# STEP: a distributed OpenMP for coarse-grain parallelism tool

Daniel Millot, Alain Muller, Christian Parrot, and Frédérique Silber-Chaussumier

GET/INT, Institut National des Télécommunications, France

**Abstract.** To benefit from distributed architectures, many applications need a coarse grain parallelisation of their programs. In order to help a non-expert parallel programmer to take advantage of this possibility, we have carried out a tool called STEP (*Système de Transformation pour l'Exécution Parallèle*). From a code decorated with OpenMP directives, this source-to-source transformation tool produces another code based on the message-passing programming model automatically. Thus, the programs of the legacy application can easily and reliably evolve without the burden of restructuring the code so as to insert calls to message passing API primitives. This tool deals with difficulties inherent in coarse grain parallelisation such as inter-procedural analyses and irregular code.

## 1 Introduction

On the one hand, parallel applications must evolve with hardware architectures ranging from computational grids to multi-core/many-core architectures. Indeed, since parallel programming seeks performance, parallel algorithms will be different depending on architectures in order to take advantage of their specificities. On the other hand, when developing a parallel version of some sequential application (or a new parallel application from scratch), parallel programmers have currently the choice between two alternatives to program their application: either MPI for message-passing programming [PIF95] or OpenMP for shared-memory programming [arb04]. Both ways have advantages and drawbacks. MPI provides a way for the programmer to control the behaviour of his parallel application precisely and especially the time spent in communication or computation. For this reason, MPI programming is considered low-level, and using it for development is time consuming. Furthermore the SPMD programming model of MPI provides a fragmented view of the program [CCZ06]. MPI code is intermingled with the original sequential code and this makes the global structure of the resulting code difficult to read. In contrast programming with OpenMP directives provides a simple way to specify which parts should be executed in parallel and helps keeping a "global view" of the original application. Driven by the directives, the actual parallelisation is delegated to a compiler. The main drawback of OpenMP is to be restricted mainly to shared-memory architectures. Furthermore hybrid programming OpenMP/MPI is necessary to exploit hybrid architectures such as clusters of many-cores.

Considering this dilemma, we propose an approach which tries to make the best of both ways : keeping the relative simplicity of programming with OpenMP directives and developing MPI applications ready to be tuned to run efficiently on distributed memory architectures. Our goal is to propose a MPI implementation of OpenMP directives specifically well-suited to express coarse-grain parallelism. The program written by the programmer remains generic in the sense that it can be either directly compiled by an openMP compiler to be run on some shared-memory architecture, or transformed into an MPI source code that can be further optimised by an expert in order to make the best of non shared-memory architectures such as clusters. In this paper, we propose a tool called STEP (*Système de Transformation pour l'Exécution Parallèle*) that, being based on the source-to-source transformation PIPS workbench [AAC+94], semi-automatically generates MPI programs for some widely-used coarse-grain parallel application structures.

The paper is organised as follows : the next section describes the context and gives a typical use case to emphasize the motivations of our work. In section 3 we describe the prototype we have developed and the results. In section 4, we describe the related work before conclusion.

## 2 Context

To benefit from distributed architectures, many applications need a coarse grain parallelisation of their programs. Therefore the goal of our work is two-fold:

– find a way to express the potential parallelism of an application that can be derived into both shared and distributed memory parallelisation;
– find a way to parallelise legacy applications without preventing evolutions.

### 2.1 The goals of our work

Expressing parallelism within an application without corrupting it is the main advantage of the OpenMP paradigm. Using techniques developed in the field of automatic parallelisation, we conceived STEP, a tool that transforms a program semi-automatically into an MPI source program and produces MPI code close to hand-written MPI code. The programmer adds OpenMP directives and then the tool generates a MPI program automatically. The collaboration between the user and the tool provides a semi-automatic parallelisation.

*Semi-automatic parallelisation.* One of our major goals in this work is to determine to what extent a compiler can help the programmer in developing MPI programs. On the one hand, coarse-grain parallelism is difficult to extract automatically. Automatic parallelism is able to extract fine-grain parallelism automatically. However these techniques could never be used to extract coarser-grain parallelism. There are multiple reasons for this as for example the coarser the grain is, the stronger the analyses must be dealing with inter-procedural analyses, irregular code... Furthermore when dealing with more statements, parallelism may logically exist but can be prevented because of the way the program is written: false dependency because of the use of variables... On the other hand, programmers of the legacy applications have a logical view of the global structure of their applications and thus can easily add simple directives to specify which tasks could be done in parallel.

Combining intelligence of the programmer with techniques of the automatic parallelisation field gives an opportunity to use powerful program transformation tools to discharge the programmer from the parallelising task.

