OpenMP and automatic parallelization in GCC

Diego Novillo
Red Hat Canada
dnovillo@redhat.com

Abstract

This paper describes the design and implementation of the OpenMP specification v2.5 in GCC. The implementation supports all the languages specified in the standard (C, C++ and Fortran), and it is generally available on any platform that supports POSIX threads.

Emphasis is placed on the internal architecture and, in particular, the intermediate representation, which could be used in the implementation of automatic parallelization techniques. The paper also presents performance results on the SPEC OMP2001 benchmark.

1 Introduction

OpenMP defines language extensions to C, C++ and Fortran for implementing shared-memory multi-threaded applications [1]. Compiler pragmas are used to define parallel regions, data and work sharing attributes. A runtime library implements the actual mechanism for creating threads, synchronization and data sharing.

This paper describes GOMP (GNU OpenMP), an OpenMP implementation for GCC. There are four main components: parser, intermediate representation, code generation and the runtime library (`libgomp`). The parser identifies and validates the OpenMP pragmas and emits the corresponding GENERIC representation. The IR used to represent OpenMP is an extension to GENERIC and GIMPLE. It serves a dual purpose: as an interface to `libgomp` and as a code generation target for auto-parallelization transformations.

2 Parser

OpenMP defines a collection of compiler pragmas for C, C++ and Fortran. As such, three separate implementations were required for each of the front ends. The new pragmas are categorized in two groups: **directives** for specifying parallelism and work-sharing, and **clauses** for specifying data sharing and thread scheduling properties.

Every OpenMP command starts with `#pragma omp` and though the standard defines quite a few of them, they are mostly straightforward to recognize in a recursive descent scan. The recognition code is hooked into the standard pragma processing code in each of the front ends: `c-parser.c:c_parser_omp_*` for C, `cp/parser.c:cp_parser_omp_*` for C++ and `fortran/parse.c: parse_omp_*` for Fortran.

Once recognized, the front ends generate the corresponding GENERIC representation as described in the next section. Some of the semantic analysis and validation is also done during parsing. Structural diagnostics such as nesting of directives is done after the representation is in GIMPLE form (`omp-low.c:diagnose_omp_structured_block_errors`). Other common diagnostics are emitted during the conversion into GIMPLE (`gimplify.c:gimplify_omp_*` and `gimplify.c:omp_*`).

3 Intermediate Representation

Most directives and clauses have a corresponding GENERIC node defined in `tree.def`. The basic code generation strategy is to outline the body of parallel regions into functions that are used as arguments to the `libgomp` thread creation routines. Data sharing is implemented by passing the address of a local structure with all the data items marked for sharing. Copy-in data is passed by value, while copy-in/copy-out data and variables that are bigger than a certain threshold are passed by address.

To illustrate at a high-level how OpenMP programs are compiled, consider the program in Figure 1 to compute the sum of all the thread IDs in parallel[1].

```
main()
{
  int sum = 0;
  #pragma omp parallel
    {
      #pragma omp atomic
      sum += omp_get_thread_num ();
    }
  printf ("sum = %d\n", sum);
}
```

Figure 1: OpenMP program to compute a sum.

```
main ()
{
  sum = 0;
  #pragma omp parallel shared(sum)
    {
      D.1324 = omp_get_thread_num ();
      D.1325 = (unsigned int) D.1324;
      __sync_fetch_and_add_4 (&sum, D.1325);
    }
  sum.0 = sum;
  printf ("sum = %d\n", sum.0);
}
```

Figure 2: High GIMPLE form for Figure 1.

Figure 2 shows the corresponding High GIMPLE representation. Note that for debugging convenience, the IL pretty-printer renders OpenMP statements using the `#pragma omp` syntax. Some transformations and mappings are done during parsing and gimplification. For instance, all predetermined or implicitly determined sharing attributes are made explicit for the benefit of code generation. In the case of Figure 2, variable `sum` is predetermined shared. Also, the atomic add operation is mapped into the corresponding `__sync` built-in.

