Generating Parallel Loops from Serial Code for PVM

Fubo Zhang and Erik H. D'Hollander
University of Ghent, Dept. of Electrical Engineering
Sint-Pietersnieuwstraat 41, B-9000, Gent, Belgium

Abstract. Enhancing high performance computing on distributed computers asks for a programming environment to help users write their parallel applications correctly, efficiently and easily. One of the open challenges for PVM is to generate parallel code from a serial program. In this contribution, a tool is presented to extract parallelism in loops with no loop-carried dependencies. Then code is generated to distribute computations and the data of the parallel loops over cooperating PVM computers.

1 Introduction

An important issue in parallel distributed computers is providing high performance computing for users' applications. This is done by assigning relatively independent components to distributed computers and executing them in parallel.

PVM [1, 2, 3] is a programming environment for the development and the execution of parallel applications on parallel distributed computers. PVM provides a strong programming interface and allows users to develop and execute their applications easily. However, users should extract program parallelism and distribute each independent component themselves. Sometimes, new errors may be introduced by users when a sequential program is translated into PVM code.

Recently, the parallelizer FPT (Fortran Parallel Transformer) has been extended to generate PVM code. FPT is able to detect parallel loops using a library of well known dependency techniques. The PVM code generator partitions the iterations of parallel loops found by FPT. Next the data are partitioned according to the partitioned code. The process allows to create and execute parallel loops starting from purely sequential Fortran code. In the next sections the major features of the PVM code generation and data partitioning are described. Section 2 describes FPT and section 3 illustrates the code and data partitioning. Results and conclusions are given in section 4.

2 Overview of FPT

FPT is a software for parallelizing programs currently under development at the ELIS laboratory of the University of Ghent. FPT reads a sequential program and extracts information such as data and control dependence. Parallel loops are generated from serial loops according to dependence information. Some important aspects are:

- Dependence Testing: Banerjee and GCD tests are used for building a data

dependence graph under which direct vectors are extracted. These methods have been extended to the trapezoidal region and give a more accurate results.
- Program Transformations: Besides automatic parallel loop generation, FPT has a link to TINY [7] which allows users to view and change the FPT program interactively.
- FPT generates parallel code for shared memory multiprocessors available at the laboratory.

3 Generation of PVM code

A parallel application of PVM consists of a master and several slaves. Slaves are distributed to a cluster of computers and are spawned by the master. A `DOALL` loop is a loop in which each iteration can run in parallel. There are two ways to partition computations of a parallel loop. One way is to partition the iterations according to an established data partitioning [6] or using existing data partitioning techniques [4, 5]. In this way it is difficult to balance computations. The other way is to distribute the data based on iteration partitioning. Since iterations of a `DOALL` are independent, it is easier to balance load by sending unfinished iterations to the least loaded computer.

In our approach, the master computer controls the data communication. At any time a slave should explicitly send or receive data via the master. A scheme for distributing code and data to slaves is described next.

Code Partitioning: Iterations are partitioned into groups of n/p, where n is the number of iterations and p is the number of slaves. When the slave computers have different capabilities, an improvement is that slaves auto-balance by taking repeatedly a number of iterations less than n/p, until all iterations are finished.

Data Distribution: Let *lhs* be an array reference to the left side of an assign statement and *rhs* be an array reference to the right side of an assign statement. Then the input data and the output data in the loop are identified by *rhs* and *lhs* respectively. The way to send data in PVM is packing data into a buffer and sending the buffer to its destination. On the other side, the receiving processor inputs the buffer and unpacks the data.

A simple method to determine the input and output data for a *slave* is to pack the *rhs* variables of a `DO` loop into a buffer one by one and send it to the slave. Likewise, the resulting data are gathered by putting *lhs* variables in the slave program one by one into a buffer and sending the result back to the master.

This data communication costs some overhead, because the data is packed iteratively and sometimes data is sent several times, when data areas overlap, especially in the *rhs*.

The method can be improved by sending data in blocks and reducing data overlap under the following conditions: 1) the loops have rectangular bounds and step size one. 2) array subscripts have the form $i+c$, where i the loop index and c is a constant.

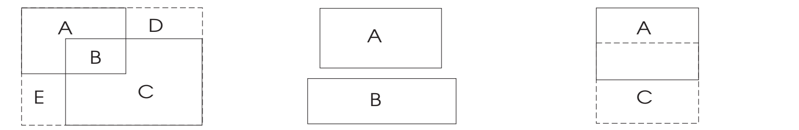
Under these assumptions a reference of array a occurring inside a loop has the form $a(e_1, \ldots, e_n)$, where n is the dimension of the array and subscripts e_k $(1 \leq k \leq n)$ are a linear expression of loop indexes. We define the area of an array reference as a tuple $Area(a(e_1, \ldots, e_n)) = (l_1 : u_1, \ldots, l_n : u_n)_a$, where l_i and u_i are the lower and upper bounds of e_i $(1 \leq i \leq n)$ respectively. The area of an array reference is also called the data area. Since rhs and lhs identify the input and output data, the combination of rhs and lhs areas gives the input and output data respectively.

