Prototyping the automatic generation of MPI code from OpenMP programs in GCC

Abdellah Kouadri-Mostefaoui, Daniel Millot, Christian Parrot, Frédérique Silber-Chaussumier

Prototyping the automatic generation of MPI code from OpenMP programs in GCC

Abdellah-Medjadji Kouadri-Mostéfaoui, Daniel Millot, Christian Parrot, and Frédérique Silber-Chaussumier

Institut TELECOM SudParis, France

Abstract. Multiprocessor architectures comprising various memory organizations and communication schemes are now widely deployed. Hence, powerful programming tools are expressly needed in order to fully exploit the computation capabilities of such architectures. Classically, a parallel program must strictly conform to a unique parallel paradigm (either shared memory or message passing). However it is clear that this approach is far from being suited to current architectures. The shared-memory paradigm is restricted to certain hardware architectures while the message-passing paradigm prevents the application from evolution. In this paper we investigated the use of GCC as a compiler to transform an OpenMP program into a MPI program. OpenMP directives are used to express parallelism in the application and to provide data dependence information to the compiler. We deeply analyze GOMP, the OpenMP implementation in GCC, providing code transformations in the GIMPLE representation and compare them to those necessary for transforming a shared-memory program (annotated with OpenMP pragmas) to a message-passing MPI-based program. In order to test some ideas, we have developed a limited prototype. The result is that, in addition to GOMP providing parser and code transformation, additional data dependency analyses are necessary to generate communications.

1 Introduction

On the one hand, HPC (High Performance Computing) 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 [arb08]. 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 keep a "global view" of the original application. Driven by the directives, the actual parallelization is delegated to a compiler. The main drawback of OpenMP is to be restricted mainly to shared-memory architectures. 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 provide a MPI implementation of OpenMP directives. This approach could also be exploited in highly specialized embedded multiprocessors systems (MPSoCs) where the ad-hoc organization of the hardware does not allow to have a global shared memory [JBW+02].

In this paper, we study the implementation of such a transformation in GCC. GCC is a tool suite from the Free Software Foundation and as such we are able to add new passes into GCC. GCC is a well-known compiler infrastructure. Thus it is widely used and a new pass in GCC could be directly used by anyone. GCC is a multi-language powerful compilation infrastructure. As such implementing the OpenMP transformation into MPI widens the potential utilization of this transformation to any application. At last, GCC provides GOMP [Nov06], a full support for the 2.5 OpenMP specifications. Therefore, we detail GOMP parsers, data structures and runtime library and compare these transformations to those necessary for a MPI transformation.

The paper is organised as follows : the next section describes the OpenMP to MPI translation in terms of execution and data management model. In section 2.2 we describe the GCC and GOMP framework. In section 4, we describe how to integrate the MPI transformation into GOMP and a first limited prototype. In section 5, we describe the related work before concluding.

2 OpenMP for distributed memory architectures

The motivation of this work is to provide the programmer with a simple programming interface based on directives to exploit parallelism in his application and to rely on a compiler to generate low-level message-passing code. The existing OpenMP standard is well-suited to express parallelism at a high level. It was first designed for loop parallelism and has been extended to express task parallelism in the 3.0 version [arb08]. The OpenMP interface targets shared-memory architectures and relies on the relaxed-consistency shared-memory model.

Particularly the standard is based on the fact that each thread can have its own temporary view of the memory. Consequently, a modification of a shared variable by a given thread is not necessarily immediately propagated to every thread of the team. In order to force the modification for every thread, the programmer has to use the `flush` directive.

Besides, the OpenMP standard is based on the fork/join execution model. Once a parallel region is encountered during the execution of the master thread, a team of parallel threads is created, and the newly created threads execute until the end of the parallel region.

2.1 OpenMP to MPI translation

Let us consider a given OpenMP program. In order to translate this program into a MPI program, both the execution and the memory models must be adapted.

Execution model. The MPI standard is based on a SPMD (*Simple Program Multiple Data*) execution model. Thus the different MPI processes start at the beginning of the program execution and they all execute the same program executable. Instructions executed by each process are determined by control structures depending on the process identifier. Compared to the fork/join execution model of OpenMP, the whole translated MPI program will then be executed by each process. Sequential parts of the original OpenMP program (outside parallel regions) will thus be redundantly executed by all MPI processes. Inside parallel regions, the programmer guarantees that the statements can be independently executed. Therefore parallel parts of the given OpenMP program (inside parallel regions) can be directly executed in parallel by all MPI processes after task partitioning. In case of loop parallelism for instance, iterations are distributed across processes as it is for threads in traditional shared-memory translation of OpenMP.