*A subset of the OpenMP standard.* Another goal of our work is to facilitate parallel programming especially for the programmer of legacy applications who is not a parallel programming specialist. OpenMP programming is relatively straightforward if we stand to main work sharing directives such as “parallel for” and “sections”. Nevertheless taking into account all possible directives, clauses and SPMD programming with runtime routines, OpenMP programming can become as tricky as MPI programming and suffer from the same disadvantages as MPI, that is to say, produce code difficult to read and maintain, and error-prone. Firstly, the idea is to restrict our MPI transformation to coarse-grain parallelism to be able to generate efficient code for distributed-memory architectures. Secondly, some clauses are difficult to use and could be determined by analysing tools. For instance, “private” and “shared” data inside a parallel construct can in some cases be determined from data dependency analyses that could also detect potential problems. Alternatively, when inserted by the programmer, those clauses could be a support to analyses that fail without them.

*Generation of source code.* Parallel programmers seek performance and for that sake, don’t necessarily trust black-boxes. Generating source code provides the user a way to understand the parallel program and furthermore gives him an opportunity to tune the generated code to improve its performance. In addition to that, the generated parallel code should propose a generic structure to allow the user to replace part of the generated code with his own code. Thus one important goal of STEP is to provide source-to-source transformation. This has also the advantage, in the end, that the written code makes no hypothesis about the target architecture.

### 2.2 An illustrative use case

It is often critical in a collaborative scientific context to easily derive parallel versions of an application which are dedicated to different types of architectures. Here we give the example of a real application which gives a flavour of applications eligible for the STEP processing. It is an application implementing a physical optics algorithm that computes the radiation patterns of non-planar aperture antennas over a range of observation angles [BL05].

The target application is composed of two disctinct phases: *computation*1 and *computation*2. Data dependencies imply that the input of the first computation phase array *array*1 is read entirely by all parallel tasks so it should be shared by all these tasks; the first computation can be processed independently on all parallel tasks; and the output of this first phase, *array*2, can then be partitioned on parallel tasks. This pattern applies to the second computation too: *array*2 should globally be shared by all parallel tasks while the output of this second phase, *array*3, can be distributed on parallel tasks.

In the shared-memory execution, the fork-join model is used. Arrays *array*1, *array*2 and *array*3 are shared. There is no need for any explicit synchronisation since there is no write conflict. A barrier is used to synchronise the two computational phases. In the distributed-memory execution, the SPMD model is used. Array *array*1 is duplicated for every process. Array *array*2 resulting from the first computation phase is distributed among the processes and then an all-to-all exchange provides necessary data for the second computation phase to all processes. The resulting array *array*3 is distributed on all processes and can be gathered on one process if necessary.

Given these execution organisations, corresponding OpenMP and MPI parallel programs can be represented as in listings 1.1 and 1.2 respectively.

**Listing 1.1.** OpenMP implementation

```
real(N) :: array1, array3
real(N) :: array2

initialisation()

!$OMP PARALLEL DO
do i=1,N
   array2(i)= computation1(array1)
end do
!$OMP END PARALLEL DO

!$OMP PARALLEL DO
do i=1,N
   array3(i)= computation2(array2)
end do
!$OMP END PARALLEL DO
```

**Listing 1.2.** MPI implementation

```
real(N) :: array1, array3
real(M) :: array2

initialisation()

TASK_PARTITIONING()
DETERMINATION_OF_LOOP_INDICES()

do i=istart,iend
   array2(i)= computation1(array1)
end do

ALL_TO_ALL_DATA_REDISTRIBUTION()
DETERMINATION_OF_LOOP_INDICES()

do i=istart2,iend2
   array3(i)= computation2(array2)
end do

DATA_GATHERING()
```

The OpenMP standard guarantees that:

- each thread will access shared arrays *array*1, *array*2 and *array*3;
- $i$ is private to a thread;
- there is an implicit data flush and an implicit barrier at the end of the *PARALLEL DO* constructs.

The MPI program is more complicated to develop. Data and tasks must be explicitly partitioned depending on the number of processes and on process identifiers. Temporary buffers might have to be allocated. Communications must be handled and may involve personalised all-to-all communication. Furthermore it makes the code much more intricate and difficult to read. Both OpenMP and MPI versions being necessary, a semi-automatic tool able to derive them from the sequential program is clearly welcome.

## 3 Description of the STEP prototype

STEP has been developed in the PIPS workbench detailed below. The PIPS project (Inter-procedural Parallelisation of Scientific Programs) [AAC+94] has been developed at ENSMP/CRI since 1988 and is distributed under the term of the GNU public license. PIPS has been used as support to develop

new analyses and program transformations for Fortran programs by several teams from CEA-DAM, Southampton University, SRU, ENST Bretagne and ENS Cachan. Nowadays PIPS is composed of more than 200 000 lines of C code. Since it was first developed for automatic parallelisation, PIPS provides powerful code transformations and analyses that can be used on real codes: source-to-source code based generation, inter-procedural analyses and the array regions analysis.

