Basic skeletons in llc

Antonio Dorta, Pablo López, Francisco de Sande *

Departamento E. I. O. y Computación, Universidad de La Laguna, 38271-La Laguna, Canary Islands, Spain

Received 30 December 2005; received in revised form 19 May 2006; accepted 7 July 2006
Available online 2 October 2006

Abstract

llc is a high-level parallel language that provides support for some of the most widely used algorithmic skeletons. The language has a syntax based on OpenMP-like directives and the compiler uses direct translation to MPI to produce parallel code. To evaluate the performance of our prototype compiler we present computational results for some of the skeletons available in the language on different architectures. Particularly in the presence of coarse-grain parallelism, the results reflect similar performances for the llc compiled version and ad hoc MPI or OpenMP implementations. In all cases, the performance loss with respect to a direct MPI implementation is clearly compensated by a significantly smaller development effort.

Keywords: Parallel programming; High-level language; Algorithmic skeletons; Compilers; OpenMP; MPI

1. Introduction

One of the main reasons that hinders the parallelization of applications is its development effort. Except for experts in the field it is not an easy task to parallelize a sequential application. While in the recent past, parallel architectures have made substantial progress in speed, scalability, number of processors, etc., the software tools we use today are not significantly different from those existing a decade ago.

MPI [1] is currently the dominant tool to develop parallel applications. When programming in MPI the user has to deal with low-level aspects of programming such as synchronization, explicit communications and tasks distribution. Parallelizing a sequential application in MPI requires a considerable effort and expertise. In a sense, we could say that MPI represents the assembler language of parallel programming, you can obtain high performance but at the cost of quite a high development investment.

Skeletal parallel languages [2–5] have been one of the most successful approaches to fill the gap between development tools and user needs. Most parallel applications are based on well-defined patterns (skeletons)
which are applied and combined repeatedly. To develop a parallel application using a skeletal language, the user has to identify the pattern that fits its needs, and use the language facilities to express it. If highly optimized skeletons are provided to the user, the performance loss with respect to a direct implementation is compensated by a considerably smaller development effort.

LLC is a parallel skeletal language that provides support for four basic skeletons: forall skeleton, parallel sections, task farms and pipelines. The LLC syntax is based on OpenMP [6] which has emerged in the last years as the industry standard for shared memory programming. The language uses compiler directives to annotate the code. LLCOMP, the LLC compiler, is a source to source compiler that translates C code annotated with LLC directives into C code with MPI calls. The compiler uses the information present in the directives to produce the parallel code. The programmer includes application dependent information in the parameters of the directives. This information complements the basic patterns available in the language. By producing MPI code, LLCOMP ensures the portability of the application, which can be executed in distributed or shared memory architectures.

We do not provide skeletons through a library, an approach that has been successfully followed in other developments [7,5,8]. Instead, we have extended a sequential language with new constructs, to express skeletal parallelism. While other groups [9,10] base their implementation in functional languages, we use C as target language due to the popularity of imperative languages among the community of scientific applications developers. The main reasons to choose OpenMP-like pragmas to provide the parallel skeletons in LLC were the reduced development effort and simplicity of this approach. While it is true that in order to achieve optimal performance in parallel architectures, the programmer has to write code specific to each architecture, we cannot ignore the fact that portability is also a major issue if our target user is not a computer science specialist. It is uncommon for the average researcher to develop and run applications in different platforms due to the complexity of maintaining different sets of code for the same application. Given the high accessibility of sequential machines, the average researcher will choose to remain in the sequential world as long as she does not have access to tools that allow for portability. Therefore, it is necessary to find a compromise between performance and portability.

A fundamental aspect to understand LLC semantics is the processor set concept. LLC is based on the OTOSP (One Thread is One Set of Processors) model. Although the reader can refer to [11] for detailed information, a brief description of the basic concepts is given next. In the OTOSP model all the processors executing a program are organized in processor sets. All the processors in a set share the same view of memory and execute the same program. Processor sets divide into subsets as a consequence of the execution of a parallel skeleton. At the end of the execution of a parallel skeleton, all the processors in the subgroups executing the skeleton are joined again to form the original set. At this point the processors use a partnership relation to communicate memory areas to processors in the complementary subsets.

The remainder of the paper is organized as follows. In Sections 2–5 we discuss the main skeletons available in LLC. For each case we present an example of use and describe the syntax of the corresponding construct. In Section 6 we discuss the main ideas behind the translation performed by LLCOMP. The experimental evaluation of the translation is presented in Section 7. Finally, we summarize a few concluding remarks and future work in Section 8.

