Available online at www.sciencedirect.com
SCIENCE DIRECT®
Journal of Computational and Applied Mathematics 185 (2006) 91–106
JOURNAL OF COMPUTATIONAL AND APPLIED MATHEMATICS
www.elsevier.com/locate/cam

An optimal schedule for Gaussian elimination on an MIMD architecture☆

Rachid Saad*

40#114 Charles Albanel Street, Gatineau (Québec), Canada J8Z 1R2

Received 15 September 2003; received in revised form 26 January 2005

Abstract

In this paper, we propose an algorithm for allocating the tasks of the well known Gaussian Elimination Algorithm on an MIMD architecture and prove that the schedule is optimal in order of magnitude, up to a polylog factor.

Keywords: MIMD; Scheduling; Gaussian elimination

1. Introduction

1.1. Basic concepts

The advent of parallel computers in the 80s and the growing expansion they assumed ever since have brought to the fore the important issue of parallel performance evaluation. Theoretical models of computation such as the well-known PRAM model do not take into account the communication time in a computer-specific manner. The reason is that the PRAM model is essentially aimed at providing a measure of parallelism based on theoretical assumptions common to most computation models. What we observe then is that the performances in practice of the algorithms oftentimes do not match the theoretical results shown within the PRAM model. Discrepancies of this kind can only be accounted

☆ This work was partially supported by a grant from the Ministry of Higher Education of Algeria as part of a research project initiated at the 'Laboratoire de recherche en informatique fondamentale et appliquée', University of Boumerdès, Algeria.
* Tel.: +819 777 5807.
E-mail address: rachid_saad2003@yahoo.com.

0377-0427/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.cam.2005.01.028

for by a detailed examination of how communication actually takes place within the architecture. A model of parallel computation has therefore to take into account the communication time as it relates to a specific type of architecture, if it is to be of any practical use. It is fair to mention in the defense of the basic PRAM model, that it has been enhanced to emulate message-routing in an interconnection network and reflect a more realistic parallel computation [10]. The emulation comes obviously at a cost which depends on the data routing through the interconnection network. Other types of parallel architecture include: the Data flow architecture and the MIMD architecture. The basic idea underlying Data Flow architecture is to allow computation to proceed in parallel regardless of any artificially induced sequencing of instructions, as built-in in most conventional computer programs. Only the logical order of the instructions is relevant. This entails for example, that an operation is allowed to execute whenever its operands are available, independently of the ordinal count of the operation in the program. The expectation is to enable the architecture to exploit at best the type of parallelism inherent with the computation itself. For all practical purposes however, this type of architecture is no different from the other ones: we run into the same technical difficulties when it comes to accessing input data or communicating output. A detailed comparison of the different parallel architectures is beyond the scope of this paper. The interested reader is referred to [8,3] for a comprehensive survey of the different models. In this paper, we will be concerned with the communication delays generated in a shared memory multiple instructions multiple data architecture (MIMD), where each processor has a local memory and communicates with the shared memory either by loading variables or transferring outputs. In this model a program may be viewed as a set T of tasks (representing instructions or blocks of instructions) related by a precedence relation $\lll$ described by a directed acyclic graph (DAG) G. A model of parallel computation which takes into account the communication delays in the context of MIMD architectures was introduced by Anderson et al. [1]. We will refer to it as the ABR model. The following is a short description of the basic concepts underlying the ABR model. All the assumptions and the attending explanations are from [4,3].

(1) Job (or cluster): It is a subset of tasks of T. If a job J is assigned to a processor b, then b loads the input to J (treated as a subprogram of T), performs some computation on J using its local memory, and transfers its outputs at the end of the execution. However, no transfer is allowed during the execution of a job. A common assumption is that the time c needed for a processor to load variables or transfer output is constant, whatever the transfer size. More precisely, $c = 1$. It is stated explicitly in article [4, p. 56 l.27] in these crude terms: “Let us assume that each job needs one unit of time to communicate with the memory”. This assumption is valid if the number of processors is small; it remains valid for a large number of processors if the loading of variables and the transfer of output is pipelined, as explicitly pointed out in [3] in these terms: “... If we assume that each memory access takes a constant time whatever is the volume of transferred data, then the communication cost is proportional to the number of tasks. This assumption is realistic in the case where the loading of variables and the output of results is pipelined” [3, p. 7 l.27]. Thus, emphasis is placed on the number of transfers involved in a parallel computation regardless of how much is transferred. In fact, the rationale of this model is to capture the computational complexity assuming an unbounded number of processors, where the communication cost is the same whatever the size of the transferred data. For more on this model issue, we refer the interested reader to the above-mentioned survey by Bampis et al. [3]. As a result, we may suppose, exactly as in [4,3], that the communication time is equal to the number of jobs, since each job gives rise to a unit-time communication with the shared memory. We will see in