Enhancing Parallelism of Data-Intensive Bioinformatics Applications

Zheng Xie, Liangxiu Han
School of Computing, Mathematics and Digital Technology
Manchester Metropolitan University
Manchester M1 5GD, UK
Z.Xie@mmu.ac.uk, L.Han@mmu.ac.uk

Richard Baldock
MRC Human Genetics Unit MRC IGMM,
University of Edinburgh Western General Hospital
Edinburgh EH4 2XU, UK
richard.baldock@igmm.ed.ac.uk

***Abstract*— Bioinformatics data-resources collected from heterogeneous and distributed sources can contain hundreds of Terra-Bytes and the efficient exploration on these large amounts of data is a critical task to enable scientists to gain new biological insight. In this work, an MPI-based parallel architecture has been designed for enhancing performance of biomedical data intensive applications. The experiment results show the system has achieved super-linear speedup and high scalability.**

Keywords- gene pattern recognition, data intensive application, Message Passing Interface, collective/point-to-point communicators, task/data parallelisms, fine-grain parallelism, super-linear speedup, super-ideal scalability.

I. INTRODUCTION

With the exploitation of advanced high-throughput instrumentation, the quantity and variety of bioinformatics data have become overwhelming. Such data, collected from heterogeneous and distributed sources, typically consists of tens or hundreds of Terra Bytes (TB) comprising ten to fifty thousand individual assays with supplementary metadata and spatial-mapping information. This is the case with the mouse embryo gene-expression databases EMAGE [13], EurExpress [14] and the Allen Brain Atlas [15]. To enable scientists to gain new biological insights, massively parallel processing on these data is required. Parallel and distributed computing, along with parallel programming models (e.g., MPI [1]), provides solutions by splitting massive data-intensive tasks into smaller fragments and carrying out much smaller computations concurrently.

This work explores how parallel processing could accelerate data intensive biomedical applications. Message Passing Interface (MPI) has been chosen to implement our parallel structure. MPI is a standardized and portable message-passing application programmer interface [4], which is designed to function on a wide variety of parallel computers. Its library provides communication functionality among a set of processes. A rich range of functions in the library enables us to implement complicated communication which we need for enhancing the effectiveness of parallel computing as computational resources increase.

Our contribution of this work lies in 1) Design and implementation of enhanced parallel algorithms for a bioinformatics data intensive application based on MPI. 2) Quantitative analyses of super-linear speedup and high scalability obtained from the parallel system. 3) Consideration of maximizing the benefits of super-linear speedup and high scalability by maximizing the benefits of caching.

The rest of the paper is organized as follows: section II describes the background and parallel solution for the biomedical use case; Section III presents the experimental evaluation. In Section IV, we conclude our work.

II. PARALLEL PROCESSING FOR DATA INTENSIVE BIOMEDICAL APPLICATIONS

A. *Background of the bioinformatics application*

In this research, we aim to accelerate a task from the biomedical science. This particular task concerns ontological annotation of gene expression in the mouse Embryo. Ontological annotation of gene expression has been widely used to identify gene interactions and networks that are associated with developmental and physiological functions in the embryo. It entails labelling embryo images produced from RNA in situ Hybridization (ISH) with terms from the anatomy ontology for mouse development. If an image is tagged with a term, it means the corresponding anatomical component shows expression of that gene. The input is a set of image files and corresponding metadata. The output will be an identification of the anatomical components that exhibit gene expression patterns in each image. This is a typical pattern recognition task. As shown in Figure1 (a), we first need to identify the features of `humerus' in the embryo image and then annotate the image using ontology terms listed on the left ontology panel. To automatically annotate images, three stages are required: at the training stage, the classification model has to be built, based on training image datasets with annotations; at the testing stage, the performance of the classification