2. Forall skeleton

Most of the parallelism in scientific codes can be expressed in the form of parallel loops. LLC provides a wide support for parallel loops, including nested parallelism and general parallel loops. We will use two sample codes to introduce the directives employed in the forall skeleton and at the end of this Section we summarize the syntax of the LLC forall skeleton.

Listing 1, illustrates the use of LLC to implement the well-known Mandelbrot Set computation. Observe that for this code, we have obtained a valid LLC application introducing only minimal changes on the original code. This is the general case when developing LLC applications.

The OpenMP construct in line 2 indicates that the for loop in line 5 will be executed in parallel. LLC is syntactically compatible with all the OpenMP constructs. Although LLC does not use most of the OpenMP clauses
it is also fully compatible with its syntax. This feature allows the development of valid llc code (and OpenMP as well) using a progressive approximation without breaking the semantics of the sequential code.

The purpose of the llc directives is similar to those of the OpenMP clauses. Most of the llc directives control the sharing attributes of variables inside the parallel construct. Using llc directives the user marks the memory regions that will be modified by each set of processors. These memory regions must be communicated to the remaining sets in order to guarantee memory coherence.

Listing 1: The Mandelbrot Set code in llc

```c
numoutside = 0;
#pragma omp parallel for default (none) reduction(+:numoutside) |
    private(i, j, ztemp, myid, nlocal, ilower, z, iupper), shared(nt, c)
#pragma llc reduction_type (int)
for (i = 0; i < npoints; i++) {
    z.creal = c[i].creal;
    z.cimag = c[i].cimag;

    for (j = 0; j < MAXITER; j++) {
        ztemp = (z.creal * z.creal) - (z.cimag * z.cimag) + c[i].creal;
        z.cimag = z.creal * z.cimag * 2 + c[i].cimag;
        z.creal = ztemp;
        if (z.creal * z.creal + z.cimag * z.cimag > THRESOLD) {
            numoutside++;
            break;
        }
    }
/* for j */
/* for i */
}
numinside = npoints - numoutside;
area_llc = 2.0 * 2.5 * 1.125 * numinside/npoints;
error_llc = area_llc/sqrt(npoints);
```

In Listing 1, the OpenMP reduction clause in line 2 indicates a reduction operation over the variable numoutside. This clause is completed with the llc reduction_type directive (line 4) which is used to specify the datatype of the reduction variable (int in this case).

In order to show different llc capabilities related to parallel loops, let's consider the synthetic code presented in Listing 2. As it is explained in [12], the limitations of the OpenMP pragma for directive do not allow a concise, scalable and portable implementation of the simple pointer-chasing loop shown in Listing 2 (lines 11 and 12). llCoMP takes this code and generates a translation, where the number of iterations required is determined by executing the loop control ignoring the loop body. Our experiments show that the overhead introduced with this approximation is negligible.

The exploitation of multi-level parallelism is another appealing feature of llc (see nested parallel loops in lines 11 and 17). If loops can be nested in any sequential programming language, it seems natural to make this property available in parallel languages. In the presence of nested parallelism, each task can generate new tasks which implies that each subset of processors will divide itself and will generate other subsets whenever enough processors are available.

A load balance problem may arise if the tasks load differ or if the number of processors does not match the number of tasks. If a measure of the work \( w_i \) per thread \( T_i \) is available, the processors/tasks distribution adapts to guarantee an optimal mapping. This is the purpose of the weight clause in line 9, Listing 2. In that example, the size values stored in each pointer are used as weights for the parallel for. The semantics of the weight construct is similar to that proposed in [13], but the mapping is computed differently. In this case we also have to deal with the additional problem of properly establishing the partnership relation. If there is no weights specification, the same work load is assumed for each task.
Listing 2: Pointer-chasing in llc

```c
void fnct (void) {
    nodeptr list, p;
    int i, j, val, v1[N*N], v2[N*N], v3[N*N];

list = generate_list();
initial (v1, v2, v3);
...
#pragma omp parallel for lastprivate (val)
#pragma llc weight (p->size)
#pragma llc result (p->data, p->size)
for (p = list; p != NULL; p = p->next) {
    val = process (p->size, p->data);

    #pragma omp parallel for
    #pragma llc result (v1+i*N, N)
    #pragma llc rnc_result (v2+i, 1, (N - 1), N)
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++) {
            tmp = calc_complex(val, i, j);
            v1[i*N + j] = tmp;
            v2[j*N + i] = tmp;
            #pragma llc no_result (v3+f(i,j), l, v3)
            v3[f(i,j)] = tmp;
        }
    ... }
}
```

