OMP2MPI: Automatic MPI code generation from OpenMP programs

Albert Saà-Garriga
Universitat Autonòma de
Barcelona
Edifici Q,Campus de la UAB
Bellaterra, Spain
albert.saa@uab.cat

David Castells-Rufas
Universitat Autonòma de
Barcelona
Edifici Q,Campus de la UAB
Bellaterra, Spain
david.castells@uab.cat

Jordi Carrabina
Universitat Autonòma de
Barcelona
Edifici Q,Campus de la UAB
Bellaterra, Spain
jordi.carrabina@uab.cat

ABSTRACT

In this paper, we present OMP2MPI a tool that generates automatically MPI source code from OpenMP. With this transformation the original program can be adapted to be able to exploit a larger number of processors by surpassing the limits of the node level on large HPC clusters. The transformation can also be useful to adapt the source code to execute in distributed memory many-cores with message passing support. In addition, the resulting MPI code can be used as an starting point that still can be further optimized by software engineers. The transformation process is focused on detecting OpenMP parallel loops and distributing them in a master/worker pattern. A set of micro-benchmarks have been used to verify the correctness of the the transformation and to measure the resulting performance. Surprisingly not only the automatically generated code is correct by construction, but also it often performs faster even when executed with MPI.

Categories and Subject Descriptors

D.3.2 [**Language Classifications**]: Concurrent, distributed, and parallel languages; D.3.4 [**Processors**]: Translator writing systems and compiler generators

General Terms

Parallel Computing

Keywords

Source to Source Compiler, Shared Memory, MPI, Parallel Computing, Program Understanding, Compiler Optimization

1. INTRODUCTION

One of the strengths of the OpenMP paradigm is the simplicity of its programming model. In it, the invocation of communication primitives are hidden from the programmer as they are implicitly introduced by compilation directives working in conjunction with the OpenMP run time. However, its use is usually limited to shared memory systems. Large HPC systems (like ones in top500 list) are often created by replicating nodes that contain some memory and a number of sockets with multicore processors or accelerators that can access the memory on the node. Memory on remote nodes is not usually visible in the address space on applications running in one node. This makes OpenMP limited to the node domain, making OpenMP applications difficult to scale to a larger number of nodes (and cores) without introducing other paradigms like MPI.

There are runtimes that can overcome this limitation usually by implementing Software Distributed Shared Memory, but they are also transparent to the programmer and, consequently, do not allow any fine tuning that could be needed to better adapt to the potential different contexts. Moreover, they cannot be generally applicable to all distributed memory platforms.

On contrast, MPI is a de facto standard commonly used for big HPC applications. In it, the communication primitives must be explicitly coded. Introducing the communication primitives to implement the cooperation patterns makes the code larger and more difficult to read and understand. Obviously, it is more complex to learn since there are a large number of functions including point to point communication primitives as well as collective communication primitives. This coding effort is justified if it is needed to execute on thousands of cores. MPI allows to communicate among cores on different nodes, and one could think that it introduces performance overheads at the node level compared with OpenMP. But this is a controversial issue with no clear answer as shown in [14, 6].

We advocate for a different approach that would let programmers use OpenMP to express the parallelism in their application while automatically generating a MPI equivalent program that can be executed in a distributed memory (DM) machine. A new tool (OMP2MPI) has been developed which transforms OpenMP source code into MPI source code. The resulting code is valid by construction, and can be executed in different kinds of DM systems, like large HPC clusters or distributed memory experimental processors like Intel Polaris, Ambric, or experimental FPGA based multi-soft-cores (like [10]). Another potential use is to test if there is any performance gain by using MPI on an application on the

same shared memory platform.

The paper is organized as follows, in Section 2 there is a review of the related work, in Section 3 compiler transformations done to translate from OpenMP to MPI, the following section present the performance obtained by several automatically created MPI codes from the Polybench benchmark [18]. and finally, in Section 5 concludes with an explanation of the obtained results and future tool improvements.

