OMPi translator internals

VVD and the OMPi team

Nov. 2010
Dec. 2009
Nov. 2007

Abstract

We describe briefly some of the internals of OMPi's translator, for anybody interested in altering it and possibly providing new functionality.

1 Introduction

This document is about OMPi's translator, used in versions $\geq$ 1.0.0. It is a source-to-source compiler written entirely in C, uses an abstract s yntax tree (AST) to represent the parsed source code and implements a number of transformation optimizations on the code it outputs. It understands fully the C99 grammar plus OpenMP V.3.0 pragmas (with the single exception of the "collapse" clause).

The most important source files comprising the translator are the following:

1. `ompi.c`
   File `ompi.c` is the main driver which, among others, calls the parsing and transformation routines.
2. `scanner.l, parser.y`
   The scanner is implemented through flex and the parser through bison. The grammar is defined in `parser.y` and this is where the AST is built up, too.
3. `ast.c, symtab.c`
   They contain the fundamental AST and symbol table routines.
4. `ast_show.c, ast_print.c, ast_free.c, ast_copy.c`
   These files take the AST and print it to stdout or to a string, they free it and they produce copies of it.
5. `ast_xform.c`
   This is the main file that drives all OpenMP transformation actions. It also implements a few of them, the shortest ones.
6. `x_parallel.c, x_single.c, x_sections.c, x_for.c, x_thrpriv.c, x_task.c`
   These files implement the 6 construct transformations which are not included in `ast_xform.c`.
7. `ast_vars.c, x_clauses.c`
   These are necessary for the analysis and transformation of variables in any piece of code and in OpenMP clauses.
8. `x_types.c`
   It handles user-defined types and declarations which include struct/union/enum entities.

2 The AST structure

The AST is built based on the structures and routines in `ast.c`. It contains 7 types of nodes: *astexpr, astspec, astdecl, aststmt, ompcon, ompdir* and *ompclause*, used correspondingly for expressions, the specifier part of a declaration, the declarator part of a declaration (see below), statements, OpenMP constructs, OpenMP directives and OpenMP clauses. Note that all those types are actually pointers to the associated structures defined in `ast.h`, as pointers are handy when building the AST.

`ast.c` contains only routines that *create* nodes of the tree; all other functionality is implemented elsewhere, e.g. free()ing of the nodes is done in `ast_free.c`, copying/duplicating trees is done in `ast_copy.c`.

The data stored at each node is beyond the scope of this document and can be discovered by looking in `ast.c` and following the grammar in `parser.y`. The root of the original AST is a statement node, usually a BlockList (see the "translation_unit" rule in `parser.y`).

Only two things need to be mentioned:

- The tree has downward pointers, from the root down towards the leaves and tree traversals occur in this direction, too. However, later stages (e.g. transformations) sometimes need to traverse the tree upwards, i.e. from lower level nodes back up towards the root of the tree. This is why each statement node (aststmt) has a 'parent' field that is initially NULL. Each statement node can store there a pointer to its parent in the tree. The initial setup of those fields occurs right after the AST is built by the parser, through a call to ast_parentize() (in `ompi.c`). Whenever portions of the tree change, calls to ast_stmt_parentize() correctly parentize the affected statement nodes.

- A declaration consists of two parts: the *specifier* and the *declarator*. The first holds the "type" and the second holds the identifier, along with various attributes (pointer, array, etc.). For example, in the following declaration:

  ```
  static long int *a[10], b = 1;
  ```

  "static long int" is the specifier and the declarator is actually a 'declarator list' with two declarators: '*a[10]' and 'b = 1'. The latter is a declarator *with an initializer.* Both declarators here share the same specifier.

3 Scanning, parsing & building the AST

`Scanner.l` does not contain many mysteries. It is a relatively straightforward lex/flex scanner which assumes that the file it scans is already preprocessed. It can identify line number information (`# <line no> <filename>`) and upon meeting an unknown word it uses the symbol table ("stab", see below) to decide whether it is a variable (identifier) or a typename.

Parsing is done through parse_file(), which is called in `ompi.c`. `Parser.y` also builds the abstract syntax tree (AST), the root of which is returned by parse_file(). However, except parsing the source file, we need to parse stored strings later on when we transform the AST. Thus the parser should be able to parse strings as well and moreover the starting rule may be different each time. In particular, the starting rule is:

- "translation_unit", when parsing the source file
- "expression" or "block_item_list" when parsing strings, depending on what the string contains.

To achieve this, we use the trick described in Bison's manual which looks like this:

```
%start start_trick;
%%
start_trick:
    translation_unit                        { pastree = $1;      }
```

```
    | START_SYMBOL_EXPRESSION expression     { pastree_expr = $2; }
    | START_SYMBOL_BLOCKLIST block_item_list { pastree_stmt = $2; }
  ;
```

Of course, the scanner provides calls to force generation of the two dummy tokens so as to trigger the corresponding rule. All is taken care by the two functions provided in `parser.y` (parse_expression_string() and parse_blocklist_string()).

When parsing the source file, a thorough check for unknown/undeclared variables is performed; every new declaration puts the variable in the symbol table ("stab"—declared in `ompi.c`) keeping track of all scopes. At the end of parsing, the symbol table is emptied and the AST is successfully built. Everything related to symbols and symbol tables is implemented in `symtab.c`.