Data handling. The OpenMP standard distinguishes two different data sharing possibilities inside a parallel construct: a data item can be either shared or private. In the OpenMP standard, a private variable is defined as: "a variable whose name provides access to a different block of storage for each task". A shared variable corresponds to: "a variable whose name provides access to the same block of storage". Entering a parallel construct or a work-sharing construct, the programmer can specify data sharing attributes for variables visible in the construct. If not predetermined nor explicitly determined, variables inside a parallel construct are shared.

On the contrary, in the SPMD execution model of MPI, variables are all private since they correspond to different blocks of storage. Thus, in a distributed-memory context, positioning a variable to "`shared`" can have different meanings:

- the variable is read by each task. In this case, the variable may be privatized and initialized;
- the array variable is modified by several tasks but modifications concern distinct array elements for each task. In this case, elements may be privatized and updated by each task. At the end of the parallel construct, a global update between all tasks will be necessary;
- the variable is read and modified by each task and synchronizations are necessarily provided by the user (using synchronization directives as `flush`, `atomic`, `critical` for instance) because of the relaxed-consistency memory model of OpenMP. In this case, the variable may be privatized and each synchronization results in update communications inside the parallel construct.

The OpenMP standard provides every information needed by the compiler to generate a correct parallel distributed-memory program. Several works [Hoe06,EHK+02,BME07] provide OpenMP translation to distributed memroy, based on SDSM systems and also generating MPI. Since sequential parts are redundantly executed, when encountering a parallel section, all data elements are up-to-date. In the parallel construct, computation is partitioned and executed in parallel. At the end of the parallel construct, data update must be performed. If data accesses can be precisely determined by the analysis, then the update can be precise at compile time. In case of poor analysis results, the update may be determined at runtime with an inspector/executor scheme for instance.

2.2 Some hints to optimize the execution

As previously stated, the OpenMP standard provides every information necessary to the compiler to generate a correct parallel program. Nevertheless in the HPC (*High Performance Computing*) context, the program should be not only correct but also efficient. Several tracks can then be followed.

Considering array regions. In order to reduce the amount of communicated data, array region analyses can be used to gather detailed array element accesses. These accesses can be propagated using interprocedural analyses.

In [EHK+02] Eigenmann *et al.* first, determine send/receive pairs for data accessed but not owned by threads; then they 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 they determine the `overlap` using the LMAD intersection algorithm of Hoeflinger and Paek [PHP98]. In [BE05,BME07] 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. 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. In PIPS [AAC+94], Irigoin *et al.* use convex regions to represent access to array elements. These regions are then propagated through the interprocedural analysis. In addition to $READ$ and $WRITE$ array regions, IN and OUT regions are computed in PIPS. 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). Thus in our previous prototype based on PIPS [MMPSC08], we use the $WRITE$ and OUT regions in order to determine which data should be updated at the end of the parallel constructs.

Considering the programmer's knowledge. Automatic code generation does not necessarily provide good performances. In case of HPC, performance is particularly critical and programmers are used to deeply tune their parallel application. This is the reason why introducing new directives to guide the compiler to achieve performances can be a trade-off between hand-written only application and click-and-play automatic code generation.

In the OpenMPI project, Boku *et al.* [BSMT04] introduce specific directives to express parallelism based on domain decomposition. They focus on data and take into account neighbor communications to exchange border elements. These new directives can be very interesting when program analyses fail to express these specific communication patterns. Several works [NPP+00,BCC+00,HCL05] introduce directives to control data placement, *à la HPF*. In standard OpenMP, Labarta et al. [LAOH00] propose the `indirect` clause that tells the compiler to provide an inspector/executor schema useful in irregular applications. At last, in [CRM+07,DPA+08,Gas08,PPJ+], authors propose to extend OpenMP with additional clauses necessary for streamization as detailed in section 5.