The translation produced by llCoMP uses communication of memory areas modified by each processor group to guarantee the consistency of memory at the end of the execution of a parallel skeleton. The code in Listing 2 shows the usage of some llc directives to ensure memory coherence. The OpenMP clause last-private (line 8) indicates that the variable val must have the same value in all the processors at the end of the parallel loop. The programmer can use either the OpenMP syntax or the llc lastresult clause. When dealing with contiguous memory locations, llc uses the result clause to mark them (see lines 10 and 15 in the code). The arguments for this directive are a pointer to the beginning of the modified region and the number of elements to be modified. Inside the nested loop in line 17, three vectors v1, v2 and v3 are updated using different access patterns. These patterns are illustrated in Fig. 1. For vector v1 a contiguous memory block of N elements is accessed and the result clause annotates this access. The memory areas updated in v2 are not contiguous, but they follow a regular pattern with stride N that repeats N times. The llc rnc_result directive in line 16 is used to annotate this access. In this clause, the programmer has to specify
four arguments: the starting address (v2+i in this example), the number of items to be modified (1), the number of items not to be changed (N-1) and finally, the number of pattern repetitions (N). The llc nc_result directive has to be used in case of non-regular or unknown memory access patterns, like that on v3 on line 23. nc_result is inserted directly in the code, just before the memory access, as shown in line 22. The arguments for this directive are: the beginning of the modified memory address (v3 + f(i,j) in this example), the number of modified items (1) and the address of the first item of the modified variable (v3). A more detailed explanation of these directives can be found in [14].

Fig. 1 illustrates the memory access pattern for vectors v1, v2 and v3 in Listing 2. For the sake of simplicity we consider N = 4 in this figure. Dark elements indicate positions in the corresponding vector that are written by the processor performing the first iteration of the loop in line 17 in Listing. The different llc directives in the code mark the memory areas that have to be communicated by that processor.

The syntax of the parallel loops in llc is presented here using extended BNF notation.

```c
#pragma omp [parallel(a)] for {omp_for_clause(b)}*
#pragma llc weight({w})
#pragma llc result({ptr, num_elem}+)
#pragma llc rnc_result({ptr, copy, nocpy, rep}+)
#pragma llc lastresult({ptr, num_elem}+)|
#pragma llc reduce({ptr_part_sol, ptr_sol, num_items, op}+)}*

for-loop(d)
```

(a): If the parallel keyword is not specified, the construct must be inside a parallel region.
(b): For compatibility reasons, all OpenMP clauses are admitted, but they have no effect in llc except for reduction and lastprivate clauses. To modify the sharing attributes of the variables inside the parallel loop, the user must use llc directives instead of OpenMP clauses.
(c): The llc #pragma llc reduction_type directive is only used to complete the OpenMP reduction clause.
(d): The code inside the for-loop may include the llc #pragma llc nc_result directive wherever it is necessary. Its syntax is as follows: #pragma llc nc_result({ptr, num_items, ptr_beginning}+)

3. Sections skeleton

The sections skeleton permits to specify code sections to be executed in parallel. The skeleton uses two constructs for its implementation: the sections construct specifies the region where portions of parallel code can be found. Portions of parallel code have to be indicated using the section construct, inside a sections region.

Listing 3: Implementation of the quicksort using sections

```c
void qs(int *v, int first, int last) {
  int pivot, temp, i, j;

  i = first; j = last;
  pivot = v[(first + last)/2];
  do {
    while (v[i] < pivot) i++;
    while (v[i] > pivot) j--;
    if (i <= j) {
      temp = v[i];
      v[i] = v[j];
      v[j] = temp;
```

A. Dorta et al. / Parallel Computing 32 (2006) 491–506
The code in Listing 3 shows the implementation of the well known Quicksort algorithm using sections. In this code, the parallel sections region covers the recursive calls to $qs()$. Each call is done inside a different section, therefore they will be executed in parallel. The weight associated to each section (number of elements to sort) is provided in order to achieve load balancing.

Fig. 2 illustrates the communication pattern produced by the llcOMP translation for the Quicksort algorithm exposed in Listing 3. The communication pattern depends on the number of processors ($P$) and the number of tasks ($T$). Each of the sections in Listing 3 is a task. In Fig 2, a set with $P=7$ processors divides in two subsets and each subset deals with the execution of one of the tasks (sections). The amount of processor on each subset (3 and 4 in the Figure) is computed accordingly with the information provided in the weight directive (lines 21 and 26 in the Listing). After the execution of the tasks, each processor set will send the results to the complementary set. The results to be communicated are specified using the llc result directives in lines 22 and 27 in Listing 3. The communications are represented by arrows in Fig. 2.