2. RELATED WORK

Many source-to-source compiler alternatives have been proposed to the MPI programming complexity, the standard idea is to reuse codes implemented in OpenMP to generate solutions that can be executed using distribute memory architectures.

Most of the existing projects dedicated to the use of OpenMP codes for distributed memory architectures rely on the use of the software layer to manage data placements on nodes (Software Distributed Shared Memory Architectures). An example of these is OMNI OpenMP[22] and his optimization proposed in [5, 23], are one way to support OpenMP in a distributed memory environment using a software distributed shared memory system (SDSM) as an underlying run-time system for OpenMP. Cluster-enabled OMNI OpenMP on SCASH is an implementation of OMNI OpenMP compiler for a software distributed shared memory system SCASH running under SCore Cluster System Software. Another important software system to mark is Cluster OpenMP proposed by Intel[12], that one, as in the aforementioned, allow the use of OpenMP programs to run in clusters, even that was discontinued few years ago. All these solutions, based on software layer, can be used on distributed architectures, without use Message Passing Interface but need some kind of runtime. In contradistinction, OMP2MPI shows the generated solution that will be executed on cluster to the programmer, an this could be optimized, if needed, by an expert offering more flexibility on how will be the code executed in cluster.

More similar ways to port OpenMP programs to Clusters are proposed in PaRADE[13] or based on OMNI compiler, [9], based in Polaris. Both combines the software layer management of data with the use of MPI primitives.

In [8, 11, 19], authors propose to extend OpenMP with additional clauses necessary for streamization as in our tool. Nevertheless, the most similar tools are proposed in[4, 5] and [16]. Both, are source-to-source compilers as our tool, the first based on Cetus[7] and the second on PIPS[1] generating solutions that could be compared to ours.

OMP2MPI is based on Mercurium Framework since it supports C/C++ source codes and gives an intermediate representation more friendly to work than the other existing frameworks as LLVM [15], PIPS , Cetus or ROSE [20]. And have a well documented API that allows to extend that one.

3. OMP2MPI COMPILER

OMP2MPI is a Source to Source compiler (S2S) based on BSCs Mercurium framework [17] that generates MPI code from OpenMP. Mercurium [3] gives us a source-to-source

compilation infrastructure aimed at fast prototyping and supports C and C++ languages. This platform is mainly used in the Nanos environment to implement OpenMP but since it is quite extensible it has been used to implement other programming models or compiler transformations as has been demonstrated in[21], providing OMP2MPI with an abstract representation of the input source code: the Abstract Syntax Tree(AST). AST provides an easy access to source code structure representation, the table of symbols and the context of these.

The specialization of Mercurium for OMP2MPI compiler is achieved using a plugin architecture, where plugins represent several phases of the compiler. These plugins are written in C++ and dynamically loaded by the compiler according to the selected configuration. Code transformations are implemented to the source code which implies that there is no need to know or modify the internal syntactic representation of the compiler.

Figure 2 shows a simplified process flow of OMP2MPI compiler, where an OpenMP input code is transformed in an MPI code by the use and analysis of the AST. OMP2MPI detect and transform OpenMP blocks (focused in *#pragma omp parallel for*), dividing the task in MPI master and slave processes that will be distributed on the available cores. To determine the OpenMP blocks that have to be transformed OMP2MPI use the directives proposed in [2], as is illustrated in the input code example shown in Table 1.

The proposed tool is able to use the combination of peer to peer communication functions (MPI_Send, MPI_Recv), and divide the code into sequential and parallel parts with the use of MPI ranks.

With these MPI functions OMP2MPI is able to create a correct implementation of a MPI parallel program that in the studied will result similar to an MPI hand-coded version of the original problem. OMP2MPI transforms the original code doing the MPI initialization and workload distribution based on the process rank of the calling process in the communicator. The master process with rank 0 will contain all the sequential code from the original OpenMP application and will manage the shared memory access being the responsible to keep this updated on all the slaves as is shown in Figure 1b, in contrast to the original OpenMP memory access represented in Figure 1a where all the created threads have access to shared memory. In these figures, lines in blue represent a read operation while lines in read represent a write operation.