Improving overall performance. Based on data dependency analysis, several performance improvements can be used in order to reduce the overhead of parallelism as for instance optimizing data transfers between two parallel regions, data prefetch and barrier elimination. Historically, these optimizations were used in distributed OpenMP projects based on Software Distributed Shared Memory (SDSM) architectures [SSKT99,HLCZ00,BME02,Hoe06]. 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. Check codes are inserted before each load/store to/from the shared data space. These

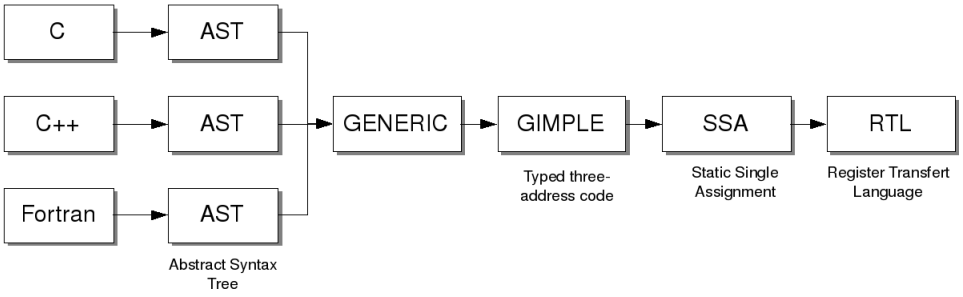

Fig. 1. GCC intermediate representations.

approaches suffer from very fine-grain communication patterns. In order to avoid unnecessary data checks and group them when possible, compiler analyses are intensively used in order to generate more efficient code: eliminate check codes outside parallel regions, eliminate redundant check codes and merge multiple check codes.

3 GCC Support for OpenMP : the GOMP project

As shown in Fig. 1, GCC implements mainly four compilation phases. The first phase (parsing) is specific for each front-end and produces an intermediate representation of the source code called GENERIC. The GENERIC representation is a language dependent tree structure where each language construct (loops, conditionals, declaration..) has a specific tree representation. At the GENERIC level, some language specific constructs may still exist. This is the reason why it is considered to be language dependent. The second phase called "gimplification" consists in building the GIMPLE trees from the GENERIC representation. GIMPLE is a language independent representation of the source. It is a simplified subset of the GENERIC representation where all the expressions are lowered into a three-address form by the introduction of new temporary variables. Later in the compilation process, GIMPLE trees are translated into SSA (*Static Single Assignment*) trees, a lower level representation suited for low level-optimization. Finally, the RTL (*Register Transfer Language*) code generation phase produces the executable code from the optimized SSA trees.

GCC provides GOMP [Nov06], a full support for the 2.5 OpenMP specifications. OpenMP pragmas are parsed by each C, C++ and Fortran frond-end and the corresponding code is translated to the appropriate intermediate format used by GCC. The code generation strategy consists in outlining the body of parallel regions into functions used as arguments to the `libgomp` thread creation routines. Data sharing is implemented by passing the address of a local structure. This transformation is performed on the GIMPLE representation. We highlight three different steps in the Figure 2.

The first step consists in the following transformations (*high* GIMPLE form): the front-end's parser produces the GENERIC intermediate representation of the code, therefore each directive and clause has a corresponding GENERIC node in the GENERIC tree (defined in the `tree.def` file); data sharing attributes are made explicit for modified variables and for implicitly determined data (`gimplify.c:omp_notice_variable()`).

The second step consists in the following transformations (*low* GIMPLE form): the code is linearized (`gimplify.c:gimplify_expr()`); `OMP` markers are inserted to determine the end of each parallel work-sharing region; a data structure containing shared data is created (`omp-low.c:scan_omp_parallel()`).

At last, the third step consists in the following transformations (*final* GIMPLE form): the body of the `omp parallel` pragma is outlined (`omp-low.c:create_omp_child_function()`); calls to `libgomp` primitives are inserted to launch threads and execute the outlined function (`omp-low.c:expand_parallel_call()`); task partitioning is performed: local loop bounds based on OpenMP thread identifiers are computed (`omp-low.c: execute_expand_omp()`).

To illustrate the different phases of the GOMP transformation process, we consider the code of a parallel sum given in Listing 1.1.

Listing 1.1. Simple parallel code: sum.

```
#include "stdio.h"
#define N 100
#define INC 2
int main()
{
  int i;
  int sum=0;
  #pragma omp parallel for reduction(+:sum)
  for (i=1;i<=N;i++)
  {
        sum+=INC;
  }
return sum;
}
```

Listing 1.2. High GIMPLE form.

```
int main() ()
{
  int D.2562;
  {
    int i;
    int sum;
    sum = 0;
    #pragma omp parallel reduction(+:sum)
      {
        {
          #pragma omp for nowait private(i)
          for (i = 1; i <= 100; i = i + 1)
            {
              sum = sum + 2;
            }
        }
      }
    D.2562 = sum;
    return D.2562;
  }
  D.2562 = 0;
  return D.2562;
}
```

