TOP^2
Tool Suite for Partial Parallelization

Ulrich Detert and *Michael Gerndt*, Central Institute for Applied Mathematics, Research Centre Juelich, D-52425 Juelich, Germany

ABSTRACT: *TOP^2 is a tool suite that aids users of parallel systems with distributed memory in porting existing sequential applications by supporting the separation of compute-intensive kernels of an application from the existing sequential code and providing a development environment for the parallelization of these code segments.*
In this scenario, the sequential and the parallel code are run simultaneously as a distributed application on both systems and automatically exchange context data between both components. Main features in this process are the provision of cross-domain message passing for the automatic distribution of program data from the sequential machine to the distributed memory system and the ability of on-line debugging of the parallelized code. The data distribution features of TOP^2 are a subset of those defined in HPF Fortran and thus especially support algorithms on regular data structures exploiting data parallelism in the context of SPMD programming.

1 Introduction

Different from vectorization, the parallelization of existing sequential programs for distributed memory parallel systems is by its nature not a local process. Due to the necessity of data distribution, it is often very hard to break up large sequential applications into handy modules for parallelization since the data distribution in one module will always affect those in other modules and, hence, also the access to the data. For this reason, it is often not naturally possible to proceed in small steps when parallelizing large sequential applications for distributed memory systems.

As a consequence, most applications that are initially ported to new distributed memory parallel systems are either rather small in code size or, in other cases, are only partially parallelized, executing major portions of the code sequentially on just one processing element (PE) using dynamic distribution of data for the parallelized program kernel. If, however, the storage requirements for data arrays are too big to be fulfilled by a single PE, the latter mode of operation will not be possible. In this case, in order to modularize the task of program parallelization, it will be necessary to first isolate and parallelize a module from the existing sequential program, write a parallel main program and include the parallel code for execution, generate and distribute realistic input data for testing, and finally run the parallelized program module and compare its results with those of the original sequential module. The whole process has then to be iterated until the entire application is parallelized and all modules can be combined together. It is obvious that this method of parallelization will be rather tedious and also error-prone if applied to really big applications.

TOP^2 attacks the problem by a different approach, providing a distributed environment for the development and testing of partially parallelized applications, where the remaining sequential portions run on the sequential machine and the parallelized program modules execute on the distributed memory parallel system. With TOP^2, all program development steps, except the parallelization itself, are automated according to user directions. I.e. the extraction of a parallelizable program module, the generation of a parallel frame program, the data exchange between the sequential and the parallel machine over the network, the distribution of the data onto the processing elements, the conversion of the internal data formats, and finally the proper comparison of the results obtained in the parallelized version of the program module with those of the original module are all realized through TOP^2 functions.

The data distribution features of TOP^2 are a subset of those defined in HPF Fortran [HPF 93] and thus especially support algorithms on regular data structures exploiting data parallelism in the context of SPMD programming. Abstract processor arrays with up to seven dimensions are the basic vehicle for data distribution specifications in this scenario. Fortran data arrays may be distributed by distributing each array dimension onto

the corresponding dimension of a processor array. The distribution scheme may be "block", "cyclic", or "replicate" (not distributed) in each dimension. The abstract processor arrays are mapped onto the physical processing elements of the target architecture which implicitly distributes the data onto the PEs. Intrinsic functions for index calculations on distributed arrays are provided in order to facilitate the development of the parallel SPMD code.

2 TOP^2 Structure

The overall functionality of TOP^2 can be subdivided into three basic phases: program analysis and annotation; code generation; distributed program execution

The resulting structure of TOP^2 is depicted in Fig. 1.

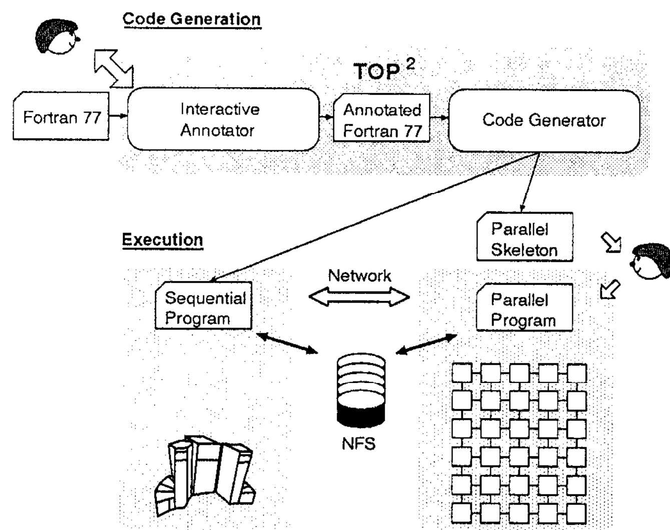

Figure 1: TOP^2 Structure

Analysis and Annotation

The annotation phase is supported by an interactive Annotator which allows the user to easily insert directives in the sequential Fortran 77 code that specify the set of input/output variables (context data) of a subroutine to be parallelized and distributions for the input/output arrays. The directives provide the information needed in the code generation phase.

