4-16-2013 12:00 AM

Integrated Development and Parallelization of Automated Dicentric Chromosome Identification Software to Expedite Biodosimetry Analysis

Yanxin Li
The University of Western Ontario

Supervisor
Dr. Peter K. Rogan
The University of Western Ontario

Graduate Program in Computer Science
A thesis submitted in partial fulfillment of the requirements for the degree in Master of Science

INTEGRATED DEVELOPMENT AND PARALLELIZATION OF AUTOMATED DICENTRIC CHROMOSOME IDENTIFICATION SOFTWARE TO EXPEDITE BIODOSIMETRY ANALYSIS

(Thesis format: Monograph)

by

Yanxin Li

Graduate Program in Computer Science

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science

The School of Graduate and Postdoctoral Studies
The University of Western Ontario
London, Ontario, Canada

Abstract

Manual cytogenetic biodosimetry lacks the ability to handle mass casualty events. We present an automated dicentric chromosome identification (ADCI) software utilizing parallel computing technology. A parallelization strategy combining data and task parallelism, as well as optimization of I/O operations, has been designed, implemented, and incorporated in ADCI. Experiments on an eight-core desktop show that our algorithm can expedite the process of ADCI by at least four folds. Experiments on Symmetric Computing, SHARCNET, Blue Gene/Q multi-processor computers demonstrate the capability of parallelized ADCI to process thousands of samples for cytogenetic biodosimetry in a few hours. This increase in speed underscores the effectiveness of parallelization in accelerating ADCI. Our software will be an important tool to handle the magnitude of mass casualty ionizing radiation events by expediting accurate detection of dicentric chromosomes.

Keywords

Parallel computing, Parallelism, Distributed system, High performance computing, Image processing, Cytogenetic biodosimetry, Human metaphase chromosomes

Acknowledgments

I would like to thank my supervisor Dr. Peter Rogan for the opportunity to study and work in his lab for the last two years. His enthusiasm for science encouraged my pursuit of our project. His expertise in biochemistry and computer science allowed me to think critically about my research. I would also like to thank my co-supervisor Dr. Jagath Samarabandu for his invaluable guidance on software engineering and image processing and Dr. Joan Knoll for providing crucial feedback on the accuracy of our software.

Asanka Wickramasinghe and Akila Subasinghe helped me design and implement the ADCI software. I feel grateful for their support. Natasha Caminsky and Wahab Khan prepared most of the metaphase slides used for our project. I appreciate their contribution. I also want to show my gratitude to Dr. Samarabandu and Dr. Marc Moreno Maza as the knowledge I acquired in their courses helped move our project forward.

I would like to thank the staff that helped us with SHARCNET and Scinet, especially Baolai Ge, Tyson Whitehead and Daniel Gruner. Finally, thanks to all members in our lab for their support in my academic and personal life in Canada. I am very thankful to Wahab Khan for his selfless help in proofreading my thesis.

Yancey Li

Table of Contents

List of Tables

List of Figures

List of Appendices

Chapter 1

1 Introduction

Ionizing radiation is ubiquitous in the environment. Its source can be natural such as radioactive materials and cosmic rays or artificial, such as nuclear power plants. Overexposure to ionizing radiation damages living tissue and could cause severe health problems (i.e. mutation, radiation sickness, cancer and death). Cytogenetic biodosimetry is the definitive assessment for exposure to ionizing radiation recommended by the World Health Organization (WHO). This involves counting the frequency of dicentric chromosomes (DCs) on metaphase cells, termed dicentric chromosome analysis (DCA) [1]. A set of algorithms implemented using MATLAB has been previously developed to automatically identify dicentric chromosomes [2] [3]. Based on these algorithms, we are developing an Automated Dicentric Chromosome Identifying software (ADCI) with C++/OpenCV. This thesis discusses the parallelization and implementation of ADCI.

1.1 Cytogenetic biodosimetry

There exist many situations in which one or more individuals have been overexposed to ionizing radiation [4]. A subset of these incidents happened in mass casualty scenarios [5] [6], among which the Fukushima Daiichi nuclear disaster is the most recent example. Such mass casualty events could be nuclear power plant related, terrorism involving radiological weapons, and/or nuclear weapon attacks [7]. Although a nuclear attack has not occurred recently, due to terrorist activities and nuclear weapons proliferation, attention must be paid to the possibility of these extreme cases. The number of individuals potentially exposed to radiation who require urgent medical evaluation could number into the thousands.

There are many accepted radiation dosimetry assessments in early-phase response to acute radiation, including monitoring the exposed individuals, observing and recording symptoms or counting white-blood-cell differential [5]. In these approaches, sampling blood for cytogenetic biodosimetry analysis usually provides an accurate estimate of the radiation dose. Ionizing radiation can cause many structural changes to the human

chromosome. These are readily identifiable by light microscopy during the metaphase stage of the cell cycle and can include chromosome breaks, marker, dicentric and ring chromosomes, as well as chromosomes with complex rearrangements. The dicentric chromosome assay (DCA) serves as the "gold standard" in dose assessment [8]. Figure 1 is a metaphase cell image generated in our laboratory in which we see a marker, a ring and at least one dicentric chromosome.

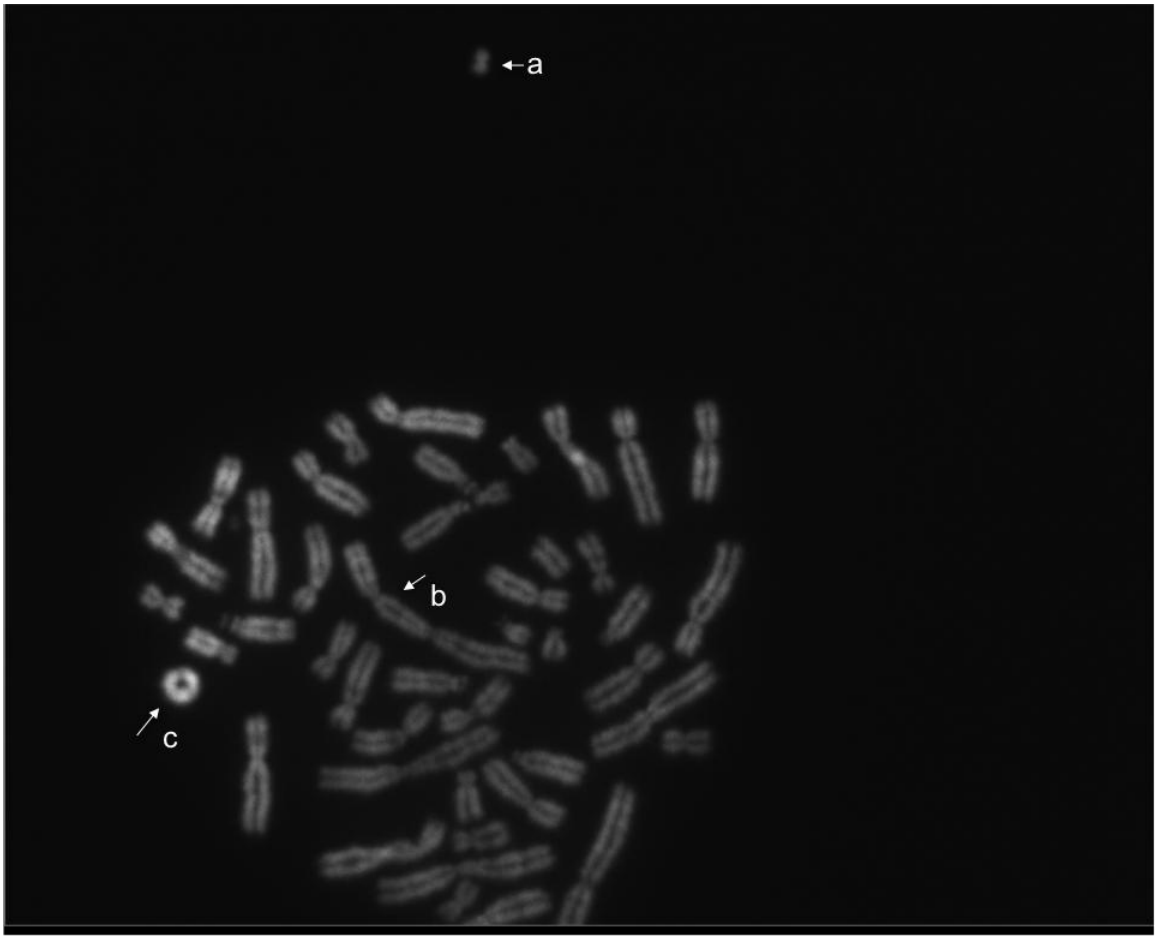

Figure 1: Metaphase image of human chromosomes shown in gray-scale. A) Structurally abnormal marker chromosome in which no part of the chromosome can be identified is flagged. B) Dicentric chromosome is shown with two primary constrictions, each containing a centromere. C) Ring chromosome in which the telomere ends are lost and the remaining chromosome components have fused.

A centromere is the part of a chromosome where sister chromatids are linked to each other. In stained metaphase cell images, the centromere can be observed as a narrow part in a chromosome exhibiting an intensity different from other regions of the chromosome [9]. A normal chromosome can only have one centromere and is always located at a certain position that is determined by the particular chromosome that contains it, which is numbered from 1 to 22, X and Y. Abnormal chromosomes with two centromeres in a single chromosome are defined as dicentric chromosomes. The frequency of dicentric chromosomes is critical in diagnosis of radiation exposure: a high frequency of dicentric chromosomes implies that the individual was exposed to a high radiation dose. Dicentric chromosomes have an extra centromere relative to normal chromosomes, but depending on shape of the chromosome, identification of the narrowest part can be subtle on certain metaphase chromosomes. Dicentric chromosomes are shown in Figures 2 and 3. Examination of dicentric chromosomes is usually made in reference cytogenetic laboratories by individuals who have experience and professional expertise in cytogenetics. A typical process of DCA for an individual includes culturing lymphocytes (white blood cells) and harvesting mitogen-stimulated cells, preparing fixed chromosomes, identifying metaphase chromosomes, selecting a small proportion for fast triage (typically, 50 metaphases), and counting the dicentric chromosomes in these selected metaphases [1]. For microscopists, this analysis is time consuming, and requires significant patience and energy, compromising rapidity of processing, especially for large numbers of samples. A rushed assessment may lead to errors in counting. In a mass casualty case, the magnitude and density of evaluation would be a challenge for early medical response, since the number of samples (patients) could be thousands, while the assessment and treatment window only lasts for a few days [1].

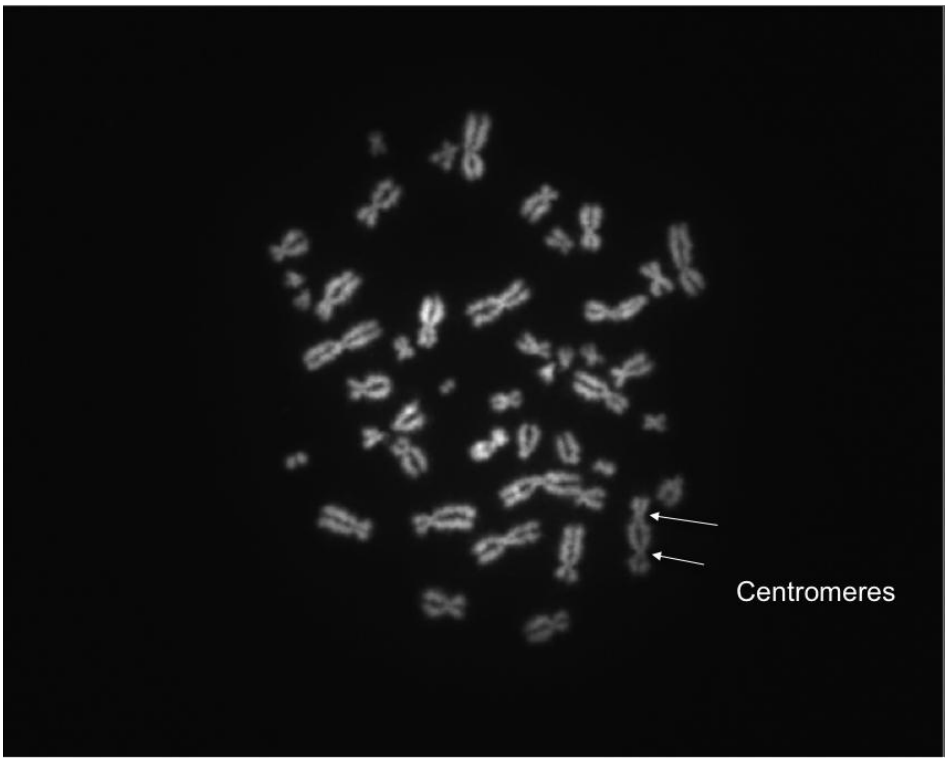

Figure 2: A metaphase with a dicentric chromosome is indicated. Arrows point to the two centromeres on this chromosome that can be identified by their narrow width. Chromosomes are counterstained with 4',6-diamidino-2-phenylindole (DAPI).

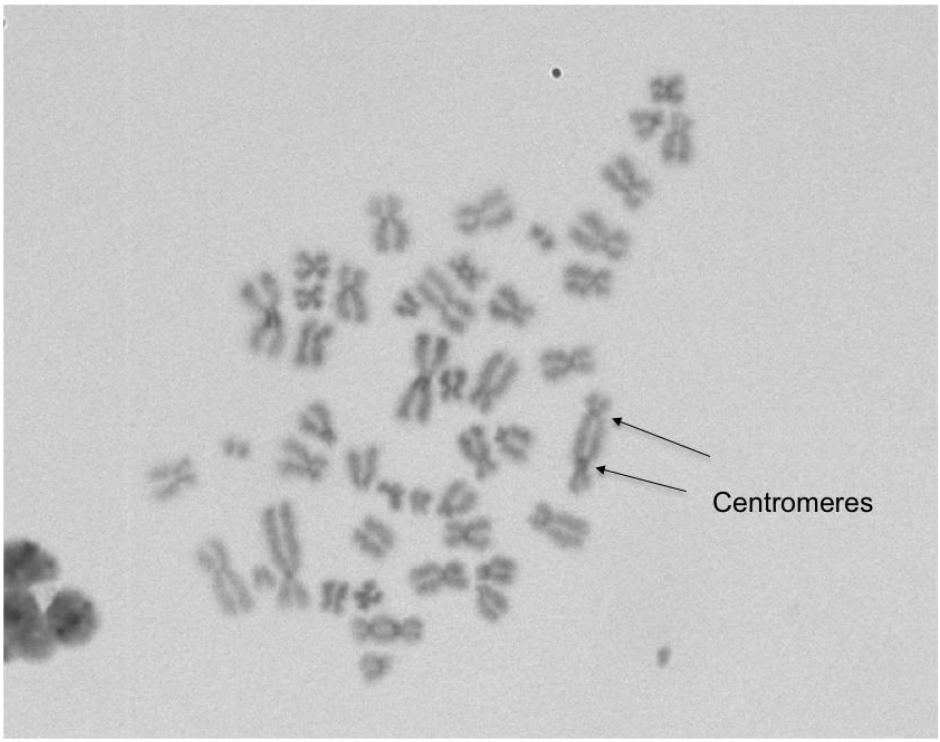

Figure 3: A metaphase with a dicentric chromosome indicated. Arrows point to the two centromeres that can be identified by their narrow width and distinguishable intensity. Cell stained method: Constitutive banding (C-banding).

1.2 Automated Dicentric Chromosome Identification

A set of algorithms was developed in MATLAB to automatically and accurately locate centromeres for dicentric analysis [3]. These algorithms have been converted to a C++/OpenCV ADCI software version and parallelization of the C++/OpenCV ADCI has been completed. MATLAB was not available for all hardware clusters that we used, such as BG/Q and Symmetric Computing. Developing software in MATLAB will also involve license issues. We have also obtained significant improvement in performance by recoding in C++/OpenCV (Table 7). The current ADCI system is comprised of six functional modules: metaphase ranking (ranking), chromosome classification (classifying), gradient vector flow contour extraction (GVF), discrete curve evolution (DCE), centerline interpolation (interpolation) and centromere detection (centromere). There are additional modules whose algorithms are still in development such as Sister Chromatid Separation with Integrated Intensity Laplacian and Chromosomes Separation

[10]. Figure 4 shows the flow chart of the ADCI system, and Figure 5 gives a visualized example of ADCI process. Two versions of ADCI, ADCI desktop and ADCI cluster, were developed for interactive use in desktops and handling mass casualty events in high performance computing clusters respectively. ADCI desktop has a GUI module and ADCI cluster has a Scheduling module.

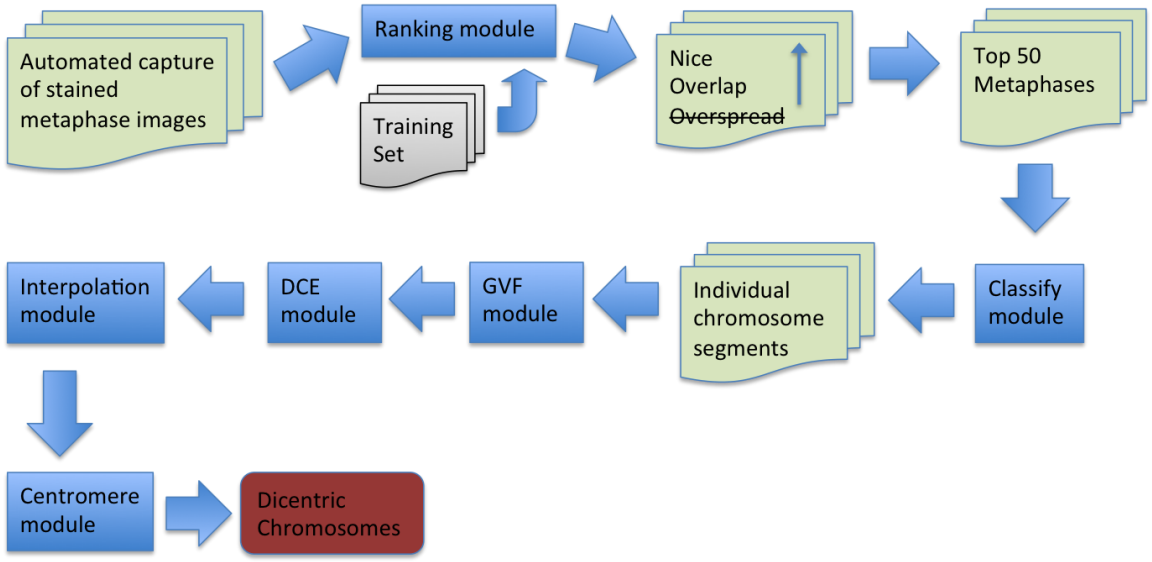

Figure 4: Flow chart of ADCI system. Green shapes depict data in ADCI. Blue boxes represent current functional modules. In the box of three categories of metaphases, the strikethrough indicates that the 'overspread' images are discarded, and arrow means the 'nice' metaphases have higher ranks than the 'overlap'.