3.1 AST Manipulation

The AST manipulation stage on Figure 2 is composed by four main steps : 1) Context Analysis, 2)Loop Analysis, 3)Workload Distribution, 4)Finalize.

3.1.1 Context Analysis

To transform the original code OMP2MPI analyze the context where the OpenMP block is originally computed, and do an accurate contextual analysis of the AST for each of the variables needed inside it. On MPI each of the executed process manage their private variables independently and the main problem to transform OpenMP to MPI is on shared

```
1   //init MPI and vars
2   const int FTAG = 0;
3   const int ATAG = 1;
4   int partSize = ((N − 0)) / (size − 1) / 10, offset;
5   if (myid == 0) {
6   int followIN = 0;
7   int killed = 0;
8   for (int to = 1;to < size;++to) {
9       MPI_Send(&followIN , 1, MPI_INT, to, ATAG, MPI_COMM_WORLD);
10      MPI_Send(&partSize , 1, MPI_INT, to, ATAG, MPI_COMM_WORLD);
11      MPI_Send(&sum[followIN], partSize, MPI_DOUBLE, to, ATAG, MPI_COMM_WORLD);
12      followIN += partSize;
13  }
14  while (1) {
15      MPI_Recv(&offset , 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
16      int source = stat.MPI_SOURCE;
17      MPI_Recv(&partSize , 1, MPI_INT, source, MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
18      MPI_Recv(&sum[offset], partSize, MPI_DOUBLE, source, MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
19      if ((followIN + partSize) < N) {
20          MPI_Send(&followIN , 1, MPI_INT, source, ATAG, MPI_COMM_WORLD);
21          MPI_Send(&partSize , 1, MPI_INT, source, ATAG, MPI_COMM_WORLD);
22          MPI_Send(&sum[followIN], partSize, MPI_DOUBLE, source, ATAG, MPI_COMM_WORLD);
23      } else if ((N − followIN) < partSize && (N − followIN) > 0) {
24              partSize = N − followIN;
25          MPI_Send(&followIN , 1, MPI_INT, source, ATAG, MPI_COMM_WORLD);
26          MPI_Send(&partSize , 1, MPI_INT, source, ATAG, MPI_COMM_WORLD);
27          MPI_Send(&sum[followIN], partSize, MPI_DOUBLE, source, ATAG, MPI_COMM_WORLD);
28      }
29      if ((followIN + partSize) > N) {
30          MPI_Send(&offset , 1, MPI_INT, source, FTAG, MPI_COMM_WORLD);
31          killed++;
32      }
33      followIN += partSize;
34      if (killed == size − 1)
35      {
36          break;
37      }
38  }
39  }
40  if (myid != 0) {
41  while (1) {
42      MPI_Recv(&offset , 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
43      if (stat.MPI_TAG == ATAG) {
44          MPI_Recv(&partSize , 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
45          MPI_Recv(&sum[offset], partSize, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
46          for (int i = offset; i < offset + partSize; ++i) {
47                  double x = (i + 0.5) * step;
48                  sum[i] = 4.0 / (1.0 + x * x);
49          }
50          MPI_Send(&offset , 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
51          MPI_Send(&partSize , 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
52          MPI_Send(&sum[offset], partSize, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
53      }
54      else if (stat.MPI_TAG == FTAG) {
55              break;
56      }
57  }
58  }
```

Table 2: Resulting piece of code from the transformation of the first OpenMP block shown in Table 1 into MPI Source Code Example that contains the calculation of an Array. In green inserted MPI funcions and created variables.

```
1   void main()
2   {
3       ...
4   #pragma omp parallel for target mpi
5       for(int i = 0; i<N; ++i) {
6               double x = (i+0.5) * step;
7               sum[i] = 4.0/(1.0+x*x);
8       }
9   #pragma omp parallel for reduction(+:total) target mpi
10      for (int j=0; j<N; ++j){
11              total += sum[j];
12      }
13      ...
14  }
```