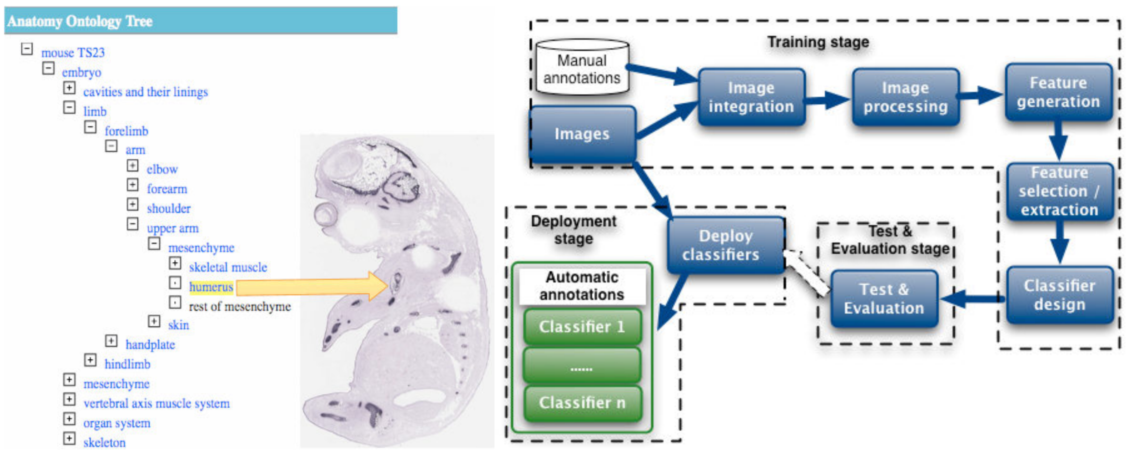


Figure 1. Automatic ontological gene annotation [3].

model has to be tested and evaluated; then at the deployment stage, the model has to be deployed to perform the classification of all non-annotated images. We mainly focus on the training stage in this case. The processes in the training stage include integration of images and annotations, image processing, feature generation, feature selection and extraction, and classifier design, as shown in Figure1 (b). Currently gene expression annotation is mainly done manually by domain experts. This is both time-consuming and costly, especially with the rapidly growing volume of image data of gene expression patterns on tissue sections produced by advanced high-throughput instruments (e.g. ISH). For example, the EuExpress-II project [2] delivered partially annotated and curated datasets of over 20 Terabytes including images for the developing mouse embryo and the ontological terms for anatomic components of the mouse embryo, which provides a powerful resource for discovery of genetic control and functional pathways of processes controlling embryo organisation. To alleviate the issues with the manual annotation, we have developed data mining algorithms for automatically identifying an anatomical component in the embryo image and annotating the image using the provided ontological terms [3], programmed these in sequential code and executed the task on a single commodity machine. Note that this task is a specific instance of a generic image-processing analysis pipeline that could be applied to many datasets. To process the everincreasing growth in the volume of stored data, it is necessary to design parallel solutions for speedup the data intensive applications.

B. *Overview of parallel approach*

It is well known that the speedup of an application to solve large computational problems is mainly gained by the parallelisation at either hardware or software levels or both (e.g., signal, circuit, component and system levels).

In general, three considerations when parallelising an application at software level include:

- How to distribute workloads or decompose an algorithm into parts as tasks?
- How to map the tasks onto various computing nodes and execute the subtasks in parallel?
- How to coordinate and communicate subtasks on those computing nodes.

There are mainly two common methods for dealing with the first two questions: data parallelism and task parallelism. Data parallelism represents workloads are distributed into different computing nodes and the same task can be executed on different subsets of the data simultaneously. Task parallelism means the tasks are independent and can be executed purely in parallel. There is another special kind of the task parallelism is called 'pipelining'. A task is processed at different stages of a pipeline, which is especially suitable for the case when the same task is used repeatedly. The extent of parallelisation is determined by dependencies of each individual part of the algorithms and tasks.

