

Giving there are many threads inside each block/SM, the computations of iterations on one wavefront inside the supernode can be done in parallel, assuming inside each block, the number of threads is equal to or greater than the maximum number of iterations on any wavefront. Then all the iterations on the same wavefront are processed in parallel thus the execution time of one wavefront is  $t_c$ . The communication cost is ignored because the communication happens between threads and is done by accessing the shared memory on the same chip. Hence the total computation time of one supernode is  $(w+h-1) \cdot t_c$ .

Next the communication costs  $t_s$  and  $t_r$  of each supernode are discussed. When a block is launched for a supernode, the first thing the threads in the block do is to read in the dependent data of the iterations on the boundary of the supernode from the global memory. As shown in Figure 8, supernode of size  $w \times h$  depends on  $h$  iterations on the left of the supernode and  $w+1$  iterations at the bottom in the original iteration space.

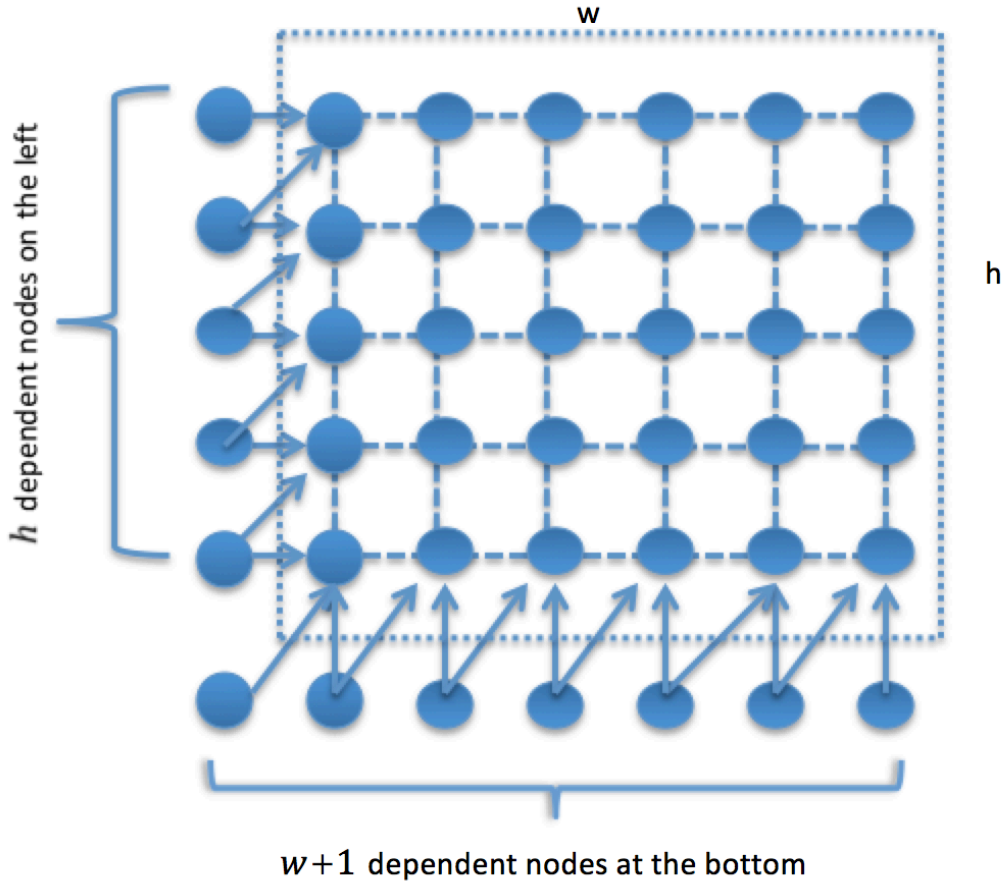
For the  $w+1$  dependent iterations to the bottom,  $w+1$  threads are used to read their data into the block from the global memory, one thread for each iteration. Since the  $w+1$  data are stored consecutively in the global memory, the  $w+1$  reads are coalesced into

$\left\lceil \frac{w+1}{s} \right\rceil$  memory segment transactions, note  $s$  is the memory segment size. So the total

reading time is  $\left\lceil \frac{w+1}{s} \right\rceil t_m$ , where  $t_m$  is one memory transaction time between the global

memory and a GPGPU block. For the  $h$  dependent iterations on the left, they are not consecutive in the global memory, so each iteration data takes one memory transaction to

read. Hence it takes  $ht_m$  to read these  $h$  data. So it takes  $\left\lceil \frac{w+1}{s} \right\rceil t_m + ht_m$  time to read in the data of the dependent iterations for a supernode.



**Figure 8: A LCS Supernode of size  $w \times h$  and its dependent nodes. The supernode is inside the dashed-line rectangular, while the dependent nodes are on the left and at the bottom.**

According to (2), there are  $w$  symbols from  $X$  sequence, and  $h$  symbols from  $Y$  sequence needed in the *LCS* computation for a supernode. So it takes  $\left\lceil \frac{w}{s} \right\rceil \cdot t_m$  and



$\left\lceil \frac{h}{s} \right\rceil \cdot t_m$  time to read these symbols in a coalesced way. Hence for a supernode, the  $t_r$  is

expressed as:

$$t_r = \underbrace{\left\lceil \frac{w+1}{s} \right\rceil \cdot t_m + ht_m}_{\text{for bottom and left dependent iterations}} + \underbrace{\left\lceil \frac{w}{s} \right\rceil \cdot t_m + \left\lceil \frac{h}{s} \right\rceil \cdot t_m}_{\text{for X and Y sequences symbols}}$$

When the computations of a supernode are done, the results of all iterations in the supernode, not just those of boundary nodes, are saved back into the global memory. Each iteration generates one datum to be saved into the global memory, note  $s$  consecutive data in a row can be coalesced into one memory transaction. Therefore, for the  $w$  data on one row,  $\left\lceil \frac{w}{s} \right\rceil$  memory segment transactions are needed, and this process

is repeated  $h$  times to finish the entire supernode, so for a supernode,  $t_s = h \cdot \left\lceil \frac{w}{s} \right\rceil t_m$ .

Thus:

$$t_{sn} = \underbrace{(w+h-1)t_c}_{\text{computation time}} + \underbrace{h \left\lceil \frac{w}{s} \right\rceil t_m + \left\lceil \frac{w+1}{s} \right\rceil t_m + ht_m + \left\lceil \frac{w}{s} \right\rceil t_m + \left\lceil \frac{h}{s} \right\rceil t_m}_{\text{communication cost}} \quad (11)$$

To simplify the analysis, all ceiling functions are removed. The error analysis due to this approximation is in section 5.3. Also  $w = \frac{M}{m}$  and  $h = \frac{N}{n}$ , then:

$$t_{sn} = \underbrace{\left(\frac{M}{m} + \frac{N}{n} - 1\right)t_c}_{\text{computation time}} + \underbrace{\left(\frac{N}{n} + 1\right)\frac{M}{sm}t_m + \frac{\frac{M}{m} + 1}{s}t_m + \frac{N}{n}t_m + \frac{N}{sn}t_m}_{\text{communication cost}} = \frac{A}{m} + \frac{B}{n} + \frac{C}{mn} - D \quad (12)$$

where

$$A = M(t_c + \frac{2t_m}{s}), \quad B = N(t_c + (1 + \frac{1}{s})t_m), \quad C = \frac{MNt_m}{s}, \quad D = t_c - \frac{t_m}{s} \quad (13)$$

The basic idea of finding the optimal solution of  $T$  in (10) on GPGPUs is similar to that of in section 5, as follows. The solution space of  $m \in [1, M]$  is divided into three subspaces  $S_1$ ,  $S_2$  and  $S_3$ :  $S_1 = \{m: m \in [1, P]\}$ ,  $S_2 = \{m: m = kP, k \in \mathbb{Z}^+, P \leq m \leq M\}$ ,  $S_3 = \{m: m = kP + r, k \in \mathbb{Z}^+, P \leq m \leq M, 1 \leq r \leq P-1\}$ . The best solution in each subspace is identified, and then the optimal solution is obtained by comparing these best solutions of the three subspaces.

## 6.2 Lemmas and Theorem

Three lemmas are introduced which lead to theorem that presents the optimal solution of supernode transformation on GPGPUs. The theorem is proved based on these lemmas. Please refer to Appendix B for the proof of these lemmas. Equations numbered (B  $x$ ) are from Appendix B.

For subspace  $S_1$ , according to (10) and (12), the total execution time  $T$  is:

$$T_{m \leq P}(m, n) = (\frac{A}{m} + \frac{B}{n} + \frac{C}{mn} - D)(n + m - 1) \quad (B0)$$

As discussed in the Appendix B,  $T_{m \leq P}(m, n)$  is convex in  $[2, P]$ , and has at most one minimum point, or stationary point. If there exists such a stationary point, it is denoted as  $(m_e, n_e)$ . Let  $(2, n_{b2})$  and  $(P, n_{bP})$  be two point in  $S_1$  such that  $\frac{\partial T}{\partial n}(2, n_{b2}) = 0$ ,

$\frac{\partial T}{\partial n}(P, n_{bp}) = 0$  . Let  $(m_1, n_1) \in S_1$  be the local minimum point, that is,

$T(m_1, n_1) = \min\{T(m, n) : m \in S_1\}$  . Then  $(m_1, n_1)$  can be found by Lemma B1.

### 6.2.1 Lemma B1

In the solution subspace  $S_1 = \{m : m \in [1, P]\}$  , if the stationary point  $(m_e, n_e)$  exists,

$(m_1, n_1) \in \{(m_e, n_e), (1, 1)\}$  , otherwise,  $(m_1, n_1) \in \{(2, n_{b2}), (P, n_{bp}), (1, 1)\}$  , where

$$n_e = \sqrt{\frac{Bm_e^2 + (C - B)m_e - C}{A - Dm_e}}, n_{b2} = \sqrt{\frac{2B + C}{A - 2D}}, n_{bp} = \sqrt{\frac{B \cdot P^2 + (C - B)P - C}{A - DP}}$$

Lemma B1 provides a candidate set for  $(m_1, n_1)$  . The candidate yielding the lowest  $T$  value is the local minimum point. Equation (B5) from Appendix B is used to obtain  $m_e$  in  $[2, P]$  :

$$a_1 m^7 + a_2 m^6 + a_3 m^5 + a_4 m^4 + a_5 m^3 + a_6 m^2 + a_7 m + a_8 = 0 \quad (\text{B5})$$

where  $A, B, C, D$  are constants defined in (13), and constants  $a_1, a_2, a_3, a_4, a_5, a_6, a_7$  are defined in terms of  $A, B, C, D$  in (B6) in Appendix B.

Equation (B5) is a polynomial in one variable with real coefficients and odd degree. So (B5) must have at least one real root [12]. (B5) may have more than one root because some extraneous roots may be generated during processing. However, as discussed in Appendix B, (B5) has at most one valid real root. For a root  $m_{root}$  and its corresponding

$n_{root}$ , if they satisfy  $\frac{\partial T}{\partial m}=0$  and  $\frac{\partial T}{\partial n}=0$  for (A0), then  $m_e = m_{root}$ . Otherwise,  $(m_e, n_e)$

does not exist.

### 6.2.2 Lemma B2

In the solution subspace  $S_2 = \{m: m=kP, 1 \leq k, m \leq M\}$ ,  $T$  defined in (10) takes

minimum at  $(m_2, n_2) = (P, \sqrt{\frac{(BP+C)(P-1)}{A-DP}})$ . Note  $m_2 = P$  is also in subspace  $S_1$ .

### 6.2.3 Lemma B3

For any  $m$  in the solution subspace  $S_3 = \{m: m=kP+r, 1 \leq k, m \leq M, 1 \leq r \leq P-1\}$ ,

$T_{m=kP+r}(m, n)$  defined in (10) is always greater than  $T_{m=P}(m, n)$  in subspace  $S_1$ .

### 6.2.4 Theorem

On GPGPU system  $(P, t_s, t_r, s)$ , the total execution time of algorithm  $(J, D)$  is minimized with supernode size  $(m_0, n_0)$  that equals  $(m_1, n_1)$ , as defined in Lemma A1.

Proof: according to Lemma B3, the optimal solution is not in  $S_3$ . Lemma B2 tells that the local best solution of  $S_2$  is at  $m_2 = P$  which is also in solution subspace  $S_1$ . So the optimal solution is in solution subspace  $S_1$ . Thus  $(m_0, n_0) = (m_1, n_1)$ .

