Parallelizing Dense Linear Algebra Operations with Task Queues in `llc` *

Antonio J. Dorta[1], José M. Badía[2], Enrique S. Quintana-Ortí[2], and Francisco de Sande[1]

[1] Depto. de Estadística, Investigación Operativa y Computación
Universidad de La Laguna, 38271–La Laguna, Spain
{ajdorta,fsande}@ull.es
[2] Depto. de Ingeniería y Ciencia de Computadores
Universidad Jaume I, 12.071–Castellón, Spain
{badia,quintana}@icc.uji.es

Abstract. `llc` is a language based on C where parallelism is expressed using compiler directives. The `llc` compiler produces MPI code which can be ported to both shared and distributed memory systems.
In this work we focus our attention in the `llc` implementation of the *Workqueuing Model*. This model is an extension of the OpenMP standard that allows an elegant implementation of irregular parallelism. We evaluate our approach by comparing the OpenMP and `llc` parallelizations of the symmetric rank-k update operation on shared and distributed memory parallel platforms.

Keywords

MPI, OpenMP, Workqueuing, cluster computing, distributed memory.

1 Introduction

The advances in high performance computing (HPC) hardware have not been followed by the software. The tools used to express parallel computations are nowadays one of the major obstacles for the massive use of HPC technology. Two of these tools are MPI [1] and OpenMP [2]. Key advantages of MPI are its portability and efficiency, with the latter strongly influenced by the control given to the programmer of the parallel application. However, a deep knowledge of low-level aspects of parallelism (communications, synchronizations, etc.) is needed in order to develop an efficient MPI parallel application.

On the other hand, OpenMP allows a much easier implementation. One can start from a sequential code and parallelize it incrementally by adding compiler directives to specific regions of the code. An additional advantage is that it

* This work has been partially supported by the EC (FEDER) and the Spanish MEC (Plan Nacional de I+D+I, TIN2005-09037-C02).

follows the sequential semantic of the program. The main drawback of OpenMP is that it only targets shared memory architectures.

As an alternative to MPI and OpenMP, we have designed `llc` [3] to exploit the best features of both approaches. `llc` shares the simplicity of OpenMP: we can start from a sequential code and parallelize it incrementally using OpenMP and/or `llc` directives and clauses. The code annotated with parallel directives is compiled by `llCoMP`, the `llc` compiler-translator, which produces an efficient and portable MPI parallel source code, valid for both shared and distributed memory architectures. An additional advantage of `llc` is that all the OpenMP directives and clauses are recognized by `llCoMP`. Therefore, we have three versions in the same code: sequential, OpenMP and `llc`/MPI, and we only need to choose the proper compiler to obtain the appropriate binary.

Different directives have been designed in `llc` to support common parallel constructs in the past as *forall*, *sections*, and *pipelines* [4,5]. In previous studies [4] we have investigated the implementation of *Task Queues* in `llc`. In this paper we focus our attention in the last feature added to `llc`: the support for the *Workqueuing Model* using *Task Queues* [6]. In order to do so, we explore the possibilities of parallelizing (dense) linear algebra operations, as developed in the frame of the *FLAME* (Formal Linear Algebra Method Environment) project [7].

The rest of the paper is organized as follows. In Section 2 we present the *symmetric rank-k update* (`SYRK`) operation as well as a FLAME code for its computation. Section 3 reviews the parallelization of this code using OpenMP and `llc`. Experimental results for both OpenMP and `llc` codes are reported and discussed in Section 4. Finally, Section 5 offers some concluding remarks and hints on future research.

2 The `SYRK` operation

The `SYRK` operation is one of the *Basic Linear Algebra Subprograms* (BLAS) [8] most often used. It plays an important role, e.g., in the formation of the normal equations in linear least-squares problems and the solution of symmetric positive definite linear systems via the Cholesky factorization [9]. The operation computes the lower (or upper) triangular part of the result of the matrix product $C := \beta C + \alpha A A^T$, where C is an $m \times m$ symmetric matrix, A is an $m \times k$ matrix, and α, β are scalars.