As for the coordination and communication among tasks or processes on various nodes or computing cores, it depends on different memory architectures (shared memory or distributed memory). A number of communication models have been developed [7][8]. Among them, the Message Passing Interface (MPI) has been developed for HPC parallel applications with distributed memory architectures and has become the de-facto standard. There is a set of implementations of MPI, for example, OpenMPI [9], MPICH [10], GridMPI [11] and LAM/MPI [12]. Two types of MPI communication functionality are point-to-point and collective communication, and there are a number of functions involved in them. Point-to-point functions deal with communication between two specific processes, in which a message needs to be sent from a specified process to another specified process. They are suitable for patterned or irregular communication. Collective functions manage communication among all processes in a process group at same time. The process group could either be the entire process pool or a program-defined process subset. Collective communication is more efficient than point-to-point communication, and ought to be used in the parallel architecture wherever it is suitable.

C. *Parallel processing using MPI for the bioinformatics application*

This section presents how we have designed and developed a parallel approach based on MPI for efficiently processing large-scale data.

In terms of the nature of algorithms used in the biomedical application use case, we have mainly employed data and task parallelisms. The communication model used here is MPI, which support both point-to-point and collective communications.

For data parallelism, we need to distribute workloads into different computing cores and execute same computation on different subsets of the data simultaneously. This functionality could be implemented by using one of MPI collective functions, 'MPI_Scatter', which takes the workload as an array and split it into a number of segments. The number of segments is program defined number of parallel processes, with which MPI programs always work. Then, at runtime, all the processes are assigned to computing cores in the order of process rank through the agent that starts the MPI program. When it needs to swap regions of data among specific processors between calculation steps, point-to-point communication functions are used, such as 'MPI_Send/Recv'. For task parallelism, especially for 'pipelining', point-to-point communication functions are used to pass new task data from a set of cores to another set of cores whenever the prior tasks are completed.

Our workflow for the application is divided into three parts according to the characteristics of the task and data processing,

1) Image Processing and Feature Generation,
2) Feature Selection and Extraction,
3) Classification.

The overview of parallel architecture is illustrated in Figure 2.

In the first part of the workflow, the image processing of each image works independently on its own dataset, and data parallelism is the suitable form for parallel processing. The efficient collective communication of MPI is able to implement the data parallel and enhance the proceeding speed with minimum communication time. It is implemented with MPI collective communicator 'MPI_Scatter' and 'MPI_Gather', and the minimum processing unit is a single image object. 'MPI_Scatter', as shown in Figure3, takes an array of image objects and distributes the objects in the order of process rank. 'MPI_Gather' is the reverse function of 'scatter'. After image processing, all the feature data vectors representing the processed images are put together in the order of the process rank to produce a new data array, which is the input for the second part of the workflow.

In the second part of the workflow, fisher-ratio algorithm is used for feature selection and extraction [3]. Taking into account of large computing resources and regular calculation of mean and standard deviation, fine-grain parallelism is implemented. The implementation is similar to the part 3, and the detail is discussed in the next paragraph. In the context of this research, 'fine-grain' means that a task is split into very small fragment to make efficient use of the computing resources, though the increasing time of communication among processors is a tradeoff.

In part 3 of the workflow, a Linear Discriminant Analysis (LDA) algorithm [16] is used to implement a classifier, considering the optimization on classification [3]. In the LDA implementation, both regular and irregular computation are involved, such as matrix multiplication, finding global and sub-data sets mean and covariance matrix, and the calculation of a discriminant function. Taking into account the complexity of the computation, fine-grain and task parallelisms are implemented by jointly using point-to-point and collective communicators. Various kinds of computations are defined as individual tasks. Sub-data sets for different computations are distributed in the way of 'MPI_Scatter'. Data transfer among tasks is managed by MPI's point-to-point communicators 'MPI_Send/Recv'. This is a task parallel implementation. Within each individual computation, fine-grain parallelism is designed, which split each vector of an array into sub-vector by using 'MPI_Scatter'. Figure 4 illustrates the fine-grained array multiplication calculation with increasing number of processes. The parallel implementation is sensitive to the number of process. In the case of a small number of process or computing cores which is less than the number of vectors in an array, it is a coarse-grained parallel implementation. It is a fine parallel implementation when the number of computing cores is greater than this number of vectors. The advantage of this structure is that it makes efficient use of the computing resources; so that no computing cores have significantly different workloads from others as the workflow progresses.