Table 1: OpenMP blocks source code example using the created *target* clause.

variables, for this reason OMP2MPI study each of shared variables used inside an OpenMP block and analyze the AST to identify when/whether they are accessed. OMP2MPI distinguish the used variables on an OpenMP blocks into IN variables (variables that are read inside the block but without modification), OUT variables (variables that are write inside the block and the result of these are needed after the block finalization) and, INOUT variables (complains both cases). Figure 3 represents the first OpenMP block implemented in Table 1, this figure is useful to show the difference between a variable x that will be read inside the OpenMP block without any modification(in variable), an *sum* that will be write inside the OpenMP block and will be necessary to have this variable updated before the next read of this(out variable). Depending on that information MPI_Send / MPI_Recv instructions are inserted to transfer the data to the appropriate slaves.

The context analysis stage will also include the study of the context situation of the OpenMP block, as could be to detect that the OpenMP block to transform is inside a loop, in which case OMP2MPI will modify where will be inserted the initialization and the task synchronization instructions.

3.1.2   Loop Analysis
This stage is the dedicated to study the loop that is included in *pragma omp for* directive to divide correctly the computation of the for loop inner statements. OMP2MPI

```
1    double work0;
2    int j = 0;
3    partSize = ((N - 0)) / (size - 1) / 10;
4    if (myid == 0) {
5    int followIN = 0;
6    int killed = 0;
7    for (int to = 1;to < size; ++to) {
8        MPI_Send(&followIN, 1, MPI_INT, to, ATAG, MPI_COMM_WORLD);
9        MPI_Send(&partSize, 1, MPI_INT, to, ATAG, MPI_COMM_WORLD);
10       followIN += partSize;
11   }
12   while (1) {
13       MPI_Recv(&offset, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
14       int source = stat.MPI_SOURCE;
15       MPI_Recv(&partSize, 1, MPI_INT, source, MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
16       MPI_Recv(&work0, 1, MPI_DOUBLE, source, MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
17       total += work0;
18       if ((followIN + partSize) < N) {
19           MPI_Send(&followIN, 1, MPI_INT, source, ATAG, MPI_COMM_WORLD);
20           MPI_Send(&partSize, 1, MPI_INT, source, ATAG, MPI_COMM_WORLD);
21       } else if ((N - followIN) < partSize && (N - followIN) > 0) {
22               partSize = N - followIN;
23               MPI_Send(&followIN, 1, MPI_INT, source, ATAG, MPI_COMM_WORLD);
24               MPI_Send(&partSize, 1, MPI_INT, source, ATAG, MPI_COMM_WORLD);
25           }
26       if ((followIN + partSize) > N) {
27           MPI_Send(&offset, 1, MPI_INT, source, FTAG, MPI_COMM_WORLD);
28           killed++;
29       }
30       followIN += partSize;
31       if (killed == size - 1)
32       {
33           break;
34       }
35   }
36   }
37   if (myid != 0) {
38   while (1)
39   {
40       MPI_Recv(&offset, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
41       if (stat.MPI_TAG == ATAG)
42       {
43           MPI_Recv(&partSize, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
44           total = 0;
45           for (int j = offset; j < offset + partSize; ++j)
46           {
47               total += sum[j];
48           }
49           MPI_Send(&offset, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
50           MPI_Send(&partSize, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
51           MPI_Send(&total, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
52       } else if (stat.MPI_TAG == FTAG) {
53               break;
54       }
55   }
56   }
57   MPI_Finalize();
58   if (myid == 0) {
59           //non parallelized source code
60   }
```