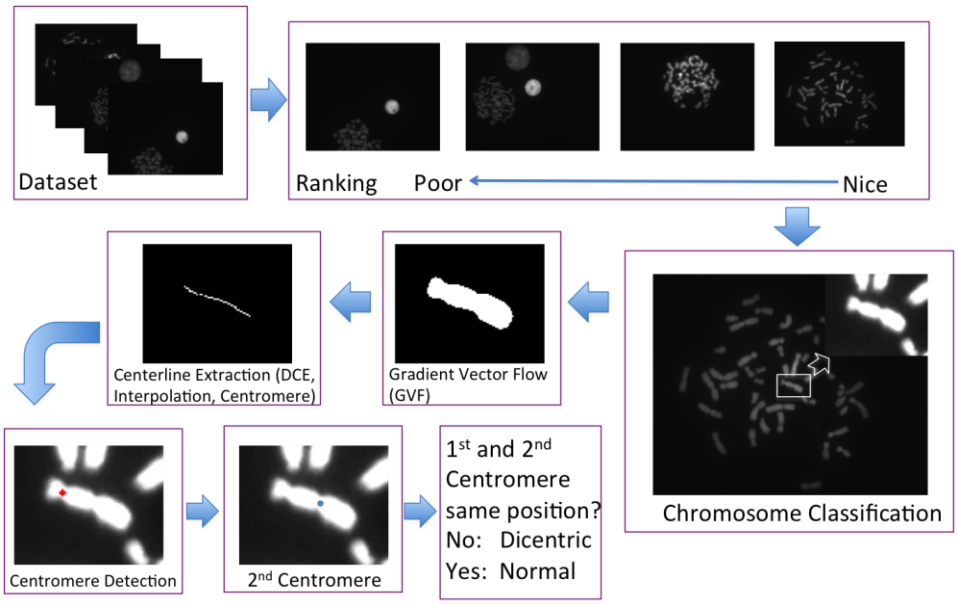

Figure 5: Example of processes in the six functional modules. The ranking module selects the best 50 images of all metaphases in a sample [11]. For each selected image, the classifying module segments individual chromosomes and classifies it into a single chromosome class or chromosome cluster class. Single chromosomes are fed into further processes [3].

1.2.1 The Ranking Module

In the ranking module, ADCI performs a preliminary selection of metaphases suitable for further analysis, as well as preprocesses of metaphases. The ranking algorithm combines content extraction and classification of metaphases to select the optimal candidates for dicentric chromosomes identification [11]. Both graphical and cytogenetic features are extracted from a metaphase based on its content. In total, 17 features are collected, including graphical data such as image contrast and foreground pixel numbers. Cytogenetic information such as number of separated connected components (blobs, corresponding to either individual or overlapping chromosomes), their average lengths and distance between separated blobs is also collected. When features of all candidate images are extracted, these are normalized with a training set consisting of the same features, obtained from several hundreds of metaphase cells previously scored by 2 certified cytogeneticists. Metaphases within this training set have been categorized into

three classes as ‘nice’; in which chromosomes are well separated but within field of view, ‘overlapped’; in which chromosomes are highly overlapped, and ‘overspread’; in which the metaphase is incomplete because some chromosomes are outside the viewing field [11] [12]. Figure 6 gives an example for every class. ‘Nice’ metaphases are the best for DCA. Metaphases with overlapped chromosomes are less ideal for DCA, as many of their chromosome boundaries are not completely distinguishable. However, when there is an insufficient number of ‘nice’ metaphases in a sample, some overlapped chromosomes may be necessary. ‘Overspread’ metaphases are not useful for DCA and are generally not considered. In ADCI, the classification of metaphases in a given sample is accomplished by the K-nearest neighbor algorithm, where K = 1 [13]. The ‘nice’ and ‘overlapped’ metaphases are provided as input to a formula to calculate scores, based on their cumulative set of features. Within a given category, ‘nice’ or ‘overlapped’ metaphases are ranked in decreasing order of their scores. In three categories, metaphases in ‘nice’ class always keep a higher rank than those in ‘overlapped’ category, while ‘overspread’ metaphases are lowest ranked. Top 50 ranked metaphase are chosen for DCA. This guarantees that ‘overlapped’ metaphase can be applied to DCA only after the set of all ‘nice’ metaphases has been exhausted [11] [12]. The ranking module provides not only a list of the most useful metaphases for further steps in ADCI, but also some intermediate results, such as the shapes of separated blobs in a metaphase. These intermediate data generated during ranking remain resident in memory to avoid duplicated operations in the subsequent software modules.

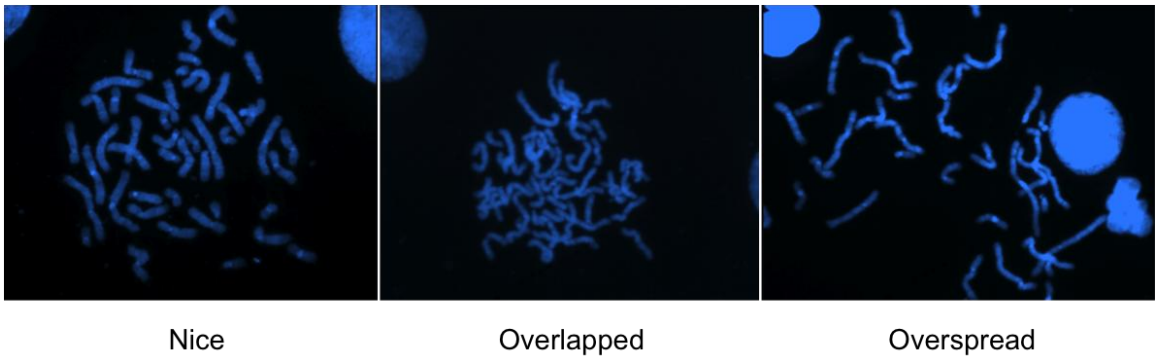

Figure 6: Examples for three metaphase categories in ranking. ‘Nice’ metaphases have a complete (46 human chromosomes) or almost complete set of chromosomes.

Most chromosomes in 'nice' metaphase category are well separated with clear boundaries, which are critical for DCA. 'Nice' chromosomes are permitted to contain a limited number of overlapped chromosomes. 'Overlapped' metaphases also have most or all of the 46 chromosomes, yet the chromosomes are under-spread. 'Overspread' metaphases contain chromosomes that are completely missing from the field of view and some are only partially visible in the image.

1.2.2 Chromosome Classification and Gradient Vector Flow Contour Extraction

The chromosome classification (classifying) module determines whether the input blob is a single chromosome or a chromosomal cluster with two or more chromosomes in overlap. The current algorithm is a variation of the algorithm proposed by Rizvandi et al. (2008) [14]. It generates and prunes a coarse centerline for an input blob, and counts the number of conjoined parts of the centerline. If conjoined part(s) exist, the input blob is considered to be a cluster of multiple chromosomes; otherwise, it is a single chromosome. Clusters of multiple chromosomes can produce false positive DC assignments. For this reason, only single chromosomes are selected for further processing by DCA.

The Gradient Vector Flow (GVF) Contour Extraction is one of the key components in ADCI [15], which produces a very descriptive contour for the input chromosome. The initial contours of chromosomes are obtained by threshold segmentation by Otsu's method, which is known to be somewhat inaccurate. We have employed active contours or snakes, to define the chromosome boundaries. Active contours are curves that can move under the effect of internal energy from the shape of the curve, and the effect of external energy from the data in the image. A parametric curve controlled by parameter s is expressed as Equation (1.1):

$$\boldsymbol{x}(s) = \big(x(s), y(s)\big) \quad 0 \le s \le 1. \tag{1.1}$$

Given an initial contour, the energy of this contour is defined as in Equation (1.2):

$$E = \int_0^1 \{\frac{1}{2}[a|\boldsymbol{x}'(s)|^2 + b|\boldsymbol{x}''(s)|^2] + E_{ext}(\boldsymbol{x}(s))\}ds. \quad (1.2)$$

The component within the square brackets is the internal energy function where parameters a and b control the active contour's tension and rigidness, respectively. The variable, $E_{ext}(\boldsymbol{x}(s))$ in Equation (1.2), represents the external energy coming from the data in the image. The active contour can expand or shrink from the initial contour while minimizing the energy defined in Equation (1.2), making the curve a function of time t. An active contour minimizing Equation (1.2) has to satisfy the Euler-Lagrange equation:

$$a\boldsymbol{x}''(s,t) - b\boldsymbol{x}''''(s,t) - \nabla E_{ext} = 0. \quad (1.3)$$

In GVF Snake model, the gradient vector flow vector field, $\boldsymbol{V}$ in Equation (1.4), serves as the external energy component. The parametric curve that can solve Equation (1.4) is a GVF snake, which is the final contour obtained from the GVF module for chromosomes.

$$a\boldsymbol{x}''(s,t) - b\boldsymbol{x}''''(s,t) + \boldsymbol{V} = 0 \quad (1.4)$$

The gradient vector flow in Equation (1.4) is defined as a vector field $\boldsymbol{V}(x,y) = [u(x,y), v(x,y)]$ and can be acquired by minimizing the energy function in Equation (1.5):

$$\varepsilon = \iint \mu\left({u_x}^2 + {u_y}^2 + {v_x}^2 + {v_y}^2\right) + |\nabla f|^2|V - \nabla f|^2 \, dxdy. \quad (1.5)$$

The f in Equation (1.5) is the edge map of the input image, which can be any gray-level or binary edge map defined in image processing. ∇f is the gradient of the edge map. In ADCI, we use the Canny edge map [16].

Compared with other external energy methods that can define active contours, for example, the distance transform, GVF snake has two advantages. The initial curve for the GVF snake can be flexible, and the GVF snake can facilitate the convergence of the curve to its boundary concavities. Concavity is very common for contours of chromosomes [15]. Figure 7 shows some examples of contours obtained from the GVF

module and that these GVF snakes can satisfactorily fit the concavities in chromosome contours [50].

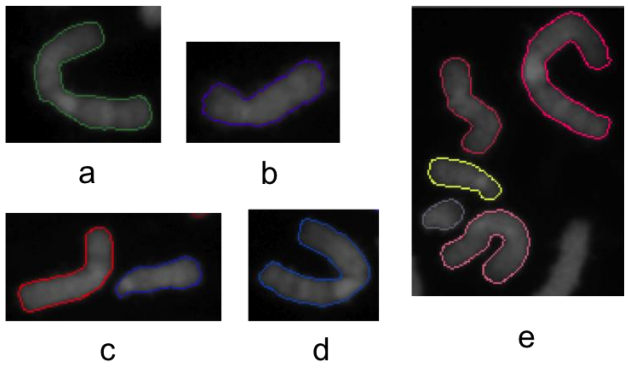

Figure 7: The images (a to e) show chromosome contours obtained from the GVF module. GVF contours of gray-scaled objects are shown as colored outlines circumscribing the chromosomes. The boundaries of concave chromosomes (d and e) are precisely matched by their corresponding GVF contours.

1.2.3 Discrete Curve Evolution and Centerline-Based Modules

Typical morphological methods for generating the centerline for an object [17], such as skeletonizing, often produce extra branches along the centerline of a chromosome. To obtain contiguous centerlines without branching, the initial centerlines acquired from skeletonizing have to be pruned. Discrete Curve Evolution (DCE) algorithm decomposes a 2D object by generating a polygon, which highly represents the input 2D object [18]. Skeleton pruning based on the polygon resulting from the DCE method, is one of its important applications [19]. In ADCI, a minimum polygon (a triangle) is obtained in the DCE module for long chromosomes, which instructs that the shortest branches along the centerline to be pruned. Centerlines of short chromosomes are obtained by medial axis thinning [20].

The DCE algorithm can be summarized in the following algorithm. The termination criterion is scenario-dependent, which is $k = 3$ in ADCI.

procedure Discrete Curve Evolution

input: polygonal decomposition D of a target curve C with k vertices: v_i $0 \leq i \leq k$

input: lines segment connecting v_i and v_{i+1} in D: s_i $0 \leq i \leq k$

input: K (s_i, s_{i+1}) the relevance measure function of two joint line segments.

for $k > 3$

Find in D a pair s_i and s_{i+1} (mod k) that minimizes K (s_i, s_{i+1});

Delete v_{i+1}, s_i and s_{i+1} in D;

Create a new line segment s connects v_i and v_{i+2};

$s_i = s$;

Update indices.

$k = k - 1$.

end for

Algorithm 1: Pseudocode of Discrete Curve Evolution in the ADCI

The relevance measure function K (s_i, s_{i+1}) represents the contribution of arc $s_i \cup s_{i+1}$ to the shape of C [19]. Equation (1.6) defines the relevance measurement function in the DCE method. Figure 8 gives a geometric example of relevance measurement function on two joint lines.

$$K\ (s_i, s_{i+1}) = \beta(s_i, s_{i+1}) l(s_i) l(s_{i+1}) / (l(s_i) + l(s_{i+1})) \qquad (1.6)$$

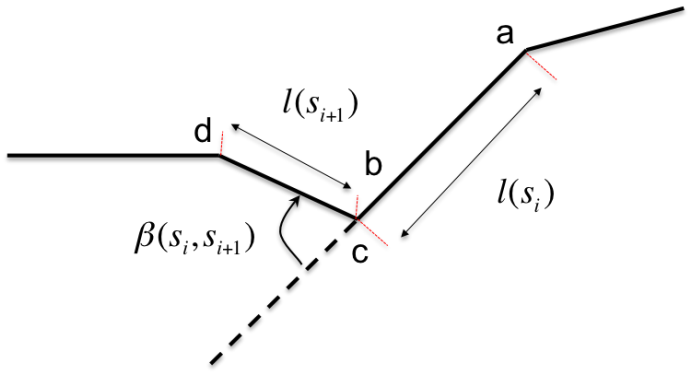

Figure 8: Example of relevance measurement function. The importance of arc a-b(c)-c to the curve can be measured by Equation (1.6).

In Equation (1.6), $\beta(s_i, s_{i+1})$ is the turn angle of line s_i and s_{i+1}. The $l(s_i)$ and $l(s_{i+1})$ variables are the lengths of line s_i and s_{i+1}. A larger turn angle and longer lines mean a more important role of the arc to the curve. This explains the rationale behind the relevance measurement function. The final triangle, resulting from the DCE module, partitions the chromosome contour curve to 3 sections. In pruning, any point in the skeleton whose maximal disk in the skeletonizing process [17] touches a same curve section at more than one point will be pruned from skeleton. After that, a centerline consisting of discrete or continuous points is obtained for every chromosome.

In order to get a curve to exactly represent a chromosome centerline, interpolation is used which connects the discrete centerline. In numerical analysis, when only some of the data points are known, interpolation is widely used to construct new data points. Interpolation is close to function approximation except that the latter does not return new data points but can be used to calculate new data points. Several common interpolation methods can be found in [21]. In ADCI, cubic spline interpolation is applied to both x and y axes for a discrete centerline. Because a final polygon in the DCE module is a triangle, a centerline obtained after interpolation still keeps an extra short branch. This extra branch is located at the end of a centerline, which means a centerline consists of a long stem branch and two short twig branches. The two twig branches are pruned and the long stem branch keeps most of the required information [20]. Figure 9 displays centerlines obtained in the ADCI desktop.

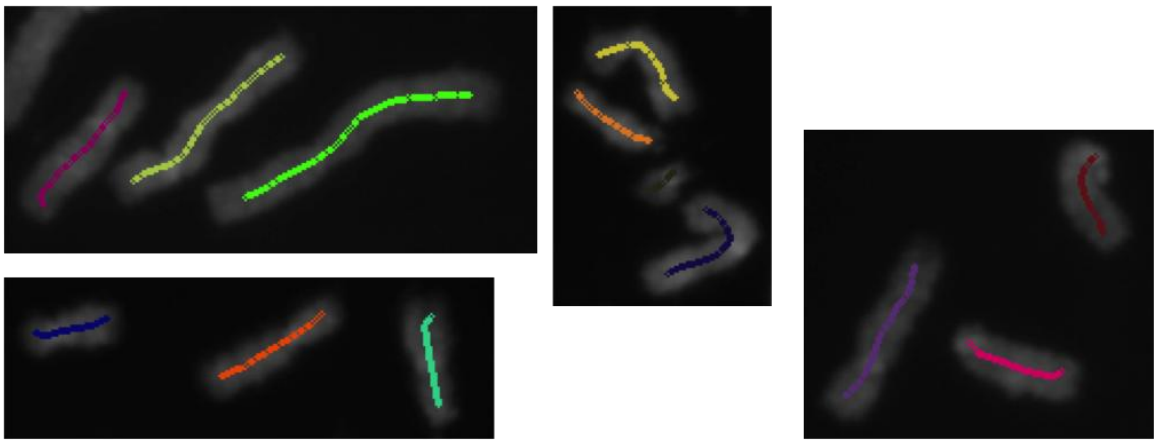

Figure 9: Examples of centerlines obtained from DCE and interpolation module. Centerlines of gray-scale chromosomes are shown as colored curve.

As stated earlier, centromeres are the most condensed and constricted parts of chromosomes, where two sister chromatids join. Locations of centromeres play a vital role in identification of chromosomes. Specifically, the designated centromere on a given chromosome dictates its cytogenetic classification as metacentric, sub-metacentric, or acrocentric [22]. When centromeres are manually identified, the location of the primary constriction (i.e. narrowest feature on the chromosome) is used to determine their location. However, the centromere region in DAPI-stained chromosomes in gray-scale metaphase images also exhibits pixel intensities that are distinguishable from other elements of the same chromosome.

In ADCI, centromeres are searched through the centerlines obtained from the DCE and interpolation modules by combining the intensity and width information along the length of the chromosomes. Metaphase chromosomes are formatted as 8-bit gray-scale images on a black background. Chromosomes shapes are obtained from the GVF module. Along the centerline, virtual lines (referred to as trellis) that are perpendicular to the centerline are generated at a unit length, with each line corresponding to a different segment of the centerline. The combined information at every centerline segment is calculated for each trellis element. Intensity information along the line element is weighted according to a Gaussian function to reduce noise at the boundaries of chromosomes [20]. Width information is determined as the length of a line in the trellis. These two sets of

information are combined into a single profile [20]. The first centromere is located by finding the global minima in the profile. Figure 10 gives examples of the first centromere in each chromosome located by ADCI. Subsequently, a regional mask covering a small neighborhood circumscribing the first centromere is applied in the profile, and the second global minimum of intensity and width is determined in the masked profile. If the first and second minima are far enough apart, this chromosome is assigned to be a dicentric; otherwise, the first and second minima are considered to represent the same centromere. Figure 11 shows two dicentric chromosomes detected by ADCI.

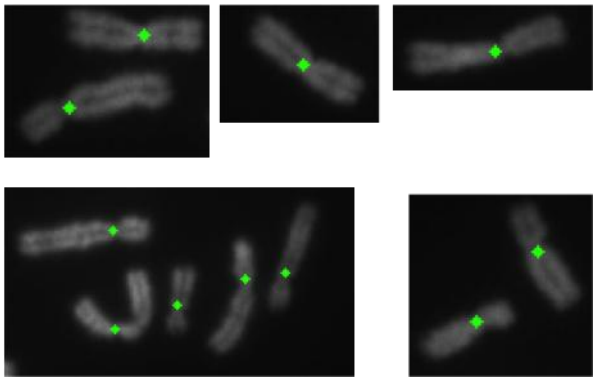

Figure 10: Examples of centromeres successfully detected in ADCI. Centromeres are marked as green dots.

