科技资料

HIGH PERFORMANCE COMPUTING II

HIGH PERFORMANCE COMPUTING II

Proceedings of the Second Symposium on
High Performance Computing
Montpellier, France, 7–9 October 1991

edited by

M. DURAND
IBM
Montpellier, France

F. EL DABAGHI
INRIA
Rocquencourt, France

1991

NORTH-HOLLAND
AMSTERDAM • LONDON • NEW YORK • TOKYO

ELSEVIER SCIENCE PUBLISHERS B.V.
Sara Burgerhartstraat 25,
P.O. Box 211, 1000 AE Amsterdam, The Netherlands

Distributors for the United States and Canada:
ELSEVIER SCIENCE PUBLISHING COMPANY, INC.
655 Avenue of the Americas
New York, N.Y. 10010, U.S.A.

ISBN: 0 444 89224 9

Printed in the Netherlands

PREFACE

The Second Symposium on High Performance Computing takes place from October 7th through 9th, 1991 in Montpellier, France. This conference is co-organized by two academic computing centers: CNUSC[1] and CNSF[2], a research institute: INRIA[3], and a computer company: IBM France.

This is the second of a series of biennal symposia which intend to be a forum for specialists working in various domains associated with Intensive Computing (Parallelism, Vectorization and Scalar) so as to discuss the state of the art. In fact, during the last decade, the growth of scientific computing devices has been firmly sustained: expanding size of memories, incredible CPU performance unheard-of just a few years ago, graphic tools transforming results treatment, networks drastically reducing communication time between computers, etc. As a matter of fact, this development has led to an indirect rapprochement, essential today, of domains which until recently were quite separated.

For instance, the interaction of computer architecture and numerical methods through new disciplines such as Parallel Numerical Analysis, allows optimal simulations as well as their applications onto complex models unreachable yesterday.

If it seems of a prime necessity for the hardware designer to take into consideration the multiple and often conflicting needs of the scientific computing community, it also seems obvious for users to steadily devote time to update their knowledge of computing environments.

So, the main purpose of this meeting is to give scientists an opportunity to investigate interactively areas such as Architecture of Supercomputers, Compilers, Algorithms, Computational Methods, Numerical Applications, etc.

This document holds the contributed papers selected for the Conference. They are arranged in three separate topics:

- Parallelism: Architectures, Algorithms and Compilers
- Numerical Methods for Supercomputers
- High Performance Computing Applications

---

1 Centre National Universitaire Sud de Calcul, Montpellier, France

2 Cornell National Supercomputer Facility, Ithaca, USA

3 Institut National de Recherche en Informatique et en Automatique, Rocquencourt, France

We greatfully thank CNUSC, CNSF, INRIA and IBM for their logistic and scientific support. We are quite appreciative of the sponsorships of Pôle Informatique, Montpellier L.R. Technopole, District de Montpellier, Département de l'Hérault, Région Languedoc-Roussillon, Multipôle Technologique Régional, IBM Europe and IBM France who have made this event possible.

In addition, we would like to express our gratitude to the members of the Scientific Committees for their active participation. Finally, special thanks to the following persons who, on account of their various and valuable contributions, greatly helped us in making this symposium a success. They include R. Glowinski, J-L. Delhaye, W. Jalby, A. Lefevre, A. Rivera and our C3NI colleagues: S. Carta, B. Cappelaere and G. Urbach.

M. Durand, F. El Dabaghi
SHPC Chairmen
Montpellier, June 1991

PROGRAM COMMITTEE

- Z. Barzilai, IBM Yorktown Heights (USA)
- C. Basdevant, Université Paris XIII / ENS Paris (France)
- C. Benoit, Université Montpellier II (France)
- L. Brown, Cornell University (USA)
- J-L. Delhaye, CNUSC, Montpellier (France)
- J. Demaille, Université Montpellier I (France)
- I. Duff, CERFACS, Toulouse (France) / RAL (UK)
- J. Erhel, IRISA, Rennes (France)
- L. Fezoui, INRIA Sophia Antipolis (France)
- P-J. van der Houwen, CWI, Amsterdam (The Netherlands)
- W. Jalby, INRIA Rocquencourt (France)
- M. Kalos, Cornell Theory Center (USA)
- E. Krause, RWTH Aachen (Germany)
- D. Laforenza, CNUCE, Pisa (Italy)
- P. Leca, ONERA, Chatillon (France)
- O. Mc Bryan, Colorado University, Boulder (USA)
- F. Marcotorchino, CEMAP IBM, Paris (France)
- O. Pironneau, Université Paris VI / INRIA (France)
- R. Scarborough, IBM Palo Alto Scientific Center (USA)
- P. Sguazzero, IBM ECSEC, Rome (Italy)
- K. Stuben, GMD, Sankt Augustin (Germany)

CONTENTS

NUMERICAL METHODS FOR SUPERCOMPUTERS

HIGH PERFORMANCE COMPUTING APPLICATIONS

PARALLELISM: ARCHITECTURES, ALGORITHMS AND COMPILERS

High Performance Computing II
M. Durand and F. El Dabaghi (Editors)

Combining allocation and scheduling

N. ABDENNADHER, J.C. ANGUE

Laboratoire d'Automatique Industrielle et Humaine. LAIH - UA CNRS 1118
Université de Valenciennes et du Hainaut Combrésis
Le Mont Houy 59326 Valenciennes cedex - France.