Using the PIPS workbench, the STEP tool implements the three following phases:

1. the outlining phase that consists in the restructuration of the sequential program by outlining statements in parallel sections,
2. the analysis phase that computes SEND array regions based on PIPS array regions analyses,
3. and the compilation phase that generates the MPI parallel code.

### 3.1 Parallel execution model

To limit changes in the sequential program, the statements delimited by directives are outlined, communications and work-scheduling take place in a new generated subroutine. Data are allocated on all processes and at the end of parallel constructs, each process communicates to other processes the data which may be used in the future. All processes redundantly execute serial regions and iterations of OpenMP "parallel do" loops are partitioned between processes.

### 3.2 PIPS array regions analysis

In PIPS, array regions are represented by convex polyhedra [Cre96]. They are used to summarise accesses to array elements. Due to region representation, the analyses are not necessarily exact and some regions can be over-approximated. Four different types of array regions are computed. The $READ$ and $WRITE$ regions represent the effects of statements and procedures on sets of array elements. However, $READ$ and $WRITE$ regions can not represent array data flow and they are not sufficient for advanced optimisations such as array privatisation. $IN$ and $OUT$ regions have been introduced for that purpose. For a block of statements or a procedure, an $IN$ region is the subset of the corresponding $READ$ region containing the array elements that are imported (i.e. read before being written in the block) and an $OUT$ region contains the exported array elements (i.e. elements assigned in the block before being potentially read outside it).

*Program example for array regions analysis.* The program given in the listing 1.3 is an example of a program $P1$ composed of a DO loop labelled 20 that is parallelised with OpenMP directives. Inside the loop, the computation subroutine $F1$ is called.

**Listing 1.3.** Fortran program for array regions analysis.

```
      PROGRAM P1
      INTEGER I,N,F1
      PARAMETER (N=10)
      INTEGER T(N,2),A(N-1)
       ...
!$OMP PARALLEL DO
      DO 20 I = 1, N-1
         A(I) = F1(T, I)
20    CONTINUE
!$OMP END PARALLEL DO
       ...
      END

      INTEGER FUNCTION F1(T,J)
      INTEGER N,J
      PARAMETER (N=10)
      INTEGER T(N,2)
```

```
      IF (MOD(J, 2).EQ.0) THEN
         F1 = T(J+1,1)-T(J,1)
      ELSE
         F1 = T(J+1,2)-T(J,2)
      ENDIF

      END
```

*Array region analysis on this example.* Based on the previous example, PIPS performs array regions analysis at every statement level of the abstract syntax tree. For example, a result of such an analysis can be seen in the listing 1.4, line 33 :

```
<T(PHI1,PHI2)-R-EXACT-{PHI2==1, J<=PHI1, PHI1<=1+J}>
```

It means that at the statement level of the assignment (see listing 1.4, line 35) the $READ$ region represented by $R$ of the array $T$ is exactly the sub-array $T(PHI1, PHI2)$ with : $J <= PHI1 <= 1 + J$ and $PHI2 == 1$.

Regions $READ$, $WRITE$, $IN$ and $OUT$ (see subsection 3.2) are tagged respectively $R$, $W$, $IN$ and $OUT$ by the pretty-printer. An over-approximated array region is tagged $MAY$; the tag $EXACT$ refers to a non-approximated array-region (under-approximation are not computed and fixed at $\emptyset$). A such approximation is shown in the listing 1.4, line 30 where the two exact regions line 33 and 37 are cumulated at the level of the IF statement line 32. These array region analyses are performed at each statement level for intra-procedural analysis, as well as gathered at function level for inter-procedural analysis (see line 24 and line 13 at the call statement).

A new source file is generated by the PIPS pretty-printer that displays the results of $READ$, $WRITE$, $IN$ and $OUT$ array regions analyses (listing 1.4).