Listing 1 presents the FLAME code for the `SYRK` operation [10]. The partitioning routines (`FLA_Part_x`, `FLA_Repart_x_to_y` and `FLA_Cont_with_x_to_y`) are indexing operations that identify regions (blocks) into the matrices but do not modify their contents. Thus, e.g., the invocation to `FLA_Part_2x1` in lines 7–8 "divides" matrix (object) `A` into two submatrices (blocks/objects), `AT` and `AB`, with the first one having 0 rows. Then, at each iteration of the loop, certain operations are performed with the elements in these submatrices (routines `FLA_Gemm` and `FLA_Syrk`). More details can be consulted in [7].

```
1  int FLA_Syrk_ln_blk_var1_seq (FLA_Obj alpha, FLA_Obj A,
2                               FLA_Obj beta,  FLA_Obj C, int nb_alg) {
3    FLA_Obj AT, AB,       CTL, CBL, CTR, CBR,
4            A0, A1, A2,  C00, C01, C02, C10, C11, C12, C20, C21, C22;
5    int     b;

7    FLA_Part_2x1(A, &AT,
8                 &AB, 0, FLA_TOP);
9    FLA_Part_2x2(C, &CTL, &CTR,
10                &CBL, &CBR, 0, 0, FLA_TL);

12   while (FLA_Obj_length(AT) < FLA_Obj_length(A)){
13     b = min(FLA_Obj_length(AB), nb_alg);
14     FLA_Repart_2x1_to_3x1(AT, &A0,
15                              &A1,
16                           AB, &A2, b, FLA_BOTTOM);
17     FLA_Repart_2x2_to_3x3(CTL, CTR, &C00, &C01, &C02,
18                                     &C10, &C11, &C12,
19                           CBL, CBR, &C20, &C21, &C22, b, b, FLA_BR);
20     /*-----------------------------------------------------------*/
21     /* C10 := C10 + A1 * A0' */
22     FLA_Gemm(FLA_NO_TRANSPOSE, FLA_TRANSPOSE, alpha, A1, A0,
23                                               beta,  C10, nb_alg);
24     /* C11 := C11 + A1 * A1' */
25     FLA_Syrk(FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE, alpha, A1,
26                                                      beta,  C11,nb_alg);
27     /*-----------------------------------------------------------*/
28     FLA_Cont_with_3x1_to_2x1(&AT, A0,
29                                   A1,
30                              &AB, A2, FLA_TOP);
31     FLA_Cont_with_3x3_to_2x2(&CTL, &CTR, C00, C01, C02,
32                                          C10, C11, C12,
33                              &CBL, &CBR, C20, C21, C22, FLA_TL);
34   }
35   return FLA_SUCCESS;
36 }
```

Listing 1. FLAME code for the SYRK operation

3 Parallelization of the SYRK operation

A remarkable feature of FLAME is its capability for hiding intricate indexing in linear algebra computations. However, this feature is a drawback for the traditional OpenMP method to obtain parallelism from a sequential code, based on exploiting the parallelism of `for` loops. Thus, the OpenMP approach requires loop indexes for expressing parallelism which are not available in FLAME codes.

Task Queues [6] have been proposed for adoption in OpenMP 3.0 and are currently supported by the Intel OpenMP compilers. Their use allows an elegant implementation of loops when the space iteration is not known in advance or, as in the case of FLAME code, when explicit indexing is to be avoided.

3.1 OpenMP parallelization

The parallelization of the SYRK operation using the Intel implementation of *Task Queues* is described in [10]. The Intel extension provides two directives to specify tasks queues. The `omp parallel taskq` directive specifies a parallel region

where tasks can appear. Each task found in this region will be queued for later computation. The `omp task` identifies the tasks. For the `SYRK` operation, the first clause is used to mark the `while` loop (line 12 in Listing 1), while the second one identifies the invocations to `FLA_Gemm` and `FLA_Syrk` as tasks (lines 21–26 in Listing 1). Listing 2 shows the parallelization using `taskq` of the loop in the

```
1  #pragma intel omp parallel taskq{
2    while (FLA_Obj_length(AT) < FLA_Obj_length(A)){
3    ...
4      #pragma intel omp task captureprivate(A0, A1, C10, C11){
5        /* C10 := C10 + A1 * A0' */
6        FLA_Gemm(FLA_NO_TRANSPOSE, FLA_TRANSPOSE, alpha, A1, A0,
7                                                  beta,  C10, nb_alg);
8        /* C11 := C11 + A1 * A1' */
9        FLA_Syrk(FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE, alpha, A1,
10                                                       beta,  C11,nb_alg);
11     }
12   ...
13   }
14 }
```