### 6.3 Simulation results

Simulations are conducted to find the optimal supernode size with the shortest running time for the LCS problem of size  $(M,N)=(600,1200)$ , using Nvidia's GeForce GTX 760 GPGPU. Its base clock is 980MHz and it has 6 SMs. The simulations use a one-dimension grid of size 6, thus  $P=6$ . For each execution of a wavefront section, a one-dimension grid of blocks is launched, and each block handles one supernode. Nvidia's CUDA provides API *clock64()* for precise time measuring in the unit of clock cycles, it is used in the simulations to record the total execution time  $T$  and the communication time  $t_s + t_r$ .

In the simulation, all possible pairs of  $(m,n)$  are tested. Hence, there are 600 x 1200 possible pairs of  $(m,n)$ . But in practice, Nvidia GPGPUs have a limit in the shared memory size. For example, the GeForce GTX 760 used in the simulation has 48KB shared memory size. This means there is one more restriction in the supernode size to ensure the entire supernode can be loaded in the shared memory for computation, that is,  $w \cdot h \cdot z < 48KB = 49152$  bytes, where  $z$  is the size of each iteration value in the shared memory. In simulation, the value of each iteration is stored as "short integer" which occupies 2 bytes of storage, thus  $z=2$ . Apparently  $(m,n)=(1,1)$  can not be the optimal solution since it treats the entire problem space as one supernode of size 600x1200, and this large supernode can not fit in the shared memory because  $600 \times 1200 \times 2 > 48KB$ , so it has to be partitioned into smaller supernodes to take advantage of GPGPU's shared memory for faster and more efficient execution.

The test results are partially shown in Table 2. In Table 2, a row corresponds to a particular value of  $m$  ranging from 1 to 24. The four columns correspond to particular values of  $n=24, 12, 8$  and  $6$ . The total execution times  $T$  from simulations are recorded and shown. For example,  $T$  for  $m=3, n=24$  is 6,339,434 clock cycles. The tests show when  $(m,n)=(5,8)$ , with the corresponding  $(w,h)=(123,152)$ , the total execution time  $T=3,320,364$  clock cycles is the shortest, out of which the total save/read time  $t_s+t_r$  recorded is 358,704 clock cycles. The simulation results are also curved in Figure 9. Note there are some empty cells in the table because they require more shared memory than 48KB. For example, when  $m=1$  and  $n=24$ , the corresponding  $w$  and  $h$  are 600 and 50, then  $w \cdot h \cdot z = 600 \cdot 50 \cdot 2 > 48KB$ .

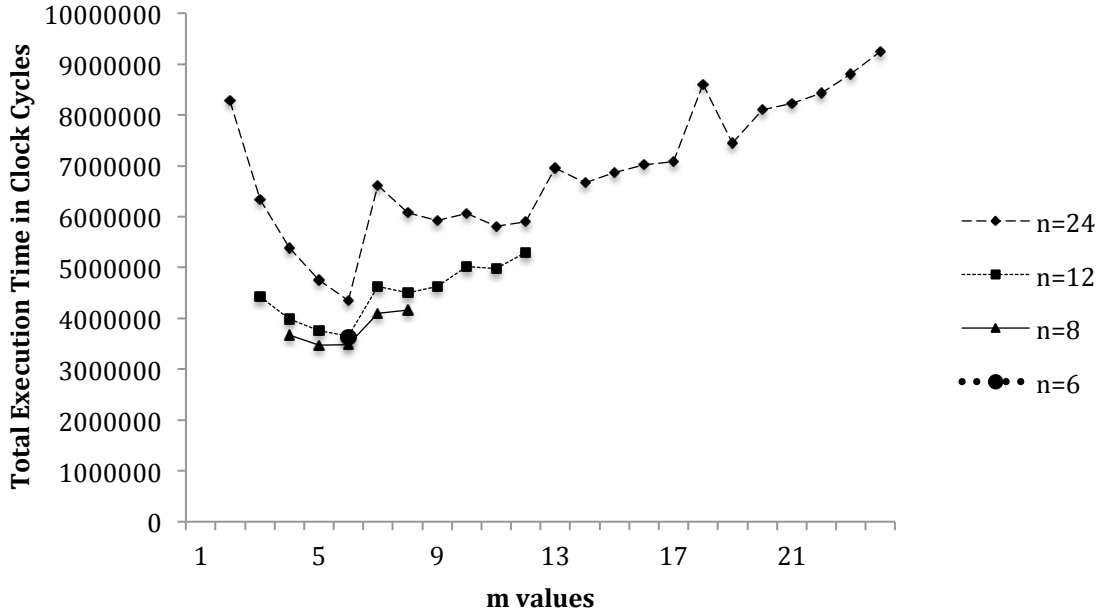
$m \leq n$	$n=24$	$n=12$	$n=8$	$n=6$
1				
2	8278296			
3	6339434	4427573		
4	5387489	3989307	3664786	
5	4755731	3764232	3472624	
6	4351107	3650690	3480312	3615454
7	6613261	4630957	4097116	
8	6072377	4507131	4164922	
9	5922144	4628155		
10	6067154	5014499		
11	5806633	4977027		
12	5900364	5289875		
13	6961389			
14	6666499			
15	6863515			
16	7016888			
17	7078705			
18	9600376			

19	7449397			
20	8101262			
21	8231596			
22	8425767			
23	8802064			
24	9250178			

**Table 2: Total execution times of the LCS problem for different values of (m,n) on a Nvidia GeForce GTX 760 GPGPU. For example, the total execution time for (m,n)=(3,24) is 6,339,434 clock cycles.**

Note Figure 9 shows recurring curve segments slanting upward to the right, the size of each segment is  $m$ . According to (B7) from Appendix B,  $T_{m=kP}$  is:

$$T_{m=kP} = (A + \frac{B}{n}m + \frac{C}{n} - Dm)(n + P - 1)(\frac{1}{P}) \quad (B7)$$



**Figure 9: Total execution times of the LCS problem for different values of (m,n) on a Nvidia GeForce GTX 760 GPGPU. Results for four n values are displayed. The chart indicates when (m,n)=(5,8), as indicated in the green curve, T is the smallest.**

Giving that  $\frac{B}{n} > D$ ,  $T'_{m=kP} > 0$ , so  $T_{m=kP}$  increases along with  $k$ , that is,  $T_{(k+1)P} > T_{kP}$ ,

thus the graph slants upward to the right.

Then for all  $m$  between  $T_{m=kP}$  and  $T_{m=(k+1)P}$ , based on (B9) from Appendix B:

$$T_{m=kP+r} = \frac{HP(k+1)(r+G)}{(kP+r)P} + E(k+1)(r+G) \quad (B9)$$

Where  $1 \leq r < P$  and  $H, E, G$  are constants with  $G = n - 1$ . It's easy to show

$$T''_{m=kP+r} = -\frac{2H(k+1)(kP-G)}{(kP+r)^3} \text{ with respect to } r. \text{ Giving that } G = n - 1 \geq m - 1 = kP + r - 1,$$

$T''_{m=kP+r} > 0$  for all  $r \in [2, P)$ , as shown as convex curves in each segment. When  $r = 1$ ,

$$T''_{m=kP+r} = 0, T_{m=kP+1} \text{ thus can be larger or smaller than } T_{m=kP+r, r \in [2, P)}.$$

Now let's check the optimal solution according to theorem. Based on the simulation, the optimal point is  $(w, h) = (123, 152)$ , that is  $(m_0, n_0) = (5, 8)$ , and  $T(5, 8) = 3,320,364$  clock cycles, out of which, the total  $t_r + t_s$  for all wavefronts is 358,704 clock cycles. Given that there are  $m + n - 1$  wavefronts, each wavefront only needs one round of execution since  $m_0 < P$ , and each execution costs one  $t_{sn}$  to complete, out of which  $(w + h - 1) \cdot t_c$  time is on computation, thus  $t_c$  is calculated as:

$$t_c = \frac{T(5, 8) - \text{total}(t_s + t_r)}{(m + n - 1)(w + h - 1)} = \frac{3320364 - 358704}{(5 + 8 - 1)(123 + 152 - 1)} = 901 \text{ clock cycles}$$

Similarly, the total communication cost is  $(t_r + t_s) \cdot (m + n - 1)$ , and equation (12) gives out the communication cost  $t_r + t_s$  in  $t_{sn}$ , thus:

$$\underbrace{\left( \frac{MN}{smn} t_m + \frac{2M}{sm} t_m + \frac{N}{n} \left( 1 + \frac{1}{s} \right) t_m + \frac{1}{s} t_m \right)}_{\text{communication of one round execution}} (m + n - 1) = 358704$$



Two bytes are used to store computation results in the simulation, so the memory segment size is 64 bytes [3], or  $s = 32$ . So:

$$t_m = \frac{358704}{\left(\frac{600*1200}{32*5*8} + \frac{2*600}{32*5} + \frac{1200}{8}\left(1 + \frac{1}{32}\right) + \frac{1}{32}\right)(5+8-1)} = 41.3 \text{ clock cycles}$$

Parameters in (13) are then obtained:  $A = 542149$ ,  $B = 1132309$ ,  $C = 929250$ ,  $D = 900$ . With these values, (B5) gives five real roots: -653.8, -30.455, -2.249, 5.18, 30.861. Apparently only root 5.18 is in valid range, thus  $m_e = m_{root} = 5.18$ , and per Lemman B1:

$$n_e = \left\lceil \sqrt{\frac{Bm_e^2 + (C - B)m_e - C}{A - Dm_e}} \right\rceil = \lceil 7.3 \rceil = 7$$

Thus  $(m_e, n_e) = (5.18, 7) \approx (5, 7)$  which is close to the simulation result (5,8).

Same simulations were also conducted on GeForce GTX Titan X, a higher end GPGPU with 24 SMs. The tests show similar results with a better optimal value of 2416220 clock cycles, a 27% improvement even compared to using GTX 760.

## 6.4 Result Analysis

According to (B0),  $T_{m \leq p}(5, 7) = \left(\frac{A}{m} + \frac{B}{n} + \frac{C}{mn} - D\right)(m+n-1) = 3254220$ , it is very close to the exhaustive test result of 3320364. This proves the mathematical model is reasonably correct.

While there is no existing formula for column-wise assignment on GPGPUs that can be used for comparison, the basic assumption of column-wise assignment is all available

processors participate in the execution. The exhaustive tests show when  $m=P=6$ , the shortest running time is 3615454 clock cycles, which is much worse than the optimal result by this research.

The efficient algorithm in [35] is an improved approach running LCS on GPGPU, but it does not use supernode transformation to take advantage of improved data locality and parallelism. The simulations on the same GPGPU show it takes 12731633 clock cycles to complete the LCS problem of the same problem size, which is about 3.8 times of the result of the optimal solution of this research.

## 6.5 The Selection of GPGUP Architecture Model Parameter $P$

The GPGPU architecture parameter  $P$  is modeled as one-dimension grid size, that is, the number of blocks. In simulation it is set to  $P=6$ , which is the number of SMs on GPGPU. Though this parameter shows up in equation (10) hence it impacts the value of optimal solution  $(m_0, n_0)$ , simulations show it does not affect the total execution time  $T$  much. Table 2 shows different  $P$  values and their corresponding shortest total execution time  $T$  via exhaustive tests.

$P=6$	$P=12$	$P=24$	$P=30$	$P=48$	$P=72$
3320364	3311516	3271551	3271625	3271625	3275971
$P=96$	$P=120$	$P=144$	$P=168$	$P=192$	
3288002	3308869	3315377	3315723	3321326	

**Table 3: Total execution time obtained using exhaustive tests under different  $P$ . The difference is  $<1.5\%$  showing the grid size has little impact on the result, due to the efficient block scheduling of GPGPU.**

This is because Nvidia's GPGPU is quite efficient in scheduling blocks and threads on its hardware multi-streaming processors. When one group of threads (a warp) stalls on a

memory operation, GPGPU will switch it out and switch in another group of threads efficiently. So All processors in GPGPU are productive, irrespective of the number of blocks, as long as there is enough parallelism (threads) to keep them busy.