**Listing 1.4.** Resulting $READ$ / $WRITE$ / $IN$ and $OUT$ array regions displayed in the program.

```
1       PROGRAM P1
2       INTEGER I,N,F1
3       PARAMETER (N=10)
4       INTEGER T(N,2),A(N-1)
5        ...
6  C   <A(PHI1)-W-EXACT-{1<=PHI1, PHI1<=9}>
7  C   <T(PHI1,PHI2)-R-MAY-{1<=PHI1, PHI1<=10, 1<=PHI2, PHI2<=2}>
8  C   <T(PHI1,PHI2)-IN-MAY-{1<=PHI1, PHI1<=10, 1<=PHI2, PHI2<=2}>
9  C   <A(PHI1)-OUT-EXACT-{1<=PHI1, PHI1<=9}>
10 !$OMP PARALLEL DO
11      DO 20 I = 1, N-1
12 C   <A(PHI1)-W-EXACT-{PHI1==I, 1<=I, I<=9}>
13 C   <T(PHI1,PHI2)-R-MAY-{I<=PHI1, 1<=PHI1, PHI1<=1+I, PHI1<=10,
14 C     1<=PHI2, PHI2<=2, 1<=I, I<=9}>
15 C   <T(PHI1,PHI2)-IN-MAY-{I<=PHI1, 1<=PHI1, PHI1<=1+I, PHI1<=10,
16 C     1<=PHI2, PHI2<=2, 1<=I, I<=9}>
17 C   <A(PHI1)-OUT-EXACT-{I==PHI1, 1<=I, I<=9}>
18            A(I) = F1(T, I)
19 20      CONTINUE
20 !$OMP END PARALLEL DO
21       ...
22       END
23
24 C   <T(PHI1,PHI2)-R-MAY-{J<=PHI1, PHI1<=1+J, 1<=PHI2, PHI2<=2}>
25 C   <T(PHI1,PHI2)-IN-MAY-{J<=PHI1, PHI1<=1+J, 1<=PHI2, PHI2<=2}>
26       INTEGER FUNCTION F1(T,J)
27       INTEGER N,J
```

```
28        PARAMETER (N=10)
29        INTEGER T(N,2)
30  C   <T(PHI1,PHI2)-R-MAY-{J<=PHI1, PHI1<=1+J, 1<=PHI2, PHI2<=2}>
31  C   <T(PHI1,PHI2)-IN-MAY-{J<=PHI1, PHI1<=1+J, 1<=PHI2, PHI2<=2}>
32        IF (MOD(J, 2).EQ.0) THEN
33  C   <T(PHI1,PHI2)-R-EXACT-{PHI2==1, J<=PHI1, PHI1<=1+J}>
34  C   <T(PHI1,PHI2)-IN-EXACT-{PHI2==1, J<=PHI1, PHI1<=1+J}>
35            F1 = T(J+1,1)-T(J,1)
36        ELSE
37  C   <T(PHI1,PHI2)-R-EXACT-{PHI2==2, J<=PHI1, PHI1<=1+J}>
38  C   <T(PHI1,PHI2)-IN-EXACT-{PHI2==2, J<=PHI1, PHI1<=1+J}>
39            F1 = T(J+1,2)-T(J,2)
40        ENDIF
41        END
```

Combining READ, WRITE, IN and OUT array region analyses helps us determine SEND array regions that represent data that need to be exchanged between processes and generate MPI code.

In the next subsections, we describe the three different steps of code transformation to produce MPI parallel code.

### 3.3 First step: outlining

To limit changes in the sequential program, the statements delimited by directives are outlined. This code transformation keeps the semantics of the original program and allows to add parallelism without alteration in the original code of sequential parts. For instance, the loop below labelled 20 in the listing 1.5 is outlined and replaced by a call to a subroutine called $P1\_DO20$ as shown the listing 1.6.

**Listing 1.5.** Before outlining

```
   PROGRAM P1
   INTEGER I,N,F1
   PARAMETER (N=10)
   INTEGER T(N,2),A(N-1)
    ...
!$OMP PARALLEL DO
   DO 20 I = 1, N-1
      A(I) = F1(T, I)
20 CONTINUE
!$OMP END PARALLEL DO
    ...
   END
```

**Listing 1.6.** After outlining

```
   PROGRAM P1
   INTEGER I,N,F1
   PARAMETER (N=10)
   INTEGER T(N,2),A(N-1)
    ...
!$OMP PARALLEL DO
   CALL P1_DO20(I, 1, N-1, N, A, T)
!$OMP END PARALLEL DO
    ...
   END

   SUBROUTINE P1_DO20(I, I_L, I_U, N, A, T)
   INTEGER I, I_L, I_U, N, F1
   INTEGER A(1:N-1), T(1:N, 1:2)
   DO 20 I = I_L, I_U
      A(I) = F1(T, I)
20 CONTINUE
   END
```

The parameters of the prototype of the new subroutine are composed of:

- loop parameters : the loop index and the loop bounds (see parameters $I$, $I\_L$ and $I\_U$ in listing 1.6),
- and variables, arrays and parameters used in the outlined statements (see parameters $N$, $A$ and $T$ in listing 1.6).

The variable and array updates between callers and callees are performed by the call by reference used in Fortran.

### 3.4 Second step: analysis

In order to generate calls to MPI primitives, data that must be exchanged are determined using PIPS array regions analysis. We distinguish four different types of array regions:

- send regions represent updated array regions which need to be sent at the end of a parallel section;
- receive regions represent data and array regions which need to be received at the beginning of a parallel section;
- private regions represent variables and arrays that could be privatised;
- used regions represent necessary memory allocation gathering READ and WRITE regions.

In our execution model (see subsection 3.1), we suppose that sequential computations are done redundantly in parallel by each process. Then at the beginning of the first parallel section, each process owns up-to-date data. At the end of the parallel section each process must send its own data to update the data of other processes. Thus in our prototype we only use the SEND regions to determine which sub-arrays must be communicated to other processes.