There are only two other things going on in `parser.y`:

1. The replacement of the "main()" function in the source code (if such a function exists). In particular, if this function exists it gets renamed to "__original_main" (the name is defined as MAIN_NEWNAME in `ompi.c`). The parser also intercepts any call to main() and replaces it with a call to __original_main(). In `ompi.c`, after building the AST, a new main() is generated, which starts by performing some OMPi-specific initializations and concludes with calling __original_main(). Finally, the original main() function is forced to have the two standard (argc, argv) arguments.[1]

2. The *replacement of threadprivate variables with pointers to them*, all over the code. This makes it smoother for later stages (when transforming) to handle threadprivate variables BUT it surely leaves the AST in an incorrect semantic state, which is good to remember.

What should be made clear is that the final AST *does not contain any information about scopes and variables*; for example, there is no link between a variable and its declaration. All such information which is implemented through the symbol table (which is empty when the AST built up is complete) must be re-created later, when traversing the tree.[2]

4 Symbols and symbol tables

Because compiling involves a lot of name (string) processing, and string operations are expensive, each name discovered by the parser is converted to a *symbol*, which is just a pointer that points to the actual string. Symbols are handled in `symtab.c` and are stored in a hash table (called "allsymbols") so as to locate them easily. Given a string X, a call to Symbol(X) uses a hash function operating on X and returns the position in the hash table where the symbol can be found. If the symbol is not there, a new entry is created for X.

The real value of symbols comes when manipulating names such as identifier names. A symbol table holds identifiers. It does not hold the actual strings but their symbols. It is, too, a hash table only now the hash function is calculated from the symbol, not from the name (which is quite faster). Among other things, for each identifier the following information is held at each symbol table entry ('stentry'):

- name space—Many different entities in a program are considered identifiers, e.g. variables, user types, labels and so on. The 'space' field is for holding the type (or 'namespace') of the identifier in question. The C language defines various name spaces, such as variables (IDNAME), user types (TYPENAME), structs/unions (SUNAME), enums (ENUMNAME), labels (LABELNAME), functions (FUNCNAME).

- scopelevel—Each symbol also has a *scope* it belongs to. A new scope starts whenever a compound statement (beginning with {) is met in the parsed code and ends when the

[1]If ompi is passed the "`--nomain`" option then the new function is actually named "__ompi_main()", instead of "main()". This is needed in order to accommodate special runtime libraries that provide their own main() function.

[2]Actually, the parser does not leave the symbol table completely empty after building the AST; the symbol table still contains all *global* declarations. This is the reason for the symtab_drain(stab) call in `ompi.c` after the call to parse_file().

closing brace (}) is met. Scopes are numbered starting with scope 0, which is the global scope, containing all global identifiers. A scope i is nested within scope $i-1$; all identifiers up to and including scope $i-1$ are visible in scope i. Whenever a scope closes, all its identifiers are removed from the symbol table.

- isarray, isthrpriv—They flag whether an identifier is non-scalar or threadprivate.
- spec, decl, idecl—These are pointers to the parts of the AST where the specifier and the declarator of the identifier's declaration lie. These are needed so as to be able to clone an identifier in various places when transforming the AST. It must be noted that if the original declarator is combined with an initializer, 'decl' does *not* point to the initializer; it points to the bare declarator (and because there is no 'parent' pointer for non-statement AST nodes, there is no way of accessing the initializer from the 'decl' field). The 'idecl' field points to the full initdeclarator node, so it provides access to both the bare declarator and the initializer. Although 'idecl' would be enough, the bare declarator is needed in 99,9% of the cases and this is why 'decl' is a handy field to have. If there is no initializer, 'idecl' is NULL.
- vval, ival—These are just extra fields that are only for saving ad-hoc info. The only situation where these fields are used is when analyzing the variables in OpenMP data clauses where for each variable ival holds the type (private, firstprivate, etc, see `x_clauses.c`) and vval holds the operator in the case of a reduction clause.

All identifiers met during parsing are entered in the symbol table of OMPi, called 'stab'. Each identifier joins at the current scope, which is kept as stab→scopelevel. Now, given a symbol and its name space, a call to symtab_get() returns the most recent identifier (i.e. the identifier that appears in the most recent scope), if any. This is because the hash table buckets are LIFO structures. If two identifiers have the same symbol, the most recent one is the last to enter the bucket.

A basic problem arises when the current scope, say scope i closes (a } was met in the parsed code) and all symbols in the scope must be removed. To avoid searching all table entries for symbols whose scope level is equal to i, the symbols are also chained, forming a stack whose top is the most recently met symbol (field stab→stacknext). Whenever a new scope is started (scope_start()) a special symbol is inserted in the table (called 'scopemark'). To remove all symbols of current scope, (scope_end()) we start from the stack's top and remove all symbols following the stacknext links until scopemark is met.

Remember that the AST is *not* annotated, so for example identifiers met inside expressions do not have any link to their declarations. The connection among identifiers, their declarations and scopes is only available through the symbol table. OMPi's symbol table is dynamic and scopes cause the addition and deletions of identifier symbols. This means that after parsing is complete, the symbol table has lost all its information about identifiers, scopes and declarations (only the global scope is still there and gets drained afterwards). In compiler jargon, OMPi's symbol table is 'imperative', or non-persistent (non-functional). Consequently, if one needs to examine, analyze, alter and/or transform the AST, there is no way of having the information needed unless *the symbol table is recreated while traversing the* AST, exactly the same way it was built-up when parsing the file. This is exactly was occurs in `ast_xform.c`.