The syntax of the llc sections is identical to its OpenMP counterpart and therefore, the llc section codes are fully compatible with OpenMP.

```c
#pragma omp parallel sections
{
    #pragma omp section
    #pragma llc weight(j - first + 1)
    #pragma llc result(v + first, (j - first + 1))
    if (first < j) qs(v, first, j);

    #pragma omp section
    #pragma llc weight(last - i + 1)
    #pragma llc result(v + i, (last - i + 1))
    if (i < last) qs(v, i, last);
}
```

The syntax of the llc sections is identical to its OpenMP counterpart and therefore, the llc section codes are fully compatible with OpenMP.

```c
#pragma omp [parallel(a)]sections [omp_sections_clause(b)]*
 [#pragma llc reduction_type(c)({type}+)]
 {#pragma llc lastresult ({ptr, num_elem}+) | 
   #pragma llc reduce ({ptr_part_sol, ptr_sol, num_items, op}+)}*
sections_code
```

Fig. 2. Communication pattern in the Quicksort algorithm.
The sections have to be specified inside the `sections_code`. Each section declaration follows the following syntax:

```c
#pragma omp section {omp_section_clause(b)*
  [#pragma llc weight ({w})]
  [#pragma llc result ({ptr, num_elem}+)*

  section_code
}
```

(a),(c): Same comments as in the corresponding notes of the syntax of the parallel loops in llc apply.

(b): Same comments as in the corresponding note of the syntax of the parallel loops in llc apply (in this case for the sections and section OpenMP clauses).

### 4. Taskq skeleton

The `taskq` skeleton is used to implement farms in llc. The syntax for this skeleton has been taken from the `taskq` construct [12] proposed by the Kai-Intel group to be included in OpenMP. In this skeleton, one processor (the *master* processor) generates tasks and the remaining processors in the current set (*slave* processors) execute these tasks and send their results to the master.

The llc syntax is fully compatible with the Intel proposal, that has been implemented in the C/C++ Intel compiler. The Intel clauses for `taskq` and `task` constructs are accepted by llCoMP, although they produce no effect. Instead of these clauses, some specific llc directives have been added in order to specify the sharing attributes of variables inside this construct. In its current state, recursion and nested tasks are not supported.

The code presented in Listing 4 is a portion of the Intel-KAI [15] implementation of the Strassen parallel algorithm for matrix multiplication. This algorithm contains several code areas, where the `taskq` construct can be used. In Listing 4 one of these cases is shown. The algorithm divides the original matrix into four smaller matrices and they are multiplied in parallel using the `taskq` construct. The Intel OpenMP directives are completed with llc directives in order to specify modified memory regions. The llc `task_slave_data` and `task_slave_rnc_data` directives indicate the memory regions modified by the slaves that have to be communicated to the master. The syntax and meaning of these directives is the same as the llc `result` and `rnc_result` directives, respectively. They were described in Section 2.

**Listing 4: Taskq in Strassen llc**

```c
1 ... #pragma intel omp taskq
2 #pragma llc taskq_master_result (& C[0], n * n)
3 {
4     /* compute tmp = a12 * b21 */
5     #pragma intel omp task
6     #pragma llc task_slave_data (& (tmp[0]), (ss * ss))
7     matrix_mult(ss, bs, ss, A, 0, ss, size, B, 0, ss, size, tmp, 0, 0, ss, 1);
8 
9     /* compute c12 = (a11,a12) * (b12,b22)^T */
10    #pragma intel omp task
11    #pragma llc task_slave_rnc_data (& (C[ss]), bs, (size - bs), ss)
12    matrix_mult(ss, n, bs, A, 0, 0, size, B, 0, ss, size, C, 0, ss, size, 1);
13 
14    /* compute c21 = (a21,a22) * (b11,b21)^T */
15    #pragma intel omp task
16    #pragma llc task_slave_rnc_data (& (C[ss*size]), ss, (size - ss), bs)
17    matrix_mult(bs, n, ss, A, ss, 0, size, B, 0, 0, size, C, ss, 0, size, 1);

A. Dorta et al. / Parallel Computing 32 (2006) 491–506 497
```
When the tasks execution has finished and the master has received all the data returned from the slaves, the master broadcasts the results specified by the llc taskq_master_result directive (line 3) to the remaining processors in the set. This llc directive has the same syntax and meaning as result.

The language provides some other directives used to guarantee memory coherence. All of them have to be used inside the task construct. The llc task_master_data directive sends initial data from the master to the slaves. Sometimes it is necessary to set up some values in the slaves every time a task is executed. In these case, the llc task_slave_set_data must be used.

The task_reduce_slave (and task_t_reduce_slave, a simpler version) directive is provided to deal with reduction operations: the master processor will perform the reduction operation specified with each single slave contribution.

The last llc directive related to the taskq skeleton is the taskq_barrier. When used, it must be inserted in the code within the taskq region but outside the tasks code. When this directive is found, the master will wait for the finalization of all the tasks before continuing execution.

