Automatic migration from PARMACS to MPI in parallel Fortran applications [1]

Rolf Hempel [a,*] and Falk Zimmermann [b]

[a] *C&C Research Laboratories NEC Europe Ltd., Rathausallee 10, D-53757 Sankt Augustin, Germany*
E-mail: hempel@ccrl-nece.technopark.gmd.de

[b] *Institute for Algorithms and Scientific Computing, GMD – German National Research Centre for Information Technology, D-53754 Sankt Augustin, Germany*
E-mail: zimmermann@ccrl-nece.technopark.gmd.de

The PARMACS message passing interface has been in widespread use by application projects, especially in Europe. With the new MPI standard for message passing, many projects face the problem of replacing PARMACS with MPI. An automatic translation tool has been developed which replaces all PARMACS 6.0 calls in an application program with their corresponding MPI calls. In this paper we describe the mapping of the PARMACS programming model onto MPI. We then present some implementation details of the converter tool.

1. Introduction

During the last decade a number of portable message passing interfaces have been developed [9]. They enable an application program to be written at an abstract level without explicit use of machine-specific functions. One such interface, called PARMACS [2, 10], has been a commercial product for several years, and was chosen as the programming environment by many projects, most of them in Europe. Whereas PARMACS-based application codes are portable among different hardware platforms, they are not compatible with codes or libraries which use other message passing interfaces, like PVM or P4. Therefore, a standard for message passing is highly desirable.

Such a standard has been defined by the Message Passing Interface Forum. Representatives from virtually all major hardware vendors, national laboratories, universities, and other interested institutions have agreed on this MPI standard, which was published in its final version 1.1 in May, 1995 [3,4,12]. Meanwhile, public domain implementations are available for virtually every parallel platform, and some hardware vendors have produced highly optimized versions for their machines.

For an applications programmer, therefore, the question arises of how PARMACS code can be translated into MPI. For large program packages it would be very desirable if there were an automatic, or at least a semi-automatic migration tool. At GMD two techniques have been used to solve this problem:

– A PARMACS implementation has been written in terms of MPI. If this library is bound to a PARMACS application, the program can run on any machine where MPI is available, without the necessity of a machine–specific PARMACS implementation. Whereas this approach guarantees the future usability of old codes, the application programmer continues to use PARMACS for message passing and does not switch to MPI for future code modifications.
  Note that this does not provide an easy route to a mixed use of PARMACS and MPI in the application program. For this, the application programmer requires deep (and implementation dependent) knowledge of the naming and mapping of processes, internal communicators, etc.
– In the long run, another approach is more satisfying: an automatic tool parses the application source code and identifies all PARMACS calls. It then replaces each subroutine reference with some MPI code sequence, thus creating a version of the application program which uses MPI instead of PARMACS.

The second approach is quite ambitious, especially if the efficiency of the resulting code is important. Although MPI covers the general functionality of PARMACS, the semantics differ considerably. MPI con-

[1]This work was supported by the Commission of the European Community through Esprit project P6643 (PPPE).

[*]Corresponding author.

tains many features for writing very efficient code. Examples are the derived data-types by which extra copy operations before and after the message transfer can be avoided, and the low-level communication primitives which implement a channel-like protocol for frequent message transfers between the same processes. Whether or not the use of those features speeds up the code depends on the program context, which is very difficult to analyze automatically. A straightforward replacement of PARMACS calls by their MPI counterparts could, therefore, not employ these advanced features. Optimizations must be done by the applications programmer.

On the other hand, the automatic PARMACS-to-MPI transformation tool produces a working MPI version of the application program, thus taking away from the programmer the burden of rewriting the code manually. An attempt has been made to use as efficient MPI mechanisms as possible. If the resulting code is to be used in a homogeneous environment where the full generality of the message passing protocol is not required, the user can select code generation for this special case which leads to much less inserted text and thus to increased readability of the resulting MPI program.