5 Analyzing variables

When transforming the AST, it is sometimes necessary to analyze the nature and the usage of variables in certain parts of the tree. For example, we may need to discover all the global or threadprivate variables used within a function. Such requirements are covered by the functions in `ast_vars.c`. In particular, the following functions:

```
ast_paracon_find_sng_vars()
ast_find_gtp_vars()
ast_find_sgl_vars()
ast_find_allg_vars()
```

discover, correspondingly, all shared non-global, all global threadprivate, all shared global and all global variables in a given subtree. They traverse the tree and visit every node in order to find occurrences of variables that are of the given type. There is a flag that can be passed to these functions: if it is set to 1, then all complying variables are treated as if they were pointers to the original variables, i.e. `x` is replaced everywhere by `(*x)`. See later (Sections 6.5 and 7) for the reason behind this.

The above functions record all discovered variables in one of the following symbol tables:

```
sng_vars, gtp_vars, sgl_vars
```

Symbol tables are the preferred data structures because they allow fast insertions and lookups.

A variable is assumed to be utilized if it appears in any expression, in any scope, unless it is shadowed at a nested scope. Thus, for example, in the following piece of code, x will be recorded by ast_find_allg_vars() if it is called to examine the AST at point A, while it won't be recorded if it is called to examine the AST at point B.

```
int x;
f() {
  x = 1;          // A
  {
    int x;
    x = 2;        // B
  }
}
```

Variables that are referenced in OpenMP data clauses within the examined portion of the AST are treated accordingly (ast_omp_dataclause_vars()). In particular they are not considered at all when they appear in a private clause, because as we will see in the next section, local copies of those variables are used instead of the original ones.

Function ast_paracon_find_sng_vars() is only called when transforming parallel constructs (Section 6.5), ast_find_gtp_vars() is called when transforming threadprivate variables (Section 7) and ast_find_sgl_vars() is used when compiling for the process model (Section 9.1).

6 Transforming the AST

After building the AST, and if there exist OpenMP pragmas in the code, the AST is transformed through ast_xform(), called from `ompi.c` and implemented in `ast_xform.c`. Just before this call, there are some necessary definitions (e.g. runtime-library definitions) inserted in the AST because generated code depends on them.

In `ast_xform.c`, ast_stmt_xform() really does almost nothing for normal tree nodes. All it takes care of is correctly recreate the symbol table scopes whenever meeting any declarations (see, though, Section 6.8). It only takes some real action when it meets OpenMP construct nodes (ast_omp_xform()).

`ast_xform.c` also keeps two other trees that transformation functions may utilize: newglobals and newtail. The first (second) one will hold statements that are generated by some transformation functions and have to be placed at the top (bottom) of the generated code, such as e.g. new global variable declarations. The function newglobalvar() takes a declaration tree, adds it to the newglobals tree and in addition declares the new global variable by inserting it in the global scope of the symbol table (using symtab_insert_global()).

All OpenMP constructs except parallel, for, sections and single are transformed within `ast_xform.c`, through ast_omp_xform(). The transformation is quite simple and involves removing the subtree rooted at the node in question and replacing it with a new tree of code. Before each such transformation xc_validate_clauses() (implemented in `x_clauses.c`) is called to check the validity of construct clauses.

Most constructs contain an implicit barrier. However, if there are two such constructs nested within each other, it may be the case that not both barriers are needed; one could be enough, avoiding thus unnecessary runtime overhead. The function xform_implicit_barrier_is_needed()

discovers such situations and is used just before actually placing the barrier call in the transformed tree. What it does is check whether the construct is the last statement in the body of another construct which does have a barrier.

The transformation of the other four constructs is handled in files `x_parallel.c`, `x_single.c`, `x_sections.c` and `x_for.c`. For those constructs, the *body of the construct is first transformed*, through xform_ompcon_body(), before the corresponding transformation function is called. In order to transform the body, however, the correct environment must first be created, since the four OpenMP directives may change the visibility of variables through data clauses such as private, firstprivate, etc. Thus, what xform_ompcon_body() does is first open a new scope, correctly *declare* those variables that must be privatized, and then call ast_stmt_xform() to transform the body. The scope closes immediately after that, and the directive's transformation function can be called with an already-transformed body.

The major complication with these four constructs (lets call them C4) results from the existence of data clauses (shared, private, firstprivate, lastprivate, reduction, copyin and copyprivate). Data clauses require some variables to be shared or privatized, and possibly initialized from other variables, etc. Consequently, when transforming C4, there are usually new declarations inserted in the top of the construct's body plus possibly new initialization statements. We may also need to insert statements at the bottom of the construct's body (e.g. for lastprivate variables).

Consequently, C4 transformations begin by collecting and analyzing all data clause variables, producing declaration and initialization statements if needed. There are two ways to do this, both of which are actually used. The first is to take each data clause in turn and for every variable it contains, perform the appropriate action. The other is to do it in two steps: first gather all data clause variables in a set, remembering what clause each variable appears in, and then perform the actions using this set. The first method is simpler and is used when transforming a single construct (a call to xc_ompdir_declarations() from xform_single()) but the second is more powerful and is used when transforming the other three constructs (collect all variables with xc_validate_store_dataclause_vars() and declare as needed with xc_stored_vars_declarations()). More on this in a while.