In both areas the Annotator provides a menu driven interface for the specification thus freeing the user from syntactical peculiarities of the directives. The specification of input/output variables is supported by an interprocedural analysis whereas the decision as to which distributions are to be used is fully up to the user.

The analysis implemented in TOP^2 first determines the sets of used and defined variables for each statement in the individual units, i.e. a subprogram, function, or the main program. This analysis takes equivalence information as well as common blocks into account. The information on the statement level is summarized for the entire unit. All parameters or global variables read or written anywhere in the unit are determined to be input or output variables, respectively. TOP^2 does not take more precise data flow information into account.

Individual units are analyzed in postorder and thus the effects of call statements or function references are known from previous analysis of the referenced unit. The input/output variables of the unit are matched to the appropriate local variables of the unit.

Due to the conservative analysis the Annotator was designed to be an interactive tool. The results of the analysis can immediately be reviewed and interactively optimized by the user.

Further user interaction in this phase allows the specification of data distributions for the input/output arrays. The distribution scheme may be "block", "cyclic", or "replicate" in any number of dimensions of one- or multi-dimensional arrays. Logical processor arrays can be interactively defined as the target for data distribution. Furthermore, a number of code generation options may be selected in this phase.

The implementation of the Annotator is based partly on the interprocedural data flow analysis of SUPERB, a semi-automatic parallelization system developed at the University of Bonn and the University of Vienna [GerZi 92], and partly on the Fortran 77 parser PAFF, developed at Research Center Juelich/KFA [Berr 89]. The final result of this phase is a Fortran program annotated with source code directives providing all information required for the code generation phase.

Code Generation

The code generation phase splits up the annotated application program into a sequential and a parallel component and generates calls to run-time routines for the exchange, conversion, and distribution of context data. The generated sequential program is complete in that it may directly be compiled, the parallel code forms a program skeleton where the user may include the code for the parallel program module. Once being included in the code, changes to the code will be preserved across multiple runs of TOP^2, thus allowing experimentation with different data distribution schemes, communication methods, debugging options, or varying numbers of PEs. If debugging is switched on, the code generator also produces code for the comparison of the results obtained in the parallelized program module with those of the original program. The implementation of the TOP^2 code generator is again based on PAFF.

Execution

The execution of the distributed application is the third phase of TOP^2. Main tasks in this phase are cross-domain message passing, data distribution and data merging, data conversion, and result comparison. All these tasks are performed through run-time routines. The cross-domain message passing allows to address individual PEs from the sequential system. Two alter-

native implementations are realized at present: The first uses shared files for communication, providing good portability across a number of hardware platforms but limited performance. The second implementation is based on UNIX sockets, still a fairly portable solution yielding better performance.

3 Source Code Directives

Source code directives are used for communication between the TOP^2 Annotator and the code generator, thus reflecting all functional details of TOP^2. Normally there will be no need for a user to directly insert or modify directives. This is also possible, however, and provides the potential of using TOP^2 as a batch tool.

The general form of TOP^2 directives is CKFA$ *keyword [value]*. The following keyword/value pairs are currently recognized by TOP^2:

IN *variable_list*
OUT *variable_list*
INOUT *variable_list*
PROCESSORS *procspec_list*
DISTRIBUTE *distspec_list*
N$PROC *number*
DEBUG
STARTSTOP *start [stop]*

The IN, OUT, and INOUT directives define variables and arrays that are input or output for the parallel program segment. *Variable_list* is a blank- or comma-separated list of variable names.

The PROCESSORS directive defines one- or multi-dimensional logical processor arrays used for data distribution. *Procspec_list* is a blank- or comma-separated list of processor specifications defining the size and shape of each processor array. TOP^2 requires that all defined processor arrays be of the same size so that a proper mapping of logical processors onto the physical PEs is possible. The size of each processor array must be defined statically, i.e. it must be known at compile-time. The use of symbolic constants (PARAMETER constants), however, is possible and is strongly recommended.

