Contents lists available at ScienceDirect

Chemometrics and Intelligent Laboratory Systems

journal homepage: www.elsevier.com/locate/chemolab

Short communication

Multi-core computing: A novel accelerating method for chemometrics calculation

Zhi-Min Zhang [a], Yi-Zeng Liang [a,*], Qing-Song Xu [b]

[a] *College of Chemistry and Chemical Engineering, Research Center of Modernization of Chinese Medicines, Central South University, Changsha 410083, PR China*
[b] *School of Mathematical Science and Computing Technology, Central South University, Changsha, 410083, PR China*

ARTICLE INFO

Article history:
Received 17 September 2008
Received in revised form 30 October 2008
Accepted 31 October 2008
Available online 11 November 2008

Keywords:
Multi-core computing
Cross-validation
Parallel algorithm
Message Passing Interface

ABSTRACT

The higher speed is the eternal pursuit of any chemometric algorithm. In order to take full advantage of the multi-core processor's computing resources (the prevailing current of personal computer) and accelerate the time-consuming algorithms in chemometrics, a novel multi-core computing method is introduced. Leave-one-out cross-validation is taken as an example to show the powerful capability of the multi-core computing. The comparison results show that the execution time drops rapidly with the increasing number of computing cores, which demonstrates that the multi-core computing is a promising tool for solving computing-intensive and data-intensive problems in chemometrics.

1. Introduction

With the development of analytical instruments, analysts can acquire more complex signals which have thousands of variables. For example, there are more than 1500 variables in the Fourier transform near infrared (FT-NIR) spectra. However, while applying the Principal Components Regression (PCR) [1] or Partial Least Squares regression (PLS) [2] to such large datasets, the speed is low, especially when the cross-validation procedure is used, which can't finish computing within acceptable time; hence there was a need to modify the classical algorithms to accelerate the cross-validation. Firstly, a fast and memory-saving PLS regression algorithm called PLS kernel algorithm was developed by Lindgren et al. [3,4]. Then Wu et al. [5] had also modified four classical PCA algorithms to their kernel versions to accelerate the speed of the PCA and cross-validation. More recently, Barrosa and Rutledge [6,7] proposed the Principal components transform-partial least squares method, which can dramatically accelerate the cross-validation of the calibration models.

All the above modifications were just to accelerate the algorithms' performance, which is the eternal pursuit of any chemometric algorithm. But nowadays the general trend in personal computer processor development is the leap from single-core to multi-core with the rapid improvement on engineering and manufacturing [8]. When single threaded leave-one-out cross-validation program is executed on a quad-core processor, only one core is busy and the others are idle. It is a waste of computing resources. Not only leave-one-out cross-validation but also the other time-consuming algorithms in chemometrics, which can be parallelized, should be redesigned and parallelized.

In this work, aside from using the faster version PCA or PLS mentioned in the previous paragraphs, the multi-core computing method is introduced to accelerate the cross-validation algorithm. The parallel method can accelerate leave-one-out cross-validation method to 3.8 times on an Intel® Core™2 Quad Q6600 processor (4 cores).

The paper is organized as follows. In Section 2, the parallel leave-one-out cross-validation method is introduced and its advantage is presented. In Sections 3 and 4, near infrared datasets are used for exploring this parallel method.

2. Parallel leave-one-out cross-validation

In this study, Message Passing Interface (MPI) [8], MATLAB® C Math Library Version 2.1 [9], Fedora 7 Linux operating system and C++ compiler of GNU Compiler Collections version 4.2 were used for implementing the parallel leave-one-out cross-validation. This work was done on a Dell Inspiron 530 PC with an Intel® Core™2 Quad Q6600 processor and 2048 M memory.

2.1. MPI and MATLAB® C Math Library Basics

The kernel techniques are MPI and MATLAB® C Math Library.

MPI is a library of standard subroutines for sending and receiving messages and performing collective operations. During an MPI based parallel computation, a fixed number of processes are created by the mpirun command (a command of MPICH which can start a parallel computation) at the initialization, and commonly, one process is created per core. These processes execute on different cores simultaneously and

* Corresponding author. Tel.: +86 731 8830824.
E-mail address: yizeng_liang@263.net (Y.-Z. Liang).

0169-7439/$ – see front matter © 2008 Published by Elsevier B.V.
doi:10.1016/j.chemolab.2008.10.010

Table 1
The used subroutines of the MPI library in parallel leave-one-out cross-validation

Subroutine	Function
MPI_Init	Initialize the MPI computation
MPI_Finalize	Shut down a computation
MPI_Comm_size	Determine number of processes
MPI_Comm_rank	Determine my rank
MPI_Send	Send a message
MPI_Recv	Receive a message
MPI_Bcast	Broadcast a message

communicate by calling MPI library subroutines to send and receive messages between or among processes [11]. The number of simultaneous processes (n_p) is fixed at program initialization in an MPI computation, which can be determined by calling the subroutine MPI_Comm_size, and the identifier of the process can be determined by calling the subroutine MPI_Comm_rank (the identifier of the process (i) in an MPI computation is a unique, contiguous integer which was numbered from 0). Although MPI is a complex library which comprises 129 subroutines, in this study, only seven subroutines are used. The used subroutines are listed as follows (see Table 1).

The MATLAB® C Math Library is a collection of mathematical routines written in C, which makes the mathematical core of MATLAB available to application programmers [9]. All the matrix computations, such as transpose, multiplication, inverse and singular value decomposition, are done by calling the MATLAB C Math Library.

