

The 11c language and its implementation

Antonio J. Dorta, Jose Rodríguez, Casiano Rodríguez and Francisco de Sande

Dpto. Estadística, I.O. y Computación
Universidad de La Laguna
La Laguna, 38271, Spain
fsande@ull.es

Abstract. The 11c language is the result of our study in the opportunities to extend OpenMP to deal with multi-level parallelism and distributed memory architectures.

In this laboratory proposal we briefly describe the language, its underlying computational model and by one example we illustrate its use. If accepted, the number of persons representing this project would be one.

1 Introduction

The simplicity of the model and the ease of use are probably the main reasons for the success of OpenMP. With a relatively short life it has become the dominant high level tool to develop parallel applications for shared memory architectures.

The 11c language represents the result of our research in the last years toward extending the OpenMP model. The language preserves the simplicity of OpenMP and extends the standard with different features. Particularly, the underlying computational model allow us to target both distributed and shared memory architectures. The syntax of the language is based in C and as in OpenMP, parallelism is expressed through the introduction of compiler pragmas in the code. Wherever it is possible 11c pragmas are compatible with those existing in OpenMP.

11CoMP, our prototype compiler for the language is a source to source compiler that translates the C code annotated with 11c pragmas into C code augmented with calls to MPI [1] functions.

2 11c syntax and semantics

The *OTOSP* (*One Thread is One Set of Processors*) model is the theoretical computational model underlying the 11c language. In this model the processors are organised in *sets*. At any time, the memory state of all the processors in the same set is identical. An *OTOSP* computation presumes that all the processors in the same set have the *same input data* and *the same program in memory*. The initial set is composed of all the processors in the machine. When a set of processors execute any kind of parallel construct, the set is divided into subsets, and each processor in a subset establishes a *partnership* relation with one or

more processors in the complementary subsets. After the execution of the parallel construct, and to keep the coherence of the memory, each processor exchanges with its *partners* the corresponding results. For a detailed description of the *OTOSP* model, please refer to [2].

```

1 void compute(int np,int nd,double *box,vnd_t *pos,vnd_t *vel,
2             double mass,vnd_t *f,double *pot_p,double *kin_p) {
3     double x, d, pot, kin;
4     int i, j, k;
5     vnd_t rij;

7     pot = kin = 0.0;
8     #pragma omp parallel for default(shared)
9         private(i, j, k, rij, d) reduction(+ : pot, kin)
10    #pragma llc reduction_type (double, double)
11    #pragma llc result(f[i], nd)
12    for (i = 0; i < np; i++) { /* Pot. energy and forces */
13        for (j = 0; j < nd; j++)
14            f[i][j] = 0.0;
15        for (j = 0; j < np; j++) {
16            if (i != j) {
17                d = dist(nd, box, pos[i], pos[j], rij);
18                pot = pot + 0.5 * v(d);
19                for (k = 0; k < nd; k++) {
20                    f[i][k] = f[i][k] - rij[k] * dv(d) / d;
21                }
22            }
23        }
24        kin = kin + dotr8(nd, vel[i], vel[j]); /* kin. energy */
25    }
26    kin = kin * 0.5 * mass;
27    *pot_p = pot;
28    *kin_p = kin;
29 }

```

Listing 1. The Molecular Dynamics code in llc

Given positions, masses and velocities of *np* particles, the routine shown in listing 1 computes the energy of the system and the forces on each particle. The code is an implementation in llc of the velocity Verlet [3] algorithm for Molecular Dynamics (MD) simulation whose Fortran version can be downloaded from the OpenMP official web site [4].

The `parallel for` at line 8 indicates that the iterations of the for loop at line 12 can be split among the processors in the current set. Processors are always available in the *OTOSP* model to deal with a *parallel construct* because an *OTOSP* machine is arbitrarily large. In a real scenario, when a set of processors reaches a `parallel for` construct two situations are possible. The number of processors in the current group can be larger or smaller than the number of tasks (in this example, loop iterations) to be done. If there are more tasks than processors (as is the usual case), the original set will be divided in subsets with a single processor. Each group (processor) will then compute several tasks. In the case of more processors than tasks, several processors belonging to the same set will replicate the computation of the same task.

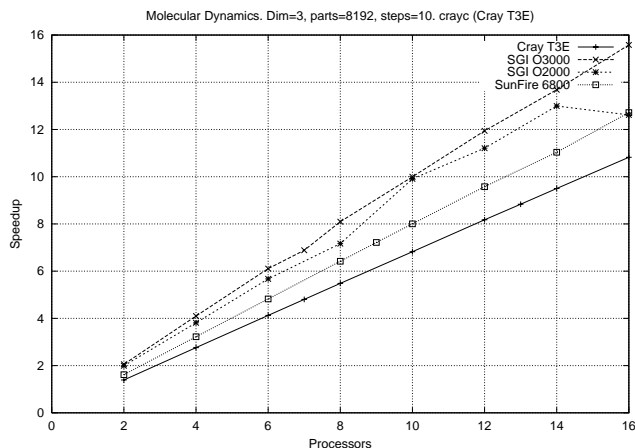


Fig. 1. Computational results for the MD algorithm

As we can see in listing 1 `llc` extends OpenMP syntax and semantic to deal with distributed memory computing. Whenever an OpenMP pragma with the required semantic exist we use it (note the use of the `omp` and `llc` prefixes in the pragmas). In fact, the clause `private` in line 9 is kept only for compatibility with OpenMP, since it is not required because all the storages are private in the *OTOSP* model. The `reduction` clause in line 9 indicates that all the local values of variables `pot` and `kin` have to be added at the end of the loop. This operation implies a collective communication amongst all processors in the set and the updating of the variable with the result of the reduction operation. Since a complete type analysis has not been included in `llcOMP`, the type of the reduction variables must be specified by the programmer. This is the purpose of the pragma `llc reduction_type` in line 10.