The DISTRIBUTE directive defines the data distribution policy for one or more arrays. Distspec_list is a list of distribution specifications, each defining the distribution policy for one array and optionally the processor array to be used as the target for data distribution.

N$PROC, finally, allows to specify a default for the number of processing elements to be used for data distribution, and DEBUG and STARTSTOP are directives used for the control of debugging.

Figure 2 shows some examples of typical source code directives.

```
          PARAMETER (IP = 8)
CKFA$ IN A, CARRAY
CKFA$ OUT A, B, D
CKFA$ PROCESSORS P(2,IP), Q(16), RS(2,2,4)
CKFA$ DISTRIBUTE A(BLOCK,CYCLIC,*) ONTO P
CKFA$ DISTRIBUTE B(*,BLOCK,*) ONTO Q,
CKFA$*           D(CYCLIC) ONTO Q
CKFA$ DISTRIBUTE CARRAY(BLOCK)
CKFA$ DEBUG
```

Figure 2: Source Code Directives

4 Data Distribution

The data distribution features of TOP^2 are a subset of those defined in HPF Fortran. Two important basic terms in this respect are "block" and "cyclic" distribution. The extension of one-dimensional distributions to multi-dimensional distributions is obtained by distributing each array dimension onto one dimension of a multi-dimensional processor array, where the number of distributed array dimensions must match the number of processor array dimensions. The processor array is then mapped onto the physical processing elements in column major order.

Details of processor mappings and index calculations for multi-dimensional distributions are given in [DeGer 94]. These are the basis for the implementation of the TOP^2 run-time routines for data distribution. For most applications it will not be necessary to directly refer to the formulae given there, since all basic index calculations for distributed arrays can be realized by means of the appropriate intrinsic functions that are available at run-time in the parallel code (see section 5).

Since the data distribution is known when the parallel program is generated[1] the declarations of distributed arrays are adjusted accordingly in the parallel skeleton. The new declaration is equal to the shape of the largest array segment assigned to a processor. The data declaration in the parallel program is very similar to that in the sequential code, i.e. the structure of COMMON blocks is maintained. If the declaration of an array is dependent on some context data, e.g. adjustable arrays, memory is allocated dynamically in the main program.

5 Intrinsic Functions for Index Calculations

TOP^2 provides a set of intrinsic functions that facilitate index calculations on distributed arrays and, thus, supports the user in writing the SPMD code for the parallel portion of the application code. This set of routines also includes functions for the inquiry of array, processor array, or distribution properties in the parallel code. The described functions are 'intrinsic' in that they are directly related to the distribution scheme chosen during the interactive phase of TOP^2, i.e. information from this phase is automatically passed to the intrinsic functions.

In order to allow for the utilization of the TOP^2 intrinsic functions even if an application has been entirely parallelized

1. The code is generated for a fixed number of processors.

and ported to the parallel machine (and, thus, should be independent of TOP^2), the implementation of these functions follows a two-level procedure:

- Level-one routines collect all information on arrays, processor arrays, and distributions and store them in an internal data base. As long as TOP^2 is used, these routines are automatically called in the parallel program for all relevant distribution items. If the TOP^2 intrinsics are to be used separately from TOP^2, however, the level-one routines have to be explicitely called in the user code.

  The following routines are available at level one:

  CALL TOP^2_DEF_PROC(proc_phd$,dim,d)

  CALL TOP^2_DEF_ARRAY(array_ahd$,dim,bounds)

  CALL TOP^2_DEF_DIST(dist_dhd$,proc_phd$, array_ahd$,dim,d)

  CALL TOP^2_UNDEF_PROC(proc_phd$)

  CALL TOP^2_UNDEF_ARRAY(array_ahd$)

  CALL TOP^2_UNDEF_DIST(dist_dhd$)

- Level-one routines return a handle for each array, processor array, and distribution that is defined. These handles are passed to the level-two routines that implement the actual index calculations for the thus identified items.

  When used with TOP^2, the naming convention for handles is such that TOP^2 appends a suffix _phd$ to each processor name (thus a processor array PROC may be identified through the handle PROC_phd$). Respectively, _ahd$ is appended to each array name and _dhd$ to the name of each array distributed onto a given processor array.