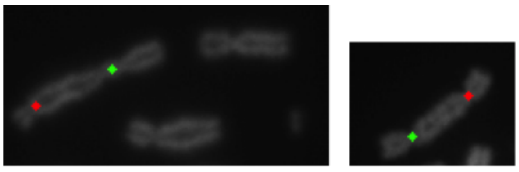

Figure 11: Two examples of dicentric detection in ADCI. The first centromeres are marked as green dots and the second centromeres marked as red dots.

1.2.4 Contribution of This Thesis

In this thesis, I describe the design and building of a parallel software version of ADCI. A parallel algorithm for binary image labeling is introduced and a fast algorithm for circulant tri-diagonal matrix inversion is described, as well as its parallelized version in Section 3. Strategies for incorporating data parallelism for processing multiple images from many different samples, and the combination of data and task parallelism on ADCI

are also discussed. We built the parallel ADCI desktop and ADCI cluster based on these principles, and tested our software on various platforms in Section 4. Results and conclusions are given in Section 4 and 5. We show that our parallelized ADCI can accelerate the DCA process by several folds and the cluster computing version of ADCI is capable of processing thousands of samples in a few hours.

Chapter 2

2 Rationale for Parallelization of ADCI

Both the traditional DCA undertaken by cytogenetic experts in reference laboratories and the automated DCA by our ADCI desktop are challenged to process thousands of samples within a clinically relevant timeframe (i.e. a few hours). As introduced in Section 1.2, metaphase images obtained from automated cytogenetic microscopy systems can be extremely variable in morphology. Thus dedicated image processing procedures are required to recognize the key features in chromosomes that are diagnostic for DCs. Furthermore, the system modules GVF snake and DCE can consume a lot of time to perform the computations necessary to extract these features. It is also difficult to find faster algorithms with the requisite accuracy to produce useful measurements of radiation exposure based on DCA. As parallel computing hardware like multicore processors and computing clusters are becoming widespread, a feasible way to accelerate ADCI is via parallelization of these existing algorithms. The need of handling thousands of samples in hours motivated us to build the ADCI cluster software that can utilize many processors in a high performance computing environment in order to efficiently accelerate DCA. We developed a set of strategies to accelerate both ADCI desktop and ADCI cluster from different perspectives. This chapter outlines background knowledge on parallel computing, general methods to parallelize programs, and the parallel strategies that we adopted for ADCI.

2.1 Background on Parallel Computing

Frequency scaling was the dominant reason for improvement in computer performance in past decades. By ramping up the frequency of a processor, faster processing speed can be obtained. Recently, the requirement for cooling associated with increasing frequency has limited the growth of CPU speed [23]. The gap in speed between CPU and memory is also become a major bottleneck, in instances when the speed of a single CPU has been comparatively fast. With the lower costs, and the availability of multi-processor CPUs, parallel computing hardware containing multiple processors can be exploited to achieve

higher throughput through scalable architectures. Software has been developed to enable parallelized computation and programming simpler, which is making significant contributions to improving computer performance. According to Hennessey and Patterson, contemporary computers can be categorized into fives classes: personal mobile devices (PMD), desktops, servers, clusters and embedded computers [24]. Parallel computing hardware has widely emerged in computers from the first four categories and examples include multi-core smartphones or tablets, multi-core desktop or laptop computers, symmetric multi-processing (SMP) systems and distributed clusters. SMP systems and distributed clusters are usually supercomputers that are much more powerful than multi-core systems. Therefore SMP supercomputers and clusters are considered to be high performance computing (HPC) systems. Although multi-core or multi-processor desktops or servers also use SMP architectures, we exclusively refer to SMP supercomputers as SMP.

On the basis of the instruction stream processed by computers and the data stream called by instruction stream, Flynn proposed a simple classification for parallel computing architecture in 1960s [25] [26], which is still widely used today. Any computer can be assigned to one of the following four classes: single instruction stream with single data stream (SISD), single instruction steam with multiple data streams (SIMD), multiple instruction streams with single data stream (MISD) and multiple instruction streams with multiple data streams (MIMD) [25]. SISD is the uniprocessor type. In computer programming, this is the standard sequential computer but it can exploit instruction parallelism, such as pipeline and speculative instructions. In SIMD, multiple processors run the same program on their own data streams, such as Graphic Processing Units (GPU), exploiting a typical data parallelism model. Computers in MISD run multiple instruction streams on the same data stream, but processors in this class have not been commercially released. Each processor in a MIMD computer fetches its own instruction and operates on its own data. It provides the most flexible way to implement parallel computing. Both task parallelism and data parallelism can be exploited on MIMD computers. We also classify parallel hardware to multi-core systems, SMP and HPC clusters, when considering implementation of parallel programs. All multi-core computers, SMP systems and HPC clusters we have used belong to the MIMD category.

Multi-core computers are computers with two or more central compute units (cores) on a single processor chip. Desktops or servers with a few processor chips sharing memory are also common. In this thesis, these computers are generally referred to as multi-core systems. Cores in a multi-core system are fully functional processors which have the same features as single-core processors, e.g. instruction pipeline, vector processing or multi-thread. Parallelizing algorithms and software on multi-core systems is a major on-going research area in parallel computing. SMP systems are computing system, where a group of identical processors are connected on a shared memory space. This differs from multi-core system as SMP systems are usually supercomputers. HPC clusters consist of a set of connected computers working together. Computing nodes in a cluster are standalone computers usually connected by fast local area networks. Nodes belonging to a same cluster are not guaranteed to be symmetric.

In considering what should be parallelized, the methods of parallelism in applications can be basically divided into data and task parallelism classes. In data parallelism, many data items are distributed to different computing units and operated at the same time. In task parallelism, a task of computation on a single data object is subdivided into multiple tasks that are processed concurrently. The boundary between data and task parallelism is not strict as data and task are highly related in most programs [27]. Based on different hardware level, computers exploit these two methods in four ways: instruction-level parallelism, data-level parallelism implemented by vector architecture and GPU, thread-level parallelism and request-level parallelism [24]. Both data and task parallelism methods can be exploited in these four ways. Instruction-level parallelism and vector architecture are determined by hardware and compilers. GPU parallelism, like CUDA, runs the same instructions on thousands of stream processors on arrays of data. Thread-level parallelism and request-level parallelism are the two major ways to program in parallel on CPUs. Thread-level is a relatively high-level parallelism method involving cooperation among threads. There have been a lot of research and applications in parallel computing focusing on thread-level parallelism. In request-level parallelism, programs are executed in large groups of tasks or processes specified by programmers, such as serial farming or message passing interface (MPI) processes.

Under optimal conditions, the speedup brought by a parallel algorithm can be linearly inverse to the number of computing units available. This means for example, doubling number of processors can halve the total process time. Most parallel algorithms are unable to achieve a level of performance at this scale, because of overhead from inter-process communication, latency due to scheduling and the specific hardware. Dependency within the software program also prevents efficient parallelism. The potential performance improvement of a parallel algorithm can be evaluated by Amdahl's law, shown as Equation (1.7), based on the extent of a program being parallelized, which was originally formulated by Amdahl [28]. In Equation (1.7),

$$S = 1/((1-\alpha)/P + \alpha) \tag{1.7}$$

α represents the portion of the program which cannot be parallelized and P stands for the number of available processors. If no part of the program can be parallelized, the speedup will always be 1. If the program can be fully parallelized, the speedup will be linearly increasing with the number of processors. In design of parallel algorithms, parallelizing the maximum portions of a program is the main objective. The degree of parallelism possible for a certain program is related to the dependency in the program, which can be analyzed by control flow and data flow diagrams [29]. A directed acyclic graph (DAG) which represents computations in a program, is a useful model (referred to as computation model) to measure performance of a parallel algorithm, as shown in Figure 12. A part of the program cannot be executed before any parts preceding it in the DAG. The series of actions in a program that cost the most time is the largest portion in the program that cannot be parallelized. This is defined as the critical path [30] [31]. Assuming every node costs a unit of time, the longest path in the DAG is the critical path. In a computation model, we define T_p, T_1 and T_∞ as the time requirements of a program when running on p, one and infinite processors respectively. In the DAG, T_1 is the sum of time cost by all nodes, defined as work, and T_∞ is the time required by the critical path, defined as span. T_p satisfies Equation (1.8):

$$T_p \geq T_1/p\,, \tag{1.8 a}$$

$$T_p \geq T_\infty \,. \tag{1.8 b}$$

If at each step in the execution of a parallel program, a scheduling mechanism attempts to do as much work as possible, this mechanism is called greedy schedule. Most current multi-core parallel platforms use a fork-join parallel schema, which means an existing thread spawns a new thread at one time. According to Graham and Brent, for any greedy schedules in fork-join schema, we have Equation (1.9) [35]:

$$T_p \leq T_1/p + T_\infty \,. \tag{1.9}$$

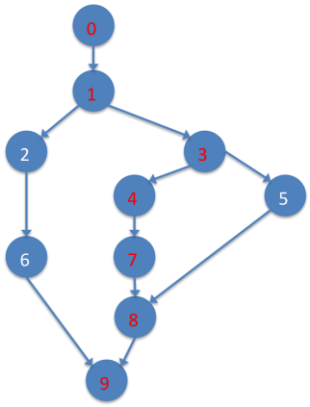

Figure 12: An example of computation model. Each node represents a part of instruction in the program. Node 0 and 9 are the start and end parts, respectively. Lines with arrows illustrate dependency. Nodes in the critical path are marked with red numbers.

The calculation above does not account for overhead due to process communication and scheduling. Implementation of communication between processors varies on hardware. Typical communication times between cores on a same chip cost 35 to 50 clock cycles, and communication among separated chips costs 100 to 500 or even a greater number of clock cycles [24]. The latency for distributed clusters mostly relies on network architecture and performance, which can be very inefficient compared with processor speed. Scheduling and data management also detract from performance. In thread-level parallel programs, computations are made within software threads. Software threads must be mapped to hardware threads at run time by the specific thread-level parallel platform

or operating system. The workload carried by the software threads is scheduled and balanced across hardware resources. For process-level (request-level) parallelism, for example programs running under MPI, programmers take the responsibility to map the workload to processes. In summary, parallelism demands time for overhead functions, which ultimately means that there is a tradeoff of benefits of multi-processor computation and the corresponding overhead cost. Therefore, theoretical estimation of parallel performance, dedicated design, and adequate experiments are required when a program is going to be parallelized.

2.2 The General Parallelizing Method

In most cases, data and tasks are related so closely that they are difficult or even unnecessary to distinguish. Therefore, only discussion of task parallelism is provided here, but it should be recognized that the same comments are also applicable to data parallelism. The general method of parallelizing software includes Decomposition, Assignment, Orchestration and Mapping phases [27]. Decomposition means subdividing tasks of the target problem into a collection of subtasks. The number of tasks may change dynamically during the course of execution of the program. Among all the tasks at a certain time in execution, some tasks can run concurrently, but others cannot due to their dependency on prior tasks or availability of data produced by prior tasks. The maximal number of tasks that can be executed simultaneously at a given time provides the upper bound of the number of processors that can be used effectively. Therefore in the Decomposition phase, the primary objective is to expose sufficient concurrency, while avoiding data race [27]. Limited concurrency is the most fundamental limitation for achieving speedup by parallelism. This is reflected in the analysis of DAG; exposure of concurrency finds a shorter critical path. Decomposition actually implies not only the procedure of decomposing target problem into pieces, but also the process of combining these pieces subsequent to their parallel execution. In the Assignment phase, there should be a mechanism to decide how tasks are distributed among threads or processes. There are four major goals in Assignment: reducing data competition, balancing workload across threads or processes, reducing communication - especially expensive communication such as message passing through network, and suppressing overhead

caused by scheduling and managing Assignment. The work to be balanced could be either computation of data, I/O operations, and communication or any combination of these. Sometimes, these four elements of parallelization can be discordant with one other. A tradeoff has to be made under these circumstances. Decomposition and Assignment form partitioning of the target problem and comprise the key steps in designing a parallel algorithm. They are usually irrelevant to the hardware architecture and programming model, but these factors do influence the performance of Decomposition and Assignment. At the Orchestration phase, different threads and processes perform their own tasks individually and simultaneously in accordance with a certain mechanism. The correctness of accessing data, exchanging data, and synchronization must be guaranteed in Orchestration [27]. The design and implementing of Orchestration is strongly related to hardware and software platforms. In most cases, parallelizing algorithms ends at the first three steps. In the Mapping step, the threads or processes created in previous steps are mapped to real physical processors or cores. Much research has evaluated different algorithms according to their resource allocation and management [32]. For most parallel software platforms, this work is accomplished by the platform itself or the operating system.

Most parallelization methods are problem-dependent and can only follow the general method stated above. However, there exist a few typical parallel models. For instance, the divide-and-conquer algorithm is an algorithmic paradigm for parallelism. This algorithm splits a problem into a series of sub-problems that are identical to the original but easier to solve because of the divided computation. If all sub-problems are solved, their solutions can be combined and a solution to the original problem can be constructed. Divide-and-conquer algorithms are usually recursively used to solve a problem. Divide-and-conquer algorithm can be efficiently parallelized, as sub-problems are independent and can be solved in parallel. In parallel divide-and-conquer algorithm, problems are commonly divided to many sub-problems to expose enough concurrency [33]. Parallelization of dividing and merging steps are also necessary to yield a high degree of parallelism.

2.3 Parallel Strategy on ADCI

In this thesis, we refer to parallelizing high-level data, such as thousands of samples, each consisting of hundreds of metaphase images, as data parallelism. The tasks performed in task parallelism refers to certain low-level basic functions performed on each image. Both data parallelism and task parallelism are used in the ADCI desktop and the ADCI cluster. Here, efforts are focused on thread-level and process-level parallelism. Data parallelism in ADCI is straightforward, as the data are organized as naturally separated objects to be processed. The data are organized in three layers: including samples, metaphase cell images, and chromosome sub-images. A sample is a set of metaphases belonging to one individual and a metaphase is an image containing chromosomes. Samples in ADCI are totally independent of each other, which mean processing samples can be perfectly parallelized. Metaphases in a sample keep dependency in the ranking module, but are independent of each other post-ranking. ADCI processes chromosomes one by one and all data are highly distributable which brings a very high degree of parallelism. In task parallelism, several functions that consume the most time in ADCI are selected and parallelized. Task parallelism can also contribute to speedup when there is still excess computing capacity remaining after data parallelism. Images are represented as matrices in OpenCV. Task parallelism on these images/matrices are not hard to design and implement, but not efficient as data parallelism. The priority and combination of data parallelism and task parallelism will be based on specific cases. The ADCI desktop is able to execute only in thread-level parallelism while the ADCI cluster can execute in thread-level parallelism, process-level parallelism or a combination of the two. In our strategy, data parallelism is the primary parallel method. In data parallelism, parallelization of higher-level data is initially performed. This helps exploit data locality.

Although some parallelization scenarios in the ADCI are suitable for GPU parallel computing, there are two major reasons why GPU is not ideal for parallelizing ADCI. GPU computing requires C kernel functions with dedicated memory management for GPU. This will cause overhead or compatibility problems in OpenCV. More importantly, it will also reduce the portability of ADCI, which discussions with end users have told us will be critical for future implementations.

C/C++ programming on raw threads is problematic for the developer and leads to high probability of errors, since it is closely related to the hardware employed. Several thread-level parallel tools/platforms have been built to address this. POSIX Threads (PThreads) is the POSIX (Portable Operating System Interface) standard for threads defined by IEEE in order to make a universal interface for different proprieties of threads implemented by hardware vendors [28]. PThreads is a low-level portable multi-threads tool defined as a set of C language programming types, constants and functions [34]. Cilk is a lightweight general-purpose language that is designed on ANSI C language for multi-threads computing [35]. Intel Threading Building Blocks (TBB) is a relatively high-level C++ programming template library providing a set of parallel algorithm and containers [36]. In the ADCI desktop, both data parallelism and task parallelism are achieved with TBB. Data are parallelized mainly with parallel loop and functions are broken into parallel tasks. Open Multiprocessing (OpenMP) is an API supporting multi-threads programming in C, C++ and Fortran, on most processor architectures and operating systems [37]. It is used in the ADCI cluster to implement thread-level data parallelism and task parallelism, and mixed with MPI in a hybrid parallel programming model.

Message Passing Interface (MPI) is a standardized portable message-passing library interface specification, widely used in a variety of parallel computers. MPI is a language-independent communication protocol that still dominates high-performance computing, especially for HPC clusters [41]. The latest version is MPI 3.0 [38]. There exist different versions of MPI implementation, such as Open MPI [39] and MPICH [40], of which both APIs can be directly called by C/C++ and Fortran. In the ADCI cluster software, MPI is used to accomplish data parallelism in distributed computing nodes.

Chapter 3

3 Implementation of Parallel ADCI

The object of parallelizing ADCI is to achieve faster processing speeds. This goal will be approached from three aspects: optimizing I/O operation on ADCI cluster, task parallelization of functions in image processing, and parallelization of data processing carried out by ADCI.

3.1 Optimization of I/O on ADCI cluster

The input data in the ADCI are organized in two layers: samples and metaphase images. Samples are collections of all metaphase images belonging to the same individual. Metaphase images are the only physical input data files in the ADCI, which are previously captured on an optical microscope and processed using chromosome karyotyping and analysis software available in many cytogenetic reference laboratories. Although the format and visual information contained in these metaphases can vary based on this software, a typical metaphase image is a one-channel gray-scale tagged image file format (tiff) or three-channel RGB tiff image, with a size ranging from several hundred kilobytes to two or three megabytes. Metaphase images from a given individual are stored in their own directory.

This organization of data works well for the ADCI desktop since secondary storage devices for desktops, such as hard disk drives, are usually connected by bus and can be accessed by one user at a certain time. The density of I/O operations on a desktop is not too large. On most desktops, the relatively slow processing speed and highly interactive interface results in I/O latency will not noticeably delay processing by the ADCI. But in distributed cluster computers, the disadvantage of this data organization has to be considered. In our test of the ADCI cluster on a distributed system, major delays caused by I/O operations were observed. In the ADCI cluster, multiple MPI processes run in parallel, so the overall processing speed is much faster than the speed of the ADCI desktop. The file system in a distributed cluster is usually connected to each computing node by a local area network (LAN). Compared with processing speed of the ADCI

cluster, I/O operation speed can be much slower. This is because, in a cluster, the file system is shared with other users. This means that I/O requests from different processes in the ADCI and from other users compete for limited I/O resources. The delay is most apparent when transporting or reading a large number of relatively small files. When an I/O request is sent, the file system server searches directories for the location of the file. A large number of independent, frequent searches in file directories on the file server is particularly time consuming.

3.1.1 Asynchronous I/O

One very common method to boost the performance of an application containing many demands on the I/O controller is to overlap the execution of the application and I/O operations. When a program issues a synchronous blocking I/O system call, the program will be blocked and have to wait until the system call succeeds or fails. The time between the program sending an I/O request and system returning is wasted since processors are idly waiting for the return of data from the I/O system. Overlapping the I/O operations and the program processing can make full use of the processing capabilities of the cluster. Programmers can define an exclusive I/O-operation thread by themselves to overlap I/O and computing. Alternatively, programmers can also use asynchronous I/O system calls to achieve this goal. As shown in Figure 13, I/O system calls can be divided into three categories based on their mechanisms: synchronous blocking I/O, non-blocking I/O, and asynchronous I/O [42]. For Unix-like systems, asynchronous I/O functionality is supported by Kernel Asynchronous I/O (AIO) library.