The transformation tool is based on the following specifications, the detailed documentation of which can be found in [11]:

(1) A set of transformation rules giving for each PARMACS call the corresponding MPI code sequence.
(2) Include files containing data structures used by the inserted MPI code.
(3) A generic main program binding the PARMACS host and node program together into an SPMD program for MPI.

This paper concentrates on Fortran 77, although PARMACS and MPI are also available as C versions. The main reasons are that the vast majority of existing PARMACS applications use Fortran 77, and the transformation of C code will look very similar to the rules presented here.

In the following section we discuss the main differences between the programming models of PARMACS and MPI.

2. Comparison of programming models

The programming models of PARMACS and MPI do not match completely. As a consequence, in porting a PARMACS application to MPI some general difficulties and restrictions are encountered, the most important of which are described in this section.

PARMACS follows an MIMD programming model, with one host process and a number of node processes. A PARMACS application is transformed into a single program, which begins with a generic main program. This program initializes the MPI environment and analyses the homogeneity/heterogeneity of the data representations in the parallel MPI processes. This is done by performing collective MPI operations on some test data. As a result, every process holds a table which for every other process contains information on representation match/mismatch. Depending on the process identifier, the generic main program then calls either a host or a node subroutine which replaces the corresponding PARMACS program.

An important feature of PARMACS is the transparent handling of process topologies. The application programmer describes this topology on a purely logical level, without reference to the hardware structure on which the program is to be mapped. This mapping is done automatically by the PARMACS runtime system. Although the process topology handling is also available in MPI, it is done in a slightly different way. In PARMACS, the host process defines the node process arrangement and creates the nodes accordingly. In MPI, all processes are created at once, and each node process itself calls the mapping routine which assigns its role in the topology.

The translated MPI version of a PARMACS program proceeds as follows: the host process sets up the mapping structure and broadcasts it to the nodes, which in turn call the mapping functions. The host process then gathers a process table which contains for each node process its position in the topology.

The PARMACS definitions allow creating a second set of node processes once the first set has finished. This does not match with the static process concept of MPI-1. Therefore, using this PARMACS feature produces an error message during conversion. The second phase of the MPI standardization activity, which started in March 1995, will lead to the inclusion of a chapter on dynamic process management in the resulting MPI-2 standard. So, eventually it will be possible to remove this limitation in a later version of the translation tool.

The specification of message data-type information for the automatic conversion in a heterogeneous invironment is handled differently in PARMACS and MPI. In PARMACS, the application passes a contigu-

```
PARMACS:      INTEGER dest,msgtag,msglen,code
              <numeric> msgbuf(*)
              CALL PMSND ( dest,msgtag,msglen,
             $             msgbuf,code)
MPI:          PM_TEMP_RANK=dest
              PM_TEMP_TAG=msgtag
              PM_TEMP_LEN=msglen
              CALL MPI_SEND ( msgbuf,PM_TEMP_LEN,MPI_BYTE,
             $                PM_TEMP_RANK,PM_TEMP_TAG,
             $                MPI_COMM_WORLD,code)
```

Fig. 1. Translation of a PARMACS send operation in an homogeneous environment.

ous buffer to the send routine, together with a description of the buffer (e.g., two integers, followed by ten doubles). On the receiving side the message is passed to the application program, after any necessary data conversions have been made automatically. The regular MPI send/receive mechanism requires that the receiving process has a priori knowledge of the message data types and layout. Thus, this mechanism cannot be used in a PARMACS-to-MPI translation without the cost of one additional message containing the structure information. Therefore, two different protocols are used instead, the choice depending on whether the data representations match or not. If they do, the message is sent as a contiguous string of type `MPI_BYTE`, which gives the best possible performance. Only in the heterogeneous case, the pack/unpack mechanism of MPI is employed. So, the additional memory copies are only done in this case where communication speed cannot be expected to be high.

This strategy ensures high message-passing performance in the most frequent case where the representations match, without loosing the full generality in a heterogeneous environment. The program, however, can only select the protocol at run-time, based on the representation match/mismatch table. Thus, it must contain code for both execution paths, which results in a considerable increase in code length. If the MPI code is to be used in a homogeneous environment only, the user can select the command line option `-hom` which results in a considerable reduction of the program size. Figure 1 shows the translation of a send operation in the homogeneous case.