## 7. Conclusion

This paper addressed problem of supernode transformation for algorithm  $(J,D)$  on multi-processor system with distributed memory, including computer cluster systems and GPGPUs, especially on the finding of the optimal supernode size for time optimal performance. First a generic mathematical model is established for two dimensional uniform dependency algorithms. Then the model is applied on those two multi-processor architectures, and the optimal supernode size is obtained. Simulations on both architectures verified the correctness of the mathematical model and the optimal supernode size solution.

The model focuses on the total running time, which comprises of computation and communication times, representing locality and parallelism. Feasible linear schedule ensures maximum parallelism to take advantage of multi-processor's parallel processing capability, while data locality is improved by tiling supported by computing nodes' cache and GPGPU's fast on-chip shared memory. The model captures these two aspects in terms of total running time. By minimizing the total running time, the optimal supernode size is obtained, leading to the optimized locality and parallelism for optimal execution performance.

## 8. Future Work

While this research focuses on two dimensional uniform dependence algorithms, it will be worth to find out if it can be applied to higher dimensional arbitrary dependence and irregular shape algorithms. One way may be to combine this research, especially the execution time model, with polyhedral model for further study.

On cluster system, the model assumes the communication cost is a constant where the start-up time dominates. This becomes inaccurate when problem size is large. In such case, a linear function of the supernode size as communication cost is more appropriate. The model can be modified to accommodate such cases. Note this is not a problem with GPGPUs, since there the size of the supernode is already limited by the shared memory size of the SM chip.

## 9. Appendix

### 9.1 Appendix A: Lemmas For Supernode Transformation On Cluster Systems

The appendix A contains three lemmas and their proofs, for supernode transformation on cluster systems. The lemmas apply to three solution subspaces, respectively, they help to derive the theorem used in section 5 for supernode transformation on cluster system.

#### 9.1.1 Lemma A1

In the solution subspace  $S_1 = \{m: 1 \leq m \leq P\}$ , with (11),  $T$  defined in (10) becomes:

$$T(m, n) = \left(\frac{A}{mn} + B\right)(n + m - 1) \quad (\text{A0})$$

As shown later, when  $m \in [2, P]$ ,  $T(m, n)$  is convex and has at most one stationary point

such that  $\frac{\partial T}{\partial m} = 0$  and  $\frac{\partial T}{\partial n} = 0$  at this point. If there exists such stationary point, it is

denoted as  $(m_e, n_e)$ . Let  $(2, n_{b2})$  and  $(P, n_{bp})$  be two points in  $S_1$  such that  $\frac{\partial T}{\partial n}(2, n_{b2}) = 0$

and  $\frac{\partial T}{\partial n}(P, n_{bp}) = 0$ . Let  $T_{\min}$  be the shortest running time in (A0), so

$T_{\min} = \min\{T(m, n): m \in S_1\}$ . Let  $(m_1, n_1) \in S_1$  be the local minimum point such that

$T(m_1, n_1) = T_{\min}$ , then  $(m_1, n_1)$  can be found from (A1) as following:

$$(m_1, n_1) = \begin{cases} (1,1) & \text{if } T(1,1) < T(m_e, n_e) \text{ and if } (m_e, n_e) \text{ exists} \\ & \text{if } T(1,1) < \min(T(m_2, n_{b2}), T(P, n_{bp})) \text{ and if } (m_e, n_e) \text{ not exist} \\ (m_e, n_e) & \text{if } (m_e, n_e) \text{ exists and } T(m_e, n_e) < T(1,1) \\ (2, n_{b2}) & \text{if } (m_e, n_e) \text{ not exist and } T(m_2, n_{b2}) < \min(T(1,1), T(P, n_{bp})) \\ (P, n_{bp}) & \text{if } (m_e, n_e) \text{ not exist and } T(P, n_{bp}) < \min(T(1,1), T(m_2, n_{b2})) \end{cases} \quad (A1)$$

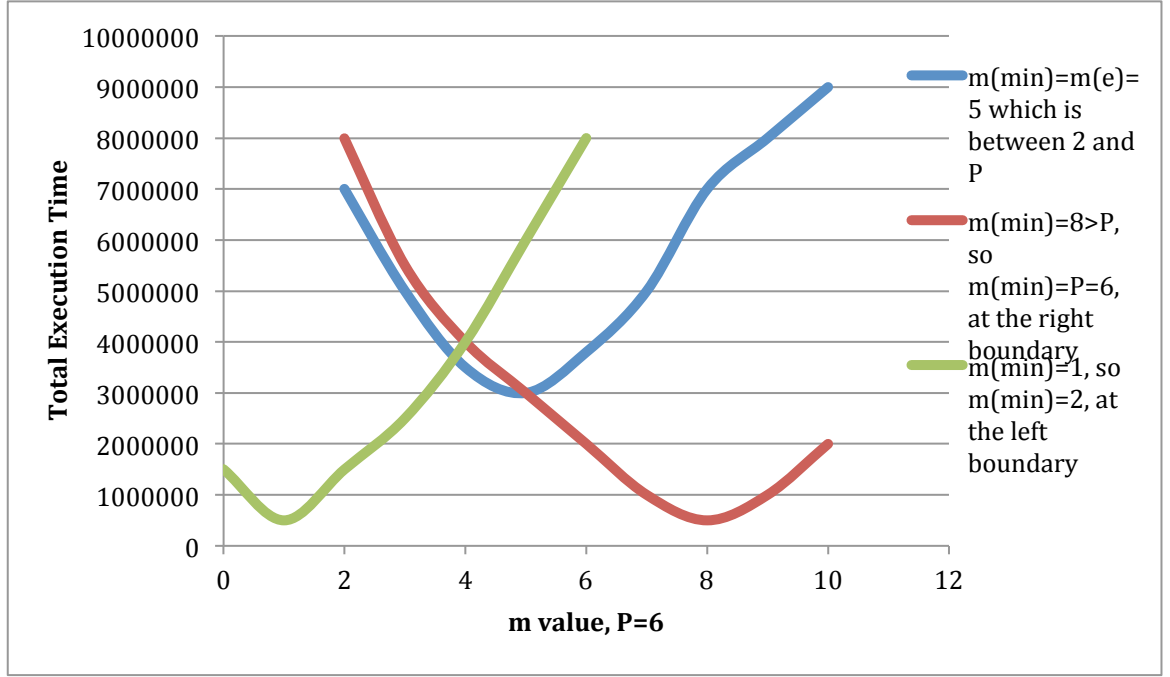
Proof: When  $m=1$ ,  $T$  in (A0) becomes  $A+Bn$ ,  $T'_n = B > 0$ , so  $n=1$  will make  $T_{m=1}$  the smallest. Thus (1,1) is the optimal point for  $T$  for the case  $m=1$ . This basically means the entire problem space is treated as one supernode and processed by one processor. This may happen when the computation time of a node  $t_c$  is extremely small compared to the communication time  $t_s + t_r$ , so the communication cost is dominant that makes it more efficient to process all iterations in one processor to minimize the communication cost.

Now consider equation (A0) for more general case  $m \in [2, P]$ . The partial derivative of  $T$  with respect to  $m$  is:

$$T'_m = \left(\frac{1}{n} - 1\right)A \frac{1}{m^2} + B \quad (A2)$$

The second derivative with respect to  $m$   $T''_m = \left(1 - \frac{1}{n}\right)A \frac{2}{m^3} > 0$  for all  $m \in [2, P]$ , so  $T(m)$  is convex, which means there is one and only one minimum point, which is called stationary point. If  $m_e$  is the stationary point, then with  $m$  increases,  $T$  decreases on the left hand side of  $m_e$  and increases on the right hand side of  $m_e$ . For the given range  $[2, P]$ , A convex can have three cases as shown in Figure 10: 1)  $m_e$  happens outside the left

boundary, then  $m=2$  is the minimum point; 2)  $m_e$  happens outside the right boundary, then  $m=P$  is the minimum point; and 3)  $m_e \in [2,P]$  where  $T'_{m,m \in [2,P]} = 0$ .



**Figure 10: On a cluster system, the total execution time for a two-dimensional uniform dependence algorithm  $T$  is convex with respect to  $m$  when  $m \in [2,P]$ .  $T_{\min}$  can happen at either of the three points:  $m_{\min} = 2$ ,  $m_{\min} = P$ , and  $m_{\min} = m_e \in [2,P]$  where  $T'_{m_e} = 0$ .**

To get the solution point  $m_e$ , take the partial derivative of  $T$  in (A0) with respect to  $n$

:

$$T'_n = \left(\frac{1}{m} - 1\right)A \frac{1}{n^2} + B \quad (\text{A3})$$

Let both (A2) and (A3) = 0, then two equations are obtained:

$$n = \sqrt{\frac{(m-1)A}{mB}} \quad (\text{A4})$$



$$B^2m^5 - B^2m^4 - 2ABm^3 + 2ABm^2 + (A^2 - AB)m - A^2 = 0 \quad (A5)$$

Equation (A5) is a polynomial in one variable with real coefficients and odd degree, so it must have at least one real root [12]. From (A5) the root  $m_{root}$  can be obtained, then the

corresponding  $n_{root}$  can be obtained based on equation (A4):  $n_{root} = \sqrt{\frac{(m_{root} - 1)A}{m_{root}B}}$ . Note

$(m_{root}, n_{root})$  may be not integer solution, then the nearest integer value will be used as the best integer solution for the convex function  $T_{2 \leq m \leq P}$ , that is,  $(m_e, n_e) = (\lceil m_{root} \rceil, \lceil n_{root} \rceil)$ .

In the case  $m_e$  is outside range  $[2, P]$ , boundaries  $m_{b2} = 2$  and  $m_{bP} = P$  are checked for smaller  $T$ , and according to (A4):

$$n_{b2} = \left\lceil \sqrt{\frac{A}{2B}} \right\rceil \text{ and } n_{bP} = \left\lceil \sqrt{\frac{(P-1)A}{PB}} \right\rceil.$$

So combining the two cases:  $m = 1$ , the optimal solution for it is  $(1, 1)$ ; and  $m \in [2, P]$ , the optimal solution for it is  $(m_e, n_e)$  if it exists in  $[2, P]$ , otherwise it is either  $(2, n_{b2})$  or  $(P, n_{bP})$ . Hence the optimal solution  $(m_1, n_1)$  for  $m \in [1, P]$  is obtained as summarized in (A1).

Equation (A5) may have up to five roots because some extraneous roots may be produced when processing equations (A2) and (A3). When multiple roots are obtained, all roots are analyzed and the real valid one can be found by checking if it is in the valid range  $[2, P]$  and if it satisfies equation (A2)=0 and (A3)=0.

### 9.1.2 Lemma A2

In solution subspace  $S_2 = \{m : m = kP, 1 \leq k, m \leq M\}$ ,  $(m_2, n_2) = (P, \left\lceil \sqrt{\frac{A(P-1)}{PB}} \right\rceil)$  is the

best solution.

Proof: According to (7) in section 4 and (11) in section 5:

$$T_{m=kP} = t_{sn} (n + P - 1) \left( \frac{m}{P} \right) = \frac{A}{P} + \frac{A(P-1)}{P} \frac{1}{n} + \frac{B}{P} mn + \frac{B(P-1)}{P} m \quad (\text{A6})$$

The derivative of  $T$  with respect to  $m$  is:  $T'_{m, m=kP} = \frac{B}{P} n + \frac{B(P-1)}{P} > 0$

So the smaller  $m$  is, the smaller  $T$  will be. Giving that  $m = kP$ , then  $m$  is the smallest when  $k = 1$ , thus the optimal solution  $m_2 = P$ , and (A6) becomes:

$$T_{m=P} = \frac{A}{P} + \frac{A(P-1)}{P} \frac{1}{n} + Bn + B(P-1)$$

Get partial derivative with respect to  $n$  and let it equal 0:

$$T'_{n, m=P} = -\frac{A(P-1)}{P} \frac{1}{n^2} + B \implies n = \sqrt{\frac{A(P-1)}{PB}}$$

So when  $m = kP$ , the best solution is:  $m_2 = P$ ,  $n_2 = \left\lceil \sqrt{\frac{A(P-1)}{PB}} \right\rceil$ .

### 9.1.3 Lemma A3

For any  $m$  in solution subspace  $S_3 = \{m : m = kP + r, 1 \leq k, m \leq M, 1 \leq r \leq P-1\}$ ,

$T_{m=kP+r}$  defined in (10) is always greater than  $T_{m=P}$  in  $S_1$ , that is:  $T_{m=kP+r} > T_{m=P}$ .