The llc syntax to declare a taskq region is the following:

```
#pragma [intel]omp [parallel(a)] taskq {omp_taskq_clause(b)}*
{#pragma llc taskq_master_result {([ptr, num_elem]+)*

    taskq_code(c)

}#pragma [intel]omp task {omp_taskq_clause(b)}*
{#pragma llc task_master_data {([ptr, num_elem]+)|
    #pragma llc task_slave_data {([ptr, num_elem, value]+)|
    #pragma llc task_slave_set_data {([ptr, num_elem]+)|
    #pragma llc task_slave_rnc_result {([ptr, copy, nocpy, rep]+)|
    #pragma llc task_reduce_slave {([ptr_part_sol, ptr_sol, num_items, op]+)|
    #pragma llc task_t_reduce_slave {([ptr_sol, type, num_items, op]+)|

    task_code

(a): Same comments as in the corresponding notes of the syntax of the parallel loops in llc apply.
(b): Same comments as in the corresponding notes of the syntax of the parallel loops in llc apply (in this case for all the taskq and task Intel OpenMP clauses).
(c): The code inside taskq_code and outside task_code can include the llc directive #pragma llc taskq_barrier.

5. Pipeline skeleton

llc provides support for pipelines. The pipe skeleton is associated with a for loop and it allows the organization of iterations as a pipeline. Each iteration is considered a stage of the pipeline.

Listing 5 presents the use of the pipe skeleton. It is a llc implementation of a Dynamic Programming algorithm solving the single resource allocation problem (SRAP) [16]. This optimization problem consists in finding the optimal allocation of $M$ units of an indivisible resource among $N$ demanding tasks. Function $f_{2r}(r)$ gives the profit obtained by the assignation of $r$ units of resource to the $r$th task. Using Dynamic Programming, the SRAP problem can be reduced to compute $G_{N-1}(M)$ using the equations:
The function \( srap() \) in Listing 5 takes as input the parameters \( N, M \) and the profit function \( f(n, r) \). It fills in tables \( G[n][r] \) and \( L[n][r] \) with, respectively, the best profits and the optimal decisions. The \texttt{llc pipeline} directive in line 4 specifies that the \( N \) iterations of the loop in line 6 will be organized as a pipeline, and each iteration will correspond to a stage of this pipeline. The current set of processors will be divided into new subsets, and each subset replicates the execution of the code associated with a loop iteration \( n \). The use of the \texttt{result} clause at line 5, ensures memory consistency at the end of the execution of the pipeline.

\begin{verbatim}
Listing 5: Implementation of the SRAP algorithm using \texttt{llc}

1 int srap(int N, int M, cost f, table G, table L) {
2    int r, n, i, s, decision_i, temp, pos, chunksize, buffersize;

4    #pragma llc pipeline schedule(chunksize, buffersize)
5    #pragma llc result (&G[0][0], M) (&L[0][0], M)
6    for (n = 0; n < N; n++) {
7        if (n==0)
8            for (r = 0; r <= M; r++) {
9                G[0][r] = f(0, r); /* assume f is non decreasing */
10               #pragma llc send (&G[0][r], 1)
11            }
12        else
13            for (r = 0; r <= M; r++) {
14               #pragma llc receive (&G[n-1][r], &s)
15                temp = G[n-1][r];
16                pos = 0;
17                for (i = 1; i <= r; i++) {
18                    decision_i = G[n-1][r-i] + f(n, i);
19                    if (decision_i > temp) {
20                        temp = decision_i;
21                        pos = i;
22                    }
23                }
24                G[n][r] = temp;
25               #pragma llc send (&G[n][r], 1)
26                L[n][r] = pos;
27            }
28        }
29    return G[N-1][M];
30 }
\end{verbatim}

The initial stage, (task \( n=0 \)), is coded from lines 7–11. Directive \texttt{send} at line 10 inserts an element in the data stream to the next stage. Communications occur between consecutive subsets and the directive has no effect for processors in the last stage.

The remaining stages perform the computation from lines 13–27. At line 14, directive \texttt{receive} takes the values from the previous stage stream, and stores them in \( G[n-1][r] \). The directive has no effect for processors in the first stage. Values stored on variable \( s \) are discarded and must be of size \texttt{sizeof(G[0][r])}. After a new value \( G[n][r] \) is computed, it is sent to the next stage at line 25.

At the end of the computation, the \( n \)th thread has computed the \( n \)th column of matrices \( G \) and \( L \). Directive \texttt{result} at line 5 forces the memory synchronization of all the processors in the current set.

The \texttt{llc scheduled(chunksize, buffersize)} clause in line 4 is used in order to indicate that \texttt{chunksize} consecutive iterations have to be cyclically mapped in the same physical processor. The size

\[
G_n(r) = \max \{ G_{n-1}(r-i) + f_n(i) : i \in \{0, \ldots, r\} \}
\]
of message buffers is given by chunksize. This information allows to llCoMP to produce “tiled code”.

llc uses its own syntax for the pipeline skeleton. The pipeline region is indicated using the following syntax:

```c
#pragma llc pipeline [schedule(chunksize, buffersize)] [weight w]
[forall llc directives]
for-loop
```

Any forall llc directive can be used also with pipelines. In addition, there are two specific directives for pipelines. These directives are used in order to send and to receive data to/from the contiguous stage, as it was exposed in the SRAP code explanation. They have to be used in the code inside the for loop:

```c
#pragma llc send ({ptr, num_items}+)
#pragma llc receive ({ptr, num_items}+)
```

6. The llCoMP Translation

Parallel programs usually follow well-known patterns or skeletons. The skeleton containing the basic parallel operations, can be stored in a code frame. These code frames can be used in different applications when they have the same parallel pattern. Unfortunately the same situation does not occur with data patterns. Despite the fact that communication patterns are the same in different codes which use the same parallel model, data patterns are usually quite different. Datatypes, sizes, number of elements, etc. are different in each code, even if they follow the same skeleton. For this reason, llCoMP uses two different kinds of code frames to translate the basic skeletons.

The first code frame is the static code frame. It does not depend on the data. The static code frame contains all the code needed for creating processor groups and other initial operations, code for saving and restoring the context, execution, code for communications operations and termination code. These codes are fixed at compile time because they only depend on the skeleton that the user chose, but they do not depend on the data.

The static code frames are divided in several stages when needed (e.g. initialization, execution, communication and finalization) and they are stored in plain text files. These files are independent of the compiler, and therefore code frames can be modified whenever needed without changing the compiler source code. llCoMP uses these code frames during the translation. From the information contained in the directives, the compiler decides which file contains the proper code frame and the proper stage. The contents of the code frames are completed with the information specified by the user using the directives and then attached to the target code.

The second type of code frames is called dynamic code frames. Static code frames do not operate on real data. They just send and receive buffers of data (streams of bytes without datatypes). There are special marks in the static code frames surrounding the code for sending and receiving buffers. While the compiler is analyzing the original code, dynamic code frames are applied and temporary files are generated. Dynamic code frames contain the code needed for allocation and management of these buffers. In order to minimize communications, data are packed/unpacked for transfer.

The separation between static and dynamic code frames allows new data management directives to be added or modified without changing the static code frames (e.g. directives for new memory access patterns). On the other hand, we can change the static code frames (e.g. change the communication patterns) without any consideration about the data.

During compile time, llCoMP analyzes the information provided by the user through the directives. With this information the compiler completes the static code frames and generates the dynamic code frames that include buffer operations and packing/unpacking data. Finally, the compiler mixes dynamic and static code frames, obtaining the target parallel code.
In the following subsections we will discuss the details of the translation for the basic skeletons that were introduced in Sections 2–5.

6.1. Forall translation

Four static code frames are used in the translation of the forall skeleton. One code frame is responsible for the initializations. It deals with processors and tasks distribution. A second code frame is used for the execution stage. This frame deals with saving and restoring the context before and after the execution. After the execution, a communication frame is applied and, at the end, a finalization frame will free all the memory used by the previous code frames.

In order to perform the translation, each single iteration of the loop is considered as a task. All the tasks are divided among the processors available in the current set of processors. Two cases, which depend on the relation between number of tasks ($n_T$) and number of processors ($n_P$), are considered.

If $n_T$ is greater than $n_P$: tasks are grouped in blocks and each block is assigned to a single processor. Blocks are built using a heuristic that will try to assign tasks in an equitable way if weight information is provided by the user. If weight information is not given, the heuristic will try to assign the same number of tasks to each processor. Following the OTOSP model [11], all the processors must share the same view of memory at the end of execution of any parallel construct. At the end of a task execution, processors have to notify the results (modified memory locations) to the parent set. This is achieved in the communication stage, which terminates with all the processors sharing the same view of memory.

If $n_P$ is greater than (or equal to) $n_T$: in this case, the processors will group into sets and a single task is assigned to each set. According to the OTOSP model, processors belonging to the same set will execute the same code with the same data. When this situation takes place, the system is able to exploit multi-level parallelism. If there is a parallel construct within a task assigned to a set containing more than one processor, the set may be subdivided. This is the way to create several levels of nested parallelism. If weight information has been provided in the construct, the number of processors on each set is computed accordingly: the heuristic associates more processors to the heavier tasks. If there is no weight information, the same number of processors will be assigned to each set, whenever possible. After execution of the construct, each set will communicate its results to the complementary sets and all the processors will have all the results. If a set contains more than a single processor, it can send and receive the results in parallel. The communication code frame is designed so that communications take place in parallel whenever possible.