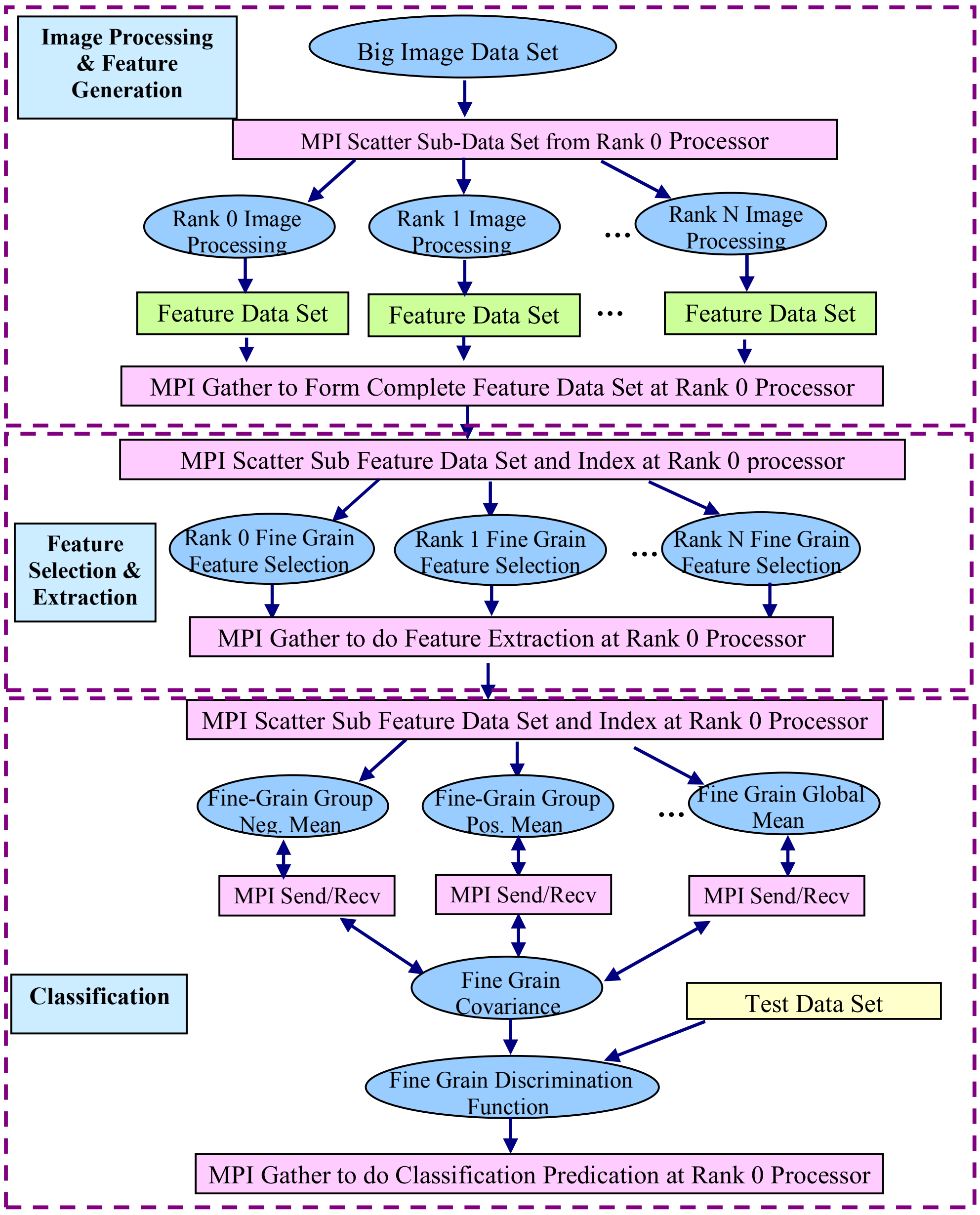


Image Processing & Feature Generation
Big Image Data Set
MPI Scatter Sub-Data Set from Rank 0 Processor
Rank 0 Image Processing
Rank 1 Image Processing
…
Rank N Image Processing
Feature Data Set
Feature Data Set
…
Feature Data Set
MPI Gather to Form Complete Feature Data Set at Rank 0 Processor
MPI Scatter Sub Feature Data Set and Index at Rank 0 processor
Feature Selection & Extraction
Rank 0 Fine Grain Feature Selection
Rank 1 Fine Grain Feature Selection
…
Rank N Fine Grain Feature Selection
MPI Gather to do Feature Extraction at Rank 0 Processor
MPI Scatter Sub Feature Data Set and Index at Rank 0 Processor
Fine-Grain Group Neg. Mean
Fine-Grain Group Pos. Mean
…
Fine Grain Global Mean
MPI Send/Recv
MPI Send/Recv
MPI Send/Recv
Classification
Fine Grain Covariance
Test Data Set
Fine Grain Discrimination Function
MPI Gather to do Classification Predication at Rank 0 Processor

Figure 2. Parallel architecture for the workflow.

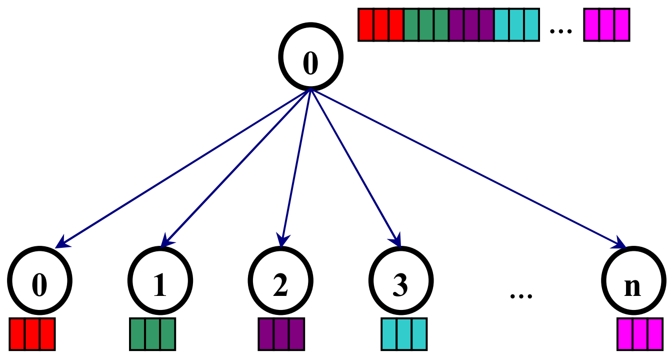
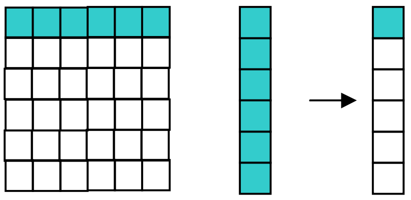
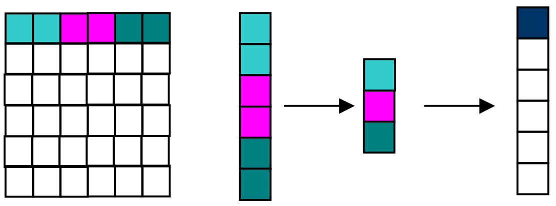


Figure 3. ‘MPI_Scatter’ operation with n processors and a data series.



(a) process number less than the number of rows.



(b) process number greater than number of rows.

Figure 4. Array multiplication under process number dependent fine-grain algorithm.