Listing 2. FLAME code for the `SYRK` operation parallelized using OpenMP

FLAME code for the `SYRK` operation. The directive `omp task` that appears in line 4 is used to identify the tasks. Function calls to `FLA_Gemm` and `FLA_Syrk` are in the scope of the `taskq` directive in line 1 and, therefore, a new task that computes both functions is created at each iteration of the loop. The first of these functions computes $C_{10} := C_{10} + A_1 A_0^T$, while the second one computes $C_{11} := C_{11} + A_1 A_1^T$. All the variables involved in these computations have to be private to each thread (A_0, A_1, C_{10}, and C_{11}), ant thus they must be copied to each thread during execution time. The `captureprivate` clause that complements the `omp parallel task` directive serves this purpose.

3.2 `llc` parallelization

In this section we illustrate the use of `llc` to parallelize the `SYRK` code. Further information about the effective translation of the directives in the code to MPI can be found in [4]. The parallelization using `llc` resembles that carried out using OpenMP, with a few differences that are illustrated in the following. After identifying the task code, we annotate the regions using `llc` and/or OpenMP directives. All the OpenMP directives and clauses are accepted by `llCoMP`, though not all of them have meaning and/or effect in `llc` [4].

We will start from the OpenMP parallel code shown in Listing 2 and we will add the necessary `llc` directives in order to complete the `llc` parallelization. The OpenMP `captureprivate` clause has no sense in `llc`, because `llCoMP` produces a MPI code where each processor has its private memory. (`llc` follows the OTOSP model [3], where all the processors on the same group have the same data in their private memories.) Unlike OpenMP, in `llc` all the variables are private by

```
1  #pragma intel omp task
2  #pragma llc task_master_data(&A0.m, 1, &A1.offm, 1, &A1.m, 1)
3  #pragma llc task_master_data(&C11.offm,1,&C11.offn,1,&C11.m,1,&C11.n,1)
4  #pragma llc task_master_data(&C10.offm,1,&C10.offn,1,&C10.m,1,&C10.n,1)
5  #pragma llc task_slave_set_data(&A1.base,1,A.base,&A0.base,1,A.base)
6  #pragma llc task_slave_set_data(&C11.base,1,C.base,&C10.base,1,C.base)
7  #pragma llc task_slave_set_data(&A0.offm,1,A.offm,&A0.offn,1,A.offn,&A0.
       n,1,A.n)
8  #pragma llc task_slave_set_data(&A1.offn,1,A.offn,&A1.n,1,A.n)
9  #pragma llc task_slave_rnc_data ((C10.base->buffer+((C10.offn*C10.base->
       ldim+C10.offm)*sizeof(double))), (C10.m * sizeof(double)), ((C10.
       base->ldim - C10.m) * sizeof(double)), C10.n)
10 #pragma llc task_slave_rnc_data ((C11.base->buffer+((C11.offn*C11.base->
       ldim+C11.offm)*sizeof(double))), (C11.m * sizeof(double)), ((C11.
       base->ldim - C11.m) * sizeof(double)), C11.n)
11   {
12   /* C10 := C10 + A1 * A0' */
13   FLA_Gemm(FLA_NO_TRANSPOSE, FLA_TRANSPOSE, alpha, A1, A0,
14                                              beta,  C10, nb_alg);
15   /* C11 := C11 + A1 * A1' */
16   FLA_Syrk(FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE, alpha, A1,
17                                                    beta, C11, nb_alg);
18   }
```

Listing 3. FLAME code for the `SYRK` operation parallelized using `llc`

default, and we have to use `llc` directives to specify shared data. Listing 3 shows the parallelization of the FLAME code for the `SYRK` operation using `llc`.

A first comparison of Listings 2 and 3 shows an apparent increase in the number of directives when `llc` is used. However, note that only three directives are actually needed, but we split those in order to improve the readability. Although `llc` code can be sometimes as simple as OpenMP code (see, e.g., [4]), here we preferred to use an elaborated algorithm to illustrate how `llc` overcomes difficulties that usually appear when targeting parallel distributed memory architectures: references to specific data inside a larger data structure (submatrices instead of the whole matrix), access to non-contiguous memory locations, etc.

