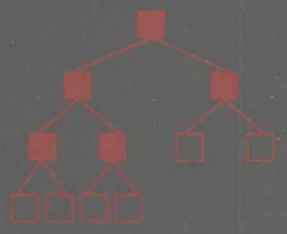

North-Holland

LANGUAGES, COMPILERS AND RUN-TIME ENVIRONMENTS FOR DISTRIBUTED MEMORY MACHINES

ADVANCES IN PARALLEL COMPUTING 3

J. Saltz
P. Mehrotra
Editors

LANGUAGES, COMPILERS AND RUN-TIME ENVIRONMENTS FOR DISTRIBUTED MEMORY MACHINES

Edited by

Joel Saltz
and
Piyush Mehrotra
Institute for Computer Applications
in Science and Engineering
NASA Langley Research Center
Hampton, VA, U.S.A.

1992

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, NY 10010, U.S.A.

Library of Congress Cataloging-in-Publication Data

Languages, compilers and run-time environments for distributed memory machines / edited by Joel Saltz and Piyush Mehrotra.
p. cm. -- (Advances in parallel computing ; v. 3)
Includes bibliographical references.
ISBN 0-444-88712-1
1. Electronic data processing--Distributed processing.
2. Programming languages (Electronic computers) 3. Compilers (Computer programs) I. Saltz, Joel. II. Mehrotra, Piyush.
III. Series.
QA76.9.D5L36 1992
004'.36--dc20 91-39628
CIP

ISBN: 0 444 88712 1

Printed in The Netherlands

LANGUAGES, COMPILERS AND RUN-TIME ENVIRONMENTS FOR DISTRIBUTED MEMORY MACHINES

ADVANCES IN PARALLEL COMPUTING

VOLUME 3

Bookseries Editors:

Manfred Feilmeier
Prof. Dr. Feilmeier, Junker & Co.
Institut für Wirtschafts- und Versicherungsmathematik GmbH
Munich, Germany

Gerhard R. Joubert
(Managing Editor)
Aquariuslaan 60
5632 BD Eindhoven, The Netherlands

Udo Schendel
Institut für Mathematik I
Freie Universität Berlin
Germany

NORTH-HOLLAND
AMSTERDAM • LONDON • NEW YORK • TOKYO

PREFACE

Distributed memory architectures have the potential to supply the very high levels of performance required to support future computing needs. However, programming such machines has proved to be awkward. The problem is that the current languages available on distributed memory machines tend to directly reflect the underlying message passing hardware. Thus, the user is forced to provide a myriad of low level details leading to code which is difficult to design, debug and maintain. The major issue is to design methods which enable compilers to generate efficient distributed memory programs from relatively machine independent program specifications. An emerging approach is to allow the user to explicitly distribute the data structures while using global references in the code. The major challenge for compilers of such languages is to generate the communication required to satisfy non-local references.

Recent years has seen a number of research efforts attempting to tackle some of these problems. Several of these groups were invited to present their work at the Workshop on Languages, Compilers and Run-Time Environments for Distributed Memory Machines held in May 1990 under the auspices of the Institute for Computer Applications in Science and Engineering (ICASE) at NASA Langley Research Center, Hampton, VA. This book is a compilation of the papers describing the research efforts of these groups.

The papers cover a wide range of topics related to programming distributed memory machines. The approaches taken by the authors are quite eclectic; in presenting these papers we chose not to impose an arbitrary classification scheme on this research. Consequently, we have chosen to list the papers in the order in which the corresponding talks were presented at the workshop.

- The paper by Gerndt & Zima describes their experiences with SUPERB, a semi-automatic parallelization tool for distributed memory machines.

- Rosing et al. survey the issues and challenges facing the developers of languages for programming distributed memory systems. They also provide a description of their own effort in this direction, the programming language DINO.

- Chapman et al. present the language extensions, Vienna Fortran, which allow the user to program in a shared memory environment while explicitly controlling the distribution of data to the underlying processors and memories.

- Pingali & Rogers discuss the techniques for generating optimal message passing code from the functional language, ID Noveau, extended with data distributions. Most of these techniques are applicable to general imperative languages as well.

- Snyder discusses the idea of phase abstractions to support portable and scalable parallel programming.