Table 3: Resulting piece of code from the transformation of the second OpenMP block shown in Table 1 that contains the calculation of a reduced variable into MPI Source Code Example. In green inserted MPI functions and created variables
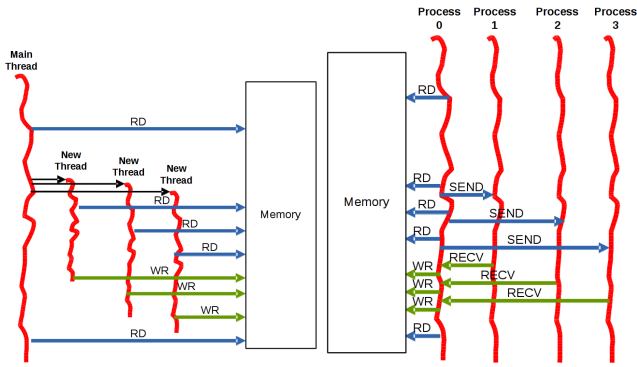
do an exhaustive analysis of the for semantics understanding and determining which is: 1) The variable iterated, 2) The variable initial value, 3) The variable final value 4) The decrement/increment after each iteration 5) The logic comparison operation. However, there are some cases in which OMP2MPI will not be able to transform loops based on the for loop semanics i.e complex not linear increments on iterator or multiple cases on condition. This cases will do that the studied blocks will not be transformed by OMP2MPI, keeping these as OpenMP blocks.

3.1.3 Workload Distribution
Having the context understanding and the proper loop semantics, OMP2MPI will divide the OpenMP block calculation to work with master/slaves MPI model, by using the producer/consumer paradigm. OMP2MPI treated all the variables studied in the context analysis stage . Figures 4 and 5 shows how OMP2MPI divide the computation for each of the OpenMP block. The iterations of the OpenMP block will be divided in a different way depending if the original OpenMP block contain in his pragma directives a

schedule clause, as static or guided. OMP2MPI divides the iterations of the for loop between all the available slaves. Figure 5 shows the division model for an OpenMP dynamic block while, 4 how an OpenMP static or guided block will be divided on master/slaves.

Using the static division the outer loop is scheduled in a round robin fashion by using MPI_Recv from specific ranks. This could lead to an unbalanced load. However, is necessary to have this kind of division because OMP2MPI is thought to be faithful with the original OpenMP code which could have this directive. On the other hand, in the case of the dynamic division, the outer loop is scheduled dynamically by using ANY_SOURCE MPI_Recv and results more efficient. Trying to overcome unbalanced load, OMP2MPI determine the range of iterations that will compute each process on execution by dividing the number of total iterations by the available slaves, and this number is finally divided by 10, as is shown in line 4 of Table 2. Table 2 illustrate an example of an array computation, OMP2MPI transforms the first OpenMP block on Table 1 into the showed source

(a) Example of a memory access pattern of an OpenMP application. Threads directly access the shared memory.

(b) Example of the proposed memory access pattern for shared variables in MPI target applications. Access to shared variables are centralized from Master node, and worker processes have to communicate with it to access them.

Figure 1: Shared memory access on different architectures(Blue lines represent a read operation. Red lines represent a write operation).
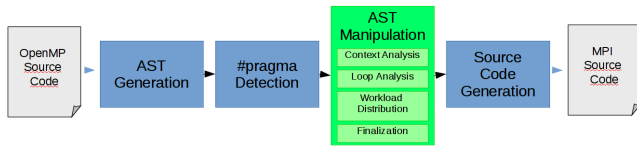

Figure 2: In blue, the functionalities already offered by the Mercurium framework. In green the AST manipulation process done by OMP2MPI.

```
void main(){
...
sum[i] = 0;
x = 3;
...
    #pragma omp parallel for target mpi
    for(int i=0;i<N;++i){

        sum[i] = 4.0/(1.0 + x*x);

    }
...
total+=sum[i];
...
}
```

Figure 3: Context example. Is possible to see that variable x is written before the parallel loop and it is not accessed after it, so it can be labeled as an IN variable. While sum is both written before but not read inside, and read after the parallel loop, so it can be labeled as OUT.

code. Two different rules to ensure that the calculation over a variable could be divided in independent executions of the original for loop are defined. In the case that the divided iterator is linear in the first dimensional access pointer in a

write operation of a variable(i.e. $var[i][j]=2*i$), MPI_Send and MPI_Recv functions will transfer to the master, just the portion of the out variable that has to be read or has been modified, from offset to the actual maximum iterator. The other studied case is when the divided iterator is not the first dimensional access pointer in a write operation but is used as that in the any of the variables on the assign operation(i.e. $var[i]=2*j$). OMP2MPI will transfer the full array but just in the case that the actual iteration is the last slave in execution.