*Computation of the SEND array regions.* The SEND array regions for a given procedure represent updated array regions which need to be sent at the end of a parallel section thus it is included into OUT array regions. Nevertheless, due to the PIPS inter-procedural array regions analysis, the $OUT$ region of an array $A$ that is a parameter of a procedure $P$, refers to the array region of $A$ updated (and used after) by all calls to the $P$ procedure through the whole program. However the SEND region of the array $A$ corresponds to only one call to the $P$ procedure. Thus the SEND region of $A$ for procedure $P$ is the intersection between the OUT region and the WRITE region of $A$:

$$SEND(A) = OUT(A) \cap WRITE(A)$$

*Handling region overlap.* Moreover, array regions can be over-approximated. That is to say SEND regions of different processes can overlap each other. Thus it prevents us from directly exchanging sub-arrays described by the SEND regions. In case of overlap, we compute at runtime data that should be exchanged. Besides, the results of array region analyses are convex polyhedra. For simplicity sake, in our prototype, we limit ourselves to communicating "rectangular" sub-arrays. Rectangular sub-arrays are represented by polyhedra with at most one dimension of the array appearing in each constraint that delimits a SEND region. In case sub-arrays are not given rectangular by array region analysis, we make an over-approximation to come back to a rectangular case. In this case also, we must compute at runtime data that should be updated.

The following listing 1.7 shows the array regions analysis performed on the outlined subroutine $P1\_DO20$.

**Listing 1.7.** Region analyses performed on the subroutine $P1\_DO20$ ( excerpt)

```
Region write P1_DO20 : 2
<A(PHI1)-W-EXACT-{I_L<=PHI1, PHI1<=I_U}>
<I-W-EXACT-{}>
Region out P1_DO20 : 1
<A(PHI1)-W-EXACT-{1<=PHI1, PHI1<=9, I==11, I_L==1, I_U==9, N==10}>
Region Send P1_DO20 : 1
<A(PHI1)-W-EXACT-{I_L<=PHI1, 1<=PHI1, PHI1<=I_U, PHI1<=9}>
Region interlaced P1_DO20 : 0
```

### 3.5 Third step: compilation

The compilation phase consists in generating the MPI parallel code and inserting communications and work-scheduling.

For a "parallel do" directive, we create a new routine suffixed with $\_MPI$ that replaces the outlined sequential routine. For instance, in the main program, the call $P1\_DO20$ is replaced by a call to $P1\_DO20\_MPI$ (see listing 1.8 line 6 and line 26).

This new procedure is divided into four parts :

– computation of the loop partitioning: the initial loop range is split in different slices for each process;
– computation of the SEND array regions according to the loop partitioning;
– call of the outlined procedure according to the loop partitioning: before the call, each process owns necessary data for the computation of its loop slice. After the call, a process owns only the updated data in its own SEND region;
– communications of the updated data between processes : to deal with the potential overlap of send data, at the first step, we merge all the SEND regions on a master process and in the second step, the master broadcasts the global send region corresponding to the initial loop bounds.

In our generated code (see listing 1.8), calls at the MPI library are encapsulated into our own Fortran functions that deal with send and receive arrays described by our own internal region representation.

**Listing 1.8.** The generated MPI parallel source file.

```
1        PROGRAM P1
2        include "step.h"
3  C     declarations
4        CALL STEP_Init
5        ...
6        CALL P1_DO20_MPI(I, 1, N-1, N, A, T)
7        ...
8        CALL STEP_Finalize
9        END
10
11       SUBROUTINE P1_DO20_MPI(I, I_L, I_U, N, A, T)
12       include "step.h"
13 C     declarations
14
15 C     Loop splitting
16       CALL STEP_SizeRank(STEP_Size, STEP_Rank)
17       CALL STEP_SplitLoop(I_L, I_U, 1, STEP_Size, I_STEP_SLI)
18
19 C     SEND region computing
20       CALL STEP_COMP_SEND(A_STEP_SR,...)
21
22 C     Where the work is done
23       I_IND = STEP_Rank+1
24       I_LOW = I_SLI(LOWER,I_IND)
25       I_UPP = I_SLI(UPPER,I_IND)
26       CALL P1_DO20(I_IND, I_LOW, I_UPP, N, A, T)
27
28 C     SEND regions communications
29       IF (STEP_Rank.EQ.MASTER) THEN
30          CALL STEP_RecvMergeRegion(...)
31       ELSE
32          CALL STEP_SendRegion(...)
33       ENDIF
34       CALL STEP_BcastRegion(...)
35       END
```

### 3.6 Results

STEP has reached several goals specified in subsection 2.1. We have implemented a source-to-source program transformation tool that semi-automatically generates MPI programs called STEP. Currently, STEP is in its early development stage. Its input (resp. output) files are Fortran 77 programs with OpenMP directives (respectively with MPI primitives) to express the parallelism.