- The paper by Malki & Snir presents Nicke, a language which supports both message passing and shared memory programming.

- Balasundaram et al. describe a system to predict the performance of Fortran D programs.
- The paper by Hiranandani et al. describes the principals behind the Fortran D compiler being constructed at Rice. Fortran D is a dialect of Fortran which includes language extensions to specify data distribution.
- The Pandore system, as described by André et al., extends C with data distribution statements. The compiler then translates this into code suitable for distributed execution.
- Das et al. discuss techniques for generating efficient code for irregular computations and for implementing directives that make it possible to integrate dynamic workload and data partitioners into compilers. The work of Das et. al. is presented in the context of Fortran D.
- Sinharoy et al. describe the equational language EPL, and discuss the issues involved in analyzing loops for aligning data and scheduling wavefronts.
- The paper by Chen et al. describes the CRYSTAL compiler which attempts to reduce communication overhead by recognizing reference patterns and using a set of collective communication routines which can be implemented efficiently.
- The papers by Ramanujam & Sadayappan, and Wolfe describe aspects of analyzing loops for optimal scheduling and distribution across a set of processors.
- Hamel et al. and Reeves & Chase present data distribution extensions to SIMD languages and discuss the issues involved in translating such programs for execution on distributed memory MIMD machines.

This book presents a set of papers describing a wide range of research efforts aimed at easing the task of programming distributed memory architectures. We very much enjoyed hosting this workshop and discussing technical issues with the authors of these papers; we hope that the reader will find these proceedings to be a valuable addition to the literature.

We would like to thank Dr. R. Voigt, Ms. Emily Todd and Ms. Barbara Stewart for all their help in organizing this workshop and the resulting book.

CONTENTS

Languages, Compilers and Run-Time Environments
for Distributed Memory Machines
J. Saltz and P. Mehrotra (Editors)

SUPERB: Experiences and Future Research

Michael Gerndt
Hans P. Zima
University of Vienna
Institute for Statistics and Computer Science
Rathausstr. 19/II/3
A-1010 Vienna, Austria
EMAIL: A4424DAN@AWIUNI11.bitnet

Abstract

In this paper, we report on experiences with the automatic parallelization system SUPERB. After analyzing the strengths and limitations of the approach, we outline the salient features of a successor system that is currently being developed at the University of Vienna and describe the direction of future research in the field of automatic parallelization.

Keywords: multiprocessors, analysis of algorithms, program transformations

1 Introduction

SUPERB (SUprenum ParallelizER Bonn) is a semi-automatic parallelization tool for the SUPRENUM supercomputer [Giloi 88, Trott 86]. The SUPRENUM machine is a distributed-memory multiprocessing system (**DMMP**) with a kernel system containing up to 256 nodes, a host computer, and a hierarchical interconnection scheme. Each node consists of an off-the-shelf microprocessor, local memory, a pipelined vector unit, and a communication unit. No shared memory exists in the system; processors communicate via explicit, asynchronous message passing.

The **programming model** for SUPRENUM can be characterized as a **Single-Program-Multiple-Data** model: a parallel program consists of a **host program** and a **node program**. The host program executes all input/output statements and performs global management functions; it runs on the host processor. The node program performs the actual computational task; it is activated as a separate **process** for each individual node.

The idea underlying this model is that the node processes execute essentially the same operations on different parts of the data domain in a **loosely synchronous** manner.

SUPERB is a source-to-source transformation system which generates parallel programs in SUPRENUM Fortran, a Fortran dialect for the SUPRENUM machine which supports the programming model outlined above and provides array operations in the style of Fortran 90 as well as message-passing primitives SEND and RECEIVE for inter-process communication. SUPERB combines coarse-grain parallelization with vectorization and thus exploits both the multiprocessing structure of the machine as well as the vector units in its individual nodes. The parallelization strategy is based on the **Data Decomposition Technique** [Fox 88] which is used for the manual parallelization of scientific codes. Starting with user-defined distributions of the program's arrays, SUPERB automatically transforms the code to exploit as much parallelism as possible. The reasoning behind this approach can be outlined as follows: While the decomposition and distribution of data is currently too hard a problem to be solved automatically (see, however, 8.4), the parallel program determined by a given data partition can be automatically generated by a restructuring system. This relieves the user from a significant amount of highly error-prone routine work.