llc provides several advantages over OpenMP when translating parallel loops. The most evident is the extension of the OpenMP paradigm to distributed memory architectures. llComP supports multi-level parallelism, a feature not always available in OpenMP compilers. The llc parallel loop implementation also allows the user to specify a dynamic load balance by indicating the weight of each task. llc supports compound variables and user defined datatypes, e.g. in reduction operations or lastprivate results, while OpenMP only provides support for simple variables. llc extends predefined reduction operations, like MIN and MAX and it also supports user defined functions in the reduction operations, which are not permitted by OpenMP. Finally, llc places no restrictions on the structure of a parallel for-loop.

Some optimizations have been added to the compiler in order to enhance performance. In situations, where optimization could mean a loss of generalization, several different code frames have been designed in order to avoid it. All the optimizations are made by the compiler without user intervention. The communications are reduced to a minimum. In most cases there is only a single communication at the end of execution of a parallel skeleton. In the case, where the user does not specify task weights, the system does not use the heuristic to compute processor sets; groups are created by applying a balanced distribution. The nc_result directive is used to indicate an irregular or unknown access pattern. The translation of this construct creates a list of modified memory regions for each variable in the directive. A heuristic is used to pack adjacent memory regions and to reduce the amount of data communicated.
6.2. Sections translation

The translation process for the sections skeleton converts parallel sections into a special case of parallel for loop. The basic idea consists in considering each section as a single iteration in a forall skeleton. With this approach, the section code frames are very similar to those used in the forall skeleton. All the considerations made for the forall skeleton translation are also valid for sections.

6.3. Tasq translation

For the taskq skeleton, the translation process is different on the slave-side and on the master-side. When a taskq directive is found in the source code, the generated code is split into two regions: The first region is the master region and only the master processor will execute the code inside this region. All the code inside a taskq region and outside a task region will be included in the master-side code. The code inside the task region will be replaced by a call to a macro. This macro contains the code needed to send data and task to an idle slave. If all the slaves are busy, the master will wait until a slave becomes idle. This code is generated by mixing a static code frame and dynamically generated data management code. After all the tasks have been sent to the slaves, the master will wait until all the tasks finalize. Finally, the master sends a message with a termination signal and the parallel region ends. If there is a taskq_result directive in the code, the master will broadcast the results to the slaves.

The translation of the second region, slave region, is enclosed in a waiting loop. At any given time, a slave processor is either executing a task code or waiting for a message from master (a new task or a finalization message). Each time the master sends a task, the slave performs the following steps: receives the task message, executes the corresponding code and sends back the results to the master. As in the previous case, all these functions are generated by mixing a static code frame with dynamically generated data management code.

When a slave receives a termination message, it finishes the waiting loop. If there is a taskq_result directive, the slaves will receive the final results from the master.

6.4. Pipeline translation

Each stage of the pipeline is implemented as a task. Tasks are assigned to the processors sets. The assignment depends on the number of task \( n_T \) and the number of processors \( n_P \).

If \( n_P > n_T \): the processors are grouped into \( n_T \) sets of processors and each task is assigned to a set. Load balancing is provided by the llc weight directive if provided. This clause specifies an expression that controls the number of processors assigned to each stage. The partner relationship is redefined to include the case when a processor has several partners in the next stage.

If \( n_T > (\text{or equal to}) n_P \): each processor composes a subset and several stages or tasks are assigned to each subset. Load balancing and data locality are considered using the information given by the llc scheduled \((\text{chunksize, buffersize})\) clause, if present. This clause cyclically maps chunk-size consecutive iterations in a processor and it buffers messages using buffers of size buffersize.

There are two special llc directives: \text{send} and \text{receive}. The first directive sends data to the next stage and it has no effect in the last stage. The second directive receives data from the previous stage and it has no effect in the first stage. Communications occur between consecutive subset of processors.

6.5. OpenMP functions and llc macros

llc also provides support for some OpenMP functions. The OpenMP functions related with getting the number of processors in the group and the processor id inside the current group (\text{omp_get_num_threads}, \text{omp_get_thread_num} and \text{omp_get_num_procs}) are recognized by llCoMP.

