From OpenMP to MPI: first experiments of the STEP source-to-source transformation tool

Daniel MILLOT [a] Alain MULLER [a] Christian PARROT [a]
Frédérique SILBER-CHAUSSUMIER [a]
[a] *Institut TELECOM, TELECOM SudParis, Computer Science Department, 91011 Évry, France* [1]

Abstract. The STEP tool allows source-to-source transformation of programs, from OpenMP to MPI, for execution on distributed-memory platforms. This paper describes tests of STEP on popular benchmarks, and analyses the results. These experiments provide both encouraging feedback and directions for improvement of the tool.

Keywords. source-to-source transformation, parallel execution, OpenMP, distributed execution of OpenMP programs, MPI

1. Introduction

In order to have good performance, HPC applications must be tuned to the target platform, be it a multicore, a cluster of multicores or a computation grid. When an end-user wants to adapt his sequential application for execution on a particular multiprocessor platform, he has to choose between two programming paradigms: shared variables or message-passing, which usually means OpenMP directives or calls to MPI communication primitives. As multiprocessor architectures nowadays tend mainly to use distributed memory, for scalability reasons, message-passing programming appears more suitable at first sight. Yet, as the distribution of both data and computation among the nodes of the target platform must be made explicit, this is reserved to expert parallel programmers. The resulting code is also generally much more intricate and fragmented, thus less easy to maintain and evolve securely, whereas OpenMP only requires high-level expression of parallelism, which does not corrupt the given code too much.

The STEP tool [1] is an attempt to reconcile the relative simplicity of OpenMP programming with the effectiveness of MPI programs that can be tuned further by experts. Based on the PIPS [2] workbench for program analysis and transformation, STEP generates MPI source code from a code annotated with OpenMP directives, thanks to the powerful inter-procedural array region analyses of PIPS. Being a source-to-source transformation tool, STEP enables the

[1] {daniel.millot, alain.muller, christian.parrot, frederique.silber-chaussumier}@it-sudparis.eu

programs of legacy applications to evolve easily and reliably without the burden of restructuring the code so as to insert calls to message passing API primitives.

This paper is organized as follows: section 2 recalls the main decisions for the design of STEP and gives some details on how it works. Section 3 then describes the experiments we have made with different scientific applications and comments on the results. Section 4 suggests some directions for improvement of the STEP tool and concludes.

2. The STEP tool

The state of the art section of the IWOMP'08 paper presenting STEP [1] describes several other attempts to provide distributed execution of OpenMP programs [3,4,5,6,7,8]. The STEP tool has been developed in the PIPS (Interprocedural Parallelisation of Scientific Programs) workbench [2], which provides powerful code transformations and analyses of Fortran codes (inter-procedural analyses, array regions analyses, etc). For a first prototype, we chose a simple parallel execution model, as shown in Figure 1: each MPI process executes the sequential code (between parallel sections) together with its share of the workshare constructs; all data are allocated for all processes and updated at the end of each workshare construct by means of communications, so as to be usable later on.

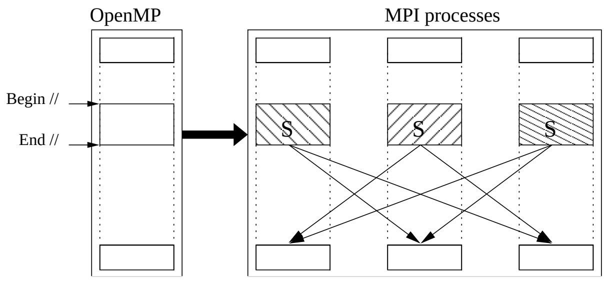

Figure 1. Execution model

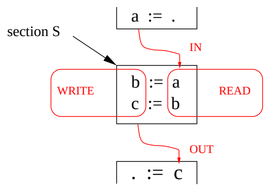

Figure 2. *READ*, *WRITE*, *IN* and *OUT* tags

The STEP OpenMP-to-MPI transformation is achieved through three successive phases: outlining (directed by OpenMP directives), computation of array regions to communicate (through region analysis) and MPI parallel code generation.