*Handling more OpenMP directives.* STEP handles "section" and "parallel do" work-sharing directives for which the number of available computation nodes is known at runtime. As previously said, we do not necessarily intend to handle the entire OpenMP standard. Nevertheless, STEP is currently limited and we want to extend it to several widely-used OpenMP directives for instance the `master` directive for I/O management. Furthermore the loops are currently partitioned using static scheduling assigning one loop chunk to each process. This should also be extended to other types of scheduling proposed by OpenMP. Although only one level of parallelism is currently supported, nested parallelism is planned to deal both coarse and fine grain parallelism.

At last, STEP does not handle explicit OpenMP synchronisation directives; only implicit synchronisations at the end of "parallel sections" and "parallel do" are performed. Although this limits the developer's possibilities, it reduces all the more the synchronisations between processes. Besides limiting the amount of synchronisations potentially provides a good execution performance.

*Handling data.* On the contrary to the OpenMP standard, data are private by default. It is the analysis phase and the computation of the SEND regions that determine which data are *shared*. These analyses specify which data must be transmitted/shared from one process to another at the end of a parallel region. Currently the same amount of memory is allocated on all processes. This is mainly due to Fortran 77. As this is a important limitation of the STEP prototype, we want as a short-term improvement distribute arrays when analyses are successful, for instance at least for arrays with regular access. Besides, our first STEP prototype converts polyhedric regions into rectangular regions in order to generate simple MPI communication datatypes. This could be improved to benefit from the array region representation in PIPS [Cre96].

*Communication patterns.* Currently, the SEND region analysis leads to use different auxiliary functions to deal with data exchanges :

- when the SEND regions are computed without over-approximation, our tool generates exact MPI messages to perform the exchange;
- when the SEND regions are computed with over-approximation and without interlacing, the same function performs the exchange of all the data specified by the SEND region;
- when the SEND regions are over-approximated and interlaced, another function is used to deal with the interlaced access. A runtime solution has been implemented. It consists in a first step in exchanging all data in the SEND region, and in the second step in systematically comparing original and updated values. Nevertheless, this simple solution with an important overhead allows to detect concurrent accesses.

Currently, the all-to-all data redistribution is centralised for simplicity sake. This could be further refined since array region analyses provide all the necessary information.

At last, current analyses need to be improved to deal with reductions: recognising a reduction pattern in the sequential code when no OpenMP reduction clause is present and then generating the appropriate MPI code. In case where a reduction pattern is not recognised, STEP should be able to handle the OpenMP reduction clause.

*Quality of the generated source code.* At last, the generated code is close to hand-written code. First, the original sequential code is not altered. The sequential parts that are executed in parallel can be found unmodified in outlined functions. Second, code transformations keep track of the original variable names, and proposes readable function names. At last, MPI communications are close to hand-written communications since the volume of the communicated data is given by array region analysis. Nevertheless it is already possible to use the STEP generated code, read it and modify to tune MPI communications if necessary. For this purpose, auxiliary functions provide several services, as for instance index conversion functions to convert array indices.

At last, the STEP tool is currently limited to Fortran 77 but an extension to C is considered.

## 4 Related work

"Distributed OpenMP" refers to research projects that help running an OpenMP application on distributed-memory architectures.

Historically, most distributed OpenMP projects addressed Software Distributed Shared Memory (SDSM) architectures. SDSM architectures rely on a software layer in order to manage data placement on the nodes of a distributed-memory architecture and keep memory consistency between nodes. For instance, the OMNI OpenMP compiler [SSKT99] is based on a DSM runtime system inserting check codes before each load/store to/from the shared data space. Those check codes are implemented using a communication library supporting one-sided remote memory transfer and synchronization between nodes. Several implementations of "distributed OpenMP" are based on the SDSM Treadmarks system [HLCZ00,BME02]. Those approaches suffer from very fine-grain communication patterns. In order to avoid unnecessary data checks and group them when possible, those projects have evolved to use compiler analyses intensively in order to generate efficient code. The OMNI compiler uses analyses to eliminate check codes outside parallel regions, eliminate redundant check codes and merge multiple check codes. In order to do so, the *static extent* and the *dynamic extent* are defined. The static extent corresponds to the statements lexically enclosed within an OpenMP construct. The dynamic extent includes the functions called from within the construct. By default, the compiler determines data mapping according to the scheduling of the loop which references the data. One further optimization proposed by the OMNI compiler is to optimize data transfers between two parallel regions. Using Treadmarks, Basumallik et al. [BME02] proposed optimizations such as data prefetch, barrier elimination and data privatization. Barrier elimination and data privatization both rely on strong compiler analyzes. Barrier separating two consecutive loops can be eliminated when permitted by data dependencies. Shared data with read-only access during a section can be privatized by copy-in during this section. Shared data that are exclusively accessed by the same processor can also be privatized. There is also some work related to automatic data placement and adding HPF-like directives to OpenMP to distribute data [MMS00]. The Cluster OpenMP software system proposed by Intel [Hoe06] allows OpenMP programs to run on clusters. It relies also on the DSM Treadmarks system. As can be assumed, it is mainly suitable for applications with small amount of read / write sharable data and few synchronisation. SDSM-based distributed OpenMP projects represent opposite approaches from ours since they start from the finest communication grain and then use program analyses to communicate less. Our tool STEP targets coarse-grain parallelism, thus aims at issuing MPI primitives only at specific points in the program.