The used workload distribution is not applicable to all the possible cases that are accepted in an OpenMP block, OMP2MPI is not able to divide the computation on variables with concurrent accesses to a shared variable, when the iterator is on second pointer of in access to that one, or when the variable is not linearly accessed.

An special case of INOUT variable is the variable that is specified as reduced variable by the OpenMP reduction clause. In this case, OMP2MPI determine the starting value of the reduced variable, depending on the reduction operation(an starting value of 0 for "+" and "-" operations, or 1 for "*" and "/") and will accumulate on the resulting variable the received results computed on slaves by the use of the operation to reduce. Table 3 show how that is preformed by OMP2MPI transforming the second block on Table 1 into the showed source code.

3.1.4 Finalization
The final stage on the AST Manipulation step is the finalization stage that is responsible to assign the remaining non MPI parallelized source code to the master node to avoid unnecessary computation, and put *MPI_Finalize* instruction before that. The resulting process is illustrated in the last lines of Table 3.
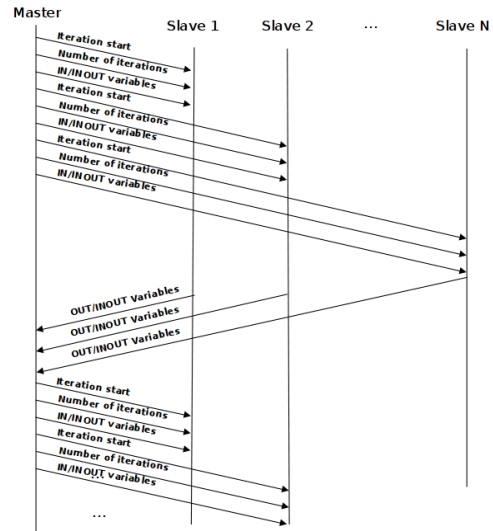

Figure 4: Workload static distribution. The work is sent in an orderly manner depending on the rank of slaves. All slaves has to finish before continue with the next piece of workload.
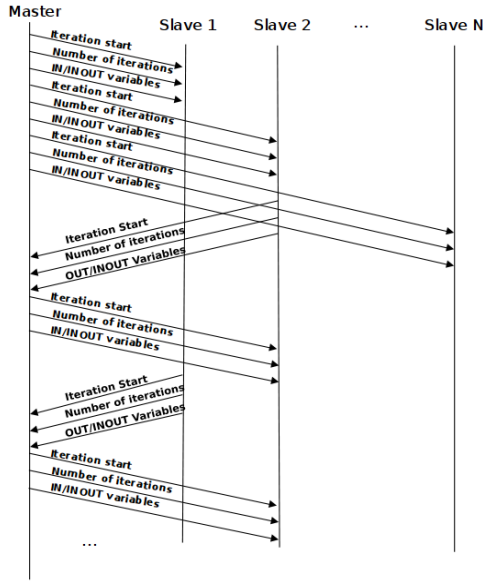
4. RESULTS

Figure 5: Workload dynamic distribution. The work is divided dynamically responding to the slave that answers with the following range of iterations and the variables needed to do the computation.

We compiled with OMP2MPI a subset of the Polybench benchmark. The generated versions were executed in 64 CPU's E7-4800 with 2.40 GHz(Bullion quadri module) and compiled with bullxmpi, compatible with MPI 2.1, enhanced by Bull with many new features such as effective abnormal pattern detection, network-aware collective operations, and multi-path network fail-over, to increase reliability, resilience and boost the performance of parallel MPI applications. We compare the codes resulting from the execution of OMP2MPI with the original OpenMP ones, and also with a sequential version of the same problem. Figure 6 shows the speed-up comparison for the selected problems. This figure shows that OMP2MPI produces good transformation of the original OpenMP code, and in most of cases the generated have better scalability than the original one with linear speed-up increment correlated with the number of processors used on execution.