The next lowering stage (`omp-low.c:pass_lower_omp`) sets up mappings for satisfying data sharing attributes and linearizes the bodies of the OpenMP directives. Converting the code into linear form, requires the addition of `OMP_RETURN` markers that indicate the end of each body. This becomes important later when the parallel work-sharing regions are expanded into the corresponding `libgomp` calls. In Figure 3, the `OMP_RETURN` at line 9 marks the end of the parallel region starting at line 3.

Data sharing is implemented using an artificial data structure (`struct .omp_data_s`) whose fields are all the variables included

[1] Yes, the program makes absolutely no sense.

```
main ()
{
 1  sum = 0;
 2  .omp_data_o.sum = &sum;
 3  #pragma omp parallel shared(sum)
 4  .omp_data_i = &.omp_data_o;
 5  D.1324 = omp_get_thread_num ();
 6  D.1325 = (unsigned int) D.1324;
 7  D.1334 = .omp_data_i->sum;
 8  __sync_fetch_and_add_4 (D.1334, D.1325);
 9  OMP_RETURN
10  sum.0 = sum;
11  printf (&"sum = %d\n"[0], sum.0);
12  return;
}
```

Figure 3: Low GIMPLE form for Figure 1.

in data sharing clauses like `shared` and `copyin`. This is why the front end is required to explicitly indicate all the variables with sharing semantics. In general, variables with sharing or copy-in/copy-out semantics are passed by reference while variables with copy-in semantics are passed by value. However, if a copy-in variable is too large, it will also be passed by reference. This is controlled by `omp-low.c:use_pointer_for_field`.

Two local variables are created: `.omp_data_o`, which is filled in with the addresses and values of every shared variable to be sent to the children threads (line 2 in Figure 3), and `.omp_data_i`, which will hold the address of `.omp_data_o` (line 4 in Figure 3). This way, every reference to variable `sum` inside the body of the `omp parallel` directive, is rewritten to use `.omp_data_i->sum`.

This seemingly convoluted rewriting is necessary for outlining the body of the `omp parallel` into a separate function as shown in Figure 4. The new function `main.omp_fn.0` receives `&.omp_data_o` in its argument `.omp_data_i`. Final expansion replaces the parallel body with calls into

```
main ()
{
 1  # BLOCK 0
 2  # PRED: ENTRY (fallthru)
 3  sum = 0;
 4  .omp_data_o.sum = &sum;
 5  __builtin_GOMP_parallel_start (main.omp_fn.0,
 6                                 &.omp_data_o, 0);
 7  main.omp_fn.0 (&.omp_data_o);
 8  __builtin_GOMP_parallel_end ();
 9  sum.0 = sum;
10  printf (&"sum = %d\n"[0], sum.0);
11  return;
12  # SUCC: EXIT
}

main.omp_fn.0 (.omp_data_i)
{
13  # BLOCK 0
14  # PRED: ENTRY (fallthru)
15  D.1324 = omp_get_thread_num ();
16  D.1325 = (unsigned int) D.1324;
17  D.1334 = .omp_data_i->sum;
18  __sync_fetch_and_add_4 (D.1334, D.1325);
19  return;
20  # SUCC: EXIT
}
```

Figure 4: Final expansion for Figure 1.

`libgomp` to launch children threads and execute `main.omp_fn.0` (lines $5-8$ in Figure 4).

The sequence of transformations proceeds as follows:

1. The front end parses the OpenMP pragmas and emits the corresponding GENERIC statements as described in Section 3.1.

2. The gimplifier determines which variables are used inside parallel regions and establishes mappings according to the data sharing clauses. It also tries to replace `omp atomic` directives with corresponding atomic update functions.

3. pass_lower_omp creates the artificial data structure to implement the data sharing mappings, rewrites variables to use the fields in struct .omp_data_s, expands some forms of synchronization and adds OMP_RETURN markers for directive bodies.

4. pass_lower_cf linearizes the directives and their bodies to remove the nested property and prepare the IL for building the flow graph.