Proof: from equation (9) in section 4:

$$T_{m=kP+r} = t_{sn} \left( \frac{m-r}{P} + 1 \right) (n+r-1) = t_{sn} (k+1)(n+r-1) \quad (\text{A7})$$

next check  $T_{m=P}$ , from (7) in section 4:

$$T_{m=P} = t_{sn} (n+P-1) \quad (\text{A8})$$

It is obvious (A7) > (A8), thus  $T_{m=kP+r} > T_{m=P}$ , Lemma A3 is true.

## 9.2 Appendix B: Lemmas For Supernode Transformation On GPGPUs

This appendix B contains three lemmas and their proofs. The lemmas apply to three solution subspaces, respectively, for supernode transformation on GPGPUs, and they form the base of the theorem used in section 6 for time optimal solution on GPGPUs.

### 9.2.1 Lemma B1

On GPGPUs, in the solution subspace  $S_1 = \{m: m \in [1, P]\}$ , with (12),  $T$  defined in (10) becomes:

$$T_{m \leq P}(m, n) = \left( \frac{A}{m} + \frac{B}{n} + \frac{C}{mn} - D \right) (m + n - 1) \quad (\text{B0})$$

where  $A = M(t_c + \frac{2t_m}{s})$ ,  $B = N(t_c + (1 + \frac{1}{s})t_m)$ ,  $C = \frac{MNt_m}{s}$ ,  $D = t_c - \frac{t_m}{s}$ . As shown later,

$T(m, n)$  is convex when  $m \in [2, P]$  and has at most one stationary point such that  $\frac{\partial T}{\partial m} = 0$

and  $\frac{\partial T}{\partial n} = 0$  at this point. If there exists such a stationary point, it is denoted as  $(m_e, n_e)$ .

Let  $(2, n_{b2})$  and  $(P, n_{bP})$  be two point in  $S_1$  such that  $\frac{\partial T}{\partial n}(2, n_{b2}) = 0$  and  $\frac{\partial T}{\partial n}(P, n_{bP}) = 0$ .

Let  $T_{\min}$  be the shortest execution time in (B0), so  $T_{\min} = \min\{T(m, n): m \in S_1\}$ . Let

$(m_1, n_1) \in S_1$  be the local minimum point, that is,  $T(m_1, n_1) = T_{\min}$ . Then  $(m_1, n_1)$  can be

found as follows:

$$(m_1, n_1) = \begin{cases} (1,1) & \text{if } T(1,1) = T_{\min} \\ (m_e, n_e) & \text{if } (m_e, n_e) \text{ exists and } T(m_e, n_e) = T_{\min} \\ (2, n_{b2}) & \text{if } (m_e, n_e) \text{ does not exist and } T(m_2, n_{b2}) = T_{\min} \\ (P, n_{bP}) & \text{if } (m_e, n_e) \text{ does not exist and } T(P, n_{bP}) = T_{\min} \end{cases} \quad (\text{B1})$$

Proof: From (B0), take partial derivative of  $T$  with respect to  $m$  and  $n$ :

$$T'_m = \frac{-An + A - C + \frac{C}{n}}{m^2} + \frac{B}{n} - D \quad (\text{B2})$$

$$T'_n = \frac{-Bm + B - C + \frac{C}{m}}{n^2} + \frac{A}{m} - D \quad (\text{B3})$$

First check the special case of  $m=1$ . In this case (B3), so  $n=1$   $T'_n = A - D > 0$  will make  $T_{m=1}$  the smallest. Thus (1,1) is the optimal point for  $T$  for the case  $m=1$ . This basically treats the entire original problem space as one supernode and processes it using one GPGPU block. This could happen when the communication time  $t_s + t_r$  is so dominant that minimizing the communication cost will efficiently reduce the total execution time  $T$ .

Next move to the more general case of  $m \in [2, P]$  for (B0). The second partial

derivative with respect to  $m$  is:  $T''_m = \frac{2(A(n-1) + C(1 - \frac{1}{n}))}{m^3} > 0$ , so  $T_m$  is convex, which

means there is one and only one minimum point, which is called stationary point.

To obtain  $m_e$  and its corresponding  $n_e$ , let equations (B2)  $T'_m = 0$  and (B3)  $T'_n = 0$ , following two equations can be obtained:

$$n = \sqrt{\frac{Bm^2 + (C - B)m - C}{A - Dm}} \quad (B4)$$

$$a_1 m^7 + a_2 m^6 + a_3 m^5 + a_4 m^4 + a_5 m^3 + a_6 m^2 + a_7 m + a_8 = 0 \quad (B5)$$

Where:  $a_1 = BD^3$

$$a_2 = B^2 D^2 - ABD^2$$

$$a_3 = -ACD^2 - ABD^2 - BD^3 + 2BCD^2$$

$$a_4 = 2BCD^2 + ACD^2 - 2ABCD + 2A^2 BD \quad (B6)$$

$$a_5 = -2ABCD - 2AC^2 D + 2A^2 CD - A^2 BD - 2BCD^2 + 2ABD^2 + BC^2 D - 2AB^2 D$$

$$a_6 = C^2 D^2 + A^2 C^2 + A^2 B^2 + 2AC^2 D - 2A^2 CD - ABC^2 - A^3 B$$

$$a_7 = -2AC^2 D - 4A^2 C^2 + ABC^2 + A^3 B - BC^2 D - A^2 BD + 2A^2 BC$$

$$a_8 = 2A^2 C^2 + AC^3 + A^3 C$$

Equation (B5) is a polynomial in one variable with real coefficients and odd degree, so it must have at least one real root [12]. From (B5) the root  $m_{root}$  can be obtained, then the corresponding  $n_{root}$  can be obtained based on equation (B4). Note  $(m_{root}, n_{root})$  may not be integer solution. Then the nearest integer value will be used as the best solution for the convex function  $T_{2 \leq m \leq P}$ , that is,  $(m_e, n_e) = (\lceil m_{root} \rceil, \lceil n_{root} \rceil)$ .

In the case  $m_e$  is outside the range  $[2, P]$ , boundaries  $m_{b2} = 2$  and  $m_{bP} = P$  are checked for smaller  $T$ , and according to (B4):

$$n_{b2} = \left\lceil \sqrt{\frac{2B + C}{A - 2D}} \right\rceil \text{ and } n_{bP} = \left\lceil \sqrt{\frac{B \cdot P^2 + (C - B) \cdot P - C}{A - D \cdot P}} \right\rceil$$

So combining the two cases:  $m=1$ , the optimal solution for it is  $(1,1)$ ; and  $m \in [2,P]$ , the optimal solution is  $(m_e, n_e)$  if it exists in  $[2,P]$ , otherwise either  $(2, n_{b2})$  or  $(P, n_{bp})$ . Hence the optimal solution  $(m_1, n_1)$  for  $m \in [1,P]$  is obtained as summarized in (B1).

Note equation (B5) may have up to seven roots because some extraneous roots may be introduced in during process. When multiple roots are produced, all roots are analyzed, the valid root must be in range  $[2,P]$  and satisfies both equation (B2)  $T'_m=0$  and equation (B3)  $T'_n=0$ .

### 9.2.2 Lemma B2

In solution subspace  $S_2 = \{m: m=kP, 1 \leq k, m \leq M\}$ ,  $T$  defined in (10) takes minimum

$$\text{at } (m_2, n_2) = (P, \left\lceil \sqrt{\frac{(BP+C)(P-1)}{A-DP}} \right\rceil).$$

Proof: From (7) in section 4, and (12) in section 5:

$$T_{m=kP} = \left( \frac{A}{m} + \frac{B}{n} + \frac{C}{mn} - D \right) (n+P-1) \left( \frac{m}{P} \right) = \left( A + \frac{B}{n}m + \frac{C}{n} - Dm \right) (n+P-1) \left( \frac{1}{P} \right) \quad (\text{A7})$$

Take the partial derivative with respect to  $m$ ,  $T'_{m,m=kP} = \left( \frac{B}{n} - D \right) (n+P-1) \left( \frac{1}{P} \right) > 0$ , so the smaller  $m$  is, the smaller  $T$  will be. Since  $m=kP$ , then  $k=1$  makes  $m$  the smallest, so  $m_2=P$ .  $n_2$  is obtained by letting  $m=P$  and (B3)=0:  $n_2 = \sqrt{\frac{(BP+C)(P-1)}{A-DP}}$ . Thus:

$$m_2 = P, n_2 = \left\lceil \sqrt{\frac{(BP+C)(P-1)}{A-DP}} \right\rceil \quad (\text{B8})$$

### 9.2.3 Lemma B3

For any  $m$  in solution subspace  $S_3 = \{m : m = kP + r, 1 \leq k, m \leq M, 1 \leq r \leq P-1\}$ ,  $T_{m=kP+r}$  defined in (10) is always greater than  $T_{m=P}$  in  $S_1$ , that is:  $T_{m=kP+r} > T_{m=P}$ .

Proof: from (9) in section 4 and (12) in section 5:

$$\begin{aligned} T_{m=kP+r} &= t_{sn} \left( \frac{m-r}{P} + 1 \right) (n+r-1) = \left( \frac{A}{kP+r} + \frac{B}{n} + \frac{C}{(kP+r)n} - D \right) (k+1)(n+r-1) \\ &= \frac{HP(k+1)(r+G)}{(kP+r)P} + E(k+1)(r+G) \end{aligned} \quad (\text{B9})$$

where  $E = \frac{B}{n} - D$ ,  $G = n-1$ , and  $H = A + F$ , also let  $F = \frac{C}{n}$ .

From (B7) and let  $m = P$ :

$$T_{m=P} = \left( \frac{A}{m} + \frac{B}{n} + \frac{C}{mn} - D \right) (n+P-1) \left( \frac{m}{P} \right) = \frac{(HP+HG)(kP+r)}{P(kP+r)} + EP + EG \quad (\text{B10})$$

Subtract (B10) from (B9):

$$\begin{aligned} T_{m=kP+r} - T_{m=P} &= \frac{HPkr + HPkG + HPr + HPG - HPkP - HPr - HGkP - HGr}{P(kP+r)} \\ &\quad + Ekr + EkG + Er + EG - EP - EG \\ &= \frac{H(P-r)(n-1-kP)}{P(kP+r)} + Ekr + Ek(n-1) + Er - EP \quad \text{since } G = n-1 \\ &\geq \frac{H(P-r)(m-1-kP)}{P(kP+r)} + Ekr + Ek(m-1) + Er - EP \quad (n \geq m = kP+r) \end{aligned}$$



$$= \frac{H(P-r)(kP+r-1-kP)}{P(kP+r)} + Ekr + Ek(kP+r-1) + Er - EP$$

Since  $\frac{H(P-r)(kP+r-1-kP)}{P(kP+r)} \geq 0$ ,  $Ekr + Ek(kP+r-1) + Er - EP > 0$ , hence

$$T_{m=kP+r} > T_{m=P}.$$

Thus Lemma B3 is proven.

## 9.3 Appendix C: Source Code of Supernode Transformation on Computer Cluster Systems

Appendix C contains source code implementing the LCS problem running on computer cluster systems. There are three parts: code running on the master computer, code running on the computing nodes, and the include file used by both codes.

### 9.3.1 Code Running on the Master Computer

The master computer drives the entire work flow. It manages the cluster, handles the synchronizations between the computing nodes. It first initializes all computing nodes by starting the client code on them, then connects to each and every computing nodes via socket and sends the X and Y sequences to them. After that, it follows the execution model explained in section 4, sends supernodes' dependent data to computing nodes to do LCS processing and receives the resultant data. It continues this process until the last wavefront is processed. The computation time and the communication time are recorded for analysis using the mathematical model obtained in section 4.