In case there are firstprivate variables present, one more complication arises because such variables must be initialized with the value of the original variable. If such a variable is scalar, the initialization can easily by included within the new declaration. This cannot be done, though, if it is an array variable. Its initialization is explicitly performed through memcpy() statements which are inserted at the top of the construct's body. All those statements are generated by xc_ompdir_fiparray_initializers() (implemented in `x_clauses.c`).

In what follows we discuss briefly the C4 transformations. For a full treatment, the reader is supposed to study the corresponding files *and* run `ompicc` with the `-k` argument on simple input programs so as to see what exactly the transformations produce.

6.1 Privatizing variables

As already mentioned, variables that appear in most data clauses are redeclared as local ones within the scope of the construct. For example,

```
extern int x;
#pragma omp <whatever> private(x)
  x = 1;
```

yields code as follows:

```
extern int x;
{ int x;
  ...
  x = 1;
  ...
}
```

This involves cloning of declarations (xform_clone_declaration() in `ast_xform.c`), which copies the specifier and the declarator part of the declaration of the original variable. Notice that the specifier is copied using ast_spec_copy_nosc() (see `ast_copy.c`), so that it leaves out all storage class specifiers (extern, static, auto, register).

The same happens for firstprivate variables, only in this case a new temporary variable is also created so as to initialize the cloned variable from the original one (xc_firstprivate_declaration() in `x_clauses.c`). Thus:

```
int x;
#pragma omp <whatever> firstprivate(x)
  x++;
```

yields code as follows:

```
int x;
{ int _fip_x = x, x = _fip_x;
  ...
  x++;
  ...
}
```

For arrays, as already mentioned, special precautions are needed since there cannot be an initializer included in the declaration:

```
int x[10];
#pragma omp <whatever> firstprivate(x)
  x[0] = 1;
```

yields code as follows:

```
int x[10];
{ int (*_fip_x)[10] = &x, x[10];
  memcpy(x, _fip_x, sizeof(x));
  ...
  x[0] = 1;
  ...
}
```

Exactly the same procedure is followed for lastprivate variables, only now the temporary ones are prefixed with `_lap_` instead of `_fip_` and, of course, there are no initializations needed (there will however be assignments back to the original variable inserted in appropriate places of the produced code).

Finally, reduction variables are treated in the same way, using a temporary variable prefixed with `_red_` (xc_reduction_declaration() in `x_clauses.c`). However, reductions require additional code for (i) performing the reduction operation on the original variable and (ii) declaring a global lock on which the code of (i) is based (xc_ompdir_reduction_code()).

6.2 Transforming a single construct

This is the simplest of them all. The original code:

```
#pragma omp single private(..) firstprivate(..) copyprivate(..)
  <body>
```

gets replaced by:

```
if (ort_my_single(W)) {
  <declarations from private/firstprivate>
  <firstprivate arrayinitializers>

  <body>

  ort_broadcast_private()
}
ort_leaving_single()
ort_barrier_me()
ort_copy_private()
```

where W is 0 if the implicit barrier must be used. In this case, one more barrier call is added after line 9. W becomes 1 if there is an explicit nowait clause present or xform_implicit_barrier_is_needed() returns false.

Basically, all variables from the private and firstprivate clauses get declared (line 2, through xc_ompdir_declarations()), with firstprivate ones initialized in-place. If there are array firstprivate variables, memcpy() statements are needed for their initializations as explained above (line 3, through xc_ompdir_fiparray_initializers()). Finally, if there are no copyprivate variables, lines 5, 8 and 9 are not generated.

6.3 Transforming a sections construct

The transformation for sections is based on the following scheme: each section becomes a case in a switch statement and the switch statement becomes the body of a loop that asks for the next section to execute. Thus:

```
#pragma omp sections private(..) firstprivate(..) lastprivate(..) reduction(..)
{
  <body0>
  #pragma omp section
    <body1>
  #pragma omp section
    <body2>
  ...
}
```

gets transformed to:

```
{
  <declarations from private,firstprivate,lastprivate,reduction>
  int caseid_ = -1, inpar_;

  <firstprivate arrayinitializers>
  if ((inpar_ = (omp_in_parallel() &&  omp_get_num_threads() > 1)) != 0)
    ort_entering_sections(W,N);
  for (;;)
  {
    if (inpar_) { if ((caseid_=ort_get_section()) < 0) break; }
           else { if ((++caseid_) >= N) break; }

     switch (caseid_)
     {
       case 0:  <body0> break;
       case 1:  <body1> break;
       case 2:  <body2> break;
       ...
     }
    )
  }

  <reduction code>
  if (inpar_) ort_leaving_sections();
}
```

where N is the total number of sections and W, as in single, becomes 0 when the implicit barrier must be used. The inpar_ variable is 0 when there is only 1 thread to execute the region. In such a case, the thread loops incrementing caseid_ by 1 each time, thereby executing the cases of the switch one by one till the last one.

The cases of the switch statement are generated by sections_cases(). Note also that at the last case of the switch, there are extra instructions inserted if there are any lastprivate variables used (xc_ompdir_lastprivate_assignments()). In addition, a synchronization call to ort_wait_all_entered() is emitted if a variable is both firstprivate and lastprivate; this is to ensure that the thread which executes the lastprivate assignment won't do so before all the other threads have initialized their own firstprivate variables.