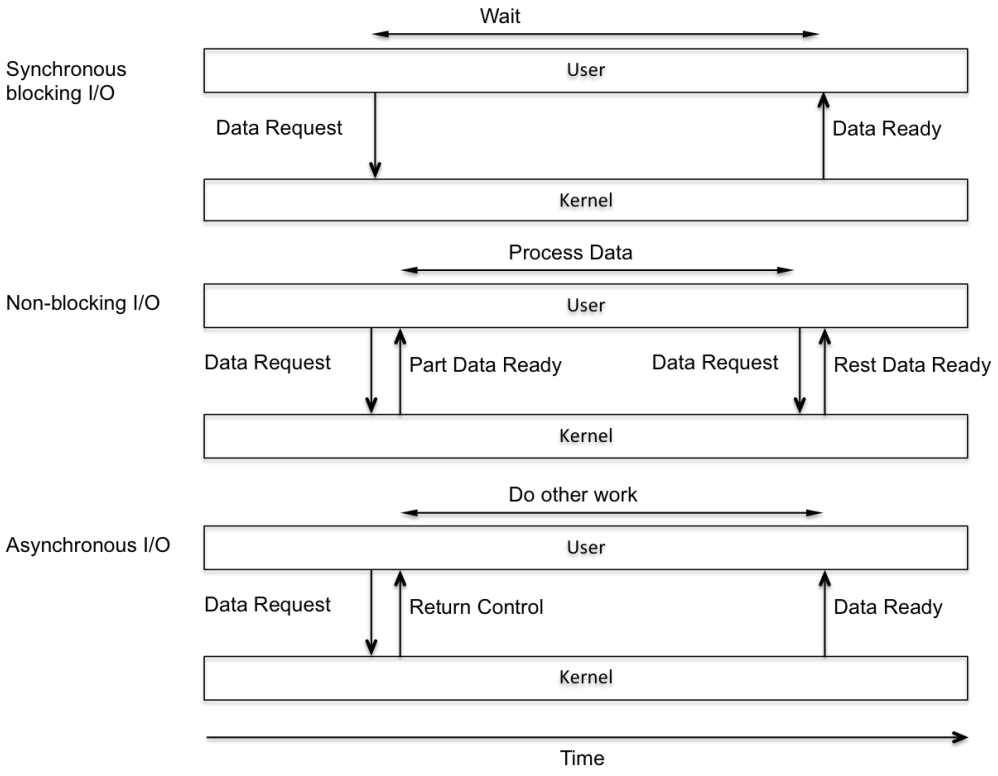

Figure 13: Comparison of three I/O mechanisms. In synchronous blocking I/O, I/O tries to fetch all data requested, while users have to wait for the return of I/O. In non-blocking I/O, I/O fetches and returns what is available currently. In asynchronous I/O, I/O returns immediately but keeps fetching data. Data are passed to user later.

3.1.2 Metaphase Gridding

Asynchronous I/O does not directly solve the I/O problem in ADCI cluster. File system servers still have to deal with a large number of I/O requests in a short timeframe. One method of addressing this is to load a small number of large files, each of which contains many metaphase images. These large files are called metaphase grids (or Bigtiff files), which use standard tiff formats. The major obstacle in this method is that almost all biodosimetry reference laboratories generate only single separated metaphases, and a separate program is required to combine them into a Bigtiff file. However, metaphase grids are used in ADCI cluster where the individual metaphases are organized into metaphase grids. A metaphase grid in the ADCI cluster is organized as a 20-by-16 image

rectangle, consisting of 16 metaphases per row, 20 metaphases per column, and 320 metaphases in total. This usually exceeds the total number of metaphases obtained for each sample.

3.2 Task Parallelism

As task parallelism involves more synchronization and communication, it is not as efficient as data parallelism. Parallelization of a task is useful only when workload for the task is big enough to outweigh the overhead of parallelism. A principle in task parallelism is profiling ADCI in order to parallelize the tasks that consume a large proportion of time in ADCI.

3.2.1 Parallelizing Binary Image Labeling

Binary connected component labeling (referred as binary image labeling) is a widely used image morphological operation. By profiling the ranking module, we know that binary image labeling costs roughly 47 percent time of total processing in ranking.

A pixel p at coordinate (x, y) in a two-dimensional image has four neighbors in vertical and horizontal directions: $(x-1, y)$, $(x+1, y)$, $(x, y-1)$ and $(x, y+1)$. The set of these four pixels is the 4-neighbor of p, denoted as $N_4(p)$. Pixel p also has four neighbors in two diagonal directions: $(x-1, y-1)$, $(x+1, y+1)$, $(x+1, y-1)$ and $(x-1, y+1)$. The set of these four pixels is the diagonal-neighbor of p, denoted as $N_D(p)$. The union of 4-neighbor and diagonal-neighbor is the 8-neighbor of p, denoted as $N_8(p)$. The set of intensity value by which adjacency is counted is defined as V. To simplify the problem, only binary image is discussed here. As we are only interested in connectivity of foreground pixels, we have $V = \{1\}$. There exist two adjacencies: 4-adjacency and 8-adjacency. Two pixels p and q with values from V are 4-adjacent if $q \in N_4(p)$. Pixels p and q are 8-adjacent if $q \in N_8(p)$ [17].

A path from pixel p with coordinate (x, y) to pixel q with coordinate (u, v) is defined as a sequence of distinct pixels:

$$(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n) \tag{3.1}$$

where $(x_0, y_0) = (x, y)$, $(x_n, y_n) = (u, v)$, and pixels $(x_i, y_i), (x_{i+1}, y_{i+1}), 0 \leq i < n$ are adjacent. Depending on specified adjacency, 4-path and 8-path are defined. In a binary image, two foreground pixels p and q are connected, if a path consisting entirely of foreground pixels exists between them. For any foreground pixel in a binary image, a set of foreground pixels connected to each other is a connected component [17].

Binary connected component labeling tries to assign an identical index number for all pixels in one connected component. Different connected components keep different indices. The returned image is an index image in which all pixels with a value $t, t > 0$ belong to the t_{th} connected component in the corresponding binary image. Pixel with index 0 is considered the background. Figure 14 shows an example of binary image labeling in a metaphase image.

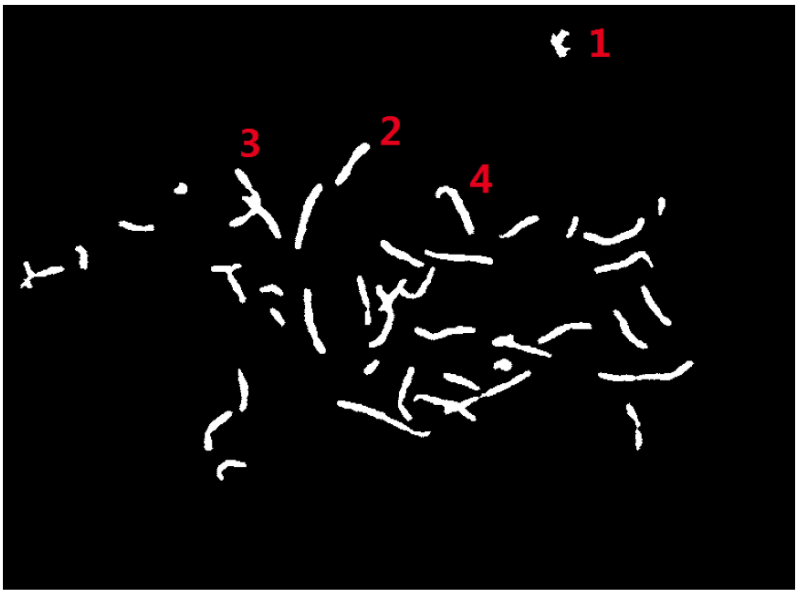

Figure 14: Binary image labeling function in a metaphase. Red numbers indicate the first to fourth connected components.

There are several serial algorithms capable of achieving binary image labeling. The algorithm used in ADCI is the same algorithm used in BWLabeln function in MATLAB [56]. Detailed description of this algorithm by Rosenfeld and Pfaltz can be found in [43]. It involves two passes of scanning a binary image and building of an equivalence table. Pseudocode of this algorithm is illustrated in Algorithm 2 below.

```
procedure BWLabeln
input: binary image IMAGE
output: index label image LABEL
LABEL = new zeros matrix of the same size of IMAGE;
create EQTABLE;
for L = 1 to last row in IMAGE              // First pass of scanning image
   for P = 1 to last column in IMAGE
      if IMAGE(L, P) == 1
         A = Neighbors((L, P));
         if A is empty
            M = new label number;
         else
            M = Min(LABELS(A));
         end if
         LABEL(L, P) = M;
         for X in LABELS(A) and X != M
            Add (X, M EQTABLE);
         end for
      end if
   end for
end for
EQLABELS = Resolve(EQTABLE);
for L = 1 to last row in IMAGE              // Second pass of scanning image
   for P = 1 to last column in IMAGE
      if IMAGE(L, P) == 1
```

```
        LABEL(L, P) = EQLABELS(LABEL(L, P));
      end if
   end for
end for
```

Algorithm 2: Pseudocode of Binary Image Labeling Algorithm [43]

The equivalence table EQTABLE has two functions. These include Add (i.e. adding a new equivalence) and Resolve. Resolve sorts out final connected component indices in its equivalences. The first pass of scanning returns a preliminary indexed image and an equivalence table containing these preliminary labels. After Resolve of the equivalence table, final label indices are assigned to the labeled image in the second pass of scanning. There could be different data structures to represent EQTABLE. The two most frequent operations in Resolve of EQTABLE are union of two equivalences and finding of an equivalence containing a certain preliminary label. A fast technique to achieve this functionality as well as to represent equivalence uses the tree data structure [44]. In this data structure, nodes represent the preliminary labels and trees represent equivalences. A new node is inserted when a new label is created. Add equivalence and Resolve EQTABLE functions are implemented by a union of two trees and sorting out all roots of trees in the equivalence table, respectively. Algorithm 3 and 4 illustrate these two functions.

```
procedure Add
input: label X, label M, EQTABLE
root A = Find(M); root B = Find(X);
if Level(A) > Level(B)
   Union(A, B);    // B is attached under A
else
   Union(B, A);    // A is attached under B
```

```
end if
```

Algorithm 3: Pseudocode of Add equivalence in Equivalence set, using the tree data structure [44]

```
procedure Resolve
input: EQTABLE
output: EQLABELS
i = 1;
initial EQLABELS[1 : last node number in EQTABLE] = {0};
for N = nodes in EQTABLE
   root = Find(N);
   if EQLABELS[root] == 0
      EQLABELS[root] = i;
      i = i+1;
      EQLABELS[N] = EQLABELS[root];
   else
      EQLABELS[N] = EQLABELS[root];
   end if
end for
```

Algorithm 4: Pseudocode of Resolve Equivalence set, using the tree data structure [43] [44]

The 4-connected and 8-connected component labeling can be solved by this algorithm depending on the choice of adjacency. An example based on 4-adjacency is given in Figure 15 and Figure 16 to illustrate the serial algorithm.

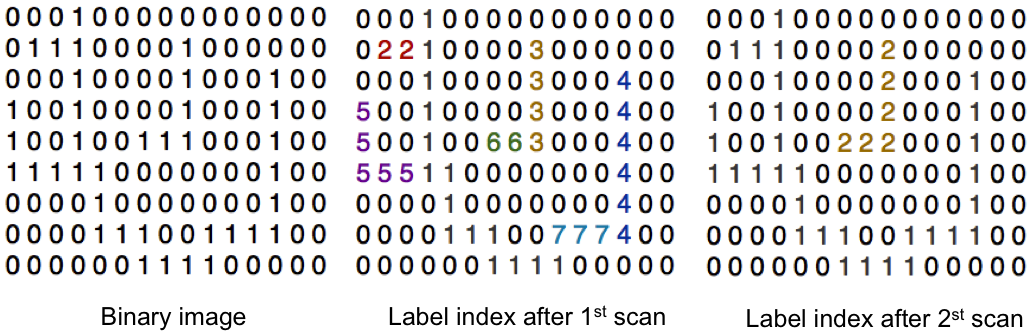

Figure 15: Example of two passes of scanning in binary image labeling algorithm. The image is scanned in row-major order. A total 7 preliminary connected components are found after the first pass of scan. Preliminary connected components are sorted out to 2 connected components in the second pass of scan after EQTABLE is Resolved.

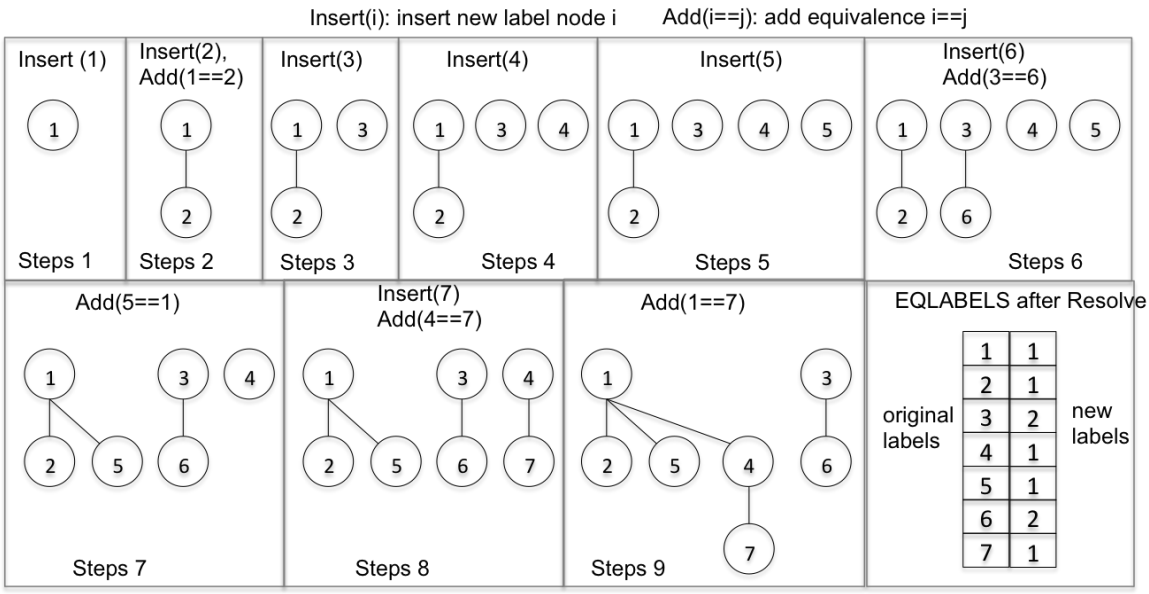

Figure 16: Equivalence set (EQTABLE) for binary image labeling in the example shown in Figure 15. Only a part of steps in building and resolving EQTABLE are demonstrated. When a new label is created, a new node is inserted as in step 3. When two preliminary labels are equal, their trees are merged as in step 7.

Time in the binary image labeling algorithm is easy to calculate. In the first pass of scanning the image, insertion of a node in the equivalence table at a foreground pixel

costs constant time. Adding equivalence is broken up into finding the roots of trees and merging the two trees (e.g. step 7). Merging two trees costs constant time. Finding the root for a certain tree depends on level of the tree, where the worst case is searching from the farthest leaf to root. The levels of trees in equivalence sets are highly relevant to the shapes of connected components. A large number of levels in the equivalence set could appear in some images with irregularly shaped connected components, especially connected components with numerous branches and spurs. Figure 17 shows a single connected component which generates a 5-level tree in its equivalence set. Only when two trees are of the same level, can the level of the merged tree increase by 1. Every tree represents a preliminary connected component in the binary image. Thus the highest level of trees in equivalence sets cannot exceed the number of preliminary connected components. To simplify the problem, we restrict binary image labeling in metaphase images. Since most metaphases selected by ranking module are 'nice' images, chromosomes in these metaphases are in a neatly separated distribution with regular shapes. A total of 46 chromosomes plus background noise is a reasonable estimation for number of connected components in a given metaphase. The regular shape of objects in metaphases limits the number of its preliminary connected components. In high ranked metaphase images, the number of total preliminary connected components can be counted as a constant. Resolving the equivalence set costs time linear to the number of preliminarily-connected components. In the second pass of scanning, finding roots of trees is called at all foreground pixels. The time cost in serial binary image labeling can be expressed as Equation (3.2):

$$t \leq c_1 f n + c_2 n + c \tag{3.2}$$

where f is the average percentage of foreground pixels in metaphases and n is number of pixels in a metaphase. In Equation (3.2), $c_1$is the time needed to label a foreground pixel and add an equivalence, c_2 is the time required to scan a background pixel, and c is the time cost to resolve the equivalence table. Values c_1 c_2 and c are constants relevant to the average number of preliminary connected components and hardware. The time requirement for serial binary image labeling is linear to the size of image.

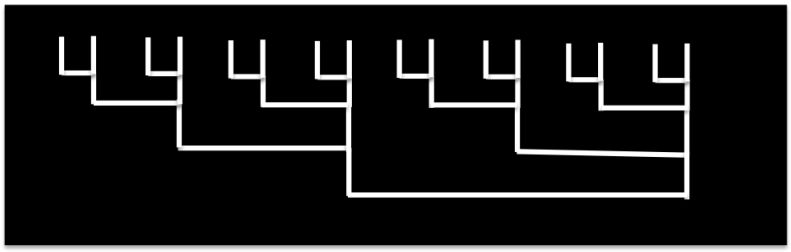

Figure 17: A binary image of a single connected component that generates a five-level tree in its EQTABLE. This situation rarely happens in metaphase images.

This algorithm can be designed with a divide-and-conquer strategy. The parallel binary labeling algorithm takes an image to be labeled and divides it into several sub-images. If these sub-images satisfy the recursive condition, they are labeled by recursively calling the parallel binary image labeling method. Otherwise sub-images are labeled by the serial algorithm. Labeled sub-images are combined to form a labeled complete image. A drawback of this method is that in the combining phase, the combination of labeled sub-images requires a re-scan and a re-label step. This includes all sub-images, except the first, in order to avoid index number conflict in different sub-images. Individually parallelizing the three steps in binary image labeling can avoid this problem. The parallel algorithm used in ADCI is explained as follows. In the first pass of scanning, a metaphase image is divided to 2 sub-images in horizontal direction, and sub-images are divided recursively in the same way until the size of sub-images meets stopping condition, which we call ending sub-images. Each ending sub-image completes its first scan in serial, returning a local equivalence set and the corresponding labeled ending sub-image. This process can be executed in parallel for all ending sub-images. Before resolving equivalence set, local equivalence sets from sub-images must be merged to a global equivalence set. Generating the global equivalence set is executed in a parallel divide-and-conquer manner. The metaphase is divided in the same way as in its first pass of scanning. The global equivalence set for the current image is combined from two local equivalence sets returned from its upper sub-image and lower sub-image. Initially, the global equivalence set is a simple union of the two local equivalence sets. The last line in the upper half-image and the first line in the lower half-image are scanned and compared. If two connected components belonging to different sub-images are actually connected,

an equivalence is added in the global equivalence set. The global equivalence set is returned as a local equivalence set if the current image is also an intermediate sub-image. If the current image is the complete metaphase, the global equivalence set is found and is resolved serially. In the second pass of scanning, each ending sub-image resulting from the first pass of scanning is labeled with help of the global equivalence set serially. This process is executed in parallel for all ending sub-images and consequently the complete metaphase is correctly labeled. Figure 18 shows evenly dividing a metaphase into two sub-images in the horizontal direction. Figure 19 shows recursively division of Figure 18 into four sub-images.