Following is the source code for the code running on the master computer MasterComputer.c

```
#include "sc.h"
```

```
int client_socket[NUM_OF_PROCESSOR];
```

```
int client_ready[NUM_OF_PROCESSOR];
```

```
double each_calc_cpu;
```

```

double each_com_time;

uint16_t LCS[M][N];

char X[M+1], Y[N+1];

/**
 * master node creates computing node
 * param i is the core id that the computing node (process) will
 * reside on the new process runs computing node image (cnode)
 */
int create_process(int i) {
    char buf[12];

    int p = fork();

    if (p < 0) {
        perror("fork failed: ");
        exit(-1);
    } else if (p == 0) { //child process
        snprintf(buf, sizeof(buf), "%d", i);
        execlp("./cnode", buf, (char *)NULL);

        perror("should not come here: ");
    } else { //parent process
        return p;
    }
}

/**

```

```

* populating a sequence (of LCS) with random values
**/
void _populate_seq(char *seq, int size) {

    char *ref = "abcdefghijklmnopqrstuvwxyz\
ABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890";

    int len = strlen(ref);

    int pos, i;

    for (i=0; i<size; i++) {

        pos = rand() % len;

        seq[i] = ref[pos];

    }

}

/**
* reset all clients' status to NOT READY
**/
void reset_clients() {

    int i;

    for (i=0; i<NUM_OF_PROCESSOR; i++)

        client_ready[i] = FALSE;

}

/**
* check if all clients are in READY state
**/

```

```

int clients_ready() {
    int i;
    for (i=0; i<NUM_OF_PROCESSOR; i++)
        if (client_ready[i] == FALSE) {
            return FALSE;
        }
    return TRUE;
}

/**
 * process computing node 'i' by first receiving data then processing
 * it. The expected data should be MSG_INIT1, or MSG_INIT2, or MSG_DATA
 * MSG_INIT1 means client is initialized, MSG_INIT2 indicates client has
 * received X and Y sequences.
 *
 * client is marked as READY when done.
 */
void process_client(int i, int node_w, int node_h) {
    int bytes_read, ii, jj;
    msg read_buf;
    struct timespec end;

    bytes_read = recv(client_socket[i], &read_buf, sizeof(msg), MSG_WAITALL);
    if (bytes_read == 0) {
        close(client_socket[i]);
    }
}

```

```

    client_socket[i] = 0;

    return;
}

msg *m = (msg *)&read_buf;

if (m->type == MSG_INIT1 || m->type == MSG_INIT2) {

    client_ready[i] = TRUE;

} else if (m->type == MSG_DATA) {

    clock_gettime(CLOCK_MONOTONIC, &end);

    if (each_calc_cpu < m->each_calc_cpu) {

        each_calc_cpu = m->each_calc_cpu; //use the largest one

    }

    double diffcom = (end.tv_sec - m->startw.tv_sec)*1000000 +

        (double)((end.tv_nsec - m->startw.tv_nsec)/(double)1000); //micro secs

    if (each_com_time < diffcom ) {

        each_com_time = diffcom; //use the largest one

    }

    if (m->base_x >= 0 && m->base_x < M &&

        m->base_y >= 0 && m->base_y < N) {

        for (ii=0; ii<m->width; ii++) {

            memcpy(&LCS[m->base_x+ii][m->base_y],

                &m->data.reply[ii][0], m->height*sizeof(uint16_t));

        }

    }

}

```

```

    }

    client_ready[i] = TRUE;

}

}

/**
 * signal handler for SIGINT and SIGTERM, make sure all child processes
 * (computing nodes) are terminated.
 **/
void sighandler(int signum) {

    kill(0, SIGUSR1);

}

/**
 * the main function which controls the entire work flow,
 * by initializing the computing nodes, send/receive data
 * to/from computing nodes and provide synchronization
 * between all computing nodes.
 **/
int main(int argc, char *argv[]) {

    int w, h, k, l, client, ii, i;

    int opt = TRUE;

    int master_socket, addrlen, new_socket;

    int max_clients = NUM_OF_PROCESSOR, activity, sd;

    int max_sd;

    struct sockaddr_in address;

```

```

fd_set readfds;

set_affinity(0);

signal(SIGINT, sighandler);

signal(SIGTERM, sighandler);

_populate_seq((char *)X, M);

_populate_seq((char *)Y, N);


for (i=0; i<max_clients; i++) {

    client_socket[i] = 0;

    client_ready[i] = FALSE;

}

if ((master_socket=socket(AF_INET, SOCK_STREAM, 0)) == 0) {

    perror("master socket error: ");

    exit(-1);

}


if (setsockopt(master_socket, SOL_SOCKET, SO_REUSEADDR, (char *)&opt,
sizeof(opt)) < 0){

    perror("set socket:");

    exit(-1);

}

```



```

address.sin_family = AF_INET;

address.sin_addr.s_addr=INADDR_ANY;

address.sin_port=htons(PORT);


if (bind(master_socket, (struct sockaddr*)&address, sizeof(address)) < 0) {

    perror("bind error: ");

    exit(-1);

}


if (listen(master_socket, 2*NUM_OF_PROCESSOR) < 0) {

    perror("listen error: ");

    exit(-1);

}


addrlen = sizeof(address);

printf("waiting for incoming connection...\n");


for (i=0; i<max_clients; i++)

    create_process(i);

// first connecting to all computing nodes
while (TRUE) {
    FD_ZERO(&readfds);

    FD_SET(master_socket, &readfds);

```

```

max_sd = master_socket;

for (i=0; i<max_clients; i++) {

    sd = client_socket[i];

    if (sd > 0)

        FD_SET(sd, &readfds);

    if (sd > max_sd )

        max_sd = sd;

}

activity = select(max_sd+1, &readfds, NULL, NULL, NULL);

if ((activity < 0) && errno != EINTR)

    printf("select error: ");

if (FD_ISSET(master_socket, &readfds)) {

    new_socket = accept(master_socket, (struct sockaddr *)&address,
(socklen_t*)&addrlen);

    if (new_socket < 0) {

        perror ("accept error: ");

        exit(-1);

    }

    for(i=0; i<max_clients; i++) {

        if (client_socket[i] == 0) {

```

```

        client_socket[i] = new_socket;

        break;

    }

}

}

for (i=0; i<max_clients; i++)
{
    sd = client_socket[i];

    if (FD_ISSET(sd, &readfds)) {

        process_client(i, 0, 0);

    }

}

// if all computing nodes are connected

if (clients_ready() == TRUE)

    break;

}

reset_clients();

// now sending X and Y to clients

msg m1;

m1.type = MSG_INIT;

```

```

m1.width = M;

m1.height = N;

memcpy(m1.data.bl.bottom, X, M);

memcpy(m1.data.bl.left, Y, N);


for (client=0; client<max_clients; client++) {

    send(client_socket[client], &m1, sizeof(msg), 0);

}


while (TRUE ) {

    FD_ZERO(&readfds);

    FD_SET(master_socket, &readfds);

    max_sd = master_socket;

    for (i=0; i<max_clients; i++) {

        sd = client_socket[i];

        if (sd > 0)

            FD_SET(sd, &readfds);

        if (sd > max_sd)

            max_sd = sd;

    }

    activity = select(max_sd+1, &readfds, NULL, NULL, NULL);

    if ((activity < 0) && errno != EINTR) {

        printf("select() error: ");

```

```

    }

    for (i=0; i<max_clients; i++) {
        sd = client_socket[i];
        if (FD_ISSET(sd, &readfds)) {
            printf("received XY reply, i=%d\n", i);
            process_client(i, 0, 0); //need to send X and Y to clients
        }
    }

    if (clients_ready() == TRUE) {
        break;
    }
}

reset_clients();

// now start the computation, it computes all possible pairs of (w,h),
// thus an exhaustive testing approach.
for (w=1; w<=M; w++) {
    for (h=1; h<N; h++) {

        int m=M%w==0 ? M/w : (M/w+1);

        int n=N%h==0 ? N/h : (N/h+1);

```

```

msg m1;

m1.type = MSG_DATA;

m1.width = w;

m1.height = h;


double Tcpu = 0;

double Tcom = 0;


if (m >= n) {

    for (k=0; k<(m+n-1); k++) {

        int wavelength;

        if (k < n)

            wavelength = k+1;

        else if (k>=n && k<m)

            wavelength = n;

        else

            wavelength = n-(k-m)-1;

        int num_of_seg = wavelength%max_clients==0 ? wavelength/max_clients :
(wavelength/max_clients+1);


        for (l=0; l<num_of_seg; l++) {

            for (client=0; client<max_clients; client++) {

```

```

        if (k < (n-1)) {
m1.base_x = (k-l*max_clients-client)*w;
m1.base_y = (l*max_clients+client)*h;

        } else if (k>=(n-1) && k<(m-1)) {
m1.base_x = (k-l*max_clients-client)*w;
m1.base_y = (l*max_clients+client)*h;

        } else { // k>=m-1 && k<n+m-1
m1.base_x = (m-1-l*max_clients-client)*w;
m1.base_y = (k-(m-1)+l*max_clients+client)*h;

        }

m1.width = w;
m1.height = h;

if (m1.base_x >= 0 && m1.base_x < M && m1.base_y >= 0 &&
m1.base_y < N) {

    m1.width = (m1.base_x+m1.width)>M ? (M-m1.base_x) : m1.width;
    m1.height = (m1.base_y+m1.height)>N ? (N-m1.base_y) : m1.height;

    if (m1.base_x == 0) {

        for (ii=0; ii<m1.height; ii++) {

            m1.data.bl.left[ii] = 0;

        }

    } else {

        for (ii=0; ii<m1.height; ii++) {

            m1.data.bl.left[ii]=LCS[m1.base_x-1][m1.base_y+ii];

```

```

    }
}

if (m1.base_y == 0) {
    for (ii=0; ii<m1.width; ii++) {
        m1.data.bl.bottom[ii] = 0;
    }
} else {
    for (ii=0; ii<m1.width; ii++) {
        m1.data.bl.bottom[ii] = LCS[m1.base_x+ii][m1.base_y-1];
    }
}

if (m1.base_x!=0 && m1.base_y!=0) {
    m1.leftbottom = LCS[m1.base_x-1][m1.base_y-1];
}

clock_gettime(CLOCK_MONOTONIC, &m1.startw);

send(client_socket[client], &m1, sizeof(msg), 0);
}

each_calc_cpu = 0;

each_com_time = 0;

```



```

while (TRUE ) {

    FD_ZERO(&readfds);

    FD_SET(master_socket, &readfds);

    max_sd = master_socket;

    for (i=0; i<max_clients; i++) {

        sd = client_socket[i];

        if (sd > 0)

            FD_SET(sd, &readfds);

        if (sd > max_sd)

            max_sd = sd;

    }

    activity = select(max_sd+1, &readfds, NULL, NULL, NULL);

    if ((activity < 0) && errno != EINTR) {

        printf("select err:");

        exit(-1);

    }


    for (i=0; i<max_clients; i++) {

        sd = client_socket[i];

        if (FD_ISSET(sd, &readfds)) {

            process_client(i, w, h);

        }

    }

}

```

```

    }

    if (clients_ready() == TRUE) {
        reset_clients();
        break;
    }
}

Tcpu += each_calc_cpu;
Tcom += each_com_time;
}
}

} else { //m<n
    for (k=0; k<(m+n-1); k++) {
        int wavelength;
        if (k < m)
            wavelength = k+1;
        else if (k>=m && k<n)
            wavelength = m;
        else //k>=n
            wavelength = m-(k-n)-1;

        int num_of_seg = wavelength%max_clients==0 ? wavelength/max_clients :

```

```

        (wavelength/max_clients+1);
for (l=0; l<num_of_seg; l++) {
    for (client=0; client<max_clients; client++) {
        if (k < (m-1)) {
            m1.base_x = (k-l*max_clients-client)*w;
            m1.base_y = (l*max_clients+client)*h;
        } else if (k>=(m-1) && k<(n-1)) {
            m1.base_x = (m-1-l*max_clients-client)*w;
            m1.base_y = (k-(m-1)+l*max_clients+client)*h;
        } else { // k>=n && k<=n+m-1
            m1.base_x = (m-1-l*max_clients-client)*w;
            m1.base_y = (k-(m-1)+l*max_clients+client)*h;
        }
    }

    if (m1.base_x >= 0 && m1.base_x < M && m1.base_y >= 0 &&
m1.base_y < N) {
        m1.width = (m1.base_x+m1.width)>M ? (M-m1.base_x) : m1.width;
        m1.height = (m1.base_y+m1.height)>N ? (N-m1.base_y) : m1.height;

        if (m1.base_x == 0) {
            for (ii=0; ii<m1.height; ii++) {
                m1.data.bl.left[ii] = 0;
            }
        }
    }
}