Fine-grain parallel algorithm corresponding to the number of process number is as below, called process number dependent fine-grain algorithm.

```
--------------------------------------------------------------------
Algorithm. Process number dependent fine-grain algorithm
--------------------------------------------------------------------
Input Process number $n_P$
Input Number of array columns $n_{column}$
Input Number of array rows $n_{row}$
If $n_{row} >= n_P$
    MPI_Scatter buffer length = $(n_{row} / n_P) * n_{column}$
    Calculation
    MPI_Gather buffer length = $n_{row} / n_P$
End
While $n_{row} < n_P$
    MPI_Scatter buffer length = $n_{column} / n_P$
    Calculation
    MPI_Gather 1 buffer length = $n_{column} / n_P$
    MPI_Gather 2 buffer length = $n_{column}$
End
--------------------------------------------------------------------
```

III. Experimental Evaluation And Discussion

A. *Experimental evaluation*

1) *Experiment setup*

A Linux cluster called Feyman, located at the Manchester Metropolitan University, is used for performance evaluation. It has a front end/storage node and 8 compute nodes, which are each 2× Intel E5430 quad-core processor running at 2.6GHz with 8GB of RAM (1 GB per core). The front end and compute nodes are all linked using gigabit Ethernet. The program runs in standalone form on the cluster, and it is compiled with FastMPJ [17] and packed into executable .jar file.