The arguments for message destination, tag, and length are assigned to temporary variables before being passed to `MPI_SEND`. The reason is that the actual arguments can be expressions containing PARMACS function references, which in turn can expand to several statements (see Section 3.5). Note that this cannot happen for the message buffer and error code which are output arguments. In the general case the inserted

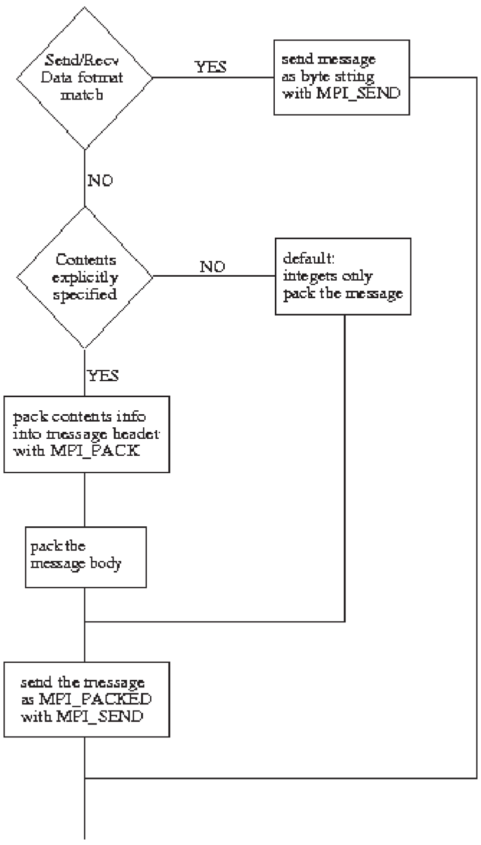
Send/Recv Data format match
YES
send message as byte string with MPI_SEND
NO
Contents explicitly specified
NO
default: integers only pack the message
YES
pack contents info into message header with MPI_PACK
pack the message body
send the message as MPI_PACKED with MPI_SEND

Fig. 2. Translation of a PARMACS send operation in an heterogeneous environment.

code contains some 50 lines, so Fig. 2 only depicts the general logic used.

Although part of the complicated structure of the heterogeneous MPI code is a result of the translation from PARMACS, a substantial fraction of it is unavoidable even if the user directly writes the MPI program. If, for the sake of saving multiple message start-ups, a message contains more than one basic MPI data-type, the set-up of derived data-types or the calling of buffer pack/unpack routines always results in relatively long code.

```
PARMACS:     tordim,torsiz(*),coords(*),code
             CALL PMQTOR (tordim,torsiz,coords,code)


MPI(HOST):   tordim=PM_MAP_SIZE
             DO PM_TEMP_I=1,PM_MAP_SIZE
               torsize(PM_TEMP_I)=PM_MAP_LIST(PM_TEMP_I)
             END DO
             code=0

MPI(NODE):   CALL MPI_CARTDIM_GET (PM_MAP_COM,tordim,code)
             CALL MPI_CART_GET (PM_MAP_COMM,PMXDIM,
            $                  torsiz,PM_TEMP_FLAGS,
            $                  PM_TEMP_COORDS,code)
             DO PM_TEMP_I=1,tordim
               coords(PM_TEMP_I)=PM_TEMP_COORDS(PM_TEMP_I)+1
             END DO
             code=0
```

Fig. 3.

3. The converter tool

3.1. Basic strategy

Transformation tools like the one presented in this paper are usually based on pattern recognition techniques. In this case there are several reasons, why the processing of input patterns requires detailed knowledge of the program context and cannot be accomplished by a straightforward string replacement. The definitions of PARMACS functions depend on their usage in a host or node program. Whereas most functions can appear in both contexts, some functions can only be used in either a host or a node routine. Even for those functions which have the same meaning in host and node programs the corresponding MPI code sequences can differ considerably.