```

```

    } else {

        for (ii=0; ii<m1.height; ii++) {

            m1.data.bl.left[ii] = LCS[m1.base_x-1][m1.base_y+ii];

        }

    }

    if (m1.base_y == 0) {

        for (ii=0; ii<m1.width; ii++) {

            m1.data.bl.bottom[ii] = 0;

        }

    } else {

        for (ii=0; ii<m1.width; ii++) {

            m1.data.bl.bottom[ii] = LCS[m1.base_x+ii][m1.base_y-1];

        }

    }

    if (m1.base_x!=0 && m1.base_y != 0) {

        m1.leftbottom = LCS[m1.base_x-1][m1.base_y-1];

    }

    clock_gettime(CLOCK_MONOTONIC, &m1.startw);

    send(client_socket[client], &m1, sizeof(msg), 0);

}

```

```

each_calc_cpu = 0; //biggest

each_com_time = 0;


while (TRUE ) {

    FD_ZERO(&readfds);

    FD_SET(master_socket, &readfds);

    max_sd = master_socket;

    for (i=0; i<max_clients; i++) {

        sd = client_socket[i];

        if (sd > 0)

            FD_SET(sd, &readfds);

        if (sd > max_sd)

            max_sd = sd;

    }

    activity = select(max_sd+1, &readfds, NULL, NULL, NULL);

    if ((activity < 0) && errno != EINTR)

        printf("select err:");


    for (i=0; i<max_clients; i++)

    {

        sd = client_socket[i];

        if (FD_ISSET(sd, &readfds)) {

```

```

        process_client(i, w, h);
    }
}

    if (clients_ready() == TRUE) {
        reset_clients();
        break;
    }
}

    Tcpu += each_calc_cpu;
    Tcom += each_com_time;
}
}
}

    printf("Tcom is %f, Tcpu=%f, total time is %f, w=%d, h=%d\n", Tcom, Tcpu,
Tcom+Tcpu, w, h);

    fflush(NULL);
}
}

    kill(0, SIGUSR1);
}

```

### 9.3.2 Code Running on the Computing Nodes

The code running on the computing nodes acts as client code in client/server model, it is driven by the master computer. It receives dependent data from the master computer, process the data based on the LCS problem logic, and then sends the resultant data back to the master computer. It continues this logic until it receives the KILL signal from the master computer indicating the completion of the entire process.

Following is the source code ComputingNode.c.

```
/**
 * the code on computing nodes
 */
#include "sc.h"

char d_X[M+1], d_Y[N+1];

msg m_reply, *m;

int my_client_id = 0;

int main(int argc, char *argv[]) {

    int sock, bytes_read;

    struct sockaddr_in server;

    msg server_data;

    int ii, jj;

    struct timespec start, end;

    double tr; // comm cost from server to this client
```

```

if (argc == 1)

    my_client_id = (int)(*argv[0]-'0');

else {

    printf("Please specify client id on command line.\n");

    exit(-1);

}


set_affinity(my_client_id + 1);

sock = socket(AF_INET, SOCK_STREAM, 0);

if (sock == -1) {

    perror("socket error:");

    exit(-1);

}


// “127.0.0.1” indicating the local host, in cluster, the real IP address of
// the master computer should be used.
server.sin_addr.s_addr = inet_addr("127.0.0.1");

server.sin_family = AF_INET;

server.sin_port = htons(PORT);

if (connect(sock, (struct sockaddr *)&server, sizeof(server)) < 0) {

    perror("connect failed: ");

    exit(-1);

}

```



```

// tell server we are ready

m_reply.type = MSG_INIT1;

send(sock, &m_reply, sizeof(m_reply), 0);


while (1) { // while loop, program is terminated via server sending KILL signal

    if ((bytes_read=recv(sock, &server_data,sizeof(msg),MSG_WAITALL)) < 0) {

        perror("received error:");

        exit(-1);

    }


    if (bytes_read != sizeof(msg)) {

        sleep(1);

        continue;

    }


    m = (msg *)&server_data;

    clock_gettime(CLOCK_MONOTONIC, &start);


    if (m->type == MSG_INIT) { //receive X and Y sequences

        memcpy(d_X, m->data.bl.bottom, m->width);

        d_X[m->width] = '\0';

        memcpy(d_Y, m->data.bl.left, m->height);

        d_Y[m->height] = '\0';

```

```

m_reply.type = MSG_INIT2;
} else if (m->type == MSG_DATA) {

tr = (start.tv_sec - m->startw.tv_sec)*1000000000 +

        (start.tv_nsec - m->startw.tv_nsec);

m_reply.type = MSG_DATA;

m_reply.width = m->width;

m_reply.height = m->height;

m_reply.base_x = m->base_x;

m_reply.base_y = m->base_y;


if (m->base_x>=0 && m->base_x<M && m->base_y>=0 && m->base_y<N) {

    for (jj=0; jj<m->height; jj++) {

        for (ii=0; ii<m->width; ii++) {

            if (m->base_x==0 && ii==0){

                m_reply.data.reply[ii][jj] = 0;

            } else if (m->base_y == 0 && jj == 0) {

                m_reply.data.reply[ii][0] = 0;

            } else if (d_X[m->base_x+ii] == d_Y[m->base_y+jj]) {

                int leftbottom;

                if (ii == 0) {

                    if (jj == 0) {

                        leftbottom = m->leftbottom;

                    } else {


```

```

        leftbottom = m->data.bl.left[jj-1];

    }
} else if (jj == 0) {

    if (ii == 0) {

        leftbottom = m->leftbottom;

    } else

        leftbottom = m->data.bl.bottom[ii-1];

} else {

    leftbottom = m_reply.data.reply[ii-1][jj-1];

}

m_reply.data.reply[ii][jj] = 1 + leftbottom;

} else {

    int left, bottom;

    if (ii == 0)

        left = m->data.bl.left[jj];

    else

        left = m_reply.data.reply[ii-1][jj];

    if (jj==0) {

        bottom = m->data.bl.bottom[ii];

    } else

        bottom = m_reply.data.reply[ii][jj-1];

```

```

        m_reply.data.reply[ii][jj] = left>bottom?left:bottom;
    }
}
}
}
}

clock_gettime(CLOCK_MONOTONIC, &end);

double diffcpu = (end.tv_sec - start.tv_sec)*1000000 +
    (double)((end.tv_nsec - start.tv_nsec)/(double)1000); //micro secs

m_reply.each_calc_cpu = diffcpu;

clock_gettime(CLOCK_MONOTONIC, &m_reply.startw);

m_reply.startw.tv_nsec -= tr; //add 'send' comm cost

send(sock, &m_reply, sizeof(msg), 0);
}
}

```

### 9.3.3 Include File

The include file defines some commonly used data structures and functions, it is included and used by both ComputingNode.c and MasterComputer.c.

```

#include <stdio.h>

#include <string.h>

```

```
#include <stdlib.h>

#include <errno.h>

#include <unistd.h>

#include <arpa/inet.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <time.h>

#include <sys/time.h>

#include <limits.h>


#define _GNU_SOURCE

#include <sched.h>

#include "signal.h"


#define TRUE      1

#define FALSE     0

#define PORT      8319


#define NUM_OF_CORE 6


#define M      600

#define N      1200
```

```
#define MSG_SIZE (M*N+32)
```

```
#define MSG_READY 0
```

```
#define MSG_DATA 1
```

```
#define MSG_INIT 2
```

```
#define MSG_INIT1 3
```

```
#define MSG_INIT2 4
```

```
typedef struct _bottomleft {
```

```
    uint16_t bottom[M + 1];
```

```
    uint16_t left[N + 1];
```

```
} bottomleft;
```

```
// the message data structure, between server and client
```

```
typedef struct _msg {
```

```
    int type;
```

```
    int base_x;
```

```
    int base_y;
```

```
    int width; //width of the array
```

```
    int height; //height of the array
```

```
    int leftbottom;
```

```
    double each_calc_cpu;
```

```
    struct timespec startw;
```

```

union {

    uint16_t reply[M + 1][N + 1];

    bottomleft bl;

} data;
} msg;

/**
 * calling Linux sched_getaffinity() to get a core to work on
 * returns 0 if successful
 * return -1 if failed
 */
int set_affinity(int which) {

    cpu_set_t set, mask;

    int i;

    CPU_ZERO(&set);

    CPU_SET(which, &set);

    printf(" setting to core %d\n", which);

    if (sched_setaffinity(getpid(), sizeof(cpu_set_t), &set)) {

        printf(" client %d sched failed:\n", which);

        perror(" client sched failed");

        return -1;

    }

```

```

CPU_ZERO_S(sizeof(cpu_set_t), &mask);

if (sched_getaffinity(0, sizeof(cpu_set_t), &mask) == -1) {
    perror("can't get it:");
    return -1;
}

if (!CPU_ISSET_S(which, sizeof(cpu_set_t), &mask)) {
    printf(" it's NOT on %d\n", which);
    return -1;
}

return 0;
}

```



## 9.4 Appendix D: Code on GPGPUs.

GPGPU programming is different from the traditional programming. A GPGPU program is divided into two parts: one running on the host CPU and the other running on GPU. The host program drives the entire work flow, sets up the data, then launches the GPU code called kernel for fast and parallel processing on GPU SMs.

Following is the LCS.cu code for running the LCS program on GPGPU, it following the execution model explained in section 4 to process the supernodes. Nvidia's CUDA development kit 6.0 is used in the code.

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <inttypes.h>


#include <cuda.h>


#define NUM_OF_BLOCK 30

#define NUM_OF_THREAD 1024


// problem size

#define M 600

#define N 1200


typedef struct {
```

```

uint16_t width;

uint16_t height;

uint16_t *elements;
} Matrix;


char X[M];

char Y[N];


__global__ void lcs_kernel(char *d_X, char *d_Y, int w, int h, int k, int l, Matrix d_L,
unsigned long long int *d_T, int m, int n, int p, int width, int height);


void _populate_seq(char *xy, int size) {
    char *ref = "abcdefghijklmnopqrstuvwxyz\
ABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890";

    int len = strlen(ref);

    int pos;

    for (int i=0; i<size; i++) {
        pos = rand()%len;

        xy[i] = ref[pos];
    }
}

```

```

//main program -- Host code

void main_program(int num_of_block)
{
    int w, h, mmin, nmin;

    int k, l;

    char *device_X, *device_Y;

    Matrix L, device_L;

    unsigned long long int *T, *d_T, Tmin, Tused, Tused0, Tmem_min, Tmem, Tmem0;

    //invoke kernel

    dim3 dimGrid(num_of_block);

    _populate_seq(X, M);

    _populate_seq(Y, N);


    T = (unsigned long long int *)malloc(2*num_of_block*sizeof(unsigned long long int));
    cudaMalloc((void **)&d_T, 2*num_of_block*sizeof(unsigned long long int));
    cudaMalloc((void **)&device_X, M);
    cudaMemcpy(device_X, X, M, cudaMemcpyHostToDevice);
    cudaMalloc((void **)&device_Y, N);
    cudaMemcpy(device_Y, Y, N, cudaMemcpyHostToDevice);


    L.width = M;

    L.height = N;

```

```

L.elements = (uint16_t *)malloc(L.width*L.height*sizeof(uint16_t));

device_L.width = M;

device_L.height = N;

cudaMalloc((void **)&device_L.elements,
device_L.width*device_L.height*sizeof(uint16_t));

```

```

Tmin = 0;

```

```

Tmem_min = 0;

```

```

for (h=1; h<=N; h++) {
    for (w=1; w<M; w++) {

```

```

        if ((w*h)>20000)

```

```

            continue; //GTX 760 shared memory per block is 48KB.

```

```

        if (w > 1024 && h > 1024)

```

```

            continue; // max 1024 threads per block

```

```

        int m = M%w==0 ? M/w : (M/w+1);

```

```

        int n = N%h==0 ? N/h : (N/h+1);

```

```

        dim3 dimBlock(1024);

```

```

        Tused = 0; Tmem = 0;

```

```

if (m >= n) {
    for (k=0; k<(m+n-1); k++) {
        int wavelength;
        if (k < n)
            wavelength = k + 1;
        else if (k >= n && k < m)
            wavelength = n;
        else
            wavelength = n-(k-m)-1;

        int num_of_seg = wavelength%num_of_block==0 ?
wavelength/num_of_block : (wavelength/num_of_block + 1);

        for (l=0; l<num_of_seg; l++) {

            lcs_kernel<<<dimGrid, dimBlock,
sizeof(uint16_t)*(w*h+w+h+w+1+h+1)>>>(device_X, device_Y, w, h, k, l, device_L,
d_T, m, n, num_of_block, M, N);

            cudaError_t cudaerr = cudaDeviceSynchronize();

            if (cudaerr != CUDA_SUCCESS) {
                exit(-1);
            }
        }
    }
}

```