We have performed experiments by varying numbers of processors and size of dataset:

- Number of processors: 4 cores, 8 cores, 16 cores, 32 cores, 64 cores.
- Data size (number of images): 1X, 2X, 4X, 8X,16X, 32X, 64X, where X=128 images.

2) *Performance evaluation metrics*

We have measured performance of the parallel solution using two metrics: speedup and scalibility.

a) *Speedup*

Amdahl's law [18] describes an ideal linear speedup for a parallel system with the following equation:

$$\frac{1}{\alpha + \frac{1-\alpha}{P}} \tag{1}$$

In this expression, α is the fraction of a calculation that is sequential, (1- α) is the fraction that can be parallelised, and P is the number of processors. The maximum linear speedup that can be achieved is *P*. In our experiments, the baseline number of processors was 4; therefore the maximum speedup was 2 for 8 processors, 4 for 16 processors and so on. However, as can be seen in Figure 5, the speedup obtained is often greater than the maximum linear speedup. This is super-linear speedup, which can be due to the cache effect resulting from the different memory hierarchies of modern computers. In parallel computing, not only do the numbers of processors change, but so does the sizes of accumulated caches from different processors. With the larger accumulated cache size, more or even all of the working set can fit into caches and the memory access time can reduce dramatically. This causes the extra speedup [1], which we strive to achieve in parallel computing. To maximize the benefits of super-linear speedup in our application, the images may be clustered according readily discernable aspects of their similarity; for example their sizes or known features. Similar images are sent to the same processor to maximize the benefits of caching.

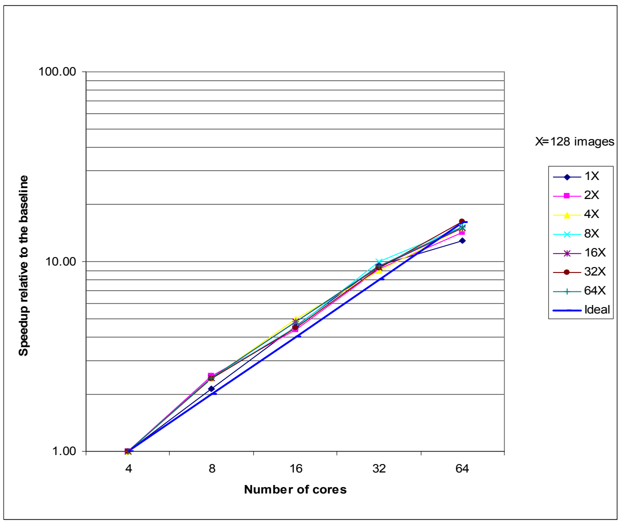


Figure 5. Speedup performance analysis

b) Scalability

The scalability concerns the relationship between the computation time and the increasing number of processors and dataset size. This can be expressed in terms of the relationship between the factor, ΔT, by which the computation time increases when the number of parallel processors increases by a factor N_P and the dataset size increases by a factor N_D. Factors less than 1 represent a decrease. ΔT may be expressed as

$$\Delta T = K(N_D, N_P)\frac{N_D}{N_P} \qquad (2)$$

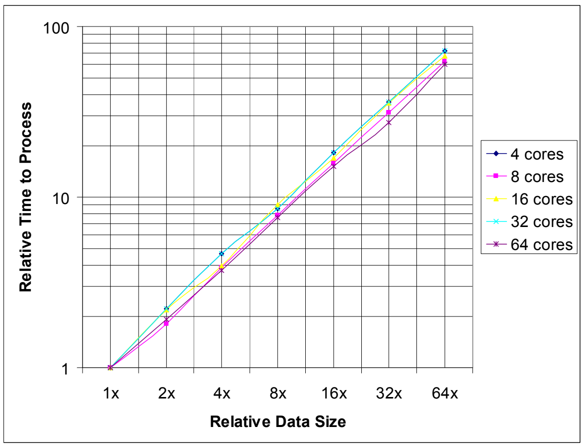
The function K may be dependent on many factors including N_D and N_P. Over ranges of N_D and N_P for which the function K remains constant, the speedup is said to be linear. Ideal linear speedup occurs where K=1 which means that if the factors N_P and N_D are equal, ΔT =1. This means that the computation time does not change when the number of parallel processors and the dataset size increase by the same ratio. The speedup becomes 'super-linear' (or 'super ideal') if K becomes less than 1 over any range of factors N_D and N_P, which means that a reduction in computation time (ΔT <1) will occur when the number of processors and the dataset size increase by the same ratio.