5. pass_build_cfg builds the control flow graph, making sure that incoming edges into parallel regions are marked abnormal to avoid CFG cleanups from making any assumptions that may violate parallel semantics. This is mostly a precautionary measure, as no such cleanups are currently implemented that may cause these problems.

   One important property about omp parallel regions is that they are guaranteed to be single-entry, single-exit. This is exploited by the expansion phase.

6. pass_expand_omp runs just before the code is put into SSA form. With the existing implementation, omp parallel regions cannot be put into SSA form because it does not support concurrency semantics.

   This pass outlines the single-entry, single-exit region of every omp parallel into a new function and expands all the other directives into calls to libgomp or the corresponding GIMPLE expansion. For instance, the computations needed to calculate iteration space bounds for statically scheduled parallel loops are expanded inline (Figures 5(a) and 5(b)).

3.1 Directives

Most OpenMP directives and clauses have a corresponding GENERIC and GIMPLE code. The exception are those that can be represented with built-in function calls (e.g. omp barrier, omp flush) or attributes (e.g. omp threadprivate are handled with the standard the thread-local storage attributes).

Calls to libgomp are encoded as built-in functions in omp-builtins.def. Directives and clauses encoded as IL statements are defined in tree.def. All the front ends emit the statements and built-ins defined in these files.

The C and C++ front ends share common code generation routines in c-omp.c while the Fortran front end converts its parse trees into GENERIC in fortran/trans-openmp.c.

OMP_PARALLEL

Represents #pragma omp parallel [clause1 ... clauseN]. It has four operands:

Operand OMP_PARALLEL_BODY is valid while in GENERIC and High GIMPLE forms. It contains the body of code to be executed by all the threads. During GIMPLE lowering, this operand becomes NULL and the body is emitted linearly after OMP_PARALLEL.

Operand OMP_PARALLEL_CLAUSES is the list of clauses associated with the directive.

Operand OMP_PARALLEL_FN is created by pass_lower_omp, it contains the FUNCTION_DECL for the function that will contain the body of the parallel region.

Operand OMP_PARALLEL_DATA_ARG is also created by pass_lower_omp. If

```
foo ()
{
  #pragma omp for
  for (i = 0; i <= 8; i = i + 1)
  do_work (i);
  OMP_CONTINUE
  OMP_RETURN
  return;
}
```

(a) Low GIMPLE form.

```
foo ()
{
       /* Lines 3-14 compute the iteration space for
          each thread.    */
 3     D.1330 = __builtin_omp_get_num_threads ();
 4     D.1331 = (unsigned int) D.1330;
 5     D.1332 = __builtin_omp_get_thread_num ();
 6     D.1333 = (unsigned int) D.1332;
 7     D.1334 = 9 / D.1331;
 8     D.1335 = D.1334 * D.1331;
 9     D.1336 = D.1335 != 9;
10     D.1337 = D.1334 + D.1336;
11     D.1338 = D.1337 * D.1333;
12     D.1339 = D.1338 + D.1337;
13     D.1340 = MIN_EXPR <D.1339, 9>;
14     if (D.1338 >= D.1340) goto <L3>; else goto <L0>;
       /* Lines 20-25 compute the first and last value of
          'i' taking the loop increment value into
          consideration.    */
17     # BLOCK 1
19   <L0>:;
20     D.1341 = (int) D.1338;
21     D.1342 = D.1341 * 1;
22     i = D.1342 + 0;
23     D.1343 = (int) D.1340;
24     D.1344 = D.1343 * 1;
25     D.1345 = D.1344 + 0;
       /* Lines 31-34 are the actual loop.    */
28     # BLOCK 2
30   <L1>:;
31     do_work (i);
32     i = i + 1;
33     D.1346 = i < D.1345;
34     if (D.1346) goto <L1>; else goto <L3>;
       /* This barrier is emitted because the loop
          was not marked with the 'nowait' clause.    */
37     # BLOCK 3
39   <L3>:;
40     __builtin_GOMP_barrier ();
41     return;
}
```

(b) Corresponding expansion.