Here, the data clause variables are declared using the two-step method: all variables are collected with xc_validate_store_dataclause_vars() and are declared as needed with xc_stored_vars-_declarations(). The reason for this is the presence of both firstprivate and lastprivate clauses. If a variable appears in both clauses (allowed in OpenMP) then the xc_ompdir_declarations() call that was used in xform_single(), would result in duplicate declarations. Thus variables are collected first from all clauses and checks are being made for variables that appear in both firsprivate and lastprivate clauses. These variables are marked as 'firstlastprivate'.

xc_validate_store_dataclause_vars() stores the variables in a symbol table—this is used purely because it is a very handy data structure and allows fast insertions and lookups. The type of clause each variable appears in is stored in the 'ival' field of the entry (see Section 4). In case of reduction variables, one must also remember the operator involved; the 'value' field of the entry is used for that.

6.4 Transforming a for construct

File `x_for.c` works almost exactly as `x_sections.c` as far as data clauses are concerned; it is lengthier mostly because is generates a lot of new code.

First, the loop index, the initial value, the upper bound and the increment are extracted from the for statement node (analyze_for()). All this is done for each associated loop if a collapse() clause is presented. Also, the schedule type and chunksize are discovered (if given). Based on that, the iteration chunks that will be given to each thread can be determined. We won't expand more on this here; the reader is referred to the "OMPi's runtime-library interface for FOR directives" document. We only note that different code is produced whenever a static schedule is requested; this is because in such a case the iteration space for each thread can be predetermined and does not require repeated calls to the runtime library.

6.5 Transforming a parallel construct

This is the most crucial and most complicated transformation. The idea is to remove the body of the construct and place it in a new function and then create threads that will execute this function. This technique is known as *outlining* and would be pretty straightforward, if it weren't for data clauses and especially shared, non-global (sng) variables.

Shared variables need no special treatment if they are global; this is because global variables are by nature shared among threads. Hence, within the new function they can be used as is. However this is no longer possible if a variable is shared but does not belong to the global scope. It is a variable that exists in the private stack of a running thread, which is about to spawn new threads. In order for this sng variable to be shared and accessed from all spawned threads, there must be explicit pointers passed to them that point to the original variable. This also means that the body of the construct must be modified: all appearances of the sng variable must be replaced with pointers that point to the original variable.

Because of the peculiarities of this construct, data clause variables are not handled through `x_clauses.c` but through similar facilities within `x_parallel.c`. All data clause variables are collected using xp_store_dataclause_vars(), similarly to xc_validate_store_dataclause_vars().

Then all used sng variables are discovered by examining the body of the construct, using ast_paracon_find_sng_vars() (see Section 5). The call to ast_paracon_find_sng_vars() is made with the transform flag set to 1 so that all appearances of the sng variables are treated as pointers (`var` is replaced by `(*var)`). Consequently, the body is correctly modified, in anticipation of pointer declarations that will point to the original variables. The discovered variables are cross-checked against the data clause variables collected by xp_store_dataclause_vars() (xp_parallel_check_shared()); this is because, according to OpenMP, all shared variables should be explicitly enlisted in shared() clauses if the default(none) clause is also present.

Given all sng variables, pointers to them are made available to the threads as follows: a C struct named `_shvars` is created, whose fields will contain pointers to the sng variables. The struct is initialized with the address of the sng variables (xp_parstruct_initializer()) and is then given to the threads through the runtime library call ort_execute_parallel. This function has three arguments: the first is the number of threads to be created, the second is the function to be called by the threads and the third is `_shvars`. Each thread gets access to the structure by calling ort_get_shared_vars from within the function they execute. Then, using the fields of the structure they get access to the original sng variables. The code that is produced for:

```
#pragma omp parallel
  <body>
```

is roughly as follows:

```
/* pragma replaced as follows: */             /* thread function created */
{                                             void *_thrFunc_(void *_arg)
  <_shvars struct>                            {
  ort_execute_parallel(-1,_thrFunc_,&_shvars);   <struct specifier> *_shvars =
}                                                          ort_get_shared_vars();
                                                <body with sngs treated as pointers>
                                              }
```

The -1 in ort_execute_parallel means that no specific number of threads is requested; it is replaced by the num_threads clause expression, if present.

Because there may exist many parallel regions, the name of the thread function should be unique, so its name is actually "_thrFuncN_" where N is a counter incremented by 1 for each parallel region met (see new_thrfunc() and thrfuncname()). Because the user program may consist of multiple translation units, each one possibly having its own parallel regions, the thread functions are declared with *static* storage class, avoiding thus name clashes. Finally, due to the possibility of recursion, a thread function must be inserted *after* the function that contains the parallel construct while its declaration should lie *above* that function (see below how this is achieved).