2.1. Outlining

This first phase prepares for the execution of the code of parallel sections in different contexts by the MPI processes: the code of each parallel section is replaced by a call to a procedure with the section code as its body (one procedure per such section). This code transformation keeps the semantics of the original program. For instance, the code of the loop labelled 20 in Listing 1 is outlined and replaced by a call to a new subroutine (called $P1\_DO20$ in Listing 2).

Listing 1: 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 2: 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 new procedure are composed of:

- loop parameters that is to say the loop index and the loop bounds (see parameters I, I_L and I_U in Listing 2),
- variables, arrays and parameters used in the outlined statements (see parameters N, A and T in Listing 2).

2.2. Region analysis

The purpose of this second phase is to determine the data that processes must exchange at the end of each parallel section, in order to build the corresponding MPI messages. It is based on PIPS array regions analyses.

For each array accessed in a parallel section, PIPS associates a convex polyhedron (or region) which it tags as $READ$ (resp. $WRITE$) [9] containing all the array elements which are read (resp. written) by statements of the parallel section (see $READ/WRITE$ tags in Figure 2). Furthermore, PIPS associates also an array region tagged as IN (resp. OUT) containing imported elements that are read in the parallel section after having been written previously by another code section (resp. exported elements used afterwards in the program continuation). As they are inter-procedural, PIPS analyses apply not only to the code of the parallel section under analysis, but also to procedures called in the section. The $READ$, $WRITE$, IN or OUT region tag is exact when all the elements of the region can themselves be tagged in the same way. Otherwise, the region is an over-approximation.

Combining tags, generated thanks to the previous analyses, STEP can minimize the size of the $COMM(A)$ region to be communicated between processes at the end of a given workshare construct for a given array A, in order to ensure data coherency. $COMM(A)$ is necessarily a subregion of $OUT(A)$, and should be restricted to $WRITE(A)$:

$$COMM(A) = OUT(A) \cap WRITE(A).$$

As it is an intersection of convex polyhedrons, a $COMM$ region itself is a convex polyhedron. To make the message construction simpler, we use a rectangular hull of $COMM$ regions as the message payload.

When at the end of a parallel section a process receives an array region which it has not altered at all during the execution of the section, the region is copied into the array straightforwardly. Otherwise, a comparative scan of both a resident copy and the received region of the array allows to decide which part of the data in the received message should be copied into the array. This time consuming operation must be done at runtime and reduces performance.

2.3. MPI code generation

Using the source code resulting from the outlining phase and the $COMM$ regions provided by the analysis phase, STEP finally generates MPI code. For instance, a new procedure (with a name suffixed by $\_MPI$) is created to replace the procedure generated from a "parallel do" directive when outlining. Listing 3 shows that the call to procedure $P1\_DO20$ of Listing 2 is replaced by a call to procedure $P1\_DO20\_MPI$. The new procedure successively:

- splits the loop, in order to distribute work between the MPI processes according to their rank (lines 15 to 17);
- determines $COMM$ regions for the different arrays accessed in the loop, according to the loop splitting (line 20);
- calls the initially outlined procedure with bounds adapted to the loop slice of each process (lines 23 to 26);
- exchanges $COMM$ regions with other processes (line 29).