Several projects generate hybrid MPI and SDSM programs from OpenMP directives. Based on the OMNI compiler, the PaRADE project [KKH03] replaces synchronization and work-sharing directives associated with small data structures by MPI explicit collective communication. The `critical` synchronization directive is translated in an `MPI_Allreduce` as well as the `atomic` directive. The `single` work-sharing directive is translated in `MPI_Bcast` whereas the `for` directive has no message-passing translation. Generating MPI, the approach also considers fine-grain communication and computation patterns while we think that MPI is best-suited for coarser-grain parallelizations. Based on the Polaris compiler, the OpenMP compiler proposed by Eigenmann *et al.* [EHK+02] generates MPI programs in case of regular data pattern and Treadmarks SDSM programs in irregular cases. Three classes of data are distinguished: *private data*, *distributed data* for simple usage pattern and *shared data* for irregular pattern. Irregular or unknown patterns are handled by the SDSM system. Regular patterns are handled with message-passing primitives (see HPF compiler techniques): first, determine send/receive pairs for data accessed but not owned by threads; then determine the intersection between the region of an array (represented by Linear Memory Access Descriptor (LMAD)) accessed in one thread and the region owned by another thread; at last determine the `overlap` using the LMAD intersection algorithm of Hoeflinger and Paek [PHP98]. Several cases of intersection between array regions accessed by processors are discussed in the paper. For a read reference, more data can be fetched (superset mode). For a write reference, the write-back must be precise otherwise it generates a failure. A non-empty `overlap` indicates necessary communications and therefore proper message-passing calls are generated. In the paper, several OpenMP extensions are mentioned: for data distribution as HPF directives, for explicit communication operations outside parallel regions with the `copy` clause and for computation distribution with the `schedule` clause to specify which thread computes which iteration and the `home` clause to specify the thread owner.

In the OpenMPI project which follows a similar objective to ours, Boku *et al.* [BSMT04] propose a programming tool for OpenMP-like incremental parallelization based on MPI scheme. They introduce specific directives to express parallelism based on domain decomposition. They focus on data and take into account neighbour communications to exchange border elements. So doing, they obtain some dedicated code which lacks genericity. These new directives can be very interesting when program analyses fail to express these specific communication patterns.

The closest approach to ours is proposed by Basumallik and Eigenmann [BE05,BME07]. They propose a source-to-source OpenMP to MPI transformation based on the Cetus infrastructure (which is the next version of Polaris). All participating processes redundantly execute serial regions and parallel regions marked by *omp master*. Iterations of OpenMP *for* loops are statically partitioned. Shared data is allocated on all processes using a producer/consumer paradigm. At the end of a parallel construct, each participating process communicates the shared data it has produced that other processes may use. This approach is based on a strong program analysis tool. As a matter of fact, the compiler constructs a control flow graph (with each vertex corresponding to a program statement) and records array access summaries with Regular Section Descriptors (RSDs) by annotating the vertices of the control flow graph. Havlak and Kennedy [HK91] compare array region representation and in particular RSD and convex regions used in PIPS. Both methods have advantages and drawbacks depending on applications. In both projects array regions are then propagated through the interprocedural analysis. In addition to READ and WRITE array regions, IN and OUT regions are computed in PIPS implying additional analysis. All in all comparing compilation, transformation and analysis tools as Cetus or PIPS is not straightforward. Nevertheless generated codes by both tools could be compared in the short term.

## 5 Conclusion and future work

Based on a solid parallel programming experience, our goal is to develop a programming environment that is a trade-off between 1) the current situation of the parallel programmer who programs the entire application by himself based on OpenMP and MPI 2) and the compilation community which has developed powerful program analyses and transformations.

The STEP tool is the first phase. Based on the PIPS workbench, STEP generates MPI source code from some OpenMP directives. Thanks to the PIPS workbench, this implementation was relatively straightforward. As a matter of fact, PIPS provided us with both a workbench for program transformation and a clearly defined internal representation as well as powerful inter-procedural array region analyses. This paper presents early achievements that we can conclude with several perspectives.