SUPERB is the first implemented tool that performs automatic parallelization for a distributed-memory machine, based on data parallelism. It has been implemented in C in a UNIX environment and currently contains 110.000 lines of code. An obvious **success** of the project is the mere fact that the tool exists and can be actually applied to real programs. We took particular care to ensure that virtually the complete ANSI standard Fortran 77 language is accepted as the source language, which (among other things) made it necessary to hand-code the scanner and handle COMMON and EQUIVALENCE as well as data and entry statements. As a result, we have an operational tool which can be used to gain experience with parallelization and serve as a starting point for new research.

Furthermore, the theory of the compilation approach has been completely developed in Gerndt's Ph.D thesis [Gerndt 89b].

There are some obvious **shortcomings**: SUPERB is a prototype and as such still contains programming errors of different degrees of severity. Since the real SUPRENUM machine was not operational until recently, our tests so far were restricted to running parallel programs on a simulator. Moreover, the size of programs that can be handled is currently restricted to about 5000 lines of code.

The rest of the paper consists of two parts. In the first part, we will discuss the experiences with the system in more detail. Parallelization can be described as a process consisting of the following steps:

1. Program splitting
2. Data partitioning
3. Interprocedural partitioning analysis
4. Initial adaptation: Masking and insertion of communication
5. Optimization

In the five sections below we will discuss each of these steps individually. The second part of the paper outlines current and future research, which extends SUPERB in various new directions (Section 8). A short summary is given in the conclusion (Section 9).

In this paper, we will not elaborate the compilation strategy of SUPERB in full detail: this material is contained in a number of published papers and theses [ZBGH 86, GerZi 87, KBGZ 88, ZBG 88, Gerndt 89a, Gerndt 89b, Gerndt 90]. The evaluation is based upon the work of people (mainly from SUPRENUM GmbH and Rice University) who have actually used the system for the parallelization of programs. In addition, we compared our strategy with the techniques of other groups working in the field of parallelizing compilers for DMMPs [KoMeRo 88, CalKen 88, RogPin 89].

2 Program Splitting

Program splitting transforms the input program into a host and a node program according to the programming model of the SUPRENUM system (Section 1). All I/O statements are collected in the host program, and communication statements for the corresponding value transfers are inserted into both programs.

In the resulting code, the host process is loosely synchronized with the node processes. Thus, the host process may read input values before they are actually needed in node processes. The output generated by the parallelized program is in exactly the same format as that of the sequential program.

In a simpler and intuitively more appealing approach described in the literature [Fox 88], I/O statements are directly inserted into the (hand-coded) node program for a Cosmic Cube and specially handled in the CUBIX operating system. The disadvantage of their approach is that additional communication has to be inserted to synchronize the node processes, if for example the values of the elements of a distributed array should be output in the original order.

Program splitting is performed automatically in the front end of SUPERB. In the current version of the system, individual I/O statements are mapped to single messages. Optimization of the message communication between host and node processes, in particular vectorization and fusion of messages along the lines of the techniques described in Section 6, will be included in a later version.

3 Data Partitioning

The major part of the node program's data domain consists of arrays. The selection of a **data partition**, by which we mean here a set of specifications for array distributions, is

the starting point for coarse-grain parallelization. It will determine the degree of parallelism, load balancing and the communication/computation ratio of the resulting parallel program.

The main restrictions on data partitioning are that the number of processes and the parameters of the distributions are compile-time constants. Furthermore, no facility for dynamic redistribution exists.

Our tool allows the specification of a number of distribution strategies. A distribution is determined by the **decomposition** of an array into rectangular **segments** and the segment-process mapping. This mapping permits the replication of segments across a range of node processes.

Decompositions are specified for each dimension of an array individually. The strategies provided by SUPERB range from stipulating the number of sections in a dimension to a precise definition of the index ranges. A specification may use variables whose values are determined by solving a system of linear equations at compile time.

Besides a canonical mapping scheme the segment-process mapping can be explicitly given using a notation which resembles that for implied do-loops in Fortran 77.