Figure 18: Dividing a binary metaphase to two sub-images evenly in the horizontal direction. Upper sub-image and lower sub-image can be processed in parallel.

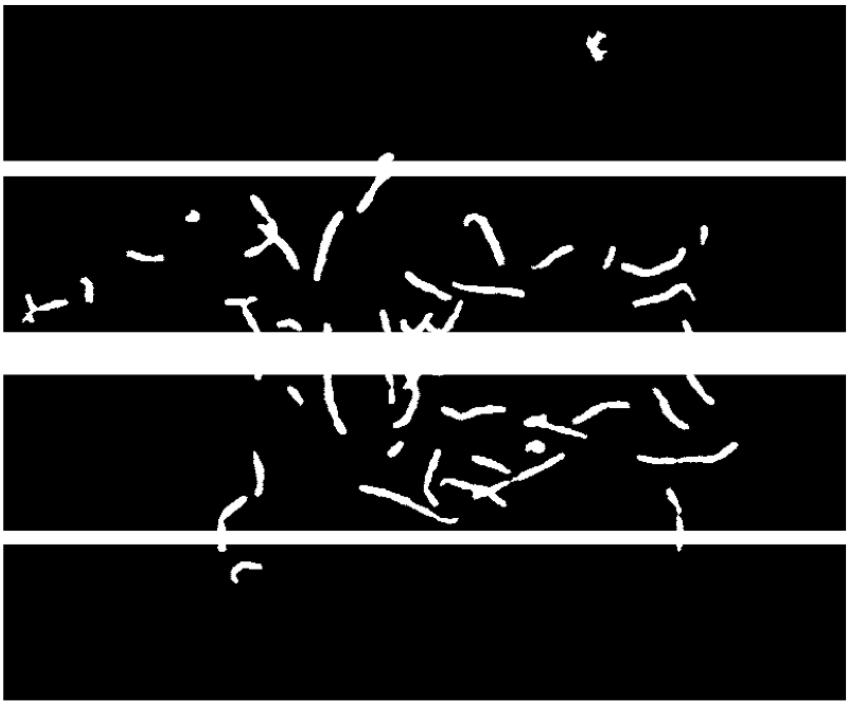

Figure 19: Dividing the binary metaphase in Figure 18 to 2 sub-images and recursively dividing them into 4 sub-images. Labeling work can be shared by 4 concurrent threads.

The time cost of parallelized binary image labeling is approximated in Equation (3.3) (overhead not included):

$$t' \leq \lceil k/p \rceil (c_1' fn/k + c_2' fn/k) + \lceil k/p \rceil r c_3' \log_2 k + c'. \tag{3.3}$$

In Equation (3.3), k is the number of ending sub-images, p is the number of processors, or more specifically threads, n is the number of pixels in metaphase, f is the average percentage of foreground pixels and r is width of a metaphase. The values c_1' and c_2' define the constant operation time of adding an equivalence in the first and second pass of scanning. The value c_3' defines the constant operation time of adding an equivalence in combining two local equivalence sets. All constants, including c', are relevant to number of preliminary connected components and hardware. From Equation (3.3), we know that the time requirement of parallel binary image labeling is approximately reciprocal to the number of processors. It is also strongly relevant to the number of ending sub-images. Figure 20 illustrates an example of the DAG analysis of the parallel binary image

labeling algorithm. When parallel overhead is not counted, length of the critical paths in the first and second passes of scanning sub-images and in the merging of equivalence tables are 1 and $\log_2 k$ respectively. This implies the rationale of Equation (3.3).

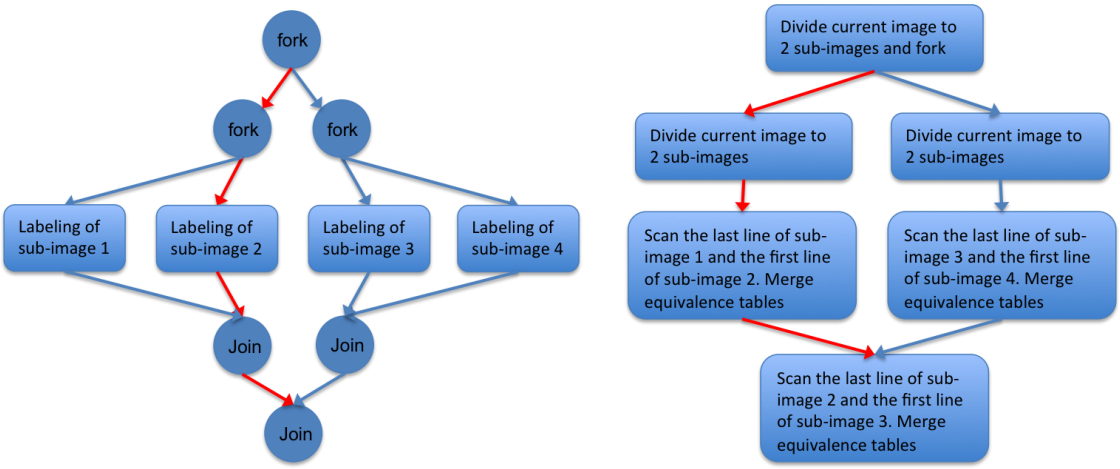

Figure 20: DAG analysis of the parallel binary image labeling when dividing a metaphase to 4 sub-images in fork-join parallel schema. Left figure is the DAG of parallel loop of the first and second scanning. Right figure is the DAG of parallel divide-and-conquer algorithm of merging equivalence tables (Fork and join are omitted). Red lines show the critical paths.

procedure Parallel Binary Image Labeling **input**: binary image IMAGE **output**: index label image LABEL
LABEL = new zeros matrix of the same size of IMAGE; Evenly divide IMAGE and LABLE to K sub-images: IMAGE[1:K] and LABEL[1:K]; **parallel for** L = 1 to K EQTABLE[L] = new EQTABLE; first scan and label of IMAGE[L]; **end parallel for** Divide-and-Conquer Merging(IMAGE, EQTABLE); **parallel for** L = 1 to K

```
   second scan and label of IMAGE[L];
end parallel for
```

Algorithm 5: Pseudocode of parallel binary image labeling

```
procedure Divide-and-Conquer Merging
input: binary image IMAGE, equivalence tables EQTABLE
output: global equivalence table GTABLE
IMAGE1 = upper half of IMAGE;
IMAGE2 = lower half of IMAGE;
if IMAGE > stop condition
   GTABLE1 = Divide-and-Conquer Merging(IMAGE1, EQTABLE);
   GTABLE2 =  new thread (Divide-and-Conquer Merging(IMAGE2, EQTABLE) );
end if
   scan the last row of IMAGE1 and the first row of IMAGE2;
   GTABLE = Merge EQTABLE[IMAGE1] and EQTABLE[IMAGE2];
```

Algorithm 6: Pseudocode of Parallel Divide-and-Conquer merging of equivalence table.

3.2.2 Parallelizing Inversion of Circulant Tri-diagonal Matrices used in GVF

To solve the energy function in GVF module, circulant tri-diagonal matrices have to be repeatedly inverted. These matrices are circulant, sparse but not strictly tri-diagonal, whose general form is expressed in Equation (3.4), where dots stand for zeros.

$$M = \begin{bmatrix} b & c & \cdots & a \\ a & b & c & \cdots \\ \vdots & \ddots & \ddots & \vdots \\ c & \cdots & a & b \end{bmatrix} \tag{3.4}$$

In terms of computational complexity, matrix inversion is a NC class problem [45]. The NC class is a set of problems which can be solved in poly-logarithmic time by parallel computers with a polynomial number of processors [45]. Belonging to NC class problems means that matrix inversion has efficient parallel algorithms available. There have been a lot of serial algorithms developed for matrix inversion. Gaussian elimination is the most general algorithm to invert a matrix. An alternative method of Gaussian elimination is LU decomposition, generating upper and lower triangular matrices which can be inverted more easily [21]. OpenCV uses this method by choosing optimal pivot elements to invert matrices. Some special matrices provide faster approaches for inverse such as eigen-decomposition and Cholesky-decomposition [21]. The time complexity of inverting an n-by-n matrix by Gaussian elimination is $O(n^3)$.

The inverses of circulant matrices are also circulant matrices. In this case, inversion of a matrix can be simplified to solving the linear system represented in the matrix. The Fourier Transform is widely used in solving circulant linear systems. The Fast Fourier Transform (FFT) method can give fast inversions for circulant matrices. In one implementation of FFT, library FFTW [46], the time complexity of inverting an n-by-n matrix in serial is $n \log n$. If infinite processors are provided, inverting an n-by-n matrix in parallel by FFTW can be achieved in $\log n$ time. As the matrices to be inverted in GVF are always sparse tri-diagonal, a fast solution for solving the linear system is the Thomas algorithm [47], which is a variation of Gaussian Elimination. To solve the target matrices in ADCI, shown as Equation (3.5), a slightly modified Thomas algorithm is demonstrated as follows.

$$\begin{bmatrix} b_1 & c_1 & 0 & \cdots & 0 & a_1 \\ a_2 & b_2 & c_2 & 0 & \cdots & 0 \\ & \ddots & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & & & \ddots & \ddots & \ddots \\ c_n & 0 & \cdots & 0 & a_n & b_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ \\ \\ x_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ \\ \\ d_n \end{bmatrix}. \tag{3.5}$$

For $1 < i < n, i = 1 + k + 2km, m = 0, 1, 2, \ldots$ we can substitute x_i in the unknown vector with the following expression:

$$x_i = (d_i - a_i x_{i-i} - c_i x_{i+i})/b_i. \quad (3.6)$$

We repeat this process with $k = 2^j, j = 0, 1, 2, \ldots$ until there are only x_1 and x_n left in the unknown vector. For an n-by-n matrix, $n - 2$ substitutions in $\log_2 n$ repeats are required. The linear system is reduced to Equation (3.7):

$$b_1 x_1 + c_1 x_2' + a_1 x_n = d_1 \quad (3.7\text{ a})$$
$$c_n x_1 + a_n x_{n-1}' + b_n x_n = d_n \quad (3.7\text{ b})$$

where x_2' and x_{n-1}' are expressed by x_1 and x_n. Equation 3.7 can be solved by Gaussian elimination quickly and all unknowns in the unknown vector can be obtained by back substitutions.

Implementation of Thomas algorithm is also called Odd-Even reduction [48]. In the forward substitution phase, even rows in the matrix are eliminated by substituting the row above and the row below with elements in the current row. This process is repeated until only the first and the last rows remain. After solving this two-row matrix, unknowns are solved in back substitution phase in the reverse order of the forward substitution phase. Every row in the matrix is only eliminated once in forward substitution and every unknown is calculated once in back substitution. This makes the time complexity of Odd-Even reduction $O(n)$.

The elimination for even rows in forward substitution and solving unknowns in back substitution can be executed in parallel. The time cost of parallel Odd-Even reduction can be calculated as Equation (3.8):

$$t = \sum_{i=1}^{\lceil \log_2 n \rceil} \left\lceil \frac{n}{2^i P} \right\rceil + C \quad (3.8)$$

where $\lceil n/2^i p \rceil$ represents the work in i^{th} forward substitution and $(\lceil \log_2 n \rceil - i + 1)^{th}$ back substitution, and C represents the work to solve Equation (3.7). Assuming infinite processors are available, the time complexity is $\log n$, which is the same as parallel FFTW. To avoid importing extra 3rd party libraries, we use parallel Odd-Even reduction

in ADCI to invert our matrices. DAG analysis and pseudocode of parallel Odd-Even reduction are given in Figure 21 and Algorithm 7.

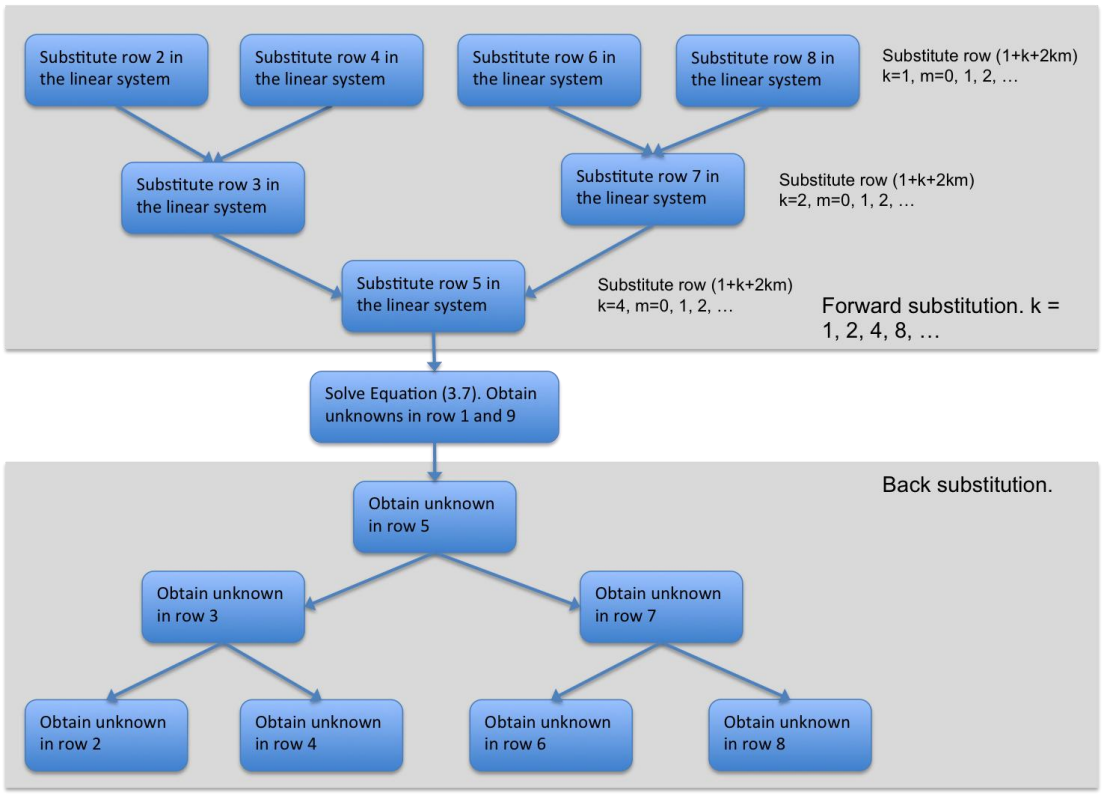

Figure 21: An example of DAG analysis of parallel Odd-Even reduction in inversion of a 9-by-9 matrix in ADCI. Fork and join are omitted. In both the forward and back substitutions, $\lceil \log_2 n \rceil$ (3 in this case) layers of substitutions are required. In each layer of substitution, execution can be parallelized in parallel loop.

procedure parallel Odd-Even reduction **input:** n-row (n+1)-column Linear system L representing a n-row n-column Circulant Tri-diagonal Matrix M

Current_rows = n;
 	
Step = 1;
 	
for Current_rows > 2
 	
  Start = 1+Step;
 	
  I = Start;

```
  parallel for I < n
    substitute row I-1 with row I in L;
    I = I + 2*Step;
  end parallel for
  I= Start;
  parallel for I <n
    substitute row I+1 with row I in L;
    I = I + 2*Step;
  end parallel for
  Step = Step*2;
  Current_rows = Current_rows/2 + 1;
end for
Solve equations represented the first line and last line in L;
```

Algorithm 7: Pseudocode of Parallel Odd-Even reduction. Back substitution is omitted as it is simply the reverse order of the forward substitution.

3.3 Data Parallelism

As mentioned in the previous sections, ADCI data parallelism is defined as parallelizing samples, metaphases or chromosomes. Unlike low-level data such as pixels in image or elements in array, parallelizing these high-level data divides the whole data set to be processed by ADCI. This yields a high degree of parallelism and notable speedup in processing time. In ADCI, there are two levels of data parallelism: multi-thread data parallelism and MPI data parallelism.

3.3.1 Multi-thread Data Parallelism

Multi-thread data parallelism is used both in the ADCI desktop (assuming the desktop is able to run in multiple threads) and in the ADCI cluster. In multi-thread data parallelism, metaphases are distributed to multiple threads to be processed. The reason why

metaphases instead of samples are parallelized is that the data set for the ADCI desktop is small, containing only a small number of samples. Under these conditions, parallelizing metaphases may be more efficient. It also keeps consistency of multi-thread parallelism in the ADCI desktop and in the ADCI, where samples are parallelized in MPI data parallelism. Division and allocation of metaphases is executed through a parallel loop. Two outstanding advantages of multi-thread parallelism are the use of shared memory by multiple threads, which saves parallel overhead on communication, and the built-in scheduling mechanism that saves programmer's work.

Multi-thread data parallelism appears in ranking module and all the modules following. In ranking module, image preprocess, feature extraction, classification and scoring for all metaphase are executed in parallel. In order to finish the preceding processes for all metaphases in a given sample, synchronization barriers are put in place before normalizing and ranking occur. Normalizing and ranking are executed in serial because of their dependency on data from other metaphases in the same sample.

3.3.2 MPI Data Parallelism

Most computing clusters are organized as distributed computing nodes connecting by a LAN. Each computing node in the cluster has its local memory space that can only be shared by processors resident on this node. MPI provides a convenient communication mechanism known as message passing which transfers data and information across computing nodes. However, the speed of passing messages depends on the size of data to be passed and the performance of the hardware. In programming MPI in ADCI cluster, we wanted to keep the frequency of communication and the size of data to be passed as small as possible. Parallelizing samples is a good way to fulfill for this constraint. The ADCI process for one sample is not dependent on the ADCI processes of other samples. The only exception is the communication required to schedule and balance workload, and there is no additional communication necessary for standalone MPI processes. As sample files are stored in file system, which is commonly managed by file system servers connected to every computing node, scheduling and assigning samples does not involve actual sample or image transfer. Messages passed to schedule work only need to indicate

MPI processes. These messages indicate which samples should be queued for the next data input. MPI processes can load samples from the file system by itself.

Scheduling work (samples) in multiple processes is important in ADCI cluster, since MPI does not handle this task automatically. This is also the reason why we do not run multiple serial ADCI programs on cluster ADCI. There are two general schedule paradigms in multi-thread parallelism: work sharing and work stealing [32]. They are mainly developed for balancing threads across processors or cores in multi-thread parallelism. In work sharing, whenever new threads are created in a processor, the scheduler tries to migrate some of them to other processors in order to reduce the load on itself. In work stealing, whenever the thread queue of a processor is empty or in low load, the scheduler tries to steal some threads from other processors to fully utilize processors resource [32]. In scheduling and balancing the workload in MPI, there are similar mechanisms. In work sharing, all work is stored in a centralized queue managed by a scheduler MPI process. Whenever a MPI process is underutilizing available resources and is available to process more work, it has to apply to the scheduler MPI process for more work, and the scheduler will respond to this application. In work stealing, all MPI processes have their local work queues. If an MPI process needs supply of data for its local work queue, it steals some work from other MPI processes.