A special problem is caused by the option in Fortran to define arrays with arbitrary index ranges, as, for example, in the following statement:

```
REAL A(-5:10).
```

If the MPI code which replaces a PARMACS call assigns values to this array in a loop, the loop index bounds have to be adjusted accordingly. This problem is discussed in more detail in Section 3.4.

To meet all these requirements, the conversion consists of an analysis and a transformation phase. For the analysis the converter employs a full-scale FORTRAN 77 front-end that collects all relevant data. This information is then used by the transformation program which again is built using compiler technology, based on the GMD development ADAPTOR [1]. Some of the problems encountered in the conversion of PARMACS calls are presented in the following sections.

Some PARMACS functions do not have counterparts in MPI-1. If they are used in an application, the automatic transformation fails. Also, the converter does not check the PARMACS input for correctness. This is the task of the application programmer, and no guarantee is made about the transformation of an erroneous program.

3.2. Context sensitivity

In PARMACS applications the host and node programs have different execution environments. Data structures initialized at program start contain different information, and some types of information are only available on either the host or node side. Although the transformed MPI program is of SPMD type, at any given time the state of a particular process depends on which kind of PARMACS process it emulates. Thus, PARMACS functions which are available on the host and node sides with identical calling sequences have to be translated into different MPI code sequences. By selecting the converter options `-host` or `-node`, the programmer specifies whether a PARMACS source code file is used in a host or node environment, repectively.

The example in Fig. 3 shows how the MPI code which replaces a simple PARMACS call depends on the host/node context.

3.3. Simple transformations

The most simple kind of transformation occurs if for a PARMACS function there exists a corresponding

```
PARMACS:    DOUBLE PRECISION wtime
            wtime=PMTIME ()

MPI:        wtime=MPI_TIME ()
```

Fig. 4.

MPI routine with identical number and types of arguments. In this case the transformation consists of a simple string substitution (see Fig. 4). Unfortunately, however, this situation is the exception rather than the rule, and in most cases the PARMACS and MPI functions differ considerably.

In general, the emulation of a PARMACS function in an MPI environment requires the generation of a more or less complicated code sequence. It may contain assignments to temporary variables as well as to the scalar or array arguments of the PARMACS procedure. If all PARMACS function arguments are scalar, the MPI code can be generated without additional knowledge about the actual arguments. The transformation in the next example is of this relatively simple kind (see Fig. 5).

The PARMACS function `PMPROB` delivers the tag and sender information of an incoming message via the scalar arguments `reqsnd` and `reqtag`, whereas the corresponding MPI function `MPI_IPROBE` uses a single array for the same purpose. The temporary array `PM_TEMP_STATUS` is passed to the MPI routine, and its return values are assigned to the scalar PARMACS arguments. Since the array indices `MPI_SOURCE` and `MPI_TAG` are fixed and context independent, the inserted MPI code is static, except for the names of the actual PARMACS arguments.

3.4. More complicated transformations

In the previous examples the transformations only require information which can be extracted from the PARMACS function reference, i.e., the number and the names of the actual (scalar) arguments. If a PARMACS function contains array arguments, the situation is more complex for two reasons: first, array indexing in Fortran 77 can start with any value (not necessarily 1), and, although not supported by the standard, it is common practice among Fortran programmers to pass an element somewhere in the middle of an array as actual argument to a routine which expects an array argument. If the converter inserts assignment statements to such an array argument, it must shift the indexing accordingly.

The following code sections exemplify these situations. In the first (standard) case, indexing of the actual array argument `sonvec` starts with 1 (see Fig. 6).