Thus SUPERB provides powerful language constructs for specifying array distributions. Experience so far indicates that real programs do not require such flexibility. The most commonly used features include the standard sectioning of an array dimension into equal-length parts and the replication of segments across several processes. A more elaborate specification may be necessary if the work load is to be optimally balanced to get very efficient programs.

The capacities for specifying array distributions in a very flexible manner considerably increase the overhead involved in the adaptation of the code according to a given partition. Further experience with SUPERB will tell us just which distribution strategies are really required in practice.

A special feature of the data partitioning language is the **workspace concept**. Applications written in Fortran 77 frequently define large arrays (workspace arrays) to circumvent the lack of dynamic storage allocation. These workspace arrays are typically accessed in a regular manner via run time dependent virtual arrays. SUPERB supports this programming method by providing a special notation for the declaration of virtual arrays, which can be distributed in the usual way. This technique has been used to parallelize multigrid programs and will be described in a forthcoming paper.

4 Interprocedural Partitioning Analysis

As discussed in the previous section, the user specifies distributions for non-formal arrays. Dummy arrays inherit their distribution at run time when the subroutine is called. Since the in-line expansion of all subroutines is practically impossible, a parallelization tool must be able to handle subroutines in an adequate manner. SUPERB performs an

interprocedural partitioning analysis to determine which dummy arrays are distributed and which distributions may occur at run-time. The resulting information is stored in a set of **partition vectors** which describe the actual distributions of arrays for every incarnation of a subroutine.

In order to be able to generate efficient code, incarnations where a dummy array is distributed are separated from those where the same array is non-distributed. This separation is implemented by cloning the subroutine and redirecting its calls appropriately. Thus, different code can be generated for these two cases.

The disadvantage of this strategy is the high memory overhead which in the worst case may grow exponentially depending on the number of calls and dummy arrays. Fortunately, this effect has not been observed for real programs, partly due to the fact that large programs generally use common blocks as their main data structures.

5 Automatic Insertion of Masking and Communication

The core of the parallelization process consists of two phases, the first of which performs an initial **adaptation** of the input program according to the given data partition. In the second phase, the adapted code is subjected to a set of optimizing transformations. In order to facilitate the implementation of the optimization, the adaptation is performed by manipulating special internal data structures without modifying the program code. The actual generation of parallel Fortran code is referred to the back end (see Section 7). We believe that this implementation strategy is more flexible than the approach described in [CalKen 88].

In this section we discuss the initial adaptation, while Section 6 deals with the optimization phase. The initial adaptation distributes the entire work assigned to the node program across the set of all node processes according to the given array distributions, and resolves accesses to non-local data via communication. The basic rule governing the assignment of work to the node processes is that a node process is responsible for executing all the assignments to its local data that occur in the original sequential program.

The distribution of work is internally expressed by **masking**: A mask is a boolean guard that is attached to each statement. A statement is executed if and only if its mask evaluates to *true*. Masks clearly separate the distribution of work from the control flow and thus can be handled efficiently.

After masking has been performed, the node program may contain references to non-local objects. The remaining task of initial adaptation is to allocate a local copy for each non-local object accessed in a process, and to insert code so that these copies are updated by communication. SUPERB uses a technique called **overlap communication** to compute the communication overhead at compile time and to organize communication at run-time in an efficient way.

The overlap concept is especially tailored to efficiently handle programs with local computations adhering to a regular pattern. For such programs, the set of non-local variables of a process can be described by an **overlap area** around its local segment.

Overlap descriptors are also computed for each reference to a distributed array. They provide important information for the user since they determine the run-time communication overhead and thus display inefficiencies resulting from an ill-chosen data partition, non-local computations or imprecise analysis information. Overlap areas may be inspected at different levels, including program, subroutine and statement level. This feature of SUPERB is especially important when handling large programs, as it facilitates the identification of those parts of the source code which must be interactively transformed to obtain efficient code.

In summary, **the overlap concept is valuable** because it allows us to

- statically allocate copies of non-local variables in the local address space of a process
- handle identically those iterations of a loop which access local data only and those which handle non-local data
- generate a single vector statement for a loop.