Listing 1.3. Final GIMPLE form.

```
int main() ()
{
  int sum;
  int i;
  int D.2562;
  struct .omp_data_s.0 .omp_data_o.2;

<bb 2>:
  sum = 0;
  .omp_data_o.2.sum = sum;
  __builtin_GOMP_parallel_start (main.omp_fn.0, &.omp_data_o.2, 0);
  main.omp_fn.0 (&.omp_data_o.2);
  __builtin_GOMP_parallel_end ();
  sum = .omp_data_o.2.sum;
  D.2562 = sum;
  return D.2562;
}
void main.omp_fn.0(void*) (.omp_data_i)
{
...
<bb 2>:
  sum = 0;
  D.2578 = __builtin_omp_get_num_threads ();
  D.2579 = __builtin_omp_get_thread_num ();
  D.2580 = 100 / D.2578;
  ...
  D.2586 = MIN_EXPR <D.2585, 100>;
  if (D.2584 >= D.2586) goto <L3>; else goto <L1>;
<L3>:;
  sum.1 = (unsigned int) sum;
  D.2572 = &.omp_data_i->sum;
  __sync_fetch_and_add_4 (D.2572, sum.1);
  return;
<L1>:;
  D.2587 = D.2584 * 1;
  i = D.2587 + 1;
  D.2588 = D.2586 * 1;
  D.2589 = D.2588 + 1;
<L2>:;
  sum = sum + 2;
  i = i + 1;
  D.2590 = i < D.2589;
  if (D.2590) goto <L2>; else goto <L3>;

}
```

The first level of transformations is illustrated in the Listing 1.2 representing the high GIMPLE form of the code. The original parallel loop is expanded into a loop, nested in a parallel region. Also, data sharing and privatization clauses are automatically added to the parallel construct. In the next stage (Listing 1.3), the body of the parallel region is outlined into a separate function (`main.omp_fn.0`) and `libgomp` thread creation and termination routines are inserted. A variable `.omp_data_o.2` is created containing the address of every shared variable contained in the scope of the parallel construct. This variable is passed as an argument to the newly created function and all references to a shared variable within the function body are replaced by the corresponding field reference (`.omp_data_o.1.sum.reference`).

The following section studies how to use GOMP to implement the transformation of an OpenMP program into a MPI program.

4 Using GOMP to transform OpenMP programs into MPI programs

This section provides a description on how to use GOMP to generate MPI. Furthermore, in order to test some ideas, we have developed a limited prototype.

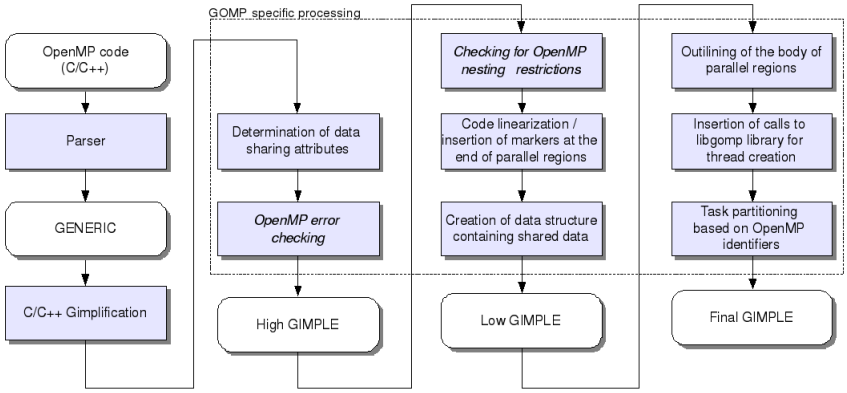

Fig. 2. GOMP transformations.

4.1 Description of GOMP adaptation for MPI generation

GOMP provides several code transformations that could be used for MPI transformation. Nevertheless, for MPI transformations, data dependency analyses are necessary to generate communications.

Code transformation. The front-end is used as is, without any modification since a MPI transformation does not introduce any new syntactic constructs. The automatic determination of data sharing attributes, code linearisation and marker insertion are also compatible and can be reused.

The outlining of the body of the parallel construct is compatible. Nevertheless, the semantics of the `parallel` pragma is very different in a message-passing context: there will be no task creation (since all tasks are launched at the beginning of the application). The `parallel` directive guarantees that there is no data dependency between two parallel constructs.

The task partitioning is necessary. But, in GOMP, local loop bounds are computed depending on thread identifiers directly calling OpenMP runtime primitives while for MPI transformations, MPI process identifiers should be used.