Listing 3: MPI source file generated by STEP

```
 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_SLICES)
18
19  C     SEND region computing
20        CALL STEP_Compute_Regions(A_STEP_SR, I_STEP_SLICES,...)
21
22  C     Where the work is done
23        I_IND = STEP_Rank+1
24        I_LOW = I_STEP_SLICES(LOWER,I_IND)
25        I_UP = I_STEP_SLICES(UPPER,I_IND)
26        CALL P1_DO20(I_IND, I_LOW, I_UP, N, A, T)
27
28  C     Regions communications
29        CALL STEP_AllToAllRegion(A_STEP_SR, STEP_Rank, ...)
30        END
```

3. Using STEP on scientific computations

In this section, we give results of experiments conducted in order to identify the conditions for a relevant usage of the STEP tool and point up its benefits and drawbacks. Different types of executions are compared for a collection of applications. They are denoted as follows:

- *SER* is execution of the serial code;
- *OMP* is execution of an OpenMP version of the code;
- *STEP* is execution of the code generated by STEP from the OpenMP version;
- *KMP* is execution of the OpenMP version through usage of the Intel Cluster OpenMP suite [10]. Cluster OpenMP is based upon a software layer implementing a distributed shared memory (DSM) which allows an OpenMP application to run on a cluster of multiprocessors (without any modification in the case of a Fortran code, which is our case);
- *MPI* is execution of a hand-written MPI version of the code.

The performance of the execution denoted *STEP* is compared with executions *SER* and *OMP* systematically, with *MPI* when available, and with *KMP* when this can give workable results.

Some of the applications have been selected because they are reference benchmarks while others have because they illustrate properties of STEP. First we have chosen two reference NAS benchmarks [11]: *FT* solves the PDE system of the 3D heat equation by using FFT; *MG* solves the Poisson equation with periodic boundary conditions by a multigrid method. These two applications differ in their communication patterns: all computing nodes need to communicate in FT whereas only some need to in MG. *SER*, *OMP* and *MPI* are applicable for both benchmarks. We selected also the molecular dynamics simulation application *MD*, for which an OpenMP version is available [12], because of its high computation/communication (comp/comm) ratio. Finally we chose the standard matrix product of square matrices *Matmul* which allows to adjust the comp/comm ratio.

The execution platform was a cluster with 20 GB/s Infiniband interconnect and 14 nodes available, with 48 GB RAM memory each and 4 Xeon-1.6 GHz quadcores (thus making it 16 cores per node). For the distributed-memory executions (*MPI*, *KMP* and *STEP*), only one core per node is used, other cores being inactive. For shared-memory executions (*OMP*), the applications run on cores of the same node. For optimal execution of the NAS benchmarks (*MG* and *FT*), the number of computing nodes we used was a power of 2, which meant respectively two, four and eight nodes on our target cluster. For both benchmarks, *KMP* did not succeed (dead lock for one and false results for the other) and is thus not compared to the other executions. *KMP* also produced large standard deviations of the execution time for benchmarks for which it could compute correct results, probably due to some cache effect of the DSM system. The execution time for each execution type is the average of the execution times over ten runs.

An in depth study of the code generated by STEP for all the selected benchmarks shows that the array region analysis optimizes the size of the MPI messages sufficiently (no over-approximation) to avoid any time consuming comparative scan at runtime, a benefit of minimising the size of messages.

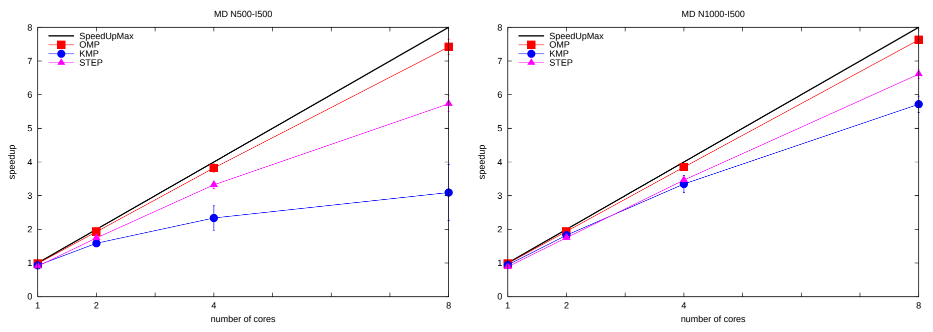

Figure 3. Speedup for MD

Figure 3 shows that *STEP* is very efficient for *MD* which has a high comp/comm ratio. In contrast to *MD*, for both *MG* and *FT*, although the MPI and OpenMP expressions of parallelism are rather similar for the former and different for the latter, the results of *STEP* are worse than those of *SER*. The performance analysis tool Vampir [13] has shown how small the comp/comm ratio of these benchmarks is for *STEP*, which is the reason for this result.