Putting all the above together, for the following piece of user code,

```
int a;
f() {
  int b, c, d;
  #pragma omp parallel private(d)
    a = b+c+d;
}
```

here is what is produced:

```
int a;                      /* shared global (sgl): nothing to be done */

static void * _thrFunc0_(void *);   /* thread function declaration */

f() {
  int b, c, d;
  struct __shvt__ { int (*b); int (*c); } _shvars = { &b, &c };

  ort_execute_parallel(-1, _thrFunc0_, (void *) &_shvars);
}

static void * _thrFunc0_(void *_arg) {
  struct __shvt__ { int (*b); int (*c); }
      *_shvars = (struct __shvt__ *) ort_get_shared_vars();
  int (*b) = _shvars->b;    /* sng var */
  int (*c) = _shvars->c;    /* sng var */
  int d;                    /* private() var */

  a = (*b) + (*c) + d;      /* original body with sng pointers */
  return (void *) 0;        /* dummy return statement */
}
```

There are only a couple more pieces to complete the puzzle. If the construct contains an 'if(condition)' clause, the code produced is:

```
if (condition)
  ort_execute_parallel(...);
else
  ort_execute_serial(...);
```

so that the condition is checked at runtime and if false, the library will call the function only through the encountering thread.

The parallel directive is the only directive to accept a copyin clause; the corresponding declarations and initializations of pointers to threadprivate variables are handled by xp_copyin_declarations(). For non-global threadprivate variables, pointers to the original ones are provided through `_shvars`.

As we have already seen, firstprivate and reduction variables produce local declarations but also need access to the original variables. In order to avoid the extra temporary variable trick used in other constructs (`_fip_` and `_red_` variables in Section 6.1), all original firstprivate and reduction variables are added to the sng variable collection (xp_sharedng_add_fpredgvars()), even if they are global. This will provide access to them through the `_shvars` struct, eliminating temporary variables.

After the new thread function is fully created, xform_parallel() calls xfrom_add_threadfunc() (implemented in `ast_xform.c`) to place the thread function definition (declaration) in the appropriate spot, right below (above) the function that contains the parallel construct (call it F), as we already mentioned. Because the transformation phase, after finishing with F, will continue with the next function in the original AST, the newly inserted thread function will never have the chance to be transformed. For this reason, `ast_xform.c` maintains a list of thread functions (thrfuncs) that will be placed in the AST after the main transformation phase is completed. For each function, it stores its definition tree plus a pointer to the function (F) it must be placed below. After the transformation of the AST is complete, ast_xform() calls xform_thread_functions() which transforms all created thread functions and place_thread_functions() which inserts them in the right places.

6.6 Transforming combined constructs

The transformation of a combined parallel-for or parallel-sections construct is actually quite simple. Such situations are handled within ast_omp_xform() itself as follows: the combined parallel-sections (-for) construct gets replaced by a new parallel construct which has as it body a sole sections (for) construct, which in turn has as its body the original body of the combined statement, e.g.:

```
#pragma omp parallel for <clauses>
  <original body>
```

gets replaced by:

```
#pragma omp parallel <some clauses>
  #pragma omp for <some clauses>
     <original body>
```

The only thing that has to be taken care of is splitting the clauses between the two new constructs: xc_split_combined_clauses() (implemented in `x_clauses.c`) does exactly that. The parallel construct takes all the clauses it can.

6.7 Transforming a task construct

To be written (it is similar in spirit to the transformation of a parallel construct).

6.8 Transforming declarations

Normally, declarations need no transformation—one only needs to add the declared identifiers to the symbol table's scope. There are only two cases where this is not enough:

- In declarations that utilize user-defined types and/or structs/unions/enums. So all declarations are handled by xt_declaration_xform() (called in `ast_xform.c`, implemented in `x_types.c`) and the whole story behind it is described in Section 8.
- In the declaration of function parameters, *when transforming function definitions* (statements of type FUNCDEF). In such cases, array parameters are substituted with their pointer equivalents, through xt_decl_array2pointer(). This is necessary because identifiers (along with their declarations) may be cloned by the transformation process in many places and array parameters may cause problems, as e.g. below:

```
int f(int x[10]) {              /* x is array */
  #pragma omp parallel shared(x)
    x[3] = 1;                   /* dummy code */
}
```

  As explained above, this will produce a new thread function, while x's declaration is cloned in 3 places so as to obtain a pointer to it:

```
int f(int x[10]) {              /* x is array */
  {
    struct { int (*x)[10]; } _shvars = { &x; };
    ort_execute_parallel(-1, _thrFunc1_, &_shvars);
  }
}

void *_thrFunc1_(void *_voidarg) {
  struct { int (*x)[10]; } _shvars = ort_get_shared_vars();
  int (*x)[10] = _shvars->x;
  (*x)[3] = 1;
  return (void *) 0;
}
```

  Of course, this crashes most systems (see why it is wrong??). So, before transforming, a function's array parameters are correctly replaced by their pointer equivalents, as in:

```
int f(int *x) {   /* equivalent to x[10] in this case */
  #pragma omp parallel shared(x)
    x[3] = 1;                   /* dummy code */
}
```

7 Threadprivate (tp) variables

As mentioned earlier, the parser changes every appearance of a threadprivate variable into a pointer (since it cannot appear in other data clauses such as private(), there will never be a need to use the original var instead of the pointer). Threadprivate variables are marked in the stab through the field 'isthrpriv'. The idea is that the original tp variables will be *renamed* and that new pointer variables with the original name will be declared and used.

File `x_thrpriv.c` handles tp variables. Tansforming an OpenMP threadprivate directive results in a call to xform_threadprivate(), which actually renames and creates the pointer declarations needed. Consequently, for every variable X in a threadprivate clause we:

- Rename the original variable to "tp_X_".