The insertion of GOMP thread creation routines should be prevented and replaced by insertion of MPI routines for initialization, communication and finalization. One major difference is that initialization and finalization must be inserted at the beginning and at the end of the main program and communication routines must be inserted after the `for` work-sharing construct whereas GOMP routines are inserted at `parallel` directive level.

The Figure 3 shows the respective positioning of GOMP processing and our transformation framework in the global GCC pipeline.

Communication generation. In order to generate the communication, we need to determine which shared data is modified inside a parallel work-sharing construct. A list of shared variables that are modified inside the parallel loop must be built. This should be limited to variables that are later used in the program. To do so, an additional analysing pass should be added, before inserting calls to MPI runtime routines, to determine shared modified variables. Since each variable must be determined with a size and a type, scalar and statically allocated arrays can directly be dealt with. For dynamically allocated arrays, the analysing pass must scan the data-flow and control-flow graphs to determine sizes when it is possible. At last, these informations must be propagated through function calls in order to be gathered at the work-sharing construct level.

These modified data will be updated on each MPI processes involving a all-to-all personalized communication at the end of the work-sharing construct. If no detailed information is available on data element accesses, then a runtime merge will be performed. If some information is available on accessed data elements (also referred as array regions in section 2.2), then only corresponding array regions will be updated.

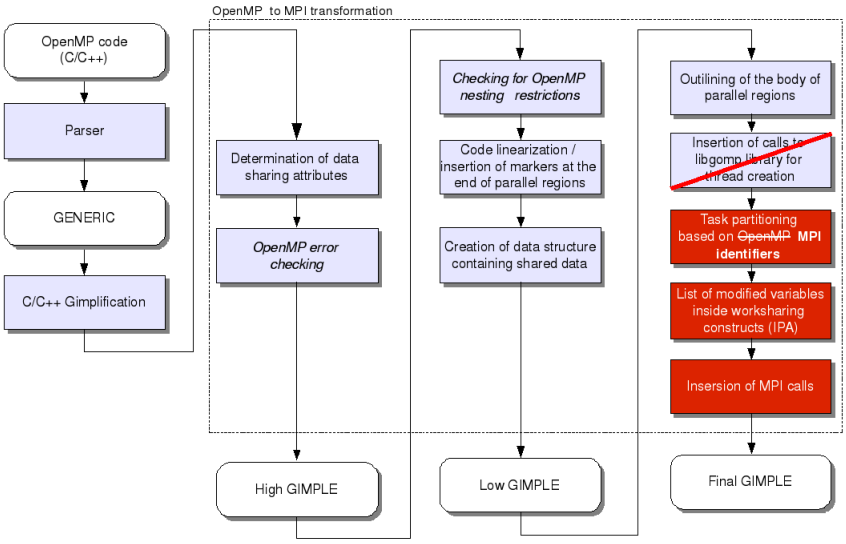

Fig. 3. GOMP and MPI transformations.

4.2 Current implementation

Our current implementation only concerns the `for` work-sharing constructs. It directly uses the main entry point for gimplification transformation (`gimplify_expr()`) and based on GOMP, we directly apply our transformation on traditional loops. The communication generation should only work on scalar or statically-allocated arrays and no function calls inside the parallel work-sharing construct.

Code transformation. During GOMP transformation, an OpenMP parallel loop is identified by the `OMP_FOR` code, and the associated function for gimplification is `c_gimplify_omp_for` for C and `cp_gimplify_omp_for` for C++. These two functions are called with a pointer to a tree which represents the body of the loop. The GIMPLE API provides a number of macros which ease the parsing and analysis of trees. For instance, macros for manipulating the `OMP_FOR` are as given in Table. 1

Table 1. GOMP GIMPLE macros: OMP_FOR accessors

Macro	Description
OMP_FOR_BODY	Body of the parallel loop
OMP_FOR_CLAUSES	List of OpenMP clauses
OMP_FOR_INIT	Initialization code ($VAR = N_1$).
OMP_FOR_COND	Loop conditional expression ($VAR\{<, >, \leq, \geq\}N_2$).
OMP_FOR_INCR	Loop index increment ($VAR\{+=, -=\}INCR$).