There is research showing that work stealing has comparative, if not better, performance than work sharing [49]. Currently, work sharing is applied in ADCI cluster. Samples are divided to work chunks, which are the basic process units for a MPI process in ADCI cluster. There is a ‘Scheduler’ MPI process maintaining a global work queue containing all samples. Every MPI process keeps a local work queue, and initially some samples are assigned to every local work queue from the global work queue. When the load in a local work queue is low, an MPI process sends a request to the Scheduler MPI process to begin to run new chunks (of samples). The Scheduler MPI process checks requests in loop within a certain time limit or within a specific thread, assigning work chunks to MPI processes that send requests until the global work queue is empty. Figure 22 illustrates work sharing mechanism in the ADCI cluster.

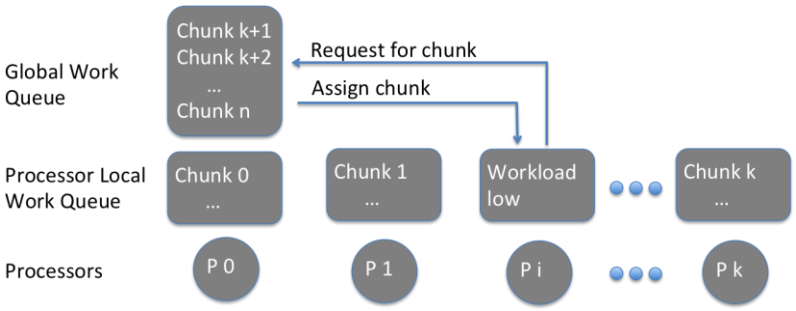

Figure 22: Work sharing scheduling in ADCI cluster. MPI process 0 is the Scheduler process and manages the global work queue.

Chapter 4

4 Experiments and Discussion

We performed several experiments to validate our method in parallelizing ADCI and to explore the effect of parallelism in accelerating ADCI. Effectiveness of task parallelism was verified in unit tests. Separate modules were experimented with data to test efficiency of task-parallelized functions when integrated into ADCI. Multi-threaded data parallelism was tested in experiments using a desktop version with images from a few samples and using a SMP supercomputer to process 18,694 samples. For comparison, MPI data parallelism was tested using a small cluster machine with 1000 samples. In order to investigate the performance of parallelized complete ADCI software, the parallel ADCI desktop and ADCI cluster were tested in a multi-core desktop and a supercomputer cluster, respectively.

4.1 Experiments of Task Parallelism

We tested task parallelism on an 8-core i7 Linux desktop, which is able to run 8 logic threads in parallel. Parallelism is implemented with Intel TBB.

4.1.1 Experiments on Parallel Binary Image Labeling

As mentioned in Chapter 2, the stopping condition in parallel binary image labeling is the size of an ending sub-image that controls the total number of ending sub-images processed in parallel. In our implementation, we used the ratio of sub-images' height to the metaphase's height as the stopping condition. This condition was optimized with several values in unit tests and adjusted to find the optimal number of sub-images.

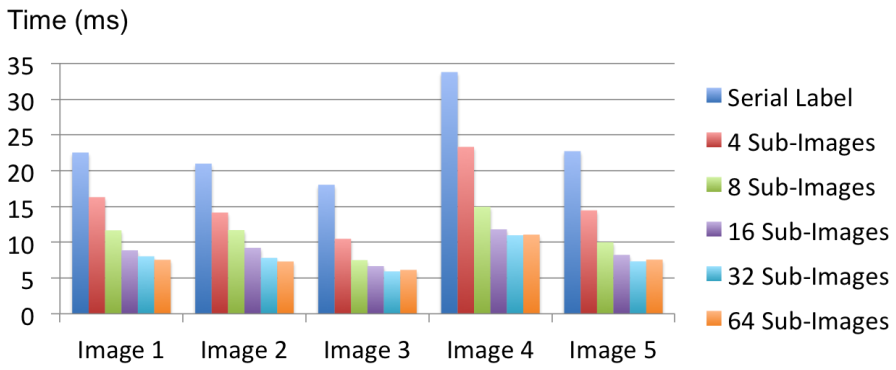

Figure 23: Comparison of serial and parallel binary image labeling for different number of ending sub-images. Binary image labeling of different versions and stopping conditions are distinguished by colors in bars. Serial binary image labeling and parallel binary image labeling with 5 different stopping conditions were tested on 5 randomly selected metaphases. For example, '4 sub-images' denote that the ending sub-images are at ¼ height of the original metaphase, and thus there are 4 ending sub-images to be processed.

Figure 23 shows the result of a unit test of parallel binary image labeling. Five tested stopping conditions includes the height of ending sub-image at 1/4, 1/8, 1/16, 1/32, or 1/64 of the height of the input metaphase. The times to label these five input metaphases by different versions of parallel binary image labeling are compared with each other and with serial binary image labeling. For all five tested metaphases, the parallel versions of binary image labeling were faster than the serial version. The largest speedup gained from parallelism came from parallel binary image labeling with 32 ending sub-images on image 5. The serial labeling on image 5 cost 22.73 milliseconds while the parallel labeling with 32 ending sub-images only took 7.332 milliseconds. This was 32.3 percent of the time cost by serial. The least improvement observed was 72.3 percent of the time requirement of serial labeling, which is obtained from parallel binary image labeling with 4 ending sub-images on image 1. On average, in these five stopping conditions, additional ending sub-images lead to a larger speedup in performance. Parallelization with 32 or 64 ending sub-images parallelism brought similar average performance improvements. Compared with serial labeling, these cost 34.1 and 33.7 percent of the time, respectively. However 32 ending sub-images parallelism were faster than 64 ending

sub-images among 3 tested metaphases. We took 32 ending sub-images as the stopping condition in following experiment.

According to the time complexity calculation in chapter 2, the theoretical time cost of parallel binary image labeling approximately decreases reciprocally with the number of processors used. In the experiment, for the best improvement we can observe 3-fold speedup. A feature of a selected metaphase is that most chromosomes and connected components are concentrated in the center of the image. If sub-images are divided evenly, the number of connected components in sub-images physically located at the center of a metaphase will be much larger than the number of connected components in other sub-images.

We tested the ranking module integrated with parallel binary image labeling on samples, with 200 metaphase images per sample. The average processing time is shown in Figure 24 (a). In serial ranking module, 200 metaphases took 10.76 seconds to rank. Ranking the same metaphases with parallel binary image labeling took 8.513 seconds, resulting an approximately 1.25 fold speedup.

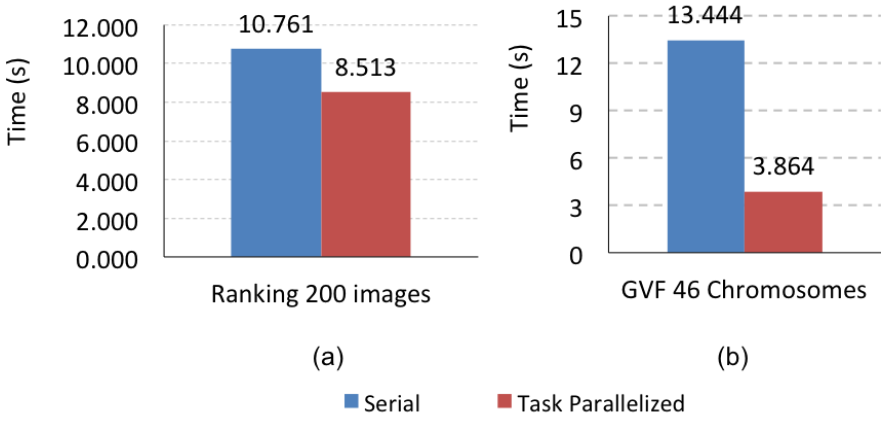

Figure 24: Comparison of serial modules and parallel modules. Panels a and b demonstrate experiments in the Ranking and GVF. Blue and red bars show the time cost by modules with serial functions versus task parallelized functions, respectively.

4.1.2 Parallelized Matrix Inversion Experiments

In the unit test for parallel Odd-Even reduction, we implemented parallelization in both Cilk and Intel TBB. Cilk provides a useful tool, Cilk View, that can collect parallel information while a Cilk program is running. It can also benchmark and analyze the collected data to give a better understanding of the performance of the parallel algorithm. Cilk was used to investigate the performance of parallel Odd-Even reduction. The TBB version of the Odd-Even reduction was used in ADCI. A 300-by-300 circulant tri-diagonal matrix was tested with Cilk Odd-Even reduction. Figure 25 shows the analysis by Cilk View. Figure 25 (a) gives the estimated possible speedup with different number of processors. In this case, a 9.97 fold increase is the upper bound of possible speedup. Figure 25 (b) plots the data in Figure 25 (a). The analysis given by Cilk View implied that parallelism could effectively accelerate Odd-Even reduction by up to this level.

In the unit test of TBB version parallelism, we compared the Odd-Even reduction method, TBB parallel Odd-Even reduction and the matrix inversion function provided by OpenCV on circulant tri-diagonal matrices of various sizes. OpenCV implements Gaussian Elimination with LU decomposition to invert matrices. When the size of the matrix is large, Gaussian Elimination may cause rounding errors. In ADCI, the matrices to be inverted are diagonal dominant and in OpenCV, Gaussian Elimination is implemented with optimal pivot element choosing. These two facts guarantee that Gaussian Elimination returns correct results in our test. Table 1 shows this comparison. In Gaussian Elimination provided by OpenCV, the time cost increases rapidly as the size of target matrix increases. Serial and parallel Odd-Even reduction can significantly control this time cost as the matrix size increases. When matrix sizes exceed 200 elements, the parallel Odd-Even reduction has better performance than the serial version. Because of parallelization overhead, the serial Odd-Even reduction is slightly faster than the parallel version for the matrices for those smaller than 100-by-100. In ADCI, Odd-Even reduction instead of the Gaussian Elimination is always utilized.

(a)

```
Parallelism :                                9.97
Burdened parallelism :                       9.97
Number of spawns/syncs:                      1467
Average instructions / strand :              17478
Strands along span :                         108
Average instructions / strand on span :      71438
Total number of atomic instructions :        0
Frame count :                                3016
Speedup Estimate
2 procs:              1.71 - 2.00
4 procs:              2.65 - 4.00
8 procs:              3.65 - 8.00
16 procs:             4.50 - 9.97
32 procs:             5.09 - 9.97
```

(b)

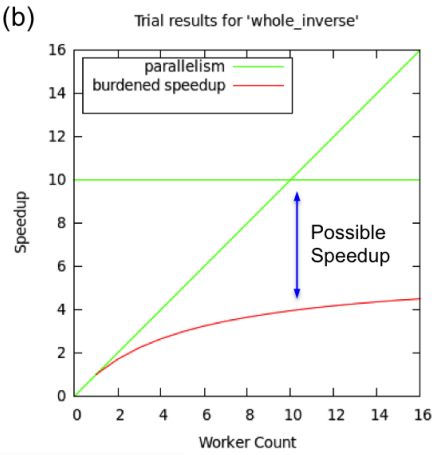

Figure 25: Parallel information Odd-Even reduction matrix inversion. Part 'a' gives the maximum speedup of the parallel program in Cilk as well as the estimation of speedup when utilizing different numbers of processors. Panel 'b' plots part 'a' where red and green lines illustrate the lower and upper bounds of possible speedup. The horizontal green line is defined by the limitation from the critical path in the program and the diagonal green line shows the speedup restricted by number of available processors.

Table 1: Comparison of Parallel Odd-Even reduction and Gaussian Elimination. Matrix size stands for the number of rows in the square matrix.

	Average time for Circulant Tri-diagonal Matrix Inversion (millisecond)		
Matrix size	Gaussian Elimination provided by OpenCV	Serial Odd-Even reduction	Parallel Odd-Even reduction with 8 cores
100-200	5.8399	0.1955	0.2134
201-300	27.0717	0.5223	0.44
301-400	75.311	1.0427	0.7223
401-500	271.7779	1.7316	1.1422
501-600	818.2553	2.6299	1.7553

We tested the GVF module integrated with parallelized Odd-Even reduction on chromosomes. The average times to process 46 chromosomes in both versions of GVF module are shown in Figure 21 (b). Original GVF module using Gaussian Elimination required an average 13.4 seconds to find contours for 46 chromosomes. The GVF module integrated with parallel Odd-Even took 3.9 seconds to process these same chromosomes.

4.2 Experiments of Data Parallelism

In experiments of data parallelism, the ADCI has been tested on a variety of computing systems. Data parallelism is performed prior to task parallelism. Therefore, in the complete ADCI, task parallelism is applied only when there is sufficient parallel computing capacity remaining after data parallelization has taken place. In the following experiments testing data parallelism, all parallel computing capacity was applied in data parallelism.

4.2.1 Experiment of ADCI desktop on Desktop

We tested data parallelism of the individual modules in the ADCI on the same 8-core i7 desktop that was used in experiments of task parallelism. Data parallelism of metaphase-level was tested in the Ranking module and data parallelism at the chromosome-level was tested in the GVF and centerline-based modules (DCE, Spline Interpolation, and Centromere detection). Table 2 shows the details of these experiments.

Table 2: Experiment summary of data parallel modules on an 8-core desktop.

	Ranking (seconds)	**GVF** (seconds)	**Centerline based modules** (DCE + Interpolation + Centromere) (seconds)

Time cost of serial version	10.8	13.4	3.50
Time cost of data parallelism version	2.22	1.76	0.70
Speedup	4.8 fold	7.6 fold	5 fold
Data size	200 images	46 chromosomes	

Data Parallelism using 8 processors on the Ranking, GVF and Centerline based modules can increase processing speeds 4.8, 7.6 and 5-fold respectively. The GVF module approximately reached the theoretical optimal speedup, which is 8-fold. The Ranking module is unable to perform data parallelism throughout, due to dependency on individual metaphase images. Therefore, it is reasonable that data parallelization only gave a 4.8-fold speedup for the ranking module. The centerline-based modules did not achieve optimal speedup due to workload unbalances. Parallelizing processing of metaphase chromosomes is likely to result in unbalanced work across multiple threads for several reasons. The A and B group chromosomes, for example chromosomes 1 and 2 are typically longer than others than others in a metaphase image, and require more time to process. Additionally, among the 'nice' metaphases, there may be a number of chromosomes that touch or overlap each other, forming a chromosome cluster. Processing a chromosome cluster requires more time than processing a normal chromosome. This results in an unequal work distribution among threads for each metaphase cell. When dividing the iteration space in a parallel loop, multi-thread parallel platforms like OpenMP and TBB can split the work on the basis of iteration index. However in our case, the workload in iterations is biased based on the size and distribution of the chromosomes in an image, neither of which is predictable until image processing begins.

The fully functional ADCI desktop was also tested on the same hardware. Average time to process a sample (250-300 images) and time stamps recorded by individual steps are shown in Table 3.

Table 3: Data Parallelized ADCI desktop on an 8-core desktop. Average time recorded to process one sample (250-300 images) in seconds

Average time to process a sample by modules	**Serial**	**Data Parallel with 8 threads**
Ranking modules	17.6034	4.12013
Other modules on chromosomes	144.803	38.2106
Fully functional ADCI	163.268	43.4368

The average time of processing a sample throughout the serial ADCI desktop is 163.27 seconds. The data parallelized ADCI desktop with 8 threads requires 43.44 seconds, approximately ¼ of the time for the same sample. However, the file loading and result committing functions increase overhead.

4.2.2 Experiment of ADCI Module on Symmetric Computing System

We tested the ranking module of ADCI on the Symmetric Computing (Boston) system with large dataset. Symmetric Computing provides two types of SMP systems with large shared memory for high performance computing [51]. We accessed the Trio shared-memory supercomputer in Symmetric Computing system, where up to 1.5 TB memory and 192 processor cores with AMD Opteron 6200 series processor are available. In this test, up to 64 cores were used to process 18,694 samples (each consisting of 250-300 metaphases) using the Ranking module parallelized with TBB, This is the same configuration that was performed in the experiment with the 8-core desktop. Figure 26 illustrates the result. When 4 processors were used, ranking module took 11.4 hours to process all of the samples. The fastest process was obtained when 40 processers were

used, taking 0.65 seconds on average to rank each sample and 3.38 hours to rank all samples. When more than 40 processors were used, the performance decreased slightly. This was not a satisfactory result. It might be caused by the imbalance in work distribution when the number of threads is large. However due to limited access time to Trio (48 hr.), we could not fully assess the cause of the plateau in performance. This experiment was performed in the early-stage development of ADCI, where only the ranking module was tested, and the parallelization implementation might not have been fine tuned. The other possibility is that, beyond 40 cores, overhead becomes a detrimental factor affecting the performance of the program.

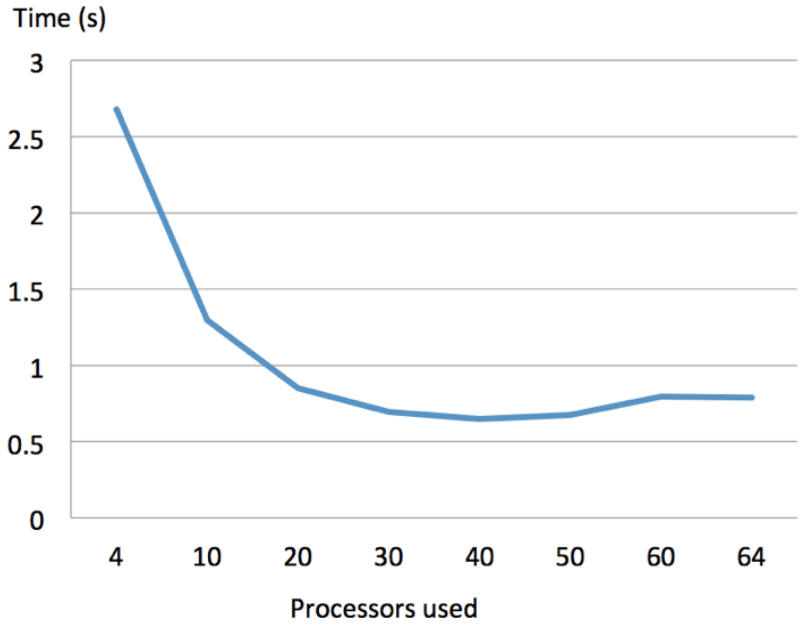

Figure 26: Parallel ranking module on a SMP system with 18694 samples. Average time to rank all metaphases in one sample is displayed on the Y-axis.

4.2.3 Experiment of ADCI Module on Cluster

To evaluate the performance of the ADCI on cluster, the ranking module on the Goblin cluster in SHARCNET was performed. SHARCNET consists of more than 20 HPC systems connected by a wide area network, including large-scale distributed clusters and SMP systems [52]. A group of workstations are also connected to the network and all systems share a global file system.