```

        cudaMemcpy((void *)T, (void *)d_T, 2*num_of_block*sizeof(unsigned
long long int), cudaMemcpyDeviceToHost);

        Tused0 = 0;

        Tmem0 = 0;

        for (int j=0; j<num_of_block; j++) {

            if (Tused0 < T[j]) {

                Tused0 = T[j];

            }

            if (Tmem0 < T[j+num_of_block]) {

                Tmem0 = T[j+num_of_block];

            }

        }

        Tused += Tused0;

        Tmem += Tmem0;

    }

}

        cudaMemcpy((void *)&L.elements[M*N-1], (void
*)&device_L.elements[M*N-1], sizeof(uint16_t), cudaMemcpyDeviceToHost);

    } else { //m<n

        for (k=0; k<(m+n-1); k++) {

            int wavelength;

```

```

if (k < m)

    wavelength = k+1;

else if (k >= m && k < n)

    wavelength = m;

else //k>=n

    wavelength = m-(k-n)-1;


int num_of_seg = wavelength%num_of_block==0 ?
wavelength/num_of_block : (wavelength/num_of_block+1);

for (l=0; l<num_of_seg; l++) {

    lcs_kernel<<<dimGrid, dimBlock,
sizeof(uint16_t)*(w*h+w+h+w+1+h+1)>>>(device_X, device_Y, w, h, k, l, device_L,
d_T, m, n, num_of_block, M, N);

    cudaError_t cudaerr = cudaDeviceSynchronize();

    if (cudaerr != CUDA_SUCCESS) {

        exit(-1);

    }

    cudaMemcpy((void *)T, (void *)d_T, 2*num_of_block*sizeof(unsigned
long long int), cudaMemcpyDeviceToHost);

    Tused0 = 0;

    Tmem0=0;

    for (int j=0; j<num_of_block; j++) {

        if (Tused0 < T[j]) {

```

```

        Tused0 = T[j];

    }

    if (Tmem0 < T[j+num_of_block]) {

        Tmem0 = T[j+num_of_block];

    }

}

Tused += Tused0;

Tmem += Tmem0;

}

}

    cudaMemcpy((void *)&L.elements[M*N-1], (void
*)&device_L.elements[M*N-1], sizeof(uint16_t), cudaMemcpyDeviceToHost);

}

if (Tmin == 0 || Tmin > Tused){

    Tmin = Tused;

    mmin = m;

    nmin = n;

    Tmem_min = Tmem;

}

printf("w=%d,h=%d,LCS=%hu Tused=%llu, Tmin=%llu, Tmem=%llu,
mmin=%d, nmin=%d\n",

```



```

        w, h, L.elements[M*N-1], Tused, Tmin, Tmem_min, mmin, nmin);

    fflush(NULL);

}

}

cudaFree(device_L.elements);

free(L.elements);

}

// one thread only handles one (x,y) in one wavefront at one time, but it's in a loop
__device__ unsigned long long int
one_thread_calculation(int k, int l, char *d_X, char *d_Y, int base_x, int base_y, int w,
int h, Matrix d_L, uint16_t *A, int width, int height, unsigned long long int *d_T, int p) {

    int i;

    int my_delta_x;

    int my_delta_y;

    uint16_t *a_bottom, *a_left;

    char *a_X, *a_Y;

    // copy global memory to A

    // A is wxh, then w+1, h+1 for d_L values, then w+h for d_X and d_Y

    a_bottom = &A[w*h];

    a_left = &A[w*h+w+1];

```

```
a_X = (char *)&A[w*h+w+1+h+1];
```

```
a_Y = (char *)&A[w*h+w+1+h+1+w];
```

```
int thread_id = threadIdx.x;
```

```
if (base_x < width && (base_y + thread_id) < height) {
```

```
    int len = (base_x + w) >= width ? (width - base_x) : w;
```

```
    int hh = (base_y + h) >= height ? (height - base_y) : h;
```

```
    int counter = 0;
```

```
    while (true) {
```

```
        int pos = counter*NUM_OF_THREAD + threadIdx.x;
```

```
        if (pos < len) {
```

```
            a_X[pos] = d_X[base_x+pos];
```

```
            counter ++;
```

```
        } else
```

```
            break;
```

```
    }
```

```
    counter = 0;
```

```
    while (true) {
```

```
        int pos = counter*NUM_OF_THREAD + threadIdx.x;
```

```
        if (pos < hh) {
```

```
            a_Y[pos] = d_Y[base_y + pos];
```

```

        counter++;

    } else

        break;

}

if (base_x==0 && base_y==0) {

    if (thread_id==0) {

        memset(a_bottom, 0, len + 1);

        memset(a_left, 0, hh + 1);

    }

} else if (base_x != 0 && base_y != 0) {

    int pos = (base_y-1)*width + base_x - 1;

    counter = 0;

    while (true) {

        int pos3 = counter*NUM_OF_THREAD + threadIdx.x;

        if (pos3 < (len + 1)) {

            a_bottom[pos3] = d_L.elements[pos + pos3];

            counter++;

        } else

            break;

    }

    counter=0;

```

```

while (true) {

    int pos3 = counter*NUM_OF_THREAD + threadIdx.x;

    int pos1 = (base_y-1+pos3)*width + base_x - 1;

    if (pos3 < (hh + 1)) {

        a_left[pos3] = d_L.elements[pos1];

        counter ++;

    } else

        break;

}

} else if (base_x == 0) { // then base_y !=0

    int pos = (base_y-1)*width;

    counter = 0;

    while (true) {

        int pos3 = counter*NUM_OF_THREAD + threadIdx.x;

        if (pos3 < len) {

            a_bottom[1+pos3] = d_L.elements[pos + pos3];

            counter ++;

        } else

            break;

    }

    a_bottom[0] = 0; //actually we may not need it

```

```

    memset(a_left, 0, hh + 1);
} else { // base_y==0 and base_x !=0
    memset(a_bottom, 0, len + 1);
    a_left[0] = 0;

    counter = 0;
    while (true) {
        int pos3 = counter*NUM_OF_THREAD + threadIdx.x;
        int pos1 = pos3*width + base_x - 1;
        if (pos3 < hh) {
            a_left[1+pos3] = d_L.elements[pos1];
            counter ++;
        } else
            break;
    }
}

__syncthreads();

if (w >= h) {
    for (i=0; i<(w+h-1); i++) { //each mini wavefront is done by all threads, each thread
handles one point

```

```

// now get each thread's position for calculation

if (i < h) {

    my_delta_x = i - threadIdx.x;

    my_delta_y = threadIdx.x;

} else if (i >= h && i < w) {

    my_delta_x = i - threadIdx.x;

    my_delta_y = threadIdx.x;

} else { //i >= w and i < (w+h-1)

    my_delta_x = w - 1 - threadIdx.x;

    my_delta_y = i - (w-1) + threadIdx.x;

}

if (my_delta_x < w && my_delta_x >= 0 && my_delta_y < h && my_delta_y >=
0 &&

    (my_delta_x + base_x) >= 0 && (my_delta_x + base_x) < width &&

    (my_delta_y + base_y) >= 0 && (my_delta_y + base_y) < height) { //so some
threads may be idle

    if (base_x == 0 && my_delta_x == 0) {

        A[my_delta_y*w] = 0;

    }

    else if (base_y == 0 && my_delta_y == 0) {

        A[my_delta_x] = 0;

```

```

}

else {

    char d_x_b = a_X[my_delta_x];

    char d_y_l = a_Y[my_delta_y];

    if (d_x_b == d_y_l) {

        int leftbottom;

        if (my_delta_x == 0 && my_delta_y == 0)

            leftbottom = a_bottom[0];

        else if (my_delta_x == 0)

            leftbottom = a_left[my_delta_y];

        else if (my_delta_y == 0)

            leftbottom = a_bottom[my_delta_x];

        else

            leftbottom = A[(my_delta_y-1)*w + my_delta_x-1];

        A[my_delta_y*w+my_delta_x] = 1 + leftbottom;

    } else {

        int left, bottom;

        if (my_delta_x == 0) {

            left=a_left[my_delta_y + 1]; //left value

```

```

    }

    else

        left = A[my_delta_y*w + my_delta_x - 1];

        if (my_delta_y == 0) {

            bottom = a_bottom[my_delta_x + 1]; //bottom value

        }

        else

            bottom = A[(my_delta_y-1)*w + my_delta_x];

        A[my_delta_y*w + my_delta_x] = left > bottom ? left : bottom;

    }

}

}

__syncthreads(); //wait till all threads are done for this wavefront in w*h
}

//copy A back to d_L
if (base_x >= 0 && base_x < width && base_y >= 0 && base_y < height) {

    int cpy_count;

    if ((base_x+w) < width)

        cpy_count = w;

    else

```



```

    cpy_count = width - base_x;

    int jj;
    for (jj=0; jj<h; jj++) {
        if ((jj+base_y) >= height)
            break;

        int counter = 0;
        while (true) {
            int d_L_pos = (base_y+jj)*M + base_x;
            int A_pos = jj*w;
            int pos3 = counter*NUM_OF_THREAD + threadIdx.x;
            if (pos3 < cpy_count) {
                d_L.elements[d_L_pos+pos3] = A[A_pos+pos3];
                counter ++;
            } else
                break;
        }
    }

}

__syncthreads(); //wait till all threads are done

```

```

} else { //h>w case, we have w threads

for (i=0; i<(w+h-1); i++) { //each mini wavefront is done by all threads

    // now get each thread's position for calculation

    if (i < w) {

        my_delta_x = i - threadIdx.x;

        my_delta_y = threadIdx.x;

    } else if (i >= w && i < h) {

        my_delta_x = w - 1 - threadIdx.x;

        my_delta_y = i - (w-1) + threadIdx.x;

    } else { //i>=h and i<(w+h-1)

        my_delta_x = w - 1 - threadIdx.x;

        my_delta_y = i - (w-1) + threadIdx.x;

    }

}

if (my_delta_x < w && my_delta_x >= 0 && my_delta_y < h && my_delta_y >=
0 &&

    (my_delta_x+base_x) >= 0 && (my_delta_x+base_x) < width &&

    (my_delta_y+base_y) >= 0 && (my_delta_y+base_y) < height) { //so some
threads may be idle

    if (my_delta_x == 0 && base_x == 0) {

        A[my_delta_y*w] = 0;

    }

    else if (my_delta_y == 0 && base_y == 0) {

```

```

    A[my_delta_x] = 0;
}
else {

    int d_x_b = a_X[my_delta_x];
    int d_y_l = a_Y[my_delta_y];

    if (d_x_b == d_y_l) {
        int leftbottom;

        if (my_delta_x == 0 && my_delta_y == 0)
            leftbottom = a_bottom[0];
        else if (my_delta_x == 0)
            leftbottom = a_left[my_delta_y];
        else if (my_delta_y == 0)
            leftbottom = a_bottom[my_delta_x];
        else
            leftbottom = A[(my_delta_y-1)*w+my_delta_x-1];

        A[my_delta_y*w+my_delta_x] = 1 + leftbottom;
    } else {
        int left, bottom;

        if (my_delta_x == 0) {
            left = a_left[my_delta_y + 1]; //left value

```

```

    }

    else

        left = A[my_delta_y*w + my_delta_x - 1];

        if (my_delta_y == 0) {

            bottom = a_bottom[my_delta_x + 1]; //bottom value

        }

        else

            bottom = A[(my_delta_y-1)*w + my_delta_x];

        A[my_delta_y*w+my_delta_x] = left > bottom ? left : bottom;

    }

}

}

__syncthreads(); //wait till all threads are done

}

//copy A back to d_L

//if (threadIdx.x == 0)

if (base_x >= 0 && base_x < width && base_y >= 0 && base_y < height) {

    int cpy_count;

    if ((base_x+w) < width)

        cpy_count = w;

```

```

else

    cpy_count = width - base_x;

int jj;

for (jj=0; jj<h; jj++) {

    if ((base_y+jj) >= height)

        break;

    int d_L_pos = (base_y+jj)*M+base_x;

    int A_pos = jj*w;

    int counter = 0;

    while (true) {

        int pos3 = counter*NUM_OF_THREAD + threadIdx.x;

        if (pos3 < cpy_count) {

            d_L.elements[d_L_pos+pos3] = A[A_pos+pos3];

            counter ++;

        } else

            break;