Our new loop structure is built from GOMP parallel loops by reusing the initialization, stop condition and increment expressions. To compute local lower and upper bounds of parallel loops depending on the current process identifier and the total number of MPI processes, we created two GIMPLE variables `lo.xx` and `hi.xx` and insert function calls to `step_Get_loop_LOW()` and `step_Get_loop_HIGH()` before the body of the parallel loop identified by the work-sharing `for` construct. The loop initializer (`OMP_FOR_INIT`) and condition (`OMP_FOR_COND`) are modified according to the new values. Finally, at the end of the loop, calls to our runtime MPI primitives for communicating array variables that have been modified in the parallel loop body are inserted.

Communication generation. A list of modified variables is manually built by listing variables taking place at left hand side of assignments. This analysis is provided for statically allocated arrays and does not follow function calls.

Currently, a centralized communication scheme is implemented. Each MPI process sends the set of modified variables to the master MPI process (with rank equal to 0). This process synchronizes each variable and then broadcasts the updated version to all other MPI processes.

A first communication optimization. The MPI transformation not only needs to identify which variables are modified, but also precisely identify modified array regions. At this point it is important to outline the fact that only array-type variables are of interest. Following the OpenMP standard [arb08], if a scalar variable is modified within a parallel loop without a guard condition or is modified without reduction, this means that it should be declared as private otherwise the parallel execution will lead to inconsistent results.

As no detailed information upon accesses to array elements is available, we have decided to implement a first optimization consisting in determining lower and higher indices of accessed elements for each array dimension. This supposes that array regions are accessed linearly within the parallel loop, in other words all array references are adjacent to each other. Therefore, such optimization is only possible on loops satisfying this condition. In order to do so, each variable which appears in the index expression has to be replaced by its maximal or minimal value depending on its arithmetical construction. Let us consider A, a n-dimensional array. Each reference (noted R_{iA}) to the array A will be associated to an interval $[A_{min}, A_{max}]$. A list of references to A in the parallel loop body is built if A is modified in different locations in the parallel loop body. The resulting interval is the union of the intervals associated with each reference.

In brief we give the algorithm of the array region analysis in Algorithm 1.

```
Input: A GENERIC tree : OpenMP loop
Output: A GIMPLE tree : MPI loop
foreach Array reference Ref do
    foreach Expression index exp of dimension dim do
        Calculate MIN(exp);
        Calculate MAX(exp);
        Write Region[ref][dim] =Union (Write Region[ref][dim],MIN(exp),MAX(exp));
    end
end
```

Algorithm 1: Array region analysis pseudo-code

This algorithm is based on some suppositions which limitates its applicability domain: index expressions must only contain references to the surrounding loop iterators, all the index expressions are affine expressions, the minimum and maximum values for nested loops iterators have constant types. Also, we notice that if the outer most loop iterators appears within any index expression, the minimal an maximal values can be computed only at runtime when local loop bounds are known. This algorithm should be correlated with previous works on the polyhedral model.

A simple transformation example. We consider the parallel RGB to grayscale transformation code as a case study. This code is a good illustrative example for applications eligible for our transformation framework. The original code is as shown in Listing 1.4. The array A is modified at a single place within the loop (line #17), and the index expression for the first dimension is $(j + i * W)$. The global iteration domain for i is given by the outer most loop header and is $[0, H-1]$, similarly the iteration domain for j is $[0, W-1]$. Since the outer most loop is executed in parallel on multiple threads, the local iteration domain is no longer as indicated by the loop header but only known at runtime. Currently in our implementation, two variables hold the minimum and the maximum values for the parallel loop respectively $lo.x$ and $hi.x$ (Listing 1.5) during the execution. The interval associated locally with the reference to the array A becomes then $[MIN(j) + lo.x * W, MAX(j) + hi.x * W]$ which becomes $[lo.x * W, H - 1 + hi.x * W]$ after replacing the minimum and the maximum values of j with the appropriate values. Now, at the end of the execution of the parallel loop, each thread knows that the region of A that has to be sent is $\{A[lo.x * W], .., Z[(H - 1 + hi.x * W)]\}$.