In this test, 8 computing nodes in Goblin system with 8 cores per node were employed to run MPI parallelized ranking module for 1000 samples. Each node in Goblin keeps an 8-core Intel Xeon processor and up to 48 GB memory. Nodes are connected by a gigabit Ethernet connection and a total 64 MPI processes were started in all these cores. Samples were scheduled and balanced by work-sharing scheduling. Table 4 shows the summary of this test. The total time of all MPI processes to complete the Ranking module was 434 seconds, which was determined based on the slowest MPI process. The fastest MPI process cost 101 seconds to complete its work. The average processing time for one sample was 0.434 seconds.

Table 4: Data-parallelized ranking module on Goblin, SHARCNET

	Total processing time of the fastest MPI process	Total processing time of the slowest MPI process	Average processing time for one sample
Time (sec)	101	434	0.434

Image files for each sample in this test were organized within the same directory, as opposed to metaphase grids (see Section 2.4.2). We discovered the bottleneck in throughput to be the I/O latency in this test. Although the processing speed of parallel ranking module was fast, loading metaphase images delayed the total time of parallel ranking module. In the worst case, the time spent on loading a sample, 250-300 individual metaphase images, could be 10-fold higher than the actual time required to process the images. We tested asynchronous I/O to overlap I/O work with real image processing in ADCI. Every MPI process maintained a local data buffer large enough to contain several samples. MPI processes fetched samples from their respective local data buffer to process by ranking module. The asynchronous I/O thread in a MPI process would load samples from file system whenever the MPI process was assigned samples and the local data buffer was not full. In this manner, the I/O capacity was fully employed. If the I/O speed is reduced due to an I/O traffic peak, MPI processes do not have to wait as long as there are samples deposited in the local data buffers. Following an I/O traffic peak, local data buffers can be supplied by a relatively high-speed I/O. Table 5

shows the comparison of ranking module with synchronous blocking I/O and with asynchronous I/O. Average time to load and rank one sample with synchronous blocking I/O strategy on Goblin is 12 seconds. With asynchronous I/O, it still requires an average 11.3 seconds to accomplish the same work.

Table 5: Influence of Asynchronous I/O on Goblin, SHARCNET

I/O strategy	Average total loading and ranking time on one sample
Synchronous Blocking I/O	12 seconds
Asynchronous I/O	11.3 seconds

4.2.4 Experiment of ADCI cluster on Blue Gene/Q

In previous experiments, modules or components in ADCI were tested. In order to collect overall data and analyze the fully functional ADCI-cluster, we tested ADCI-cluster with 1025 simulated samples on the IBM Blue Gene/Q hardware. The version of ADCI-cluster that was specifically modified for Blue Gene/Q is called ADCI-BG/Q, because of the requirements of Blue Gene/Q's special PowerPC-based hardware architecture.

Blue Gene is an IBM project aimed at designing supercomputers that can achieve 10^{15} level floating-point operations per second with low energy consumption. Blue Gene/Q is the third and latest generation of Blue Gene, which is equipped and configured according to the individual requirements of each customer. The Blue Gene/Q (referred as BG/Q in the following text) that we used for testing ADCI-cluster belongs to the Southern Ontario Smart Computing Innovation Platform (SOSCIP) and is located at the University of Toronto's SciNet HPC facility. BG/Q has a very dense hardware architecture that can be divided to several layers. The basic computing units in BG/Q are comprised of computing cards. A computing card contains a single chip with a 16-core 1.6 GHz 64-bit PowerPC A2 processor and 16 GB of RAM. Every core in a PowerPC A2 processor is 4-way simultaneous multi-thread, which means 4 concurrent threads or MPI processes can be executed in one core. Computing cards are bundled in groups of 32, making a node

board. Every 2 boards are associated with a specific I/O node. Every 16 boards make up a mid-plane and 2 mid-planes make up a rack. SciNet has a 2.5-rack BG/Q with a ½-rack development part and a 2-rack production part. Computing cards are connected by a 5-dimensional optical interconnect network, with direct links to its positive and negative nearest neighbors in every dimension. From the perspective of a distributed cluster a computing card in BG/Q is considered a computing node with 64 logic cores and 16 GB memory. A computing node is directly connected with 10 nearest neighbor nodes by an optical interconnect. This network topology gives BG/Q a very fast communication speed relative to other clusters interconnected by LAN. A group of 64 computing nodes are associated with a single I/O node. Therefore, not all nodes in the cluster compete for a single file system. Figure 27 shows the sequential diagram for ADCI cluster, which is tested on BG/Q.

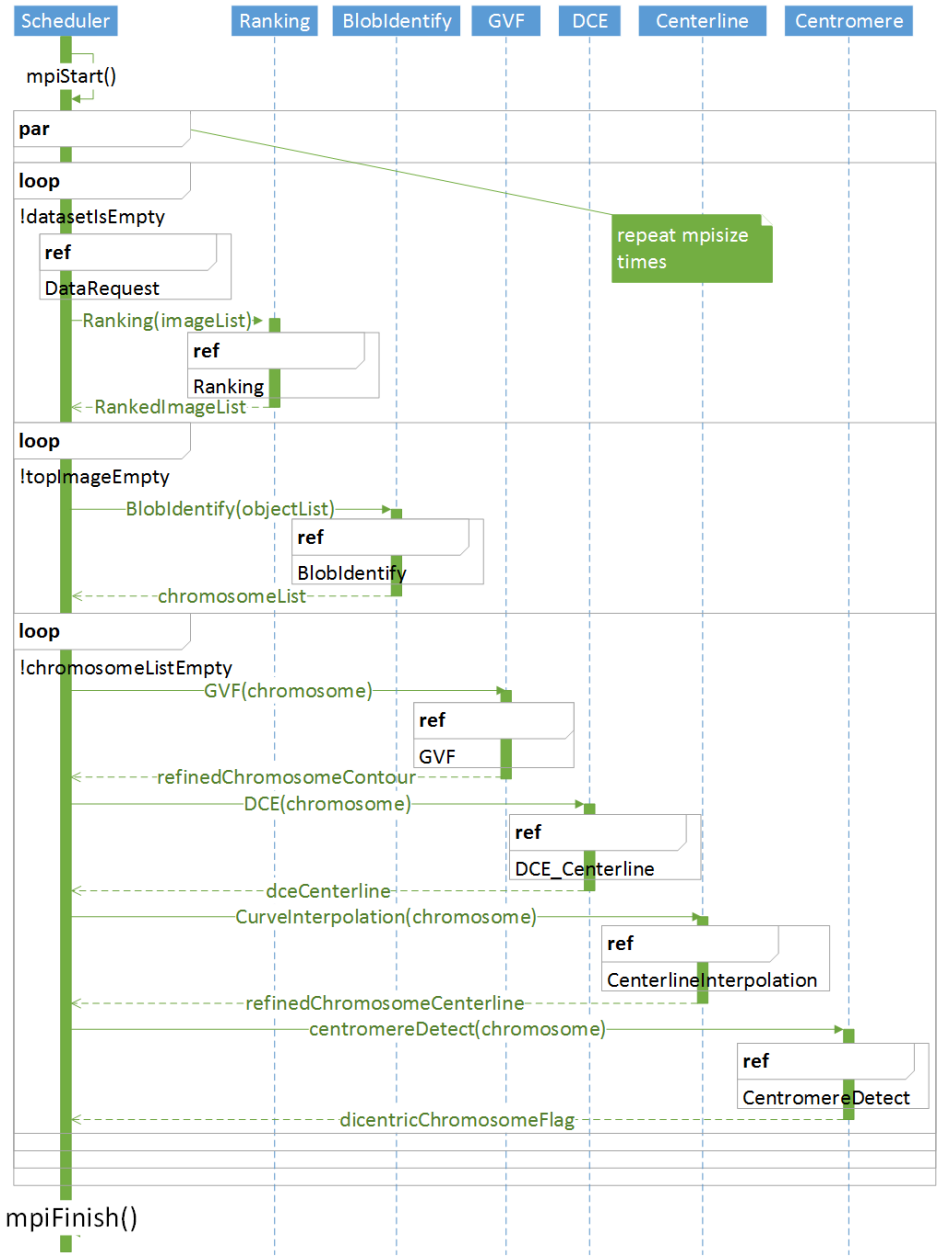

Figure 27: Sequential diagram of ADCI cluster. Blue boxes represent modules in ADCI cluster. Functional modules are resident in memory. The scheduler module parallelizes data in parallel loop and call functional modules.

In this test, 1025 samples were processed throughout ADCI. Samples were organized as metaphase grids. A total of 64 nodes containing1024 physical cores in BG/Q were utilized to process these samples. A MPI process was created for every computing node, to process samples in serial. In total, 64 MPI processes were run in parallel. Inside a

node, 64 OpenMP threads executed ranking of a sample in parallel. After the Ranking completed for a sample, 50 OpenMP threads were issued to process the top 50 metaphases in parallel. In scheduling, the chunk size was kept at 4 samples. Figure 28 shows the deployment of ADCI on BG/Q.

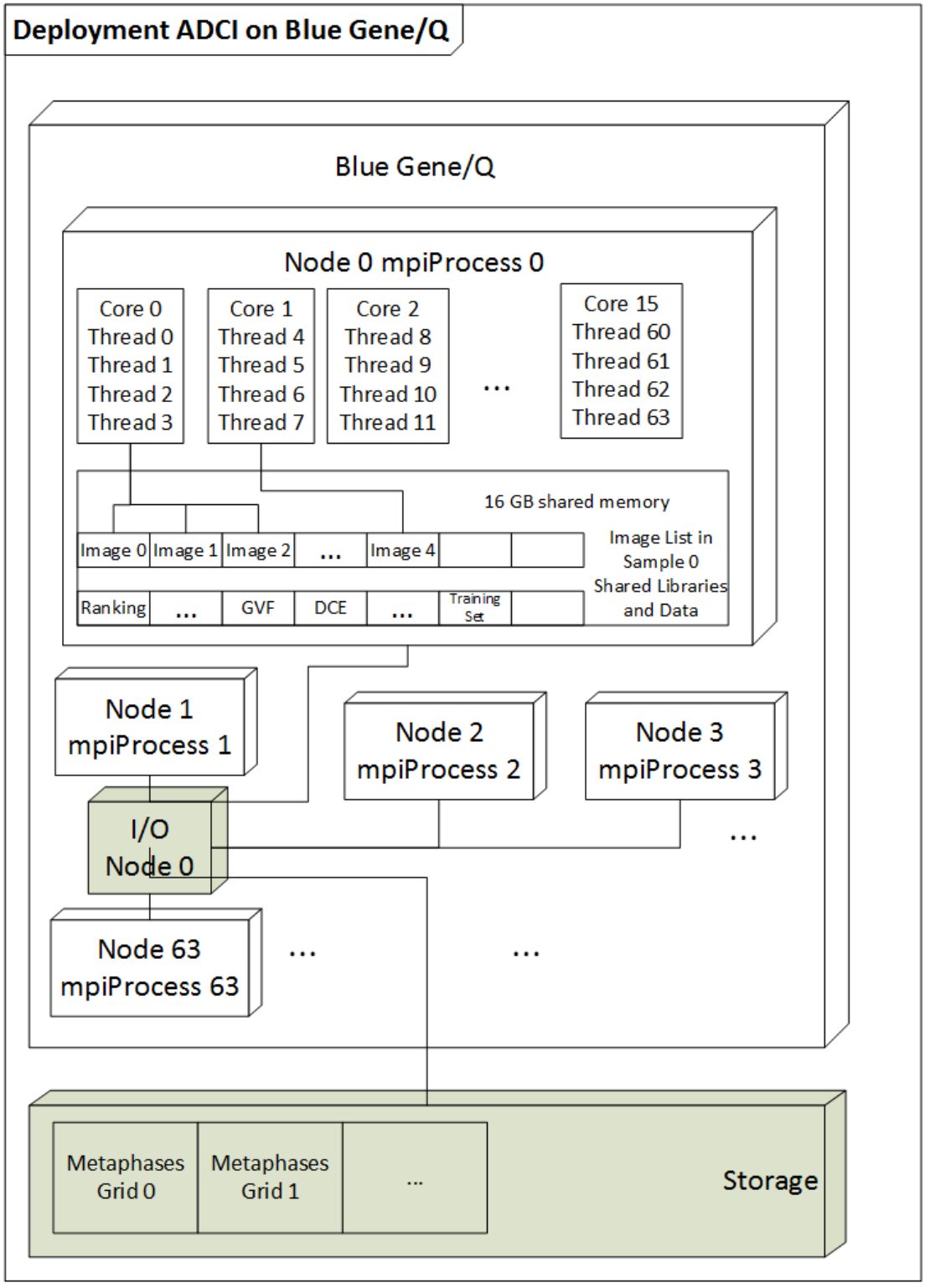

Figure 28: Deployment of ADCI cluster on BG/Q. A total of 64 MPI processes resided on 64 nodes individually with 64 or 50 OpenMP threads in each MPI process.

The total time for processing all samples on BG/Q was 5090 seconds. Some intermediate statistical data are summarized in Table 6:

Table 6: Summary of Experiment on BG/Q, time recorded by samples.

	Tiff File Loading	**Ranking Module**	**Chromosome Based Modules**	**Complete ADCI**
Data Scale	1025 tiff files	1025 tiffs * (250-300) images	1025 tiffs * (250-300) images * (~46) chromosomes	1025 tiff files
Parallel mode	64 MPI processes	64 MPI processes * 64 OpenMP threads	64 MPI processes * 50 OpenMP threads	
Accumulated Time for all samples	4 hr. 45 min. 25 sec.	19 hr. 24 min. 47 sec.	44 hr. 44 min. 7 sec.	68 hr. 54 min. 19 sec.
Maximum time among all samples	18 sec.	1 min. 12 sec.	5 min. 33 sec.	7 min.
Minimum time among all samples	16 sec.	1 min. 7 sec.	2min 16 sec.	3 min. 41 sec.
Average Time per Sample	16.7 sec.	1 min. 8.2 sec.	2 min. 37.1 sec.	4 min 2sec.
Standard Deviation of Time	0.48 sec.	0.73 sec.	53.6 sec.	53.8 sec.

We divided ADCI-BG/Q into 3 major steps (based on parallelization modes) when collecting execution information by samples. These included Tiff file loading, Ranking and Chromosome-based steps, including the chromosome classification, GVF, DCE, interpolation, and centromere modules. The Tiff file loading step was parallelized with MPI at sample-level. The Ranking and Chromosome-based steps were parallelized with MPI at sample-level and with OpenMP at metaphase-level. In Table 6, a tiff file represents a metaphase grid tiff file containing all metaphases in a sample. Accumulated time sums the time spent on all samples for each step. It gives the total workload for processing all samples. Statistical data such as the maximum and minimum time helps to understand the workload distribution among samples at each step. The processing time of a sample in File loading and Ranking steps were very consistent, as indicated by a relatively small standard deviation. A large standard deviation in Chromosome-based step signifies the unbalanced workload in this step and emphasizes the importance of scheduling workload among processors.

Figure 29 and Figure 30 shows parallelization statistics from the perspective of the compute nodes of BG/Q. Figure 29 displays the real processing time of all nodes in the experiment. The longest time is the total processing time of ADCI-BG/Q, which was 5090 seconds (1 hr. 24 min. 50 sec.) required by node 63. Figure 30 shows the number of samples processed by nodes. Most nodes processed 16 samples. Some nodes processed 20 or 12 samples, one chunk more or less compared with nodes of 16 samples.

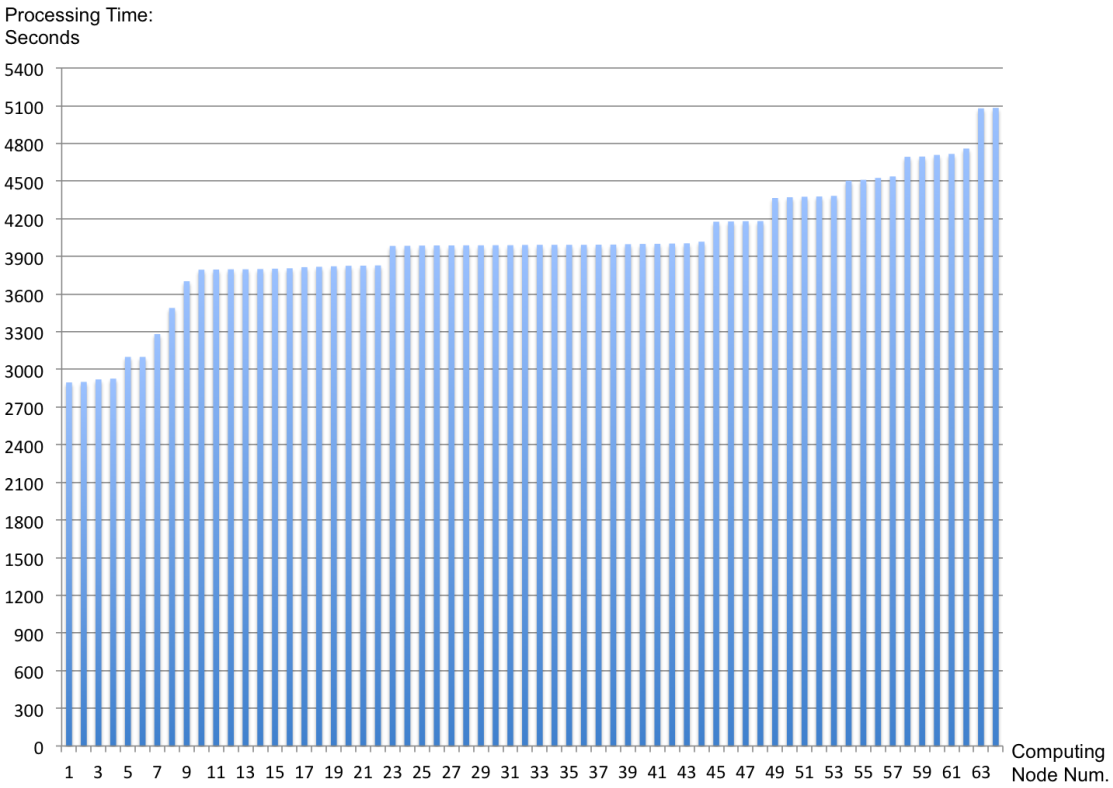

Figure 29: Distribution of processing time on BG/Q nodes. The longest processing time, 5090 seconds (1 hr. 24 min. 50 sec.), was observed on node 63.