llCoMP support is completed with some macros related with I/O operations. By using these macros, the programmer can manage files, read/write to files and to standard input/output. These macros provide safe mode I/O operations and support for the basic I/O functions in ANSI C. All the operations will be performed by the master processor only and all data read will be broadcasted to the remaining processors in the same group.
7. Computational results

In order to evaluate the performance of the `llCoMP` translation we have used four algorithms. Three of them have been formerly presented in this work: the Mandelbrot Set computation, the single resource allocation problem (SRAP) and the Strassen matrix multiplication algorithm. The fourth algorithm is an implementation in `llc` of the velocity Verlet algorithm for molecular dynamics (MD) simulation. The source code for all the examples can be downloaded from the OpenMP Source Code Repository [17].

Five different target platforms were used in the experiments: a PC cluster with Intel 2.4 GHz Xeon processors, with 1 GByte of RAM memory each, and connected through a Myrinet switch. We used the MPICH implementation on top of the vendor’s communication library GM-1.6.3. Two Sun machines (SunFire 6800 and E15K) both using UltraSPARC-III processors with clock rates of 750 MHz and 900 MHz, respectively, and, finally two SGI machines, an Origin 2000 with R10000-250 MHz MIPS processors and an Origin 3000 with R14000-600 MHz MIPS processors. The Sun machines are shared memory SMP systems, and the SGIs are cc-NUMA architectures with 2 and 4 processors per node, respectively. Only in the SGI Origin 2000, exclusive use of the processors was guaranteed during the executions.

The `llc` Mandelbrot Set computation code was exposed in Listing 1. Fig. 3 presents the results obtained in the SunFire 6800, using 4096 points in the complex plane and in the PC-Cluster, computing 8192 points. In the case of the Sun platform three versions are shown: pure OpenMP and MPI and a `llc` implementation using the `forall` skeleton. In the PC-Cluster we compare `llc` using the `taskq` skeleton against a Master–Slave MPI implementation. We observe that up to 20 processors, both `llc` skeletons behave similarly to their corresponding OpenMP and MPI versions.

The results in Fig. 4 compare the `llc` and MPI implementations of the Molecular Dynamics code in the PC-Cluster. The performance loss in `llc` with respect to a direct MPI implementation is acceptable.

The code for the SRAP [16] was formerly outlined in Listing 5. Fig. 5 shows the computational results for this algorithm implemented using the `llc` `pipeline` skeleton in three of the target platforms. In the case of the SGI O2K and the PC-Cluster the input parameters were \( N = 350 \) and \( M = 4000 \) while in the SGI O3K the results correspond to \( N = 150 \) and \( M = 2000 \).

![Mandelbrot. (A) SunFire 6800 (N=4096). (B) PC-Cluster (N=8192)](image)

Fig. 3. Computational results for the Mandelbrot Set computation.
The code in llc for the Strassen matrix multiplication algorithm was outlined in Listing 4. The results in Fig. 6 compare the multiplication of square matrices with size $1600 \times 1600$ in the Sun E15K and PC-Cluster platforms.

Fig. 4. Computational results for the molecular dynamics code.

Fig. 5. Results for the SRAP code.
8. Conclusions and future work

llc is a C based language that provides support to some of the most widely used skeletons. We believe that preserving the sequential semantics of programs is a major key to achieve the objective of alleviating the difficulties in the development of parallel applications. In llc the parallel skeletons are expressed using compiler directives. An important advantage obtained through this approach is that the same source code represents three valid code versions. It is a sequential code as far as a sequential compiler that ignores OpenMP and llc directives accepts it. It is also a parallel MPI code because that is the result of compilation with llCoMP. Finally, if the OpenMP directives are used in the source code, any OpenMP compiler would translate it to a pure parallel OpenMP binary.

We have shown that a llc compiler using direct generation of MPI code for communications can produce good results, even in the case of fine-grain parallel algorithms. The performance loss with respect to a direct MPI implementation is compensated by a much smaller development effort. The use of MPI at the back-end of the compiler guarantees the portability of the target code to distributed and shared memory environments.

At the present time, llc is in a status of reasonable stability and maturity. Nevertheless, the system can still be considered in a development stage. Work in progress in our project includes the following issues:

- To unburden the final user from the specification of memory regions to be communicated (using the result clauses).
- To explore the potential sources of improvement for the target code produced by our compiler prototype. The use of parallel profiling tools is mandatory to analyze and discover opportunities for enhancement.
- To extend the group of applications suitable to be targeted by llCoMP.
- To add new parallel skeletons to the language.

Acknowledgements

The authors thank the anonymous reviewers for their suggestions on how to improve the paper. This work has been partially supported by the Canary Islands government, contract PI2003/113, and also by the EC.
(FEDER) and the Spanish MCyT (Plan Nacional de I + D + I, contract TIN2005-09037-C02-01). This work was carried out under the HPC-EUROPA project (RII3-CT-2003-506079), with the support of the European Community – Research Infrastructure Action under the FP6 “Structuring the European Research Area” Programme.

References