2.2. Broadcast data to processes

In the parallel leave-one-out cross-validation, the data (**X** matrix and ***y*** vector) of every process are the same, and the source code of every process is also the same, but during the execution the identifiers are different, so the process's start index s_i and end index e_i of samples are different which are determined by the identifier of the process. There is only one copy of the source code; the source code is compiled to a program. When the program is started to run by the mpirun command, a group of processes are created and running on different cores and each process with a unique identifier. So one can use the unique identifier to partition task and assign different tasks to different processes in the source code. (Please see the source code in the Supplementary information.)

The root process (the identifier is 0) reads the training set of spectra (**X**) and the standard concentration vector (***y***), then it calls the MPI_Bcast subroutine, which distributes the data to all the other processes. The operation is illustrated in Fig. 1, each row means a process and their memory layouts are denoted by the grid of each row, the root process is denoted by the first row of the matrix. Before the dataset is broadcasted, the memory layouts of these processes are denoted by the left matrix. After the dataset is broadcasted, the memory layouts of these processes are denoted by the right matrix.

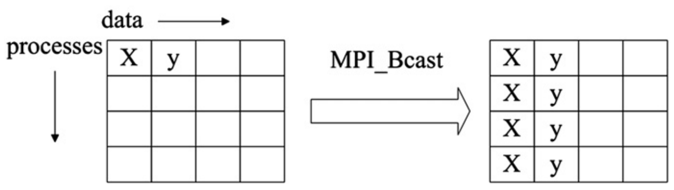

Fig. 1. Broadcast the training set of spectra and the standard concentration matrix to cores using the MPI_Bcast subroutine, each row represents data locations in a different process. At the beginning, the training set of spectra **X** and the standard concentration vector ***y*** are located only in process 0, after the calling of MPI_Bcast, they are replicated in all processes.

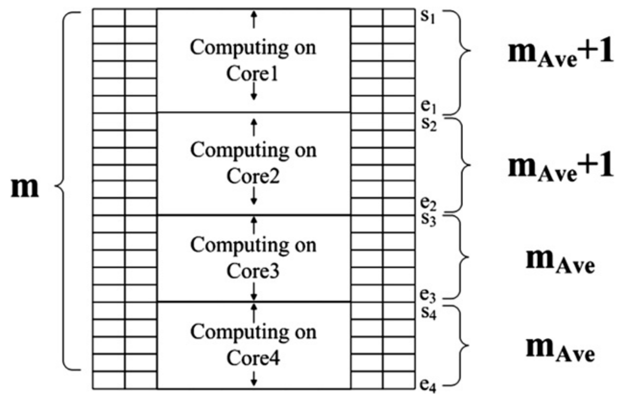

Fig. 2. Partition the training set of spectra according to the number of the processor's cores.

2.3. Partition computing task

Let us assume the training set of spectra is represented by a matrix **X** of order $m \times n$, the number of processes is n_p and the identifier of the calling process is i ($0 \le i < n_p$). The partition of leave-one-out cross-validation computing task can be illustrated in Fig. 2 and described as follows:

1. $m_{\mathrm{Ave}} = m/n_p$, $m_{\mathrm{Left}} = m \cdot \mathrm{mod}(n_p)$.
   In step 1, m_{Ave} is the result of the integer division m/n_p. m_{Left} is arithmetical remainder to m modulo n_p.
2. Assign $m_{\mathrm{Ave}}+1$ samples to each of the m_{Left} first processes and m_{Ave} samples to the remaining processes to deal with (see also Fig. 2), which can be mathematically described as follows:

$$\begin{cases} T_i = m_{\mathrm{Ave}} + 1 & i < m_{\mathrm{Left}} \\ T_i = m_{\mathrm{Ave}} & i \ge m_{\mathrm{Left}} \end{cases}, \quad s_i = \sum_{k=0}^{i-1} T_k + 1 \text{ and } e_i = \sum_{k=0}^{i} T_k.$$

In step 2, T_i is the number samples assigned to process i, s_i and e_i are the start and end indices of samples of the process, whose identifier is i, respectively.

2.4. Computing

The start index s_i and end index e_i of leave-out samples treated by process i can be obtained according to Section 2.3, also the training set of spectra **X** and the standard concentration vector ***y*** have been distributed to all processes. The leave-one-out cross-validation subroutine is called with the parameters s_i and e_i in process i to finish the task which has been assigned to process i.

2.5. Results collection

After computing, the prediction residual error (denoted by $\mathbf{P_i}$ in Fig. 3) should be gathered from processes. The memory layout of FORTRAN's multi-dimension array is different than that of C language.

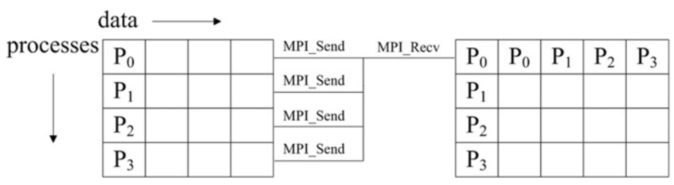

Fig. 3. Gather the prediction residual error array of each process to the root process for concatenating. **P** is the prediction residual error array that is sent by calling the MPI_Send subroutine in each process, and then it is received by calling the MPI_Recv subroutine in the root process.