*Short-term perspectives.* This STEP tool must be completed with several technical improvements described earlier in subsection 3.6 as OpenMP, communications... The most important one is to really distribute and not allocate entire arrays since this strongly limits the scope of potential parallel applications. Furthermore this work must be completed with tests on popular benchmarks and speedup figures.

*Middle-term perspectives.* We want to take into account multilevel parallelism and generate hybrid MPI/OpenMP applications from OpenMP programs. Focusing on coarse-grain parallelisation for MPI parallel code generation, we could delegate fine-grain parallelisation to OpenMP directives for shared-memory execution on multi-core/many-core nodes for instance. In this perspective, a program analysis could compute the ratio between computation and communication and propose to distribute the data on different nodes or not.

*Long-term perspectives.* The STEP generated MPI code is close to hand-written parallel code in specific cases. We want to further focus on the interaction between STEP and the user. Our goal is to build a tool that is able to work in collaboration with the user and for instance ask for more guidance from the user in case analyses for data dependencies fail. In these "failure" cases, OpenMP programming could be refined by the user adding OpenMP clauses such as `private`, `shared`, `reduction`...

To conclude, we believe that this approach is very promising and we intend to improve this work in these three directions.

## 6 Acknowledgements

The work reported in this paper was partly supported by the European ITEA2 ParMA (Parallel programming for Multicore Architectures) Project [1].

[1] http://www.parma-itea2.org/

## References


[AAC+94] Corinne Ancourt, Batrice Apvrille, Fabien Coelho, Franois Irigoin, Pierre Jouvelot, and Ronan Keryell. PIPS — A Workbench for Interprocedural Program Analyses and Parallelization. In *Meeting on data parallel languages and compilers for portable parallel computing*, 1994.

[arb04] OpenMP architecture review board. *OpenMP Application Program Interface*, 2004.

[BE05] Ayon Basumallik and Rudolf Eigenmann. Towards Automatic Translation of OpenMP to MPI. In *Proceedings of the 19th ACM International Conference on Supercomputing (ICS)*, 2005.

[BL05] Amir Boag and Christine Letrou. Multilevel Fast Physical Optics Algorithm for Radiation From Non-Planar Apertures. *IEEE Transactions on Antennas and Propagation*, 53(6), jun 2005.

[BME02] Ayon Basumallik, Seung-Jai Min, and Rudolf Eigenmann. Towards OpenMP Execution on Software Distributed Shared Memory Systems. In *International Workshop on OpenMP: Experiences and Implementations, WOMPEI'2002*, 2002.

[BME07] Ayon Basumallik, Seung-Jai Min, and Rudolph Eigenmann. Programming Distributed Memory Systems Using OpenMP. In *12th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, 2007.

[BSMT04] T. Boku, M. Sato, M. Matsubara, and D. Takahashi. OpenMPI - OpenMP like tool for easy programming in MPI. In *Sixth European Workshop on OpenMP*, 2004.

[CCZ06] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel Programmability and the Chapel Language. *The International Journal of High Performance Computing Applications, Special Issue on High Productivity,Programming Languages and Models*, 2006.

[Cre96] Beatrice Creusillet. *Array region analyses and applications.* PhD thesis, École Nationale Supérieure des Mines de Paris, 1996.

[EHK+02] R. Eigenmann, J. Hoeflinger, R. Kuhn, D. Padua, A. Basumallik, S.-J. Min, and J. Zhu. Is OpenMP for Grids. In *NSF Next Generation Systems Program Workshop held in conjunction with IPDPS*, 2002.

[HK91] Paul Havlak and Ken Kennedy. An Implementation of Interprocedural Bounded Regular Section Analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, 1991.

[HLCZ00] Y. Charlie Hu, Honghui Lu, Alan L. Cox, and Willy Zwaenepel. OpenMP for Networks of SMPs. *Journal of Parallel and Distributed Computing*, 60(12), 2000.

[Hoe06] J.P. Hoeflinger. Extending OpenMP to Clusters. Technical report, Intel Corporation, 2006.

[KKH03] Yang-Suk Kee, Jin-Soo Kim, and Soonhoi Ha. ParADE: An OpenMP Programming Environment for SMP Cluster Systems. In *Conference on High Performance Networking and Computing*, 2003.

[MMS00] John Merlin, Douglas Miles, and Vincent Schuster. Distributed OpenMP: Extensions to OpenMP for SMP Clusters. In *Second European Workshop on OpenMP, EWOMP'2000*, 2000.

[PHP98] Y. Paek, J. Hoeflinger, and D. Padua. Simplication of Array Access Patterns for Compiler Optimizations. In *Proceedings of ACM SIGPLAN'98*, 1998.

[PIF95] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, 1995.

[SSKT99] M. Sato, S. Satoh, K. Kusano, and Y. Tanaka. Design of OpenMP Compiler for an SMP Cluster. In *Proceedings of First European Workshop on OpenMP (EWOMP)*, 1999.