In the case of *MG*, the volume of data communicated could be reduced, thus the comp/comm ratio increased. Indeed, with the current execution model, data can be tagged $COMM$ and communicated but not accessed by the next code section; worse, it can even be communicated many times uselessly, i.e. without being used. To get around this problem, STEP could use another execution model with "producer" and "consumer" parallel sections being tied more closely. Such a model would require a more in depth array region analysis, not only computing $OUT(A) \cap WRITE(A) = COMM_O(A)$, but also $IN(A) \cap READ(A) = COMM_I(A)$ to determine whether a section consumes data from array A or not (a code section producing $COMM_O$ regions and consuming $COMM_I$ ones). This would relax synchronisation of memory updates, delaying communication of data until required, and allow overlap of communication by computation. It could also be effective for *FT*, with a slight simplification of the given source code in order to ease the work of STEP.

The influence of the comp/comm ratio over the efficiency of *STEP* can be assessed by means of *Matmul*, the ratio increasing with the matrix size. Figure 4 shows that the speedup increases faster with the comp/comm ratio for *STEP* and *KMP* than for *OMP*. The *STEP* speedup is also better than the *KMP* one for the smallest value of the comp/comm ratio (N=1000). Fewer memory pages means more competition for access when the number of client processes of the Cluster OpenMP DSM system increases.

All these experiments confirm that STEP is relevant for applications with high comp/comm ratio. Of course, STEP performance cannot compete with those of MPI code written by an expert MPI programmer. On the contrary, the use of STEP has been shown to lead to interesting speedups, compared to that of OMP or KMP, for a certain class of applications.