The `result` clause in line 11 is associated with the `parallel for` at line 8. It has the purpose of informing the compiler of the new “ownership” of the part of the memory that is going to be modified. The clause indicates that the set of processors that perform thread i “owns” (and potentially could modify) the memory area starting at `f[i]` and whose size is `nd`.

Figure 1 shows the results obtained for the MD code on four different platforms: a Cray T3E, a SunFire 6800, and two SGIs models 2000 and 3000. Only the Cray T3E is a pure distributed memory architecture. In the SGI Origin 2000 exclusive use of the processors was guaranteed during the executions. The results correspond to `np = 8192` particles and 10 simulation steps.

Moreover `llc` extends the OpenMP data parallel construct in several ways. Let us explain some of them.

2.1 Multi-level parallelism

Multi-level parallelism enables the generation of work from different simultaneously executing threads. The exploitation of multiple levels of parallelism has not been clearly addressed in OpenMP, despite it has been stated as a positive property [5], [6], [7]. Although nesting of parallel regions is allowed in the OpenMP specification, most of the compilers simply serialise any nested parallel region.

```
1 #pragma omp parallel for
2 #pragma llc weight(size[i])
3 #pragma llc result(res[i], size[i])
4   for(i = 0; i < 2; i++)
5     qs(v, start[i], end[i]);
```

Listing 2. Weighted recursive calls

The parallel for loop in listing 2 appears in *qs()*, a parallel function written in llc that implements the widely known Quicksort algorithm and it is an example of multi-level parallelism. Each instance of *qs()* makes two recursive calls. Following the *OTOSP* model, each recursive call to *qs()* divides in two the current set of processors. The divisions proceed until processor sets with size one are reached.

2.2 Load balancing

We can also use the code in listing 2 to explain another capability present in llc. When the situation in a `parallel for` is that the number of processors available in the set is larger than the number of tasks to compute, a load balance problem arises. If a measure of the work w_i per thread T_i is available, the processors distribution can be adapted to guarantee an optimal mapping. This is the purpose of the `weight` clause in line 2 of listing 2. In this Quicksort example, the size of the generated sub-vectors are used as weights for the `parallel for`. Remember that, depending on the goodness of the pivot chosen, the new subproblems may require rather different computational efforts.

The semantic of the `weight` construct is similar to that proposed in [6], but the mapping is computed differently. In this case we also have to deal with the additional problem of properly establishing the *partnership* relation.

The interested reader may refer to [8] to see examples and results that make use of these features and also others that have not been presented here for the sake of brevity.

3 Conclusions

The llc language extends the OpenMP directives with new annotations. An appealing property of llc is that keeping the simplicity of the OpenMP model,

it can be targeted to both shared and distributed memory architectures. From the sequential program and using OpenMP directives we can, without destroying the sequential program, produce a new parallel shared memory version. From the OpenMP program and through the addition of supplementary `llc` directives, we also can, without destroying the OpenMP program, produce a new parallel distributed memory version. Still, if the efficiency is unsatisfactory, the programmer can add explicit MPI send/receive operations.

Furthermore, the language extends OpenMP to deal with multi-level and pipeline parallelism.

The portability both for the code generated and for the compiler itself have been two of the goals in the design of `llCoMP`. The target architectures where we have tested our implementations include Cray T3E, SGI Origin, SunFire E15K, IBM RS-6000 and Intel-based Beowulf clusters.

From the `llc` project web page [9], the reader can retrieve additional information about the algorithms that we have successfully implemented and tested using `llc`.

References

1. Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. University of Tennessee, Knoxville, TN, June 1995. <http://www.mpi-forum.org/>.
2. Casiano Rodríguez, Francisco de Sande, Coromoto León, and Luis García. Parallelism and recursion in message passing libraries: an efficient methodology. *Concurrency: Practice and Experience*, 11(7):355–365, June 1999.
3. W. C. Swope, H. C. Andersen, P. H. Berens, and K. R. Wilson. A computer simulation method for the calculation of equilibrium constants for the formation of physical clusters of molecules: Application to small water clusters. *Journal of Chemical Physics*, 76:637–649, 1982.
4. OpenMP Architecture Review Board. OpenMP official web site. <http://www.openmp.org/>.
5. Sanjiv Shah, Grant Haab, Paul Petersen, and Joe Throop. Flexible control structures for parallelism in OpenMP. *Concurrency: Practice and Experience*, 12(12):1219–1239, October 2000.
6. Marc González, Eduard Ayguadé, Xavier Martorell, Jesús Labarta, Nacho Navarro, and José Oliver. NanosCompiler: supporting flexible multilevel parallelism exploitation in OpenMP. *Concurrency: Practice and Experience*, 12(12):1205–1218, October 2000.
7. Y. Tanaka, K. Taura, M. Sato, and A. Yonezawa. Performance evaluation of OpenMP applications with nested parallelism. *Languages, Compilers, and Run-Time Systems for Scalable Computers*, 2000.
8. Antonio J. Dorta, Jesús A. González, Casiano Rodríguez, and Francisco de Sande. Towards structured parallel programming. In *Proc. Fourth European Workshop on OpenMP (EWOMP 2002)*, Rome, Italy, Sep 2002.
9. Francisco de Sande and Antonio J. Dorta. *llc Project web site*, 2002. <http://nereida.deioc.u11.es/~llCoMP/>.