We can thus derive the following three equations based on Eq.(2). For fixed number of computing cores with increasing dataset size, the scalability can be represented in Eq.(3). For both the size of dataset and number of computing cores increase, the scalability can be represented in Eq.(4)

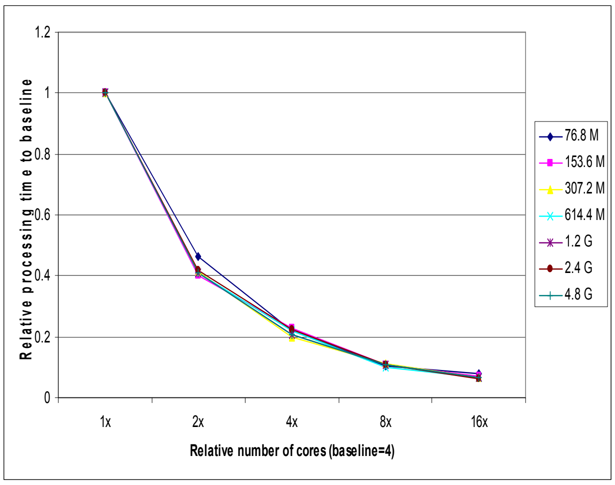
$$\frac{\partial \Delta T}{\partial N_D} = \left(\frac{N_D}{N_P}\right)\frac{\partial K(N_D, N_P)}{\partial N_D} + \left(\frac{1}{N_P}\right)K(N_D, N_P) \qquad (3)$$

$$\frac{\partial \Delta T}{\partial N_P} = N_D\frac{\partial K(N_D, N_P)}{\partial N_P} - \Delta T \qquad (4)$$

In Figure 6 (a), it may be seen that the increasing factor of computation time is almost equal to the increasing factor of dataset size for all numbers of cores, and are close to the ideal linear relationship. Figure 6 (b) shows that the decreasing factor of computation time is almost equal to the increasing factor of number of computing cores. All these results demonstrate good scalability of the parallel processing.



(a)



(b)

Figure 6. Scalability analysis: (a) time increasing factor vs dataset size increasing factor; (b) time decreasing factor vs increasing factor of computing core number.

Figure 7 shows the value of $K(N_D, N_P)$ in Eq.(2) plotted over the ranges of N_D and N_P used in the experiments. In the figure, label 'log2NP' and 'log2ND' represent increasing factor of computing cores and dataset size respectively. For example, that 'log2NP' is 3 means the core number increase by a factor of 8. It may be seen in Figure 7 that the function K becomes less than one over a significant range of values of (N_D, N_P) meaning that the scalability becomes super-ideal.

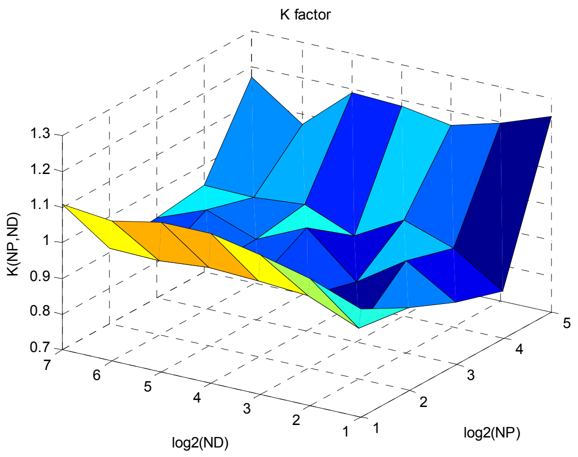


Figure 7. Scalability K(ND, NP) for factors N_D and N_P .