    }

}

__syncthreads();

}

```

```

    return 0;
}

__global__ void lcs_kernel(char *d_X, char *d_Y, int w, int h, int k, int l, Matrix d_L,
unsigned long long int *d_T, int m, int n, int p, int width, int height)
{
    unsigned long long int clock1=0, clock2=0, delta=0;

    extern __shared__ uint16_t A[]; //A should be w*h size, thread size is either w or h

    memset((void **)(d_T+blockIdx.x), 0, sizeof(unsigned long long int));
    memset((void **)(d_T+p+blockIdx.x), 0, sizeof(unsigned long long int));

    if (threadIdx.x == 0)
        clock1 = clock64();

    int block_id = blockIdx.x;

    // base_x, base_y is the lower left corner coordinate of the block in d_L
    int base_x;
    int base_y;

    if (m>=n) {
        if (k < (n-1)) {
            base_x = (k-l*p-block_id)*w;

```

```

    base_y = (l*p+block_id)*h;
} else if (k >= (n-1) && k < (m-1)) {
    base_x = (k-l*p-block_id)*w;
    base_y = (l*p+block_id)*h;
} else { // k>=m-1 && k<n+m-1
    base_x = (m-1-l*p-block_id)*w;
    base_y = (k-(m-1)+l*p+block_id)*h;
}
} else { //m<n
    if (k < (m-1)) {
        base_x = (k-l*p-block_id)*w;
        base_y = (l*p+block_id)*h;
    } else if (k >= (m-1) && k < (n-1)) {
        base_x = (m-1-l*p-block_id)*w;
        base_y = (k-(m-1)+l*p+block_id)*h;
    } else { // k>=n && k<=n+m-1
        base_x = (m-1-l*p-block_id)*w;
        base_y = (k-(m-1)+l*p+block_id)*h;
    }
}
}

if (base_x >= 0 && base_x < width && base_y >= 0 && base_y < height)

```

```
    one_thread_calculation(k, l, d_X, d_Y, base_x, base_y, w, h, d_L, A, width, height,  
d_T, p);
```

```
    __syncthreads(); //wait till all threads are done  
    if (threadIdx.x == 0) {  
        clock2 = clock64();  
        delta = clock2-clock1;  
        d_T[block_id]=delta;  
    }  
}
```

```
int main(int argc, char *argv[])  
{  
    int num_devices, d;  
    int num_of_core;  
    cudaDeviceProp deviceProp;  
  
    cudaGetDeviceCount(&num_devices);  
    printf("num of devices is %d\n", num_devices);  
    cudaGetDevice(&d);  
  
    cudaGetDeviceProperties(&deviceProp, d);
```



```

    printf("device %d has compute capability %d.%d.\n", d, deviceProp.major,
deviceProp.minor);

    printf("device %d multiProcessorCount is %d.\n", d, deviceProp.multiProcessorCount);

    printf("device name is %s, totalGlobalMem is %d\n", deviceProp.name,
deviceProp.totalGlobalMem);

    printf("device shared mem per block %d, regs per block %d\n",
deviceProp.sharedMemPerBlock, deviceProp.regsPerBlock);

    printf("device max threads per block %d, max grid size [%d %d %d]\n",
        deviceProp.maxThreadsPerBlock, deviceProp.maxGridSize[0],
        deviceProp.maxGridSize[1], deviceProp.maxGridSize[2]);

    num_of_core = deviceProp.multiProcessorCount;

    main_program(NUM_OF_BLOCK);

}

```

## 10. References

- [1]. E. Hodzic and W. Shang, "On supernode transformation with minimized total running time," *Parallel and Distributed Systems, IEEE Transactions on* (Volume :9, Issue:5), P417-428, May 1998.
- [2]. D. Hirschberg, "A linear space algorithm for computing maximal common subsequences," *Communications of the ACM*, Volume 18, Issue 6, P341-343, June 1975
- [3]. Nvidia CUDA Programming Guide 2.3, Nvidia Corporation, 2009.
- [4]. F. Irigoien and R. Triolet, "Supernode Partitioning," *Proceedings of the 15<sup>th</sup> ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, P319-329, San Diego, California, 1988.
- [5]. W. Shang and J. Fortes, "Time Optimal Linear Schedules for Algorithms with Uniform Dependencies," *IEEE Transactions on Computers*, Volume 40 Issue 6, June 1991. Page 723-742.
- [6]. J. Steinbrecher and W. Shang, "On Supernode Transformations And Multithreading For The Longest Common Subsequence Problem," *Proceedings of the Tenth Australasian Symposium on Parallel and Distributed Computing*, Melbourne, Australia, 2012.
- [7]. J. Xue, "On Tiling as a Loop Transformation," *Parallel Processing Letters*, Vol. 7, P409-424, 1997.
- [8]. R. Andonov, S. Rajopoldhye, and N. Yanev, "Optimal Orthogonal Tiling," *Proc. Fourth Int'l Euro-Par Conf.*, D. Pritchard and J. Reev, eds. P480-490, September 1998.
- [9]. B. Sinharoy, B. Szymanski, "Finding Optimum Wavefront of Parallel Computation," *Journal of Parallel Algorithms and Applications*, Vol. 2, No. 1, 1994, P5-26.
- [10]. T. Cormen, C. Leiserson, R. Rivest, C. Stein, "Introduction to Algorithms," MIT Press, Cambridge, MA, USA, 2001.
- [11]. E. Hodzic, W. Shang, "On Time Optimal Supernode Shape," *IEEE Transactions on Parallel and Distributed Systems*, December 2002, P1220-1233.
- [12]. A Jeffrey, "Complex Analysis and Applications," Second Edition, P22-23, 2005.
- [13] T. Andronikos, N. Koziris, "Optimal Scheduling for UET-UCT Grids Into Fixed Number of Processors," *Proceedings of 8<sup>th</sup> Euromicro Workshop on Parallel and Distributed Processing*, P237-243, IEEE, 2000.
- [14] M. Athanasaki, A. Sotiropoulos, G. Tsoukalas, N. Koziris, "Pipelined Scheduling of Tiled Nested Loops onto Clusters of SMPs using Memory Mapped Network Interfaces,"

Proceedings of the 2002 ACM/IEEE conference on Supercomputing (SC2002), Baltimore, Maryland, November 2002.

[15] P. Y. Calland, J. Dongarra, Y. Robert, "Tiling with Limited Resources," Proceedings Conference Application Specific Systems, Architectures, and Processors, IEEE Computer Society, P229-238, 1997.

[16]. M. Athanasaki, E. Koukis, N. Koziris, "Scheduling of Tiled Iteration Spaces onto a cluster with a Fixed Number of SMP Nodes," Proceedings of the 12<sup>th</sup> Euromicro Conference on Parallel, Distributed and Network-Based Processing, IEEE, 2004.

[17] P. Boulet, J. Dongarra, Y. Robert, and F. Vivien, "Tiling for Heterogeneous Computing Platforms," Technical Report UT-CS-97-373, Univ. of Tennessee, Knoxville, 1997

[18] L. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, P. Sadayappan, N. Vasilache, "Loop Transformations: Convexity, Pruning and Optimization," Proceedings of the 38<sup>th</sup> annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, P549-562, Austin, TX, USA 2011

[19] G. Goumas, N. Drosinos, N. Koziris, "Communication-Aware Supernode Shape," IEEE transactions on Parallel and Distributed Systems, Volume 20, Issue 4, P498-511, 2009

[20] S. Parsa, Sh. Lotfi, "Wave-Fronts Parallelization and Scheduling," 4<sup>th</sup> International Conference on Innovations in Information Technology, IEEE, Dubai, UAE, P382-386, 2007

[21] L. Liu, L. Chen, C. Wu, X Feng, "Global Tiling for Communication Minimal Parallelization on Distributed Memory Systems," 14<sup>th</sup> International Euro-Par Conference, Las Palmas de Gran Canaria, Spain, P382-391, 2008

[22] J. Yang, Y. Xu, Y. Shang, "An Efficient Parallel Algorithm for Longest Common Subsequence Problem on GPUs," Proceedings of the World Congress on Engineering June, 2010 Vol I, London, U.K.

[23] J. Kloetzli, B. Strege, J. Decker, M. Olano, "Parallel Longest Common Subsequence using Graphics Hardware," Proceedings, Eurographics Symposium on Parallel Graphics and Visualization, 2008, Crete, Greece.

[24] J. Xue, "Communication-Minimal Tiling of Uniform Dependence Loops," Journal of Parallel and Distributed Computing, Vol. 42, No. 1, P42-59, 1997

[25] R. Andonov, S. Balev, S. Rajopadhye, N. Yanev, "Optimal Semi-Oblique Tiling," IEEE Trans. On Parallel and Distributed Systems, Vol. 14, No. 9, P944-960, September, 2003

- [26] H. Ohta, Y. Saito, M. Kainaga, and H. Ono, "Optimal tile size adjustment in compiling general DOACROSS loop nests." In *1995 International Conference on Supercomputing*, pages 270-279, ACM Press, 1995.
- [27] G. Goumas, A. Sotiropoulos, and N. Koziris, "Minimizing Completion Time for Loop Tiling with Computation and Communication Overlapping." In *Proceedings of IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS'01)*, San Francisco, Apr 2001.
- [28] A. Sotiropoulos, G. Tsoukalas, and N. Koziris, "Enhancing the Performance of Tiled Loop Execution onto Clusters using Memory Mapped Network Interfaces and Pipelined Schedules." In *Proceedings of the 2002 Workshop on Communication Architecture for Clusters (CAC'02)*, Int'l Parallel and Distributed Processing Symposium (IPDPS'02), Fort Lauderdale, Florida, April 2002
- [29] P. Boulet, A. Darte, T. Risset, Y. Robert, "(Pen)-ultimate tiling?" *INTERGRATION, The VLSI Journal*, volume 17, Pages 33-51, 1994
- [30] A. Cohen, S. Girbal, D. Parello, M. Sigler, O. Temam, N. Vasilache, "Facilitating the Search for Compositions of Program Transformations", *ACM ICS 2005: Proceeding of the 19<sup>th</sup> Annual International Conference on Supercomputing*, P151-160, New York, NY, USA
- [31] P. Feautrier, "Some efficient solutions to the affine scheduling problem. Part I: one-dimensional time," *International Journal of Parallel Programming*, 21(5), P313-348, 1992
- [32] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, O. Temam, "Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies," *International Journal of Parallel Programming*, 34(30) P261-317, 2006
- [33] U. Bondhugula, A. Hartono, J. Ramanujam, P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer." In *PLDI 2008 Proceedings of the 29<sup>th</sup> ACM SIGPLAN Conference on Programming Language Design and Implementation*, Tuscon, USA, Jun. 2008
- [34] S. Pop, A. Cohen, C. Bastoul, S. Girbal, P. Jouvelot, G.-A. Silber, N. Vasilache, "GRAPHITE: Loop optimizations based on the polyhedral model for GCC," In *Proc. Of the 4<sup>th</sup> GCC Developer's summit*, Ottawa, Canada, Jun. 2006
- [35] J. Yang, Y. Xu, Y. Shang, "An efficient parallel algorithm for longest common subsequence problem on GPUs." *WCE 2010 – Proceedings of the World Congress on Engineering 2010*, P499-504
- [36] N. Ukiyama, H. Imai, "Parallel multiple alignments and their implementation on

CM5,” Genome Informatics, Yokohama, Japan, P103-108, Dec. 1993

[37] A. Lim, S. Liao, M. Lam, “Blocking and array contraction across arbitrarily nested loops using affine partitioning,” in proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming, P103-112, 2001

[38] A. Lim, G. Cheong, M. Lam, “An affine partitioning algorithm to maximize parallelism and minimize communication,” in proceedings of the 13<sup>th</sup> international conference on supercomputing, P228-237, 1999

[39] N. Ahmed, N. Mateev, K. Pingali, “Synthesizing Transformations for Locality Enhancement of Imperfectly-Nested Loop Nests,” International Journal of Parallel Programming, 29(5), P493-544, Oct. 2001

[40] T. Grosser, S. Verdoolaege, A. Cohen, “Polyhedral ast generation is more than scanning polyhedra,” ACM Transactions on Programming Languages and Systems, Volume 37, Issue 4, 2015

[41] C. Nugteren, P. Custers, H. Corporaal, “Algorithmic species: A classification of affine loop nests for parallel programming,” ACM Transactions on Architecture and Code Optimization, Volume 9, issue 4, 2013