Listing 1.4. Example: parallel conversion of a RGB image to a grayscale image.

```
1  #include <stdio.h>
2  #include <omp.h>
3  #define W  10
4  #define H  10
5  int main()
6  {
7  int i,j;
8  unsigned char A[W*H];
9  unsigned char R[W*H],G[W*H],B[W*H];
10 i=0;
11 j=0;
12 #pragma omp parallel for
13 for(int i = 0; i < H; i++)
14 {
15  for(j = 0; j< W; j++)
16  {
17          A[j+i*W] =R[j+i*W]*0.299+\
18                  G[j+i*W]*0.587+\
19                  B[j+i*W]*0.114;
20  }
21 }
22 return 0;
23 }
```

Listing 1.5. Low GIMPLE form.

```
1
2  int main() ()
3  {
4  ...
5  {
6   {
7     ...
8    step_Get_loop_LO (&lo.0);
9    step_Get_loop_HI (&hi.1);
10   i = lo.0;
11   goto <D.2237>;
12   <D.2235>:;
13   j = 0;
14   goto <D.2241>;
15   <D.2240>:;
16   D.2242 = 0 * 10;
17   D.2243 = D.2242 + j;
18   D.2244 = i * 10;
19   /*Loop Body*/
20   ...
21   D.2267 = __step_get_rank ();
22   if (D.2267 == 0)
23   {
24    step_RecvMerge (A, 0, 99,MPI_TYPE_INT);
25    step_BcastRegion (A, 0, 99,MPI_TYPE_INT);
26   }
27   else
28   {
29    D.2268 = hi.1 * 10;
30    D.2269 = D.2268 + 9;
31    D.2270 = lo.0 * 10;
32    D.2271 = D.2270 + 0;
33    step_SendRegion (A, 0, D.2271, D.2269,MPI_TYPE_INT);
34    step_RecvRegion (A, 0, 99,MPI_TYPE_INT);
35   }
36   }
37   }
38 ...
39 }
```

As shown in the GIMPLE representation of the code in Listing 1.5, when the work-sharing `for` construct ends, each MPI process sends the modified region of the array. The MPI process with rank equal to 0 gathers all the regions and builds up a new consistent copy which is sent to all other MPI processes. Lines #21 to #24 show the code which performs computation of array region to send.

5 Related work

In this section, we consider two main related works: first we study a transformation close to ours concerning streams [PPJ+], second we study attempts to add GIMPLE-level analyses in GCC through the GRAPHITE framework [PSC+06].

In [CRM+07,DPA+08,Gas08,PPJ+], authors propose to extend OpenMP with additional clauses necessary for streamization. In the European IST ACOTES project [CRM+07] and also in Duran *et al.* [DPA+08], the OpenMP 3.0 `task` directive is extended with `input` and `output` clauses. These new clauses are used to schedule parallel tasks in a producer/consumer way whereas data-dependent tasks cannot be implemented so far using OpenMP. Gaster [Gas08] propose to add a new datatype for handling streams. A stream is created from an array, using the `stream create` directive, and is particularly characterized by a chunk size. Streams will be explicitly connected (via the `connect` clause) to each thread in a team defined by a parallel region. This chunk size corresponds to "the number of elements that will copied to a particular thread". Inside parallel regions, threads will explicitly read, using the `stream read` directive, stream elements. Thus in this work, streams are really considered as a "collection of data that can be processed in parallel" and the extensions of OpenMP are a means to distribute this collection of data. This is valid both for data parallelism traditionally provided by OpenMP but also for task parallelism. Let us notice that these extensions also provide a way to express data-dependent tasks.

A first implementation of streams as an OpenMP extension inside GCC has been provided by Pop *et al.* [PPJ+]. Pop *et al.* automatically detect tasks that present a producer/consumer relationship. These tasks correspond to statements with flow dependences between each other. When detected, OpenMP parallel sections are automatically inserted with stream function calls. This approach encounters limitations due to the computation grain. The automatic detection of tasks will be limited to the static analysis of data dependencies. Coarse-grain dependencies are very complex to extract. Therefore fine-grain tasks only can be extracted thus limiting the potential concurrency. Two major improvements are proposed to extract coarser-grain tasks: improve the static data dependency analysis with interprocedural array region analysis and handle tasks directives with input and output additional clauses provided by the

programmer. We could also use these two improvement directions to generate efficient MPI communications. At last, the stream data structure proposed by Gaster is interesting for us because the chunk size associated to each stream provides a way to distribute data in case of distributed OpenMP. As a matter of fact, so far in our MPI translation proposal, data is replicated on all computing nodes.

In [PSC+06], a intermediate representation is proposed called GRAPHITE to host high-level loop transformations. The support for polyhedral methods in GCC is also used to add new static analyses. Particularly, the idea of performing "on-demand" computations on part of the dependence graph only is interesting for us since we need to precisely analyse parallel loops. Furthermore, this could also be used to implement interprocedural array regions as proposed by PIPS [AAC+94].

6 Conclusion

Our goal is to use existing tools to develop a parallel 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/or MPI 2) and the compilation community which has developed powerful program analyses and transformations.