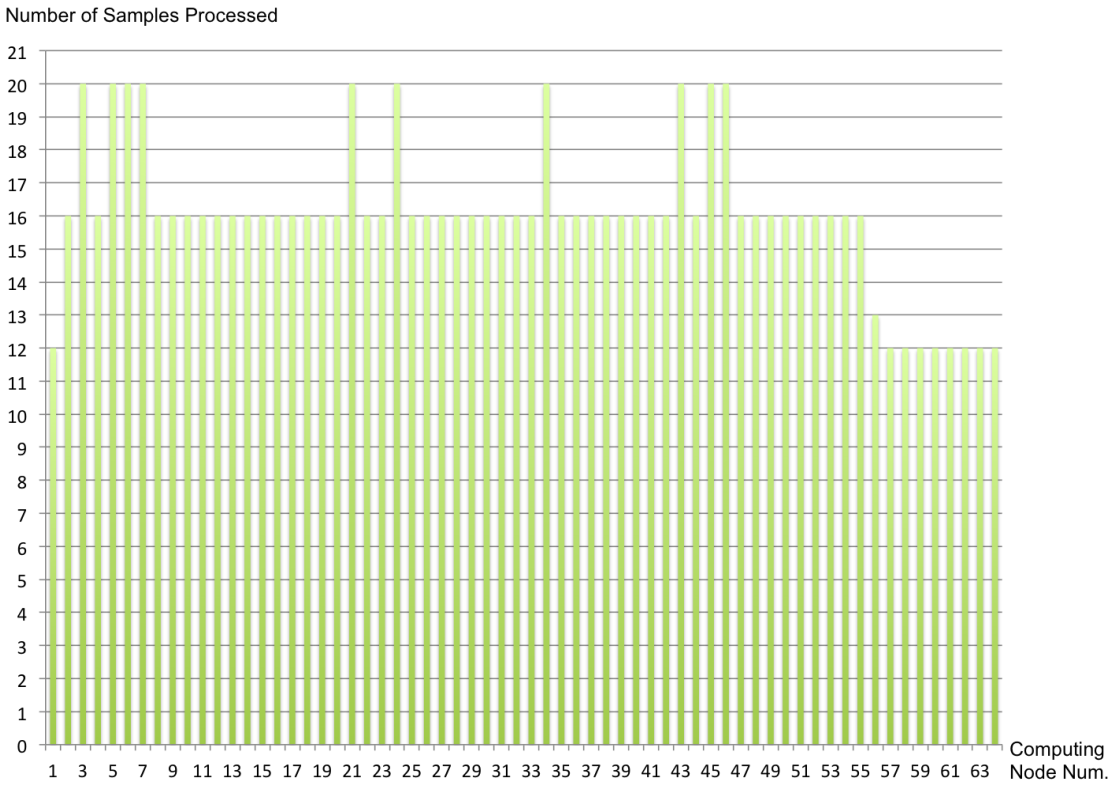

Figure 30: Distribution of samples in BG/Q nodes, 4 samples per chunk. Most nodes were scheduled with 4 chunks with a few scheduled for 3 or 5 chunks.

It should be noted that BG/Q is not dedicated for image processing and there is no library to handle tiff files in its operating system. Therefore, we had to build a tiff library from source code. The tiff specification suggests that every tiff library should be able to handled tiff files in both little Endian order and big Endian order. However, we found that on BG/Q, OpenCV integrated with the tiff library worked well with 8-bit 1-channel tiff images, but failed at decoding 8-bit 3-channel RGB tiff images. A 24-bit uncompressed RGB tiff image stores the bytes of each color channel in the order blue, green and red, when little Endian is used, and in the order red, green and blue when big Endian is used. In the tiff header, there is a 2-byte value indicating which Endian is used. After decoding the image data and representing the image as a matrix in OpenCV, the triple-byte structure to represent a pixel is always organized as blue, green and red. In our test of ADCI on BG/Q, we found that the triple-byte structures were organized as green, blue and a hexadecimal value 0xFF. Therefore when converting the RGB to a gray-scale

image, the gray-scale value for every pixel had a wrong value. The final centromere location for a given chromosome was therefore off-target. We suspect that this may be caused by an internal problem of the tiff library or compatibility issues with OpenCV. However the time and speed measurement of ADCI BG/Q in this test is still trustworthy since all modules and steps in ADCI were executed for all tested images. The error in decoding tiff files actually means the green channel and blue channel of an input image were switched and the red channel was always of maximal value. For an RBG metaphase, a wrong input image resulted from incorrect decoding but the boundary and intensity information are still able to differentiate chromosomes from background.

Chapter 5

5 Conclusion and Future Work

In this chapter, conclusion of the parallelization of our ADCI software and future work are discussed.

5.1 Conclusion

In endeavoring to accelerate ADCI, we parallelized it in two ways: task parallelism and data parallelism. In task parallelism, two basic functions costing the most time in ADCI were parallelized. Parallelized binary image labeling on an 8-core desktop can reduce the time of ranking to approximately 80 percent of its serial time requirement. The parallel GVF module with the help of parallel Odd-Even reduction on an 8-core desktop can be 3.4 times faster than the serial GVF module. We can also conclude from unit tests of parallel Odd-Even reduction that parallel Odd-Even reduction is able to control the increase in computation caused by the growing size of matrices. In data parallelism, parallelized ranking, GVF and centerline based modules were 4.8, 7.6 and 5-fold faster than their serial versions respectively. Data-parallelized ADCI desktop on an 8-core systems required ¼ of the time needed by serial ADCI desktop. These results supported our hypothesis: parallelization can effectively and efficiently accelerate ADCI.

Besides comparative speedup, we can also analyze the performance of parallel ADCI. The ranking module on a SMP supercomputer utilizing 40 cores was able to handle 18,649 samples in 4 hours, average 0.64 seconds per sample. The average time to rank a sample on the 64 core Goblin cluster nodes was 0.434 seconds, equivalent to ranking a thousand of samples in 80 minutes. From these experiments, we know that parallel ADCI is capable of processing thousands of samples through each of the 6 functional modules within a few hours. In experimentation on Blue Gene/Q, we concluded that ADCI cluster is capable of processing thousands of samples to identify DCs in 2 hours when the raw data are organized as metaphase grids. Although centromere detection in ADCI BG/Q returned incorrect locations due to a problem resulting from compatibility between the

OpenCV and Tiff library implementations on the PowerPC architecture, the assessment of execution time was still correct.

We give a summary and comparison of ADCI from different platforms in Table 7. Current ADCI includes image ranking and chromosome-based modules. Meanwhile we have chromosome separation and sister chromatid separation in development and expected ADCI may include these modules [54] [55]. The summarized accuracies on stages of ADCI, also shown in Table 7, give a comprehensive description for ADCI, whose details can be found in reference. In ranking and chromosome-based modules, we observed an approximate 25-times speedup when code is converted from MATLAB to serial C++. Converting MATLAB to parallel C++ with 8 available cores brought us 64-fold and 92-fold speedup in ranking module and chromosome-bases modules, respectively. Overall, C++ ADCI software is 3.7-times and 30-times faster when parallelized on an 8-core desktop and on BG/Q. The 8-core parallel ADCI desktop and ADCI BG/Q are able to process ~1000 samples in 12 and 1.5 hours respectively.

Table 7: Summary and comparison of ADCI of different versions

	Time to process one sample (sec)			**Accuracy, compared with cytogeneticists** [2] [3]
	MATLAB	C++/OpenCV		
		Serial	8-core Parallel	
Image ranking	266	10.7	4.12	~98%
Classifying to centromere detection	3540	145	38.2	96.6% Normal 85% DCs
Sister chromatid separation				93.1%

Complete ADCI	63.4 min.	2.6 min.	0.7 min.	
	Time to process 1000 samples			
	MATLAB	Serial	8-core Parallel	BG/Q, 32 nodes
Complete ADCI	44 days	1.8 days	11.7 hours	1.4 hours

We are encouraged to see that our parallel strategy can reduce the required time of processing cytogenetic biodosimetry data in large casualty events from several days to a few hours. This parallel strategy is able to achieve thorough image processing demanded by the short diagnostic and treatment windows to analyze a large number of individuals exposed to varying levels of ionizing radiation.

5.2 Future Work

In future, we will fix the error caused by Endian in BG/Q codes by reinstalling tiff library or build our own library. An assisting program to create metaphase grids from individual metaphases will be implemented. Alternative parallel schema, such as work-stealing in scheduling MPI workload and parallelization parameters, such as the size of chunk will be tested. Our work will focus on BG/Q as it provides superior computing. We also plan to explore implementations which use hybrid MPI and OpenMP to improve the efficiency of the overall process.

References or Bibliography

[1] W. F. Blakely et al., "Early-response biological dosimetry-recommended countermeasure enhancements for mass-casualty radiological incidents and terrorism," *Health Physics*, 89(5), pp. 494-504, 2005.

[2] A. Subasinghe et al., "An accurate image processing algorithm for detecting FISH probe locations relative to chromosome landmarks on DAPI stained metaphase chromosome images," in *IEEE Canadian Conference on Computer and Robot Vision*, pp. 223-230, DOI: 10.1109/CRV.2010.36. 2010.

[3] A. Subasinghe et al., "An image processing algorithm for accurate extraction of the centerline from human metaphase chromosomes," in *IEEE International Conference on Image Processing*, pp. 3613-3616, DOI: 10.1109/ICIP.2010.5652017. 2010.

[4] A. J. González, "An international perspective on radiological threats and the need for retrospective biological dosimetry of acute radiation overexposures," *Radiation Measurements*, 42(6-7), pp. 1053-1062. 2007.

[5] G. A. Alexander et al., "BiodosEPR-2006 meeting: Acute dosimetry consensus committee recommendations on biodosimetry applications in events involving uses of radiation by terrorists and radiation accidents," *Radiation Measurements*, 42(6), pp. 972-996, 2007.

[6] P. Prasanna et al., "Diagnostic Biodosimetry Response for Radiation Disasters: Current Research and Service Activities," in *NATO Medical Surveillance and Response, Research and Technology Opportunities and Options*, Budapest, Hungary, 2004.

[7] J. K. Timins and J. A. Lipoti. "Radiological terrorism," *N. J. Med.*, 100(6), pp. 14-21, quiz 22-4, 2003.

[8] P. G. Prasanna et al., "Triage dose assessment for partial-body exposure: Dicentric analysis," *Health Physics*, 98(2), pp. 244. 2010.

[9] T. D. Pollard et al., *Cell Biology*, 2nd ed. Philadelphia: Saunders, 2007.

[10] Y. Li et al., "Towards large scale automated interpretation of cytogenetic biodosimetry data," in *IEEE 6th Annual International conference on Automation for Sustainability*, pp. 30 - 35, DOI: 10.1109/ICIAFS.2012.6420039. 2012.

[11] T. Kobayashi et al., "Content and classification based ranking algorithm for metaphase chromosome images," in *IEEE Conference on Multimedia Imaging*, Taipei, 2004.

[12] Y. Panala et al., "Automated detection of metaphase chromosomes for FISH and routine cytogenetics," in *54th Annual ASHG Meeting*, Toronto, pp. 195, 2004.

[13] T. Cover and P. Hart, "Nearest neighbor pattern classification," *Information Theory, IEEE Transactions*, 13(1), pp. 21-27, 1967.

[14] N. B. Rizvandi et al.,
"Skeleton Analysis of Population Images for Detection of Isolated and Overlappe d Nematode C.elegans," in *16th European Signal Processing Conference (EUSIPCO 2008),* 2008.

[15] C. Xu and J. L. Prince, "Snakes, shapes, and gradient vector flow," *Image Processing, IEEE Transactions,* 7(3), pp. 359-369, 1998.

[16] J. Canny, "A computational approach to edge detection," *Pattern Analysis and Machine Intelligence, IEEE Transactions,* vol. 6, pp. 679-698. 1986.

[17] R. C. Gonzalez and R. E. Woods. *Digital Image Processing,* 3rd ed. Boston: Addison-Wesley, 2008.

[18] L. J. Latecki and R. Lakämper, "Polygon evolution by vertex deletion," in *Scale-Space Theories in Computer Vision,* 1999.

[19] X. Bai et al., "Skeleton pruning by contour partitioning with discrete curve evolution," *Pattern Analysis and Machine Intelligence, IEEE Transactions,* 29(3), pp. 449-462, 2007.

[20] A, Subasinghe, "Image Processing Techniques for Detecting Chromosomes Abnormalities," M.S. thesis, Dept. ECE, University of Western Ontario, London, Ontario, 2010.

[21] R. L. Burden and J. D. Faires, *Numerical Analysis,* 7th ed. Stamford: Brooks Cole, 2001.

[22] R. C. King et al., *A Dictionary of Genetics,* 7th ed. Oxford, UK: Oxford University Press, 2006.

[23] J. M. Rabaey, *Digital Integrated Circuits: A Design Perspective,* Upper Saddle River: Prentice Hall, 1996.

[24] J. L. Hennessy et al. *Computer Architecture: A Quantitative Approach,* 4th ed. Burlington, MA: Morgan Kaufmann, 2007.

[25] M. J. Flynn, "Some computer organizations and their effectiveness," *Computers, IEEE Transactions,* 100(9), pp. 948-960, 1972.

[26] R. Duncan, "A survey of parallel computer architectures," *Computer,* 23(2), pp. 5-16, 1990.

[27] D. E. Culler et al., *Parallel Computer Architecture: A Hardware/Software Approach,* Burlington, MA: Morgan Kaufmann, 1999.

[28] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," presented at *Spring Joint Computer Conference*, 1967.

[29] R. S. Pressman, *Software Engineering: A Practitioner's Approach,* 5th ed. New York: McGraw-Hill, 2001.

[30] A. J. Bernstein, "Analysis of programs for parallel processing," *Electronic Computers, IEEE Transactions,* vol. 5, pp. 757-763, 1966.

[31] C. Q. Yang and B. P. Miller, "Critical path analysis for the execution of parallel and distributed programs," in *8th International Conference on Distributed Computing Systems,* 1988.

[32] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," in *35th Annual Symposium on Foundations of Computer Science*, 1994.

[33] G. E. Blelloch and M. M. Bruce, "Parallel algorithms," in *Algorithms and Theory of Computation Handbook*, London, UK: Chapman and Hall/CRC, 2010.

[34] D. R. Butenhof, *Programming with POSIX Threads*, Boston: Addison-Wesley, 1997.

[35] R. D. Blumofe et al., "Cilk: An efficient multithreaded runtime system," *Journal of Parallel and Distributed Computing*, 37(1), pp. 55-69, 1996.

[36] J. Reinders, *Intel Threading Building Blocks: Outfitting C for Multi-Core Processor Parallelism*, Sebastopol: O'Reilly, 2010.

[37] L. Dagum and R. Menon, "OpenMP: An industry standard API for shared-memory programming," *IEEE Computational Science & Engineering*, 5(1), pp. 46-55, 1998.

[38] "MPI Forum," http://www.mpi-forum.org/docs/.

[39] E. Gabriel et al., "Open MPI: Goals, concept, and design of a next generation MPI implementation," *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, vol. 3241 2004.

[40] W. Gropp et al., "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Computing*, 22(6), pp. 789-828, 1996.

[41] W. Gropp et al., *Using MPI-: Portable Parallel Programming with the Message Passing Interface*, 2nd ed. Cambridge, MA: MIT Press, 1999.

[42] A. Silberschatz et al., *Operating System Concepts*, 7th ed. Hoboken, NJ: John Wiley and Sons, 2005.

[43] R. M. Haralick and L. G. Shapiro, *Computer and Robot Vision*, Boston: Addison-Wesley, 1992.

[44] R. Sedgewick, *Algorithms in C*, 3rd ed. Boston: Addison-Wesley, 1998.

[45] S. Arora and B. Barak, *Computational Complexity: A Modern Approach*, Cambridge, UK: Cambridge University Press, 2009.

[46] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceeding of the IEEE*, 93(2), pp. 216-231, 2005.

[47] L. H. Thomas, "Elliptic Problems in Linear Differential Equations over a Network," Watson Laboratory Report, Columbia University, 1949.

[48] H. S. Stone, "Parallel tridiagonal equation solvers," *ACM Transactions on Mathematical Software (TOMS)*, 1(4), pp. 289-307, 1975.

[49] J. Dinan et al., “Dynamic load balancing of unbalanced computations using message passing,” in *IEEE International Parallel and Distributed Processing Symposium*, 2007.

[50] C. Li et al., “Segmentation of edge preserving gradient vector flow: An approach toward automatically initializing and splitting of snakes,” in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2005.

[51] “Symmetric Computing (Boston),” http://www.symmetriccomputing.com.

[52] SHARCNET, https://www.sharcnet.ca/my/front/

[53] *IBM Blue Gene/Q Solution*, http://www.redbooks.ibm.com/abstracts/sg247948.html.

[54] A. Subasinghe et al., “Intensity integrated Laplacian algorithm for human metaphase chromosome centromere detection,” in *25th IEEE Canadian Conference on Electrical & Computer Engineering*, pp. 1-4, DOI: 10.1109/CCECE.2012.6334866, 2012.

[55] A. Subasinghe et al., “Intensity Integrated Laplacian Based Thickness Measurement for Detecting Human Metaphase Chromosome Centromere Location,” *IEEE Transactions on Biomedical Engineering*, DOI: 10.1109/TBME.2013.2248008.

[56] BWLabeln function in MATLAB, http://www.mathworks.com/help/images/ref/bwlabeln.html

Appendices

Appendix A: Acronym.

DAPI	4',6-diamidino-2-phenylindole
C-banding	Constitutive banding
GVF	Gradient Vector Flow
DCE	Discrete Curve Evolution
DCs	Dicentric Chromosomes
DCA	Dicentric Chromosome Assay
ADCI	Automated Dicentric Chromosome Identification
SISD	Single Instruction stream, Single Data stream
SIMD	Single Instruction stream, Multiple Data stream
MISD	Multiple Instruction stream, Single Data stream
MIMD	Multiple Instruction stream, Multiple Data stream
PThread	POSIX Threads
TBB	Intel Threading Building Blocks
MPI	Message Passing Interface
AIO	Asynchronous I/O
DAG	Directed Acyclic Graph
LAN	Local Area Network
BG/Q	Blue Gene/Q

Curriculum Vitae

Name: Yanxin Li

Post-secondary Education and Degrees:
Zhejiang University
Hangzhou, Zhejiang, People's Republic of China
2007-2011 B.Eng., School of Computer Science and Technology

The University of Western Ontario
London, Ontario, Canada
2011-2013 M.Sc. Candidate, Department of Computer Science

Honours and Awards:
Western Graduate Research Scholarship
2011-2012

Faculty of Science, The University of Western Ontario
Graduate Thesis Research Award
2012-2013

Related Work Experience
Teaching Assistant
The University of Western Ontario
2011-2012

Publications and Presentations:

Li, Yanxin; Wickramasinghe, Asanka; Subasinghe, Akila; Samarabandu, Jagath; Knoll, Joan; Wilkins, Ruth; Flegal, Farrah.; Rogan, Peter K. "Towards large scale automated interpretation of cytogenetic biodosimetry data." IEEE 6th Annual International conference on Automation for Sustainability, 2012. pp. 30 – 35; DOI: 10.1109/ICIAFS.2012.6420039.

Li, Yanxin; Wickramasinghe, Asanka; Subasinghe, Akila; Samarabandu, Jagath; Knoll, Joan; Rogan, Peter K. "Towards large scale automated interpretation of cytogenetic biodosimetry data" Presented at Great Lakes Bioinformatics Conference (GLBIO), Ann Arbor MI, 2012.

Li, Yanxin; Wickramasinghe, Asanka; Subasinghe, Akila; Samarabandu, Jagath; Knoll, Joan; Rogan, Peter K. "Towards large scale automated interpretation of cytogenetic biodosimetry data" Presented at SHARCNET Research Day, 2012.

Li, Yanxin; Wickramasinghe, Asanka; Subasinghe, Akila; Samarabandu, Jagath; Knoll, Joan; Rogan, Peter K. "Towards large scale automated dicentric analysis for cytogenetic" Presented at University of Western Ontario Research at Computer Science (UWORCS), 2012.