- Declare a pointer X, which will be initialized to the address of the thread-specific copy of the original variable (through a call to ort_get_thrpriv()).
- Declare a global key like "tp_X_key_" to make the thread-specific storage possible (see the pthreads manual). Notice though that for non-global tp (ngtp) vars the name of the key is "tpngN_X_key_", where N is a unique numeric id, since there may exist many such vars with the same name, in different scopes (and all of them would have used the same key).

The second item above is easy for static block-scope variables (ngtp); the declaration is made in-place. It is tougher for global tp (gtp) vars: the declaration/initialization *must be done in every function that references the gtp variable.* This is the reason behind the function tp_fix_funcbody_gtpvars(). This one takes the body of a function, discovers all (gtp) vars (utilizing ast_find_gtp_vars() in `ast_vars.c`) and declares pointers to them at the top of the function's body. It is called from `ast_xform.c`, when transforming a function definition.

Here are some finer points:

- A subtle detail is that the original var is renamed to "tp_X_" (actually, to whatever tp_new_name(var) returns) but this is *only done in the original declaration.* I.e.:
  - When cloning the declaration to declare the pointers, one must remember to change the name in the copy back to the original tp var name
  - The symbol table entry is NOT changed. I.e. the variable is still known as "X"; the new name has never entered the symbol table.
- In `x_single.c` and `x_parallel.c`, the produced code for copyprivate calls, structure initializations etc use the tp vars as-is (i.e. not through the address operator) since they are already pointers.
- In `x_parallel.c`, tp vars are handled as follows:
  - there is a function to handle the copyin variables (xp_copyin_declarations()).
  - all non-copyin ngtp variables that are used within the body are handled through xp_sharedng_declarations() (i.e. like shared non-global) but the produced declarations contain calls to ort_get_thrpriv() to properly initialize them
  - all non-copyin gtp variables are ignored since they will be appropriately handled by tp_fix_funcbody_gtpvars() when the new thread function definition is later transformed.

8 User-defined types and struct-like declarations

We face one problem which is solely due to the omp parallel transformation. This is the only transformation that may move / copy code to new functions, along with all necessary variable declarations. If variables that were declared in the original scope have to be redeclared in a different, non-nested scope then *if those variables have user-defined types, the corresponding type definitions must be visible to the new scope.* This means that not only variables but also `typedef`s and related stuff should also be carried around.

A simple strategy is to make all user-defined types global. This way, there is potentially no visibility problem. This scheme is certainly far from elegant and in addition it is problematic in many aspects.

In OMPi we follow a different approach (file `x_types.c`): *we substitute all user-defined types with their primitive constituents*, so in essence at the end there exist no user-defined types in the AST. This way, variables have no type dependencies and can move around (almost) freely. The substitutions occur through xt_declaration_xform(), which is implemented in `x_types.c` and is called from `ast_xform.c` whenever a declaration is met. For example, this:

```
typedef int type1;
typedef type1 *type2;
static type2 x, *y;
```

is transformed to this:

```
static int *x;
static int *(*y);
```

i.e. all user types are (recursively) substituted. Also notice that during this transformation a multi-variable declaration (`static type2 x, *y;`) is broken into single-variable ones (function xt_break_multidecl()).

The original `typedef`s are removed from the AST but kept around (in `retiretree`) since there may exist more declarations later on, which depend on those types. For the same reason, the names of those types are not removed from the symbol table. Remember, that all symbols (hence user types, too) have pointers to their original specifier and declarator tree nodes.

The substitutions occur through `xt_barebones_substitute(spec,decl)`, given the specifier and the declarator part of the the declaration. The substitution is relatively simple: the user type in `spec` is replaced by the specifier of the user type definition and `decl` replaces the user type name in the declarator of the user type definition (huh?!). Thus (utd stands for 'user type definition'):

```
typedef type1 *type2; // utd: type1 is the specifier
                      //      *type2 is the declarator
static type2 *y;      // spec = type2, replaced by utd's spec
                      // decl = *y, replaces utd name in utd's decl
```

becomes:

```
static type1 *(*y);
```

There is a possibility that `decl` may contain stuff that are based on other user-defined types, for example:

```
type2 func(type4 x, type5);
```

Consequently, before the actual substitution takes place, `decl` is checked recursively through xt_barebones_decl(). Just notice in the above that `type4 x` is a *concrete or direct* parameter declarator while `type5` is an *abstract* one since it just mentions the type and not the actual name of the parameter. Substitutions of course take care of such situations (e.g. concrete_to_abstract_declarator()).

One last thing that gets transformed is declarations that combine the identifier with a struct, union or enum ("sue") definition, such as:

```
struct s { int f1; char *f2; } x;
```

The specifier here is '`struct s { int f1; char *f2; }`' which defines a *named* structure. Such declarations are broken into two statements (using xt_suedecl_xform()); the first one defines the sue and the second one uses the named sue to declare the identifier:

```
struct s { int f1; char *f2; };  // no declarator part here
struct s x;                      // the identifier declaration
```

The same occurs for *unnamed* sue, which acquire a unique name during the process:

```
struct { int f1; char *f2; } x;  // unnamed struct
```

becomes:

```
struct _unnamed3_ { int f1; char *f2; };  // generated name
struct _unnamed3_ x;
```

Of course, all sue fields get checked for user types and may be substituted if needed.