Abstract

This paper deals with a heuristic algorithm for Task Allocation Problem which takes into account the precedence rules between tasks and the scheduling policy implemented on processors of the target machine (Time sharing in our case).

This algorithm does not only aim at mapping the tasks into a given architecture but also at "building" the processor network which suits best the structure of the parallel program. This construction must meet hardware constraints such as number of processors, number of links per processor, basic structure of the network, etc ... The algorithm is evaluated in a concrete case of parallel algorithm written in OCCAM and executed on Transputer network.

I. INTRODUCTION

Recent computer applications require systems which process faster and faster. The attempts to meet this requirement based on technological improvements alone have failed because of the limitations of physical laws. The best way to solve this problem is to share the processing among several processors : The sequential program is thus transformed into a parallel program composed of several inter-communicating tasks. However, one of the most important problems encountered in parallel programming is that of allocating tasks to processors so as to achieve minimal execution time. This problem, known as "Task Allocation Problem" (or Mapping Problem) is ***NP-COMPLET*** : Its complexity grows exponentially with the size of data. Two types of algorithms are used to solve the task allocation problem :

- Exact algorithms which lead to an optimal solution (allocation) but whose complexity is minored by an exponential function.

- Empirical algorithms whose complexity is majored by polynomial functions, but which do however supply solutions nearing the optimum.

Task allocation has to take into account two opposite criteria :

- Communication costs : Intraprocessor communication costs are quite trivial against those of interprocessor communications. Thus, minimizing these costs amounts to assigning tasks to a single processor.

- Processor load : To take advantage of the parallelism inherent to the parallel program, tasks need to be allocated to different processors.

Solulion attempts for the mapping problem can be classified into five categories : graph theory [1-3], integer 0-1 programming [4-5], branch and bound methods [6 -7], heuristic approaches [8-10] and simulated annealing approaches [11-12].

In this paper, we propose a heuristic algorithm for task allocation problem which takes into account the precedence rules between tasks and the local scheduling policy implemented on each processor : this is necessary to give a realistic definition of the processor load. This algorithm does not only aim at mapping the tasks into a given architecture but also at "building" the processor network which will perform the application. This construction must meet hardware constraints such as number of processors, number of links per processor ... etc. This algorithm is used to map parallel programs written in OCCAM and implemented on Transputer networks.

This paper is organised as follows : section II presents a heuristic algorithm for task allocation problem. Section III gives experimental results. Finally, section V offers conclusions as well as directions for future research.

II. HEURISTIC ALGORITHM FOR TASK ALLOCATION PROBLEM

The parallel program is represented by a data flow oriented graph $G_t = (\mathcal{T}, \mathcal{A}_t)$ where nodes are tasks ($\mathcal{T}$) and links are data flows between tasks ($\mathcal{A}_t$) (Fig. 1). (T_i, T_j) belongs to $\mathcal{A}_t$, means that T_i sends data to T_j : T_i is an immediate predecessor of T_j. A task is a sequential program which integrates communication routines : sending and receiving messages. Two virtual tasks, corresponding to the start and the end of the parallel program, are added to the graph : **Start** and **End**.

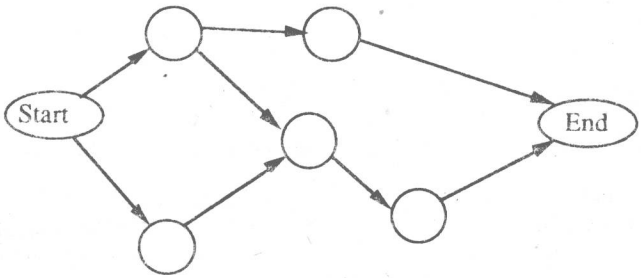

Fig. 1 : data flow graph.