5. CONCLUSIONS

We have presented OMP2MPI, a tool that facilitates the portability of an OpenMP source code to MPI, we shown how it effectively automatically translates OMP2MPI being able to go outside the node. Allowing that the program exploits non shared-memory architectures such as cluster, or NoC-based MPSoC.

This automatic task is very useful because the programmer could keep working with the OpenMP model, being easily readable and just compile over OMP2MPI compiler to take the advantages of the MPI model offer (speed-up, scalability, etc.). The readability of the code generated is acceptable so that allows further optimization by an expert intending to improve performance results. The experiments made using Polyhedral Benchmark are promising for this effortless version and produce better scalability than the original

(a) GEMM (b) 2mm
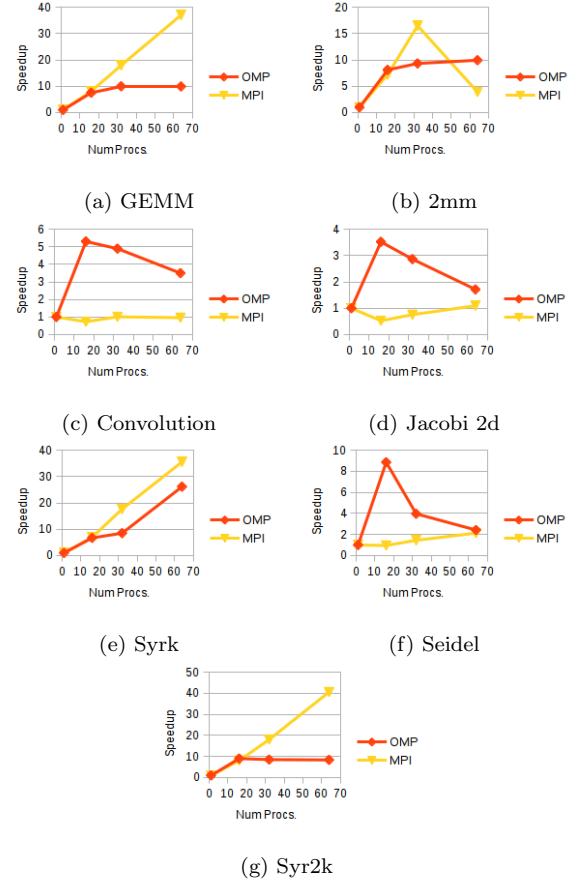(c) Convolution (d) Jacobi 2d
(e) Syrk (f) Seidel
(g) Syr2k

Figure 6: Speed-up of the tested problems by using 16, 32 and, 64 processors compared with the sequential version.

OpenMP code, Speed-up figures for 64 cores in most of cases are higher than 20× compared to the sequential version, and also higher than 4× compared to the original OpenMP code. These results show again, as mentioned in the introduction, that OpenMP does not always perform better than MPI in shared memory systems.

Future improvements on OMP2MPI will be done to include all the possible uses of shared variables inside OpenMP block and to allow the use of *target mpi* clauses on more OpenMP directives as example on *critical* sections that could be transformed by the use of *MPI_AllReduce* or *atomic* sections transformed to *MPI_Bcast*.

6. ACKNOWLEDGMENTS

7. REFERENCES

[1] C. Ancourt, F. Coelho, B. Creusillet, F. Irigoin, P. Jouvelot, and R. Keryell. Pips: A framework for building interprocedural compilers, parallelizers and

optimizers. Technical report, Technical Report A/289, Centre de Recherche en Informatique, Ecole des Mines de Paris, 1996.

[2] E. Ayguade, R. M. Badia, D. Cabrera, A. Duran, M. Gonzalez, F. Igual, D. Jimenez, J. Labarta, X. Martorell, R. Mayo, et al. A proposal to extend the openmp tasking model for heterogeneous architectures. In *Evolving OpenMP in an Age of Extreme Parallelism*, pages 154–167. Springer, 2009.