  Level two contains the following routines:

  CALL TOP^2_GET_OWNER(dist_dhd$,inx,own)

  CALL TOP^2_GET_LOCAL(dist_dhd$,gx,lx)

  CALL TOP^2_GET_GLOBAL(dist_dhd$,own,lx,gx)

  CALL TOP^2_GET_RANGE(dist_dhd$,owner, low,high,inc)

  CALL TOP^2_GET_SHAPE(dist_dhd$,owner, low,high)

  CALL TOP^2_GET_PROC(proc_phd$,node,inx)

  CALL TOP^2_GET_NODE(proc_phd$,inx,node)

  CALL TOP^2_GET_ALLNODES(proc_phd$,nodes)

  CALL TOP^2_GET_PSHAPE(proc_phd$,dim,inx)

  CALL TOP^2_GET_ARRSHAPE(arr_ahd$,dim,low,high)

  CALL TOP^2_GET_DIST(dist_dhd$,proc_phd$, arr_ahd$,dim,dist)

6 Cross-Domain Message Passing

With respect to functionality and run-time performance, TOP^2 heavily depends on the communication between the sequential and the parallel system. Functionally, the required communication model is that of a cross-domain message passing, where individual processing elements of the parallel machine can be addressed, such that sections of distributed arrays can be directly exchanged between the sequential and the parallel program modules. Since, in general, the sequential and the parallel machine will have a different system architecture, there is a need for implicit data conversion during communication. Generally, TOP^2 uses the External Data Representation (XDR) format for system independent data representation [Sun 90]. On Cray T3D, proprietary conversion routines are used for the conversion between Cray and IEEE data format.

Two alternative solutions have been realized in the current implementation of TOP^2, one using shared files, the other UNIX sockets for communication.

The protocol for communication via shared files is implemented through a fairly simple and robust mechanism: One file is used per addressed PE for each communication direction. An additional lock file ensures proper synchronization of read and write operations on the sequential and the parallel machine. Fig. 3 shows the principles of operation for communication via shared files.

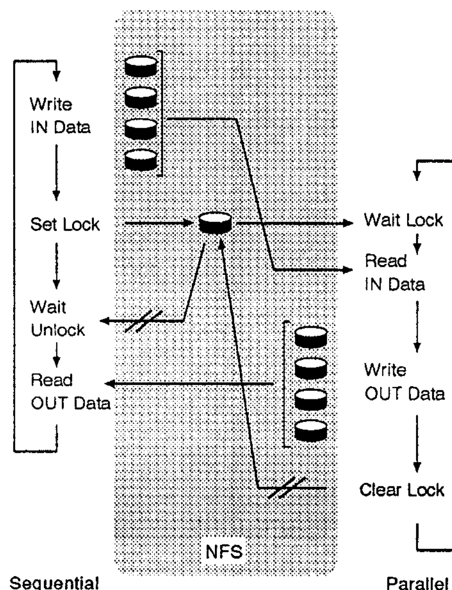

Figure 3: Protocol for Communication via Shared Files

The protocol for communication via UNIX sockets is more complex (Fig. 4). The sequential communication partner is implemented as a communication client, the parallel partner as a server. As there is a need to address individual PEs, *n* communication ports are required for communication, where *n* is the number of PEs. The first handshaking between server and client is realized via a fixed port, called the command port. This port is used to exchange the port identifiers of the *n* dynamically allocated data ports that carry the actual data transfer. The command port is only used for a short period of time and, thus,

may be shared by multiple TOP^2 applications. The data ports, on the other hand, are exclusively dedicated to a single application. In order to ensure that only valid client/server pairs are connected, a password mechanism is used during communication setup. The password is automatically generated during code generation.