In conclusion, the type of a variable ends up being either one of the base types or some named sue entity (or a combination of both). Thus, the only thing that needs to be done when moving variables and declarations around is to carry along any named sue the declarations depend on.[3]

[3]Because the author got too lazy at some point, instead of checking for sue dependencies, at the top of every new thread function all visible non-global sue definitions are repeated obliviously.

An esoteric note: Due to the above substitutions, the code that is generated (e.g. in `ast_xform.c`) cannot use any non-primitive types. E.g. even though the type `FILE` may have been defined (the user code has #included stdio.h), since `FILE` gets retired there should be no generated declaration like ‘`FILE *f;`’—it will cause an error since `FILE` won’t be defined there.

9 Miscellanea

9.1 The thread and the process models

The code produced by OMPi follows either the *thread* (by default) or the *process model* (TM, PM). In TM the parallel execution ‘vehicles’ are threads, which means that *all global variables are by nature shared between them.* This explains why one needs the threadprivate directive: it is the only way to make some global variables non-shared. I.e. wrt global variables, everything is shared unless explicitly stated otherwise (through threadprivate directives). This is exactly what OpenMP requires, so the TM model is the natural choice.

Nevertheless there are some cases where the execution vehicles are not threads. For example, one could support process parallelism, through fork() calls. Another example is supporting OpenMP over software DSM environments to make shared-memory programming available to clusters. However, if the execution vehicles are processes then *all global variables are by nature private to each process*, i.e. there is nothing shared between them. This is good for global threadprivate variables since they are process-private anyways. But to follow the OpenMP model, one has to *explicitly make all global non-threadprivate (gntp) variables shared between the processes.* As a result, the code that OMPi produces for the PM model is somewhat different than that for the TM model.

We like to have only 1 parser and 1 runtime system for OMPi, even if it supports 2 execution models. Consequently, the differences in the produced code as well as the ORT library should be as few as possible. Here is exactly what OMPi does when it is called to compile for PM (i.e. when executed with the `--procs` option):

- The threadprivate directive is essentially ignored since the marked variables will be private to each process anyway.

- All gntp variables are turned into pointers (through sgl_fix_sglvars() in file `x_shglob.c`) and are replaced accordingly all over the code (done by ast_find_sgl_vars(), which is called by ast_statement_xform() in `ast_xorm.c` when transforming FUNCDEF bodies) Those pointers will somewhere, sometime, somehow be explicitly initialized to point to some shared memory area.

  where: all global variable initializations are gathered in one function called _init_shvars_() which is placed at the bottom of the produced code. All this is implemented through sgl_fix_sglvars() in file `x_shglob.c`, which is called by ast_xform(), after all other transformations on the AST have been completed.

  when: all actual allocations should be made before fork()ing new processes. Consequently the produced _init_shvars_() must be called quite early by somebody. Keep in mind that the user may possibly compile/link many different modules, each one having its own _init_shvars_() function. This precludes the possibility of injecting allocation calls at the beginning of the main() function (since main() will only be present in one of the modules). Consequently we face the problem of *making some functions be called before the execution of main().* There is no really portable way of doing this.[4] The solution we adopted is GCC-specific so after all OMPi supports the PM only when the base compiler is GCC, or a few others for which a similar mechanism exists. GCC-produced executables call all functions labelled as ‘constructors’ before executing the main() function. All one has to do then is label _init_shvars_() as a constructor, and this is done through the __attribute__ extension:

[4]well actually there exists one Unix-specific and far-from-elegant way, which was used in early versions of OMPi to support threadprivate variables

```
static void __attribute__ ((constructor)) _init_shvars_(void);
```

For Sun Studio compilers, the same effect is achieved by listing the function in a '#pragma init' line:

```
#pragma init(_init_shvars_)
```

how: the actual allocations should be done through mmap(), shmget(), etc. calls. Actually, since this is runtime-specific, ORT provides such calls: ort_shared_allocate(void **ptr, int size, void *initvalue). Thus _init_shvars_() contains a series of ort_shared_allocate() calls, one for each global variable. ORT could do the allocations immediately but it really defers them for a later time; ort_shared_allocate() just marks the allocation requests but it actually grabs memory and inits the pointers later, when ort_initialize() is called from main(). This has the additional benefit of making only 1 allocation in total, since the total size of all variables will be known in advance. This is however a runtime library issue so we stop the story here.

The 'initvalue' pointer contains the address of the initial value of the variable. If the variable was declared with no initializer, initvalue is NULL; otherwise, the parser moves the initializer code to a new global variable (new_insertdummyinitvar()) and that variable's address is passed on initvalue. I.e. '`int a = 10;`' results in

```
int _sglini_a = 10, *a;
```

and a call to ort_shared_allocate(&a, sizeof(int), &_sglinit_a).

That' it. Nothing else is needed. The rest of the code produced is the same for TM and PM. A good question here would be: what about shared variables that are non-global, as in:

```
{
  int x;
  #pragma omp parallel shared(x)
    x = ...;
```

We will answer this only partly here. There are two ways to handle this issue. One is to do nothing (!) and another is to produce code that explicitly: (a) allocates space in shared memory, (b) copies the value of the original variable there just before fork()ing, (c) frees the space after the team seizes execution. The second scheme has two disadvantages. First, the produced code would no longer be similar for the two execution models. Second, it is clearly slow. So OMPi follows the first scheme: it does nothing! It lets the runtime library handle this issue. How? In a few words the trick is to have the process stacks reside in shared memory. Then all local variables will be shared anyways.