Data overlap can be reduced by merging areas. The merging operation is defined by creating rectangular regions as follows:

$$(L_1 : U_1, \ldots, L_n : U_n)_a = (l_1 : u_1, \ldots, l_n : u_n)_a \cup (l'_1 : u'_1, \ldots, l'_n : u'_n)_a$$

where $L_i = min(l_i, l'_i)$ and $U_i = max(u_i, u'_i)$. Figure (1.a) gives an example where areas `A` and `C` overlap in `B`. When `A` and `C` is merged, the data in `B` is sent only once. However, then redundant data is introduced, see data areas `D` and `E`. Since redundant data in the output could overwrite valid data in the master, we only merge areas which do not create redundant data.

Two areas can be merged when 1/ they have no element in common (then they are effectively appended one after the other), (figure 1.b), 2/ one area is included in the other area (then the largest area is sent) or 3/ areas are equal in all but one dimension (figure 1.c).



(a) Area B is overlapped (b) No overlap (c) Areas can be merged

Figure 1. Areas overlap and unoverlap

The technique is illustrated by an example. A segment code of a slave is given by the following program.

```
do j=j0,u
  a(j,i)=a(j,i)-c(j)*a(j,i-1)
enddo
```

In order to determine the input and output data, lhs and rhs areas are identified. Two rhs areas are $Area(a(j, i)) = (j0 : u, i : i)_a$ and $Area(a(j, i-1)) = (j0 : u, i-1 : i-1)_a$. The lhs area is $Area(a(j, i)) = (j0 : u, i : i)_a$. Therefore the input data area $(j0 : u, i-1 : i)_a$ is the combination of $(j0 : u, i : i)_a$ and $(j0 : u, i-1 : i-1)_a$; and the output data area is $(j0 : u, i : i)_a$.

Since vector `c` is only input data, the master sends vector `c` only once. Using the input and output areas of matrix a, the generation of PVM code for sending and receiving data is straight forward.

4 Results and conclusions

Two examples,GJ (Gauss-Jordan Elimination) and MM (matrix multiplication), illustrate the use of FPT-generated PVM code. The following table gives speedups when problems with dimensions 400 and 800 are executed on 1 to 4 processors.

	GJ		MM	
#procs	400	800	400	800
1	1	1	1	1
2	1.3	1.33	1.77	1.7
3	1.7	2.48	2.28	1.39
4	1.9	3.17	2.77	2.11

Generating PVM code is one of important aspects of using a PVM environment. FPT-PVM enhances high performance computing in a distributed network of computers by translating a sequential application into parallel PVM code. At present our method doesn't remove all overlap areas and slaves don't preserve the output data for the next use. This will be addressed in future work. Data distribution is improved by eliminating areas of overlap and relaxing the assumptions; data communication is reduced by preserving output data. Furthermore, the generation of PVM code will be extended to other sources of parallelism, i.e parallel sections, parallel procedure calls.

References

1. Beguelin A., Dongarra J., Geist G.A., Manchek R., Sunder, Moore K., Wade R., Pank J., "HeNCE: A User' Guide", Oak Ridge National Laboratory and University of Tennessee, December, 1992.
2. Dongarra J., Geist G.A., Manchek R., Sunderam V.S., "Integrated PVM framework supports heterogeneous network", Tennessee Univ., Knoxville, TN, USA Computers in Physics Vol: 7 Iss:2 pp. 166-74, 1993.
3. Geist A., Beguelin A., Dongarra J., Jiang W., Manchek R., Sunder, "PVM 3 User's Guide and Reference Manual", Oak Ridge National Laboratory, Oak Ridge TN 37831-6367 Technical Report, ORNL/TM-12187, May, 108pp., 1993.
4. Gupta M., Banerjee P., "Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compilers on Multicomputers", *IEEETPDS*, Vol. 3, No. 2, March, pp. 179-93, 1992.
5. Hovland P. D., Ni L. M., " A Model for Automatic Data Partitioning", *ICPP93*, 2, August, pp. 251-259, 1993. 1990.
6. Su E., Palermo D.J., and Banerjee P., " Automating Parallelization of Regular Computations for Distributed-Memory Multicomputers in the Paradigm Compiler", *ICPP93*, 2, August, pp. 30-38, 1993.
7. Wolfe M. TINY. A Loop Restructuring Research Tool. Oregon Grad. Inst. of Sc. and Tech., Dept. Comp. Sc.& Eng., Beaverton. *Technical Report*, 19-14-21, 1991.

This article was processed using the LaTeX macro package with LLNCS style