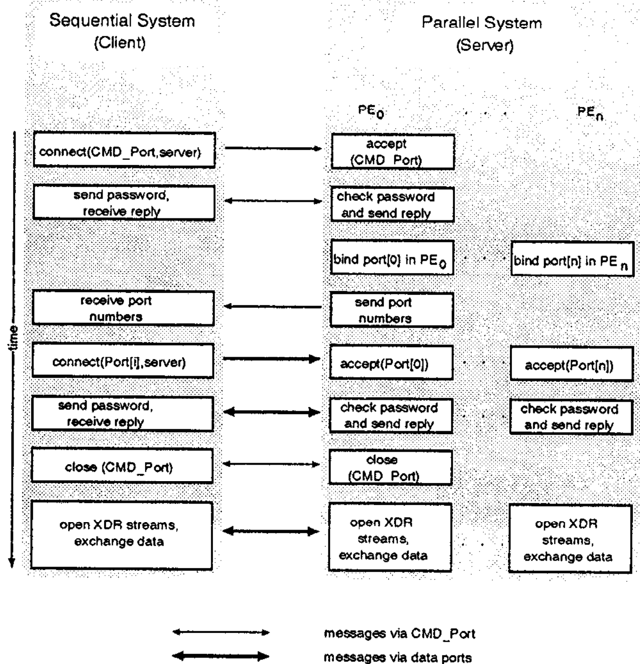

Figure 4: Protocol for Communication via Sockets

7 Debugging

An important feature of TOP^2 is its ability to automatically check the correctness of the results computed in the parallelized program module. If debugging is switched on, the control and data flow during execution of the distributed application is slightly different from the normal case (Fig. 5). In this case, not only the parallelized program module is executed, but redundantly also the original sequential module. After execution, the results of both components (all variables that are declared to be output) are compared. Integer, Logical, and Character variables are compared for identity, Real, Double Precision, and Complex variables are compared for equality within a certain allowed relative absolute error. Approximately two deviating decimal digits are allowed due to round-off errors in the current version.

If the results of the sequential and the parallel program modules do not match, the TOP^2 run-time system issues error messages showing the name of the involved variable (for arrays also the global array index), the PE number, and the computed result of the sequential and the parallel module.

If non-distributed arrays or variables are declared to be output, TOP^2 checks if all PEs redundantly return the same

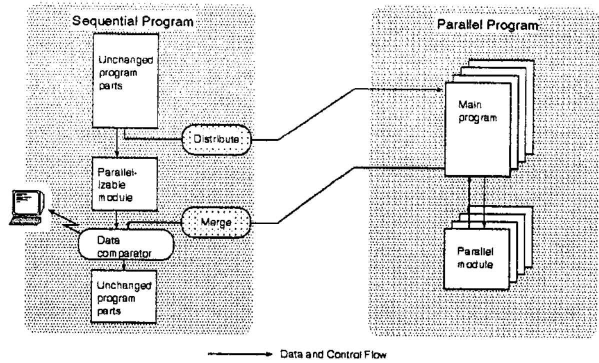

Figure 5: Program Structure with Debugging

result. If not, a warning message is displayed. In either case, only the result of PE zero is copied back to the sequential program.

8 Conclusion

Experiences with application programs have shown that TOP^2 is reasonably suited to be applied to the parallelization of big application codes. Yet, in order to increase its flexibility and usefulness, some improvements can be made. A key feature of TOP^2 is its ability to perform run-time data distribution across the network according to user-specific directives. The currently implemented distribution schemes form a subset of the HPF data distribution features. However, alignment of data is not supported. For a number of applications the implementation of this feature could improve the ease of use of TOP^2 and also the efficiency of the resulting parallel code. In other cases, the provision of overlap areas might be profitable as well.

For the parallelization of complete applications it will be essential to parallelize several modules one after the other and to finally combine them together. In the current version of TOP^2, this is not directly supported. Instead, the user has to apply TOP^2 to each module and then manually combine all parallelized components. A better solution will be, to allow for the parallelization of multiple modules and provide a scheduling mechanism that executes the proper parallel module at run-time according to the control structure of the sequential program. This mechanism would also make the parallelization of nested modules easier, a prerequisite for the incremental parallelization of entire applications.

References

[Berr 89]R. Berrendorf, *Der FORTRAN-Parser PAFF als wiederverwendbares Modul fuer Programmier Tools*, Research Centre Juelich, Technical Report, Jul-Spez-537, 1989

[DeGer 94]U. Detert, H.M. Gerndt, *TOP^2 Tool Suite for Partial Parallelization, Version 3.01 User's Guide*, Research Centre Juelich, Internal Report, KFA-ZAM-IB-9418, 1994

[GerZi 92]H.M. Gerndt, H.P. Zima, *SUPERB: Experiences and Future Research*, In: Languages, Compilers and Run-time Environments for Distributed Memory Machines, Editors: J. Saltz, P. Mehrotra, North-Holland 1992, pp. 1-15

[HPF 93] HPFF, *High Performance Fortran Language Specification*, High Performance Fortran Forum, Version 1.0, Rice University Houston Texas, May 1993

[MWW 92]M. Mihelcic, H. Wenzl, K. Wingerath, *Flow in Czochralski Crystal Growth Melts*, Bericht des Forschungszentrums Juelich, No. 2697, ISSN 0366-0885, December 1992

[Sant 91]M. Santifaller, *TCP/IP and NFS Internetworking in a UNIX Environment*, Addison-Wesley 1991

[Sun 90]Sun Microsystems, Inc., *Network Programming Guide*, Part No. 800-3850-10, 1990