The performance of this parallel structure has been tested on a large size of dataset, 78.6 G. It shows super-linear speedup of 2.66, 5.24, 9.87 when the number of processors increased with factor of 2, 4, and 8 respectively.

IV. CONCLUSIONS

The process number dependent fine-grained algorithm enhances the parallel calculation needed in Linear Discriminant Analysis (LDA) classification. The advantage of this parallel structure is that it makes efficient use of the computing resources; so that no computing cores have significantly different workloads from others as the workflow progresses. The parallel architecture designed with technically using MPI communicators has enhanced the parallel processing performance in both aspects of speedup and scalability, and super-linear speedup and super-ideal scalability have occurred under this parallel structure. The method of quantitative analysis of scalability may be used for complicated high performance system. To maximize the benefits of super-linear speedup applications, the objects may be clustered according readily discernable aspects of their similarity. Similar objects are sent to the same processor to maximize the benefits of caching.

ACKNOWLEDGMENT

This research was sponsored by the Biotechnology and Biological Science Research Council (BBSRC). The authors acknowledge the financial support from BBSRC and the collaboration among the people of the AGILE project.

REFERENCES

[1] J. Benzi, M. Damodaran, "Parallel Three Dimensional Direct Simulation Monte Carlo for Simulating Micro Flows," Implementations and Experiences on Large Scale and Grid Computing, Parallel Computational Fluid Dynamics, p. 95, Springer2007, Retrieved 2013-03-21.

[2] Eurexpress website, http://www.eurexpress.org/ee/ , accessed May, 2013.

[3] L. Han, J. van Hemert, and R. Baldock, "Automatically identifying and annotating mouse embryo gene expression patterns," Bioinformatics, vol. 27(8), pp. 1101–1107, 2011.

[4] A.Yukiya, Nakano, "*Practical MPI Programming*", http:/ /www. redbooks. ibm. com/ abstracts/ sg245380. html), ITSO, Jun 1999.

[5] EURExpress-II project, http://www.eurexpress.org/ee/, Retrieved 10, May, 2010.

[6] L. Silva, and R. Buyya, "High Performance Cluster Computing: Programming and Applications," ch. Parallel Programming Models and Paradigms, pp. 4–27. No. ISBN 0-13-013785-5. Prentice Hall, PTR, NJ, USA, 1999

[7] P. S. Pacheco Parallel Programming with MPI. Morgan Kaufmann Publishers, Inc., 1997.

[8] PVM, 2009, http://www.csm.ornl.gov/pvm/, Retrieved 5 May, 2010.

[9] OpenMPI, 2009, http://www.open-mpi.org/, Retrieved 5 May, 2010.

[10] MPICH, http://www.mcs.anl.gov/research/projects/mpi/mpich1/, Retrived 5 May, 2010.

[11] GridMPI, http://www.gridmpi.org/index.jsp, Retrieved 5 May, 2010.

[12] LAMMPI, http://www.lam-mpi.org/, Retrieved 5 May, 2010.

[13] Jeffrey H. Christiansen, Yiya Yang, Shanmugasundaram Venkataraman, Lorna Richardson, Peter Stevenson, Nicholas Burton, Richard A. Baldock and Duncan R. Davidson. EMAGE: a spatial database of gene expression patterns during mouse embryo development. Nucl. Acids Res. 34 (2006): D637

[14] Diez-Roux et al, "A High-Resolution Anatomical Atlas of the Transcriptome in the Mouse Embryo," PLoS Biol 9(1) 2011: e1000582.

[15] E. S. Lein, M. J. Hawrylycz, et al, "Genome-wide atlas of gene expression in the adult mouse brain.," *Nature*, vol. 445, no. 7124, pp. 168–176, Jan. 2007.

[16] G. Perriere, J. Thioulouse, "Use of Correspondence Discriminant Analysis to predict the subcellular location of bacterial proteins", Computer Methods and Programs in Biomedicine, vol.70, pp.99-105, 2003.

[17] "High Performance Java Message Passing Library", http://www.fastmpj.com/, accessed January 2013.

[18] Amdahl, Gene, "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities," AFIPS Conference Proceedings (30), pp. 483–485, 1967.