In this paper, we propose to implement the transformation of an OpenMP program into a MPI program using the GOMP framework. The relaxed-consistency memory of the OpenMP standard guarantees that such a transformation can provide a correct code. This paper show that GOMP provides useful parsers, data structures and routines for code transformations that we should reuse. Nevertheless, in GOMP multithreaded environment, shared data are directly handled through shared memory. In a distributed-memory MPI environment, data must be explicitly updated via communications.

Thus, in order to generate MPI communications, a list of modified variables (including type and size of each variable) inside a work-sharing region must be built. At the GOMP level, no such pass is available. Furthermore this analysis should be interprocedural since the information is needed at the OpenMP work-sharing construct level. At last, the analysis must provide type and size of data. It is straightforward for statically-allocated arrays but it involves additional computation for dynamically-allocated arrays.

In a future work, we have to study the analyses provided by GCC at SSA level and determine whether a higher level framework for analysis should be relevant as proposed by the GRAPHITE project.

7 Acknowledgements

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

References

[AAC+94] Corinne Ancourt, Béatrice 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.

[arb08] OpenMP architecture review board. *OpenMP Application Program Interface, version 3.0*, 2008.

[BCC+00] John Bircsak, Peter Craig, Raelyn Crowell, Zarka Cvetanovic, Jonathan Harris, C. Alexander Nelson, and Carl D. Offner. Extending OpenMP for NUMA Machines. In *Proceedings of the ACM/IEEE Supercomputing Conference*, 2000.

[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.

[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.

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

[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.

[CRM+07] Paul Carpenter, Alex Ramirez, Xavier Martorell, David Rodenas, and Roger Ferrer. Deliverable D2.1: Streaming Programming Model and Abstract Streaming Machine description. Technical report, IST ACOTES (Advanced Compiler Technologies for Embedded Streaming) project, 2007.

[DPA+08] Alejandro Duran, Josep M. Perez, Eduard Ayguadé, Rosa M. Badia, and Jesus Labarta. Extending the OpenMP Tasking Model to Allow Dependent Tasks. In *OpenMP in a new area of parallelism, 4th international workshop on OpenMP, IWOMP*, 2008.

[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.

[Gas08] Benedict R. Gaster. Streams: Emerging from a Shared Memory Model. In *International Workshop on OpenMP IWOMP'08, OpenMP in a New Era of Parallelism*, 2008.

[HCL05] Lei Huang, Barbara Chapman, and Zhenying Liu. Towards a more efficient implementation of OpenMP for Clusters via translation to Global Arrays. *Parallel Computing*, 31(10-12), 2005.

[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.

[JBW+02] Ahmed Amine Jerraya, Amer Baghdadi, Cesario Wander, Gauthier Lovic, Lyonnard Damien, Nicolescu Gabriela, Paviot Yanick, and Yoo Sungjoo. Application-specific Multiprocessor Systems-On-Chip. *Microelectronics Journal*, 33(11):891–898, 2002.

[LAOH00] Jesus Labarta, Eduard Ayguadé, José Oliver, and David Henty. New openMP Directives for Irregular Data Access Loops. In *2nd European Workshop on OpenMP (EWOMP'00)*, 2000.

[MMPSC08] Daniel Millot, Alain Muller, Christian Parrot, and Frédérique Silber-Chaussumier. STEP: a distributed OpenMP for coarse-grain parallelism tool. In *International Workshop on OpenMP IWOMP'08, OpenMP in a New Era of Parallelism*, 2008.

[Nov06] Diego Novillo. OpenMP and automatic parallelization in GCC. In *GCC Summit 2006*, 2006.

[NPP+00] D.S. Nikolopoulos, T.S. Papatheodorou, C.D. Polychronopoulos, Jesus Labarta, and Eduard Ayguadé. Is Data Distribution Necessary in OpenMP? In *IEEE/ACM Supercomputing'2000: High Performance Computing and Networking Conference (SC'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.

[PPJ+] Antoniu Pop, Sebastian Pop, Harsha Jagasia, Jan Sjdin, and Paul H.J. Kelly. Improving GNU Compiler Collection Infrastructure for Streamization. In *GCC Summit 2008*.

[PSC+06] Sebastian Pop, Georges-Andr Silber, Albert Cohen, Cdric Bastoul, Sylvain Girbal, and Nicolas Vasilache. GRAPHITE : Polyhedral Analyses and Optimizations for GCC. In *GCC Summit 2006, Ottawa, Canada*, 2006.

[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.