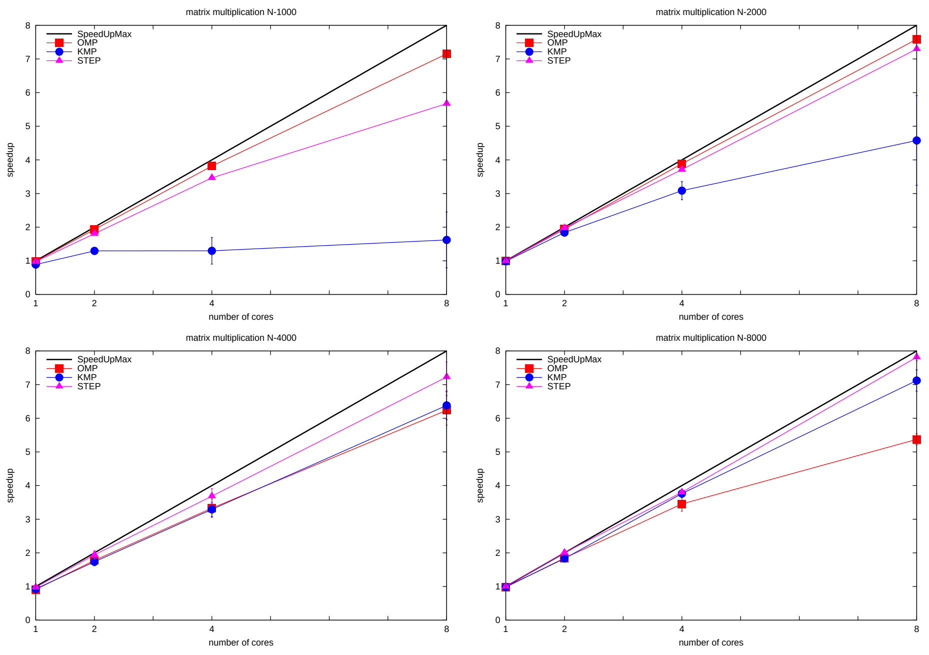

Figure 4. Speedup for matrix multiplication

4. Conclusion and future work

We have presented the source-to-source transformation tool called STEP which generates MPI source code, from a program annotated with OpenMP directives. We have also presented the results of experiments with STEP on typical scientific applications. STEP relies on PIPS inter-procedural region analyses, the quality of which allows STEP to reduce both the volume of data that need to be communicated and the extent of time consuming scan at runtime when updating arrays at the end of parallel sections. The experiments have shown that STEP is relevant for applications with a high enough comp/comm ratio and given some hints for future work.

Namely, the current execution model of STEP requires systematic data updates, that may be very expensive for some applications. As illustrated by *MD*, a high comp/comm ratio can compensate for these updates and lead to interesting speedups. The execution model of STEP could then evolve as suggested in the previous section with more complete region analyses in order to optimize the volume of communicated data.

Besides, in the short term, we plan to allow STEP to deal with multilevel parallelism and generate hybrid MPI/OpenMP code. Coarse-grain parallelism could be dealt with by means of calls to MPI primitives, and exploited across distributed memory nodes, whereas fine-grain parallelism could be expressed through OpenMP directives, and exploited within each multicore node. The resulting parallel code should be relevant for clusters of multicores. As PIPS analyses could compute the comp/comm ratio for each parallel section, STEP could decide the best execution mode for a parallel section, either distributed or shared memory.

Since it simplifies the production of a parallel code adapted to distributed memory architectures, STEP can be used by the designer of an application straightforwardly. Therefore, an objective of STEP in the long term is to build on the designer's expertise to guide STEP in certain decisions, in order to improve the quality of the source code it produces.

5. Acknowledgements

The research presented in this paper has partially been supported by the French Ministry for Economy, Industry and Employment (DGCIS) through the ITEA2 project "ParMA" [14] (No 06015, June 2007 – May 2010).

References

[1] D. Millot, A. Muller, C. Parrot, and F. Silber-Chaussumier. STEP: a distributed OpenMP for coarse-grain parallelism tool. In *Proc. International Workshop on OpenMP '08, OpenMP in a New Era of Parallelism, Purdue University, USA, IWOMP*, 2008.

[2] C. Ancourt, B. Apvrille, F. Coelho, F. Irigoin, P. Jouvelot, and R. Keryell. PIPS — A Workbench for Interprocedural Program Analyses and Parallelization. In *Meeting on data parallel languages and compilers for portable parallel computing*, 1994.

[3] 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.

[4] A. Basumallik, S.J. Min, and R. Eigenmann. Towards OpenMP Execution on Software Distributed Shared Memory Systems. In *International Workshop on OpenMP: Experiences and Implementations, WOMPEI'2002*, 2002.

[5] R. Eigenmann, J. Hoeflinger, R. H. 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.

[6] Y.S. Kee, J.S. Kim, and S. Ha. ParADE: An OpenMP Programming Environment for SMP Cluster Systems. In *Conference on High Performance Networking and Computing*, 2003.

[7] 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.

[8] A. Basumallik, S.J. Min, and R. Eigenmann. Programming Distributed Memory Systems Using OpenMP. In *12th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, 2007.

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

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

[11] NAS parallel benchmarks. http://www.nas.nasa.gov/Resources/Software/npb.html.

[12] B. Magro. OpenMP samples. Kuck and Associates Inc. (KAI), http://www.openmp.org/samples/md.f.

[13] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel. The vampir performance analysis tool-set. In *Tools for High Performance Computing*, 2008.

[14] ParMA: Parallel Programming for Multi-core Architectures. ITEA2 Project (06015) http://www.parma-itea2.org.