The proposed algorithm aims at searching the allocation (of tasks $\{T_j\}$ among processors $\{P_Q\}$) which minimizes the execution time of the parallel program, in other words the ending date ($\mathbf{ED_{end}}$) of the task **End**. This algorithm is composed of three phases :

1- The first one, called ***greedy phase***, consists in distributing the tasks among processors. During this phase material constraints such as the number of links or the initial structure of the network are not taken into account. This phase has two goals :

a) to assign each task to a processor,

b) to build an "ideal" network which may not be achieved. This network is made of the set of processors $\mathcal{P}$, linked by connections (called ***virtual links***), which support inter-tasks communication channels

2- The second phase, called ***multiplexing phase***, consists in changing the ideal processor network defined in the first phase into an effective network which meets the material constraints. This aims at multiplexing virtual links on one or more physical links. Multiplexing should minimize idle times due to the use of the same physical link by several channels of communication. When the waiting time increases, it is sometimes better to use a longer path including many physical links : this is the "routing".

3 -The third phase, called ***routing phase*** aims at defining the optimum path for each virtual link not assigned to a physical link during the second phase.

II.1. Greedy phase

The greedy phase searches for an optimal partitioning of the program into p groups (p is the number of processors). We suppose that the processor network is completely connected. The optimal partitioning is that which minimizes the execution time of the whole program ; that is to say the ending date (ED_{end}) of the last task **End**. To process this date, it is necessary to compute the starting date (SD_j) and the ending date (ED_j) of all tasks :

$$SD_i = \max_{T_j \in PRED_i} (ED_j + c_{ji})$$

$$ED_i = SD_i + w_{iq} + a_{iq}.$$

Where SD_i (resp. ED_i) is the start (resp. ending) date of the tasks T_i.

w_{iq} is the execution cost, expressed in time, of the task T_i when it is executed by the processeor P_q. c_{ij} is the communication cost, expressed in time, between T_i and T_j. a_{iq} is the idle time of task T_i (on processor P_q) due to the execution of several tasks on processor P_q.

a_{iq} depends on two factors :

- the temporal allocation of the tasks assigned to processor P_q.

- the scheduling policy implemented on each processor.

At each iteration, the algorithm tries to assign one task T_i, whose predecessors have already been mapped, to a processor P_q belonging to $\mathcal{P}$. The ending date of the latest task already mapped ($ENDD_q$) is calculated according to the new allocation. The algorithm holds the processor P_{elu} which minimizes $ENDD_q$:

$$ENDD_{elu} = \min_{P_q \in \mathcal{P}} ENDD_q$$

$$ENDD_q = \max_{T_j \text{ already assigned or } T_j = T_i} ED_j$$

The number of processors is not necessarily known from the beginning. Thus, the definition of the ideal architecture of the processor network and the number of processors p are both needed from the start. The set of processors $\mathcal{P}$ is assumed to be empty at the initialisation. At each iteration, a new processor P_p is added to $\mathcal{P}$. The algorithm tries to assign a Task T_i, whose predecessors have already been mapped, to one of the processors belonging to $\mathcal{P}$. The maximum ending date of the already assigned tasks ($ENDD_q$) is calculated according to the new allocation. The algorithm keeps the processor P_{elu} which minimizes ENDD.

Two situations may arise :

1 -P_{elu} is equal to P_p : this means that the new processor P_p has contributed to a faster execution of the parallel program. P_p is kept in P and the number of processors, p, is incremented by 1.

2 -P_{elu} differs from P_p. Adding a new processor P_p does not produce any improvement in the execution time of the parallel program. P_p is thus removed from $\mathcal{P}$.

The multi-tasking phenomena involves waiting times due to the share of the processor among several tasks. These idle times depend on both the temporal allocation of the tasks and the scheduling policy implemented on each processor. In this paper, we assume that this policy is a ***Time Sharing*** one which consists in sharing processor time between all the tasks assigned to a given processor. During its execution, the task can have one of the following three states :

1) waiting state : the task is waiting for data from its predecessors.

2) processing state : the task is processing the data (received from its predecessors) in a specific way. The task is said to be ***ready***.

3) sending state : the taks is sending the results to its successors.

At each task T_i which is ready, correpond two variables which represent its descriptive :

- a_{iq} : the idle time of T_i (counted when T_i goes from a waiting state to the processing one) due to the execution of several tasks on a same processor,

- $exec_{iq}$: the "performed part" of the task T_i (expressed in time) : $exec_{iq} \leq w_{iq}$.

To process the idle time a_{iq} of all tasks belonging to T_q (T_q being the set of tasks assigned to P_q), it is important to process intervals during which the number of "ready" tasks is constant (Fig. 2). These intervals are bound by two events (e_h and e_{h+1}) which represent the starting and/or the ending date of one or several tasks.

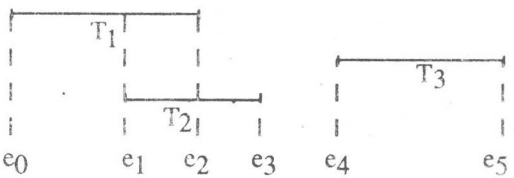

Fig. 2 : Temporal allocation of tasks

Events e_h are however difficult to process as they mainly depend on the idle times a_{iq}.

If e_h is assumed to be known, two situations may occur :