The same MPI code is generated if the PARMACS call is replaced by

```
call PMGLTR(nrsons,sonvec(1),father,code)
```

If, however, the actual PARMACS function argument is not the first array element, as in the following example

```
call PMGLTR(nrsons,sonvec(5),father,code)
```

the index space of the generated loop assignment has to be shifted:

```
DO PM_TEMP_I=1,PM_NUM_SONS
  sonvec(PM_TEMP_I+4)
     = PM_TREE_SONS(PM_TEMP_I)
END DO
```

```
PARMACS:     INTEGER result
             result=PMPROB (reqsnd,reqtag,sender,msgtag,code)


MPI:         CALL MPI_IPROBE( reqsnd,reqtag,MPI_COMM_WORLD,
            $                 PM_TEMP_FLAG,PM_TEMP_STATUS,
            $                 code)
             IF(PM_TEMP_FLAG)THEN
               PM_TEMP_L=1
               sender=PM_TEMP_STATUS(MPI_SOURCE)
               msgtag=PM_TEMP_STATUS(MPI_TAG)
             ELSE
               PM_TEMP_L=0
             END IF
             code=0
             result=PM_TEMP_L
```

Fig. 5.

```
PARMACS:     INTEGER nrsons,sonvec(*),father,code
             CALL PMGLTR (nrsons,sonvec,father,code)

MPI:         nrsons=PM_NUM_SONS
             DO PM_TEMP_I=1,PM_NUM_SONS
               sonvec(PM_TEMP_I)=PM_TREE_SONS(PM_TEMP_I)
             END DO
             father=PM_TREE_FATHER
             code=0
```

Fig. 6.

Therefore, the converter scans all actual array arguments for offsets and inserts them at the appropriate positions of the transformation sequence.

Finally, the same shift as in the last example has to be generated if, for some reason, the array declaration is replaced by

```
INTEGER sonvec(5:max_sons+4)
```

and the PARMACS function is called as in the original case. In this example the transformation is more difficult, because the index bounds of the actual array argument are not known at the point where the PARMACS function is called. Therefore, the converter looks up the array bounds in the internal tables generated by the compiler front-end.

Another strategy to handle arrays with non-standard offsets would be to copy them into temporary vectors with standard indexing. For long vectors, however, this would lead to a considerable overhead, both in memory usage and execution time.

3.5. Nested PARMACS references

PARMACS function references can occur in various different contexts, which the converter has to take into account in the transformation. In the most simple situation a subroutine call is replaced by some statement sequence. The transformation is more complicated for Fortran function references which may appear in any Fortran expression. In the easiest case where a PARMACS function is converted into a single MPI function, the converter replaces the PARMACS function with its MPI counterpart, and the structure of the code remains unchanged.

This straightforward technique fails if the function reference is expanded into several lines of code. We call the case where the function reference occurs in a simple assignment statement a *basic occurrence*. The expanded code is inserted directly preceeding the function reference, and the resulting function value replaces the function call in the original PARMACS statement.

If the function reference occurs in a more complex context, as, for example, in a numeric expression or an argument list of another function, the converter proceeds in two steps. First, it restructures the original PARMACS code by inserting an assignment to a temporary variable before the function occurrence, and replacing the original function reference with the temporary variable. This reduces the PARMACS function reference to a *basic occurrence*, and in the second step the converter can process the code as described before.

Fig. 7 shows an example of this situation.

The converter recognizes the PARMACS function `PMFORM()` in the argument list of the subroutine `PMSND` and replaces it with the temporary variable `TMP1`. At this point the converter is capable of transforming the `PMSND` call to MPI, using the still undefined temporary variable. A new statement which assigns the value of `PMFORM()` to the temporary variable `TMP1` is inserted preceeding the original statement. Since this assignment contains a *basic occurrence* only, the conversion can be completed.

4. Applications

An automatic converter is only useful if it is able to process large program packages correctly without manual intervention. Therefore, testing the PARMACS-to-MPI converter with large industrial applications has been of high priority. Large codes which have been successfully processed include:

– the operational European medium-range weather forecasting program (IFS) of the ECMWF at Reading, U.K. [5],
– the combustion simulation code FIRE of AVL,
– the market leading crash simulation package PAM-CRASH of the ESI group [6,7],

```
 CALL PMSND (dest,msgtag,PMFORM(nrsegs,sgtype,sglens,code),
$            msgbuf,code)

                         ↓ Normalization

 TMP1=PMFORM (nrsegs,sgtype,sglens,code)
 CALL PMSND (dest,msgtag,TMP1,msgbuf,code)

                         ↓ Transformation

 PM_TEMP_I=nrsegs
 PM_NUM_SEGMS=PM_TEMP_I
 PM_TEMP_L=0
 DO PM_TEMP_I=1,PM_NUM_SEGMS
   PM_TEMP_L=PM_TEMP_L+
$            PM_TYPE_EXTENT (sgtype(PM_TEMP_I)*sglens(PM_TEMP_I))
 END DO
 code=0
 TMP1=PM_TEMP_L


 PM_TEMP_RANK=dest
 PM_TEMP_TAG=msgtag
 PM_TEMP_LEN=TMP1
 CALL MPI_SEND( msgbuf,PM_TEMP_LEN,MPI_BYTE,
$               PM_TEMP_RANK,PM_TEMP_TAG,
$               MPI_COMM_WORLD,code)
```

Fig. 7.

– and the CLIC communications library of GMD which is the basis of the message-passing versions of the German industrial CFD codes for aircraft development [8].

The transformation of the first three codes was tested by the application code owners as part of the Esprit project P6643 (PPPE).

The communications library CLIC which contains more than 100,000 lines of communication-intensive code was transformed in the German national project POPINDA. The whole task was accomplished in a few days by a project partner who had no detailed knowledge of the library. They only reported one problem: Since the transformer removed all comment lines from the source code, they had to move them all back to their original locations.

In all test cases the converter proved to work in a stable way, and the generated code ran correctly.

Acknowledgment

We thank our colleague Robin Calkin for checking the manuscript and correcting our English.

References

[1] Th. Brandes and F. Zimmermann, ADAPTOR – A Transformation Tool for HPF Programs, in: *Programming Environments for Massively Parallel Distributed Systems*, K.M. Decker and R.M. Rehmann, eds, Birkhäuser, 1994, pp. 91–96.

[2] R. Calkin, R. Hempel, H.-C. Hoppe and P. Wypior, Portable programming with the PARMACS message-passing library, *Parallel Computing* **20** (1994), Special issue on message-passing interfaces.

[3] *Message Passing Interface Forum*. Document for a standard message-passing interface. Technical Report CS-93-214, University of Tennessee, November 1993.

[4] *The MPI Forum*. The MPI message-passing interface standard. WWW home page http://www.mcs.anl.gov/mpi/standard.html, Argonne National Laboratory, May 1995.

[5] U. Gärtel, W. Joppich and A. Schüller, Parallelizing the ECMWF's weather forecast program. Arbeitspapiere der GMD 740, Gesellschaft für Mathematik und Datenverarbeitung mbH, March 1993.

[6] E. Haug and D. Urlich, The PAM-CRASH code as an efficient tool for craschworthiness simulation and design, in automotive simulation, in: *Proceedings of the Second European Cars Trucks Simulation Symposium*, May 1989.

[7] J. Haug, E. Clinckemaillie and F. Aberlenc, Contact-impact problems for crash, in: *Post Symposium Short Course of the Second International Symposium of Plasticity*, August 1989.

[8] R. Hempel and H. Ritzdorf, The GDM communications library for grid-oriented problems. Arbeitspapiere der GMD 589, Gesellschaft für Mathematik und Datenverarbeitung mbH, November 1991.

[9] R. Hempel, A. Hey, O. McBryan and D. Walker (eds), *Parallel Computing* **20** (Special issue on message-passing interfaces), North-Holland, 1994.

[10] R. Hempel, H.-C. Hoppe and A. Supalov, *PARMACS 6.0 Library Interface Specification*, GMD, December 1992.

[11] R. Hempel, A. Supalov and F. Zimmermann, The conversion of parmacs 6.0 calls into their MPI counterparts in Fortran application programs. Arbeitspapier 879, Gesellschaft für Mathematik und Datenverarbeitung, November 1994. Also available by anonymous ftp from ftp.gmd.de, file name gmd/ppe/parmacs_to_mpi.uu.

[12] D.W. Walker, The design of a message passing interface for distributed memory concurrent computers, *Parallel Computing* **20** (1994).