On the other hand, the overlap concept cannot adequately handle computations with irregular accesses as they arise in sparse-matrix problems, for example, since it requires the allocation of memory for any potentially non-local variable and an additional overhead for the resulting communication which may be superfluous.

6 Optimization

The code resulting from the initial adaptation is usually not efficient, since the updating is performed via single-element messages and work distribution is enforced on the statement level. In the optimization phase, special transformations are applied to generate more efficient code.

First, communication is extracted from surrounding do-loops [Gerndt 90] resulting in the **fusion of messages** and secondly, loop iterations which do not perform any computation for local variables are suppressed in the node processes. This transformation is called **mask optimization** and will be described in a forthcoming paper. Additional transformations are applied to eliminate redundant communication and computation.

SUPERB determines **standard reductions** and treats them in an efficient way. This transformation is a good example to demonstrate the advantages of the two-level approach. For example, let the reduction, e.g. a dot product of two distributed arrays, be implemented via an assignment statement to a scalar variable in a loop. This reduction is computed entirely in each node process since scalar variables are replicated.

```
S=0
Update local copies of entire arrays in each process
EXSR (A(*),...)
EXSR (B(*),...)
DO I=2,100
S: ALWAYS→S=S+A(I)*B(I)
ENDDO
```

The optimizing transformation first generates a new mask from one of the arrays accessed on the right side. This mask enforces that each node process computes the dot product only for its local values. Then a call to a subroutine is inserted behind the reduction code, to combine the local values to the global dot product and to communicate this value to all node processes.

```
S=0
no communication since all references are local
DO I=2,200
S: owned(A(I))→S=S+A(I)*B(I)
ENDDO
CALL COMBINE(S,"+")
S now contains the global dot product
```

If the access patterns of both arrays are different it may be necessary to insert communication to perform the local part of the dot product. In the two-level approach the communication is automatically inserted since the analysis algorithm depends on the mask of the statement and the reference to the distributed array. For the same reason the code is correctly transformed even if one of the arrays on the right side is non-distributed.

```
S=0
Updating of the copy for the accessed non-local variable
EXSR (B(*),...)
DO I=2,200
S: owned(A(I))→S=S+A(I)*B(I-1)
ENDDO
CALL COMBINE(S,"+")
```

Another important optimization is **Mask Propagation**. This transformation eliminates redundant computation in the node processes. Scalar variables are used frequently inside loops to store intermediate results. Since scalar variables are replicated, all intermediate results are computed in each node process, although an individual node process does not need all of them.

```
Update local copies of the entire array A in each process
EXSR (A(*),...)
DO I=2,48
S: ALWAYS→S=...A(2*I)...
S1:owned(B(2*I))→B(2*I)=0.5*(S+A(2*I+3))
ENDDO
```

When propagating the mask *owned(B(2*I))* to S, the redundant computations in the node processes as well as the accesses to non-local array elements have been eliminated.

```
DO I=2,48
If A and B have the same distribution, a node process only accesses local elements
S: owned(B(2*I))→S=...A(2*I)...
S1:owned(B(2*I))→B(2*I)=0.5*(S+A(2*I+3))
ENDDO
```

The implemented transformations are very useful, but have to be extended in several directions. For example, the pattern matching algorithm for detecting reductions is only able to detect these reductions if they are implemented via scalar variables. Mask propagation which eliminates redundant computation in the case of scalar temporaries in loops, works only inside a single loop and cannot detect the same situation for temporary arrays.

One important optimization area has not been dealt with up to now. All optimizing transformations are local transformations, i.e. they are applied to a single loop nest. There are no global transformations in the sense that entire procedure calls are masked or communication is propagated in the flow graph or extracted from subroutines.

7 System Structure

The structure of SUPERB is shown in Figure 1. The system is made up of six main components: the kernel, the frontend components, the backend and the reconstructor, which utilize the program database. The different components perform the following tasks:

Frontend 1:	scanning, parsing and normalization of individual program units
Frontend 2:	program splitting, collection of global information, e.g. call graph
Frontend 3:	execution of a predefined transformation sequence on each program unit
Backend:	machine dependent transformations
Reconstructor:	reconstructor of the final code from the syntax tree
Kernel:	system organization, execution of analysis services and transformations