Figure 5: Expansion of a statically scheduled parallel loop.

there are shared variables to be communicated to the children threads, this operand will contain the VAR_DECL that contains all the shared values and variables.

OMP_FOR

Represents #pragma omp for [clause1 ... clauseN]. It has 5 operands:

Operand OMP_FOR_BODY contains the loop body.

Operand OMP_FOR_CLAUSES is the list of clauses associated with the directive.

Operand OMP_FOR_INIT is the loop initialization code of the form VAR = N1.

Operand OMP_FOR_COND is the loop conditional expression of the form VAR {<,>,<=,>=} N2.

Operand OMP_FOR_INCR is the loop index increment of the form VAR {+=,-=} INCR.

Operand OMP_FOR_PRE_BODY contains side-effect code from operands OMP_FOR_INIT, OMP_FOR_COND and OMP_FOR_INC. These side-effects are part of the OMP_FOR block but must be evaluated before the start of loop body.

The loop index variable VAR must be a signed integer variable, which is implicitly private to each thread. Bounds N1 and N2 and the increment expression INCR are required to be loop invariant integer expressions that are evaluated without any synchronization. The evaluation order, frequency of evaluation and side-effects are unspecified by the standard.

OMP_SECTIONS

Represents #pragma omp sections [clause1 ... clauseN].

Operand OMP_SECTIONS_BODY contains the sections body, which in turn contains a set of OMP_SECTION nodes for each of the concurrent sections delimited by #pragma omp section.

Operand OMP_SECTIONS_CLAUSES is the list of clauses associated with the directive.

OMP_SINGLE

Represents #pragma omp single.

Operand OMP_SINGLE_BODY contains the body of code to be executed by a single thread.

Operand OMP_SINGLE_CLAUSES is the list of clauses associated with the directive.

OMP_MASTER

Represents #pragma omp master.

Operand OMP_MASTER_BODY contains the body of code to be executed by the master thread.

OMP_ORDERED

Represents #pragma omp ordered.

Operand OMP_ORDERED_BODY contains the body of code to be executed in the sequential order dictated by the loop index variable.

OMP_CRITICAL

Represents #pragma omp critical [name].

Operand OMP_CRITICAL_BODY is the critical section.

Operand OMP_CRITICAL_NAME is an optional identifier to label the critical section.

OMP_ATOMIC

Represents #pragma omp atomic.

Operand 0 is the address at which the atomic operation is to be performed.

Operand 1 is the expression to evaluate. The gimplifier tries three alternative code generation strategies. Whenever possible, an atomic update built-in is used. If that fails, a compare-and-swap loop is attempted. If that also fails, a regular critical section around the expression is used.

OMP_RETURN

This does not represent any OpenMP directive, it is an artificial marker to indicate the end of the body of an OpenMP. It is used by the flow graph (tree-cfg.c) and OpenMP region building code (omp-low.c).

OMP_CONTINUE

Similarly, this instruction does not represent an OpenMP directive, it is used by OMP_FOR and OMP_SECTIONS to mark the place where the code needs to loop to the next iteration (in the case of OMP_FOR) or the next section (in the case of OMP_SECTIONS).

In some cases, OMP_CONTINUE is placed right before OMP_RETURN. But if there are cleanups that need to occur right after the looping body, it will be emitted between OMP_CONTINUE and OMP_RETURN.

3.2 Clauses

Clause codes are defined in tree.h as subcodes for the main OMP_CLAUSE code. This was necessary because of code space overflow in tree.def. GCC does not support more than 256 IL codes, so clauses are all represented by a main code (OMP_CLAUSE) and a sub-code, which can be one of OMP_CLAUSE_PRIVATE, OMP_CLAUSE_SHARED, OMP_CLAUSE_FIRSTPRIVATE, OMP_CLAUSE_LASTPRIVATE, OMP_CLAUSE_COPYIN, OMP_CLAUSE_COPYPRIVATE, OMP_CLAUSE_IF, OMP_CLAUSE_NUM_THREADS, OMP_CLAUSE_SCHEDULE, OMP_CLAUSE_NOWAIT, OMP_CLAUSE_ORDERED, OMP_CLAUSE_DEFAULT, and OMP_CLAUSE_REDUCTION.