[3] J. Balart, A. Duran, M. Gonzàlez, X. Martorell, E. Ayguadé, and J. Labarta. Nanos mercurium: a research compiler for openmp. In *Proceedings of the European Workshop on OpenMP*, volume 8, 2004.

[4] A. Basumallik and R. Eigenmann. Towards automatic translation of openmp to mpi. In *Proceedings of the 19th annual international conference on Supercomputing*, pages 189–198. ACM, 2005.

[5] A. Basumallik, S.-J. Min, and R. Eigenmann. Towards openmp execution on software distributed shared memory systems. In *High Performance Computing*, pages 457–468. Springer, 2002.

[6] D. Buono, T. De Matteis, G. Mencagli, and M. Vanneschi. Optimizing message-passing on multicore architectures using hardware multi-threading. *22nd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 262–270, 2014.

[7] C. Dave, H. Bae, S.-J. Min, S. Lee, R. Eigenmann, and S. Midkiff. Cetus: A source-to-source compiler infrastructure for multicores. *Computer*, 42(12):36–42, 2009.

[8] A. Duran, J. M. Perez, E. Ayguadé, R. M. Badia, and J. Labarta. Extending the openmp tasking model to allow dependent tasks. In *OpenMP in a New Era of Parallelism*, pages 111–122. Springer, 2008.

[9] R. Eigenmann, A. Basumallik, S.-J. Min, J. Hoeflinger, R. H. Kuhn, D. Padua, and J. Zhu. Is openmp for grids? In *Parallel and Distributed Processing Symposium, International*, volume 2, pages 0171b–0171b. IEEE Computer Society, 2002.

[10] E. Fernandez-Alonso, D. Castells-Rufas, S. Risueño, J. Carrabina, and J. Joven. A noc-based multi-{soft} core with 16 cores. In *Electronics, Circuits, and Systems (ICECS), 2010 17th IEEE International Conference on*, pages 259–262. IEEE, 2010.

[11] B. R. Gaster. Streams: Emerging from a shared memory model. In *OpenMP in a New Era of Parallelism*, pages 134–145. Springer, 2008.

[12] J. P. Hoeflinger. Extending openmp to clusters. *White Paper, Intel Corporation*, 2006.

[13] Y.-S. Kee, J.-S. Kim, and S. Ha. Parade: An openmp programming environment for smp cluster systems. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 6. ACM, 2003.

[14] G. Krawezik. Performance comparison of mpi and three openmp programming styles on shared memory multiprocessors. In *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, pages 118–127. ACM, 2003.

[15] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis &

transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[16] D. Millot, A. Muller, C. Parrot, and F. Silber-Chaussumier. Step: a distributed openmp for coarse-grain parallelism tool. In *OpenMP in a New Era of Parallelism*, pages 83–99. Springer, 2008.

[17] Nanos. Mercurium, Mar. 2004.

[18] PolyBench. The polyhedral benchmark suite, Mar. 1997.

[19] A. Pop, S. Pop, H. Jagasia, J. Sjödin, and P. H. Kelly. Improving gnu compiler collection infrastructure for streamization. *Proceedings of the 2008 GCC Developers Summit*, pages 77–86, 2008.

[20] D. Quinlan. Rose: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(02n03):215–226, 2000.

[21] A. Saà-Garriga, D. Castells-Rufas, and J. Carrabina. Omp2hmpp: Hmpp source code generation from programs with pragma extensions. In *High Performance Energy Efficient Embedded Systems*. ACM, 2014.

[22] M. Sato, S. Satoh, K. Kusano, and Y. Tanaka. Design of openmp compiler for an smp cluster. In *Proc. of the 1st European Workshop on OpenMP*, pages 32–39, 1999.

[23] V. Schuster and D. Miles. Distributed openmp, extensions to openmp for smp clusters. In *Proc. of the Workshop on OpenMP Applications and Tools (WOMPAT 2000)*, 2000.