Clauses associated with the same directive are chained together via OMP_CLAUSE_CHAIN. Those clauses that accept a list of variables are restricted to exactly one, accessed with OMP_CLAUSE_VAR. Therefore, multiple variables under the same clause *C* need to be represented as multiple *C* clauses chained together. This facilitates adding new clauses during compilation.

4 Auto parallelization

The new GENERIC and GIMPLE codes used for OpenMP can also be the target for an auto parallelization pass. Although GCC does not currently implement such a transformation, all the necessary data dependency and code generation tools are already present.

It is possible to emit both task and data parallel code using OMP_SECTIONS and OMP_FOR respectively. Data sharing semantics can be implemented with the corresponding OMP_CLAUSE_* codes and synchronization needed to preserve sequential data dependency semantics may use the appropriate OMP directive or call the libgomp routines directly.

Once parallel GIMPLE code is generated, pass_expand_omp may be used to do the outlining and low-level expansion work, and schedule the new function into the call-graph. Currently, care should be taken to take the function out of SSA form prior to these transformations because the call graph manager currently expects functions to be in normal form. However, this limitation may be lifted in the future.

5 Runtime Library

The runtime library (libgomp) is essentially a wrapper around the POSIX threads library, with some target-specific optimizations for systems that support lighter weight implementation of certain primitives. For instance, locking primitives in some Linux targets are implemented using atomic instructions and futex system calls. To support libgomp, the target must also implement thread-local storage.

The implementation is in gcc/libgomp and most entry points into the library are defined as built-in function calls inside the compiler.

5.1 Thread creation

The main entry point is GOMP_parallel_start, which takes as arguments the function to run on each thread, a pointer to the .omp_data_s structure as described earlier and the number of threads to be launched. If the specified number of threads is 0, the number of threads is computed automatically.

Once the parallel region ends, threads are docked so that they can be re-used at a later time. The master thread keeps executing the code after GOMP_parallel_start, which in this case is just another invocation to the same function that the children threads are executing. A call to GOMP_parallel_end Tears down the team and returns to the previous parallel state.

There are alternate entry points for combined parallel and work-sharing constructs that avoid one extra synchronization at the start of the work-sharing construct. The compiler tries to emit these combined calls whenever possible (omp-low.c:determine_parallel_type).

5.2 Synchronization

With few exceptions, most synchronization is just a direct mapping to the underlying POSIX routines. The exceptions are omp master and omp single:

omp master simply blocks the thread with a thread-id different than 0.

omp single has two alternate entry points, with and without the copyprivate clause. Since copyprivate is used to broadcast the values computed inside the omp single body, the compiler emits a call to GOMP_single_copy_start, which will block all the threads except one. On return, the blocked threads receive a pointer into a common area which will have been filled by the thread that entered the region. That area contains the broadcast data. See omp-low.c:lower_omp_single_copy for details.

5.3 Work sharing

Every scheduling variant of omp for has been implemented in the library. There are three main functions, GOMP_loop_*_start to initialize the loop bounds, GOMP_loop_*_next to get the next chunk of iteration space to work on, and GOMP_loop_*_end to finalize the parallel loop.

The omp sections construct is simpler. The compiler transforms the construct into a switch statement using the section id as index. The call to GOMP_sections_start sets up the work-share construct and record the number of sections found in the body. GOMP_sections_next returns the next section id to execute. Once all the sections have been executed, a barrier after the switch synchronizes all the threads.

Benchmark	ICC 9.0	GCC 4.2.0	% Diff
wupwise	227.0	224.0	-1.3%
swim	140.0	138.0	-1.4%
mgrid	146.0	140.0	-4.1%
applu	154.9	147.3	-4.9%
equake	267.2	264.5	-1.0%
apsi	179.0	179.0	0.0%
fma3d	139.0	133.0	-4.3%
ammp	140.0	153.0	9.3%
Mean	**169.11**	**167.31**	-1.1%

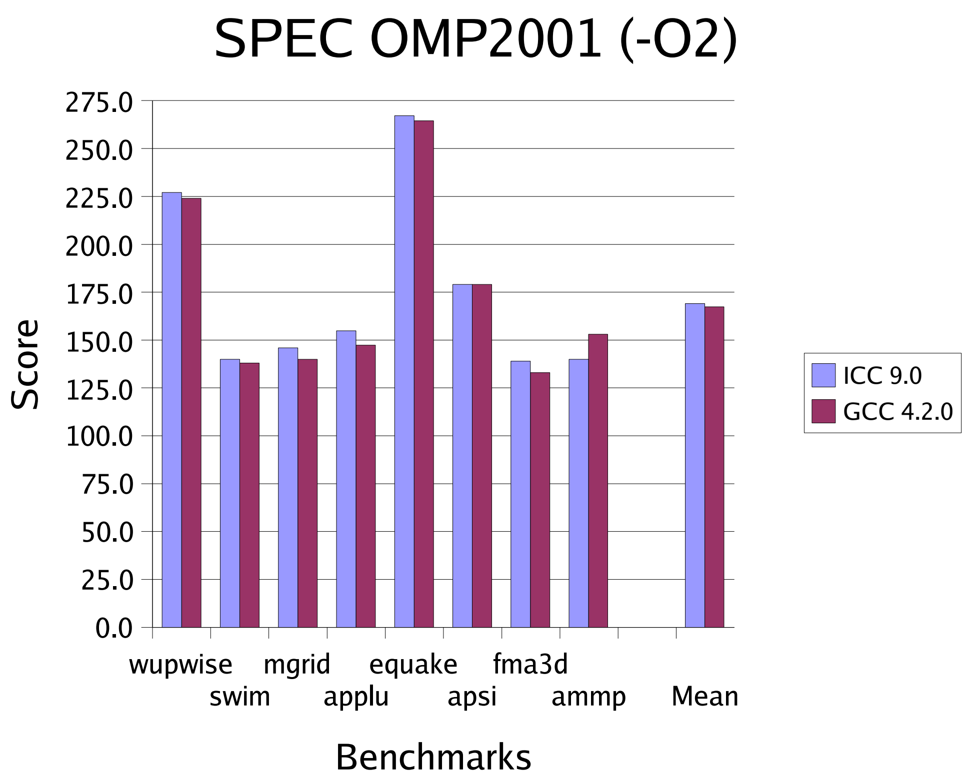

Figure 6: SPEC OMP2001 scores for GCC and ICC on a dual processor EM64T. Higher scores indicate better performance.

6 Implementation Status

At the time of this writing, the implementation is feature complete for all the three languages defined in the standard and scheduled to be released with GCC 4.2. It has also been ported to the GCC 4.1 version included in Fedora Core 5.

The focus over the next few months will be bug fixing and performance tuning. No firm plans exist yet for an auto-parallelization pass as described in the previous section, but it should not be an exceedingly complex project to implement.

To assess the performance of the code generated by GCC, I used SPEC OMP2001 on a dual processor Intel EM64T at 3.4Ghz with 2Gb of RAM, running Fedora Core Linux 3. The compilers tested were GCC v4.2.0 20060406 (experimental) and ICC v9.0 20050914.

As shown in Figure 6, the performance differences between the two compilers are negligible. GCC has a slight edge in some tests and vice versa, but the geometric mean is almost identical.

Both compilers used the standard `-O2` optimization level. Note that the goal was to get a rough idea on how the GCC implementation compares to other compilers. This was not a valid SPEC run as neither GCC nor ICC were able to run all the benchmarks without errors. GCC failed to execute `gafort` and `art`, while ICC failed to build `galgel` and failed to execute `gafort`. Tests that failed in either compiler were taken out of the chart.

References

[1] OpenMP Architecture Review Board. Openmp application program interface v2.5. May 2005. http://www.openmp.org/drupal/mp-documents/spec25.pdf.