In the `llc` implementation of *Task Queues*, a master processor handles the task queue, sends subproblems to the slaves, and gathers the partial results to construct the solution. Before the execution of each task, the master processor needs to communicate some initial data to the slaves, using the `llc task_master_data` directive. As the master and slaves processors are on the same group, they have the same values in each private memory region. Exploiting this, the master processor only sends those data that have been modified. With this approach the number of directives to be used is larger than in the OpenMP case, but the amount of communications is considerably reduced.

The master needs to communicate to each slave the offset and number of elements of the objects A_0, A_1, C_{10}, and C_{11} (lines 2–4). After each execution, the slave processors "remember" the last data used. To avoid this, we employ the `llc task_slave_set_data` directives in lines 5–8 that initialize the variables before each task execution to certain fixed values (with no communications involved).

The code inside the parallel task computes $C_{10} := C_{10} + A_1 A_0^T$ and $C_{11} := C_{11} + A_1 A_1^T$. The slaves communicate to the master the results obtained (C_{10} and

C_{11}). These data are not stored in contiguous memory positions and therefore can not be communicated as a single block. However, the data follow a regular pattern and can be communicated using the `llc task_slave_rnc_data` directive (lines 9–10). This directive specifies *r*egular *n*on-*c*ontiguous memory locations.

4 Experimental Results

All the experiments reported in this section for the `SYRK` operation ($C := C + AA^T$, with an $m \times m$ matrix C and an $m \times k$ matrix A) were performed using double-precision floating point arithmetic. The results correspond to the codes that have been illustrated previously in this paper (FLAME Variant 1 of the `SYRK` operation, Var1) as well as a second variant (Var2) for the same operation [10].

Three different platforms were employed in the evaluation, with the common building block in all these being an Intel Itanium2 1.5GHz processor. The first platform is a shared-memory (SM) Bull NovaScale 6320 with 32 processors. The second platform is a SM SGI Altix 250 with 16 processors. The third system is a hybrid cluster composed of 9 nodes connected via a 10 Gbit/s InfiniBand switch; each node is a SM architecture with 4 processors, yielding a total of 36 processors in the system. An extensive experimentation was performed to determine the best block size (parameter `nb_alg` in the algorithms) for each variant and architecture. Only those results corresponding to the optimal block size (usually, around 96) are reported next.

The OpenMP implementations were compiled with the Intel C compiler, while the `llc` binaries were produced with `llCoMP` combined with the `mpich` implementation of MPI on the SGI Altix and hybrid cluster, and MPIBull-Quadrics 1.5 on the NovaScale server.

The goal of the experiments on SM platforms is to compare the performance of the `SYRK` implementation in OpenMP and `llc`. The results on the hybrid system are presented to demonstrate that high performance can be also achieved when the portability of `llc` is exploited.

Table 1 reports the results for the `SYRK` codes. In particular, the second row of the table shows the execution time of the sequential code, while the remaining rows illustrate the speed-up of the OpenMP and `llc` parallelizations on the SGI Altix and the Bull NovaScale.

The results show a similar performance for OpenMP and our approach on both architectures. OpenMP obtains a higher performance than `llc` when the number of processors is small. The reason for this behavior is that in the `llc` implementation one of the processors acts as the master. As the number of processors grows, the speed-up of `llc` increases faster than that of OpenMP. When the number of processors is large, `llc` yields better performance than OpenMP because it is less affected by memory bandwidth problems. The second variant of the algorithm exhibits a better performance than the first one, because it generates a larger number of tasks with finer granularity during the computations following a bidimensional partitioning of the work; see [10].

#Proc.	Var1 SGI		Var1 Bull		Var2 SGI		Var2 Bull	
seq.	*19.0 sec.*		*176.5 sec.*		*19.0 sec.*		*176.5 sec.*	
–	omp	llc	omp	llc	omp	llc	omp	llc
3	2.13	1.58	2.75	1.85	2.83	1.89	2.94	1.98
4	2.85	2.22	3.48	2.65	3.72	2.82	3.84	2.96
6	3.97	3.49	4.25	4.16	5.51	4.72	5.34	4.91
8	4.60	4.68	5.16	5.59	7.16	6.52	6.74	7.21
10	5.78	5.70	6.83	6.98	8.82	8.33	8.16	8.62
12	6.76	7.41	7.34	7.81	10.24	10.09	9.53	10.65
14	6.69	7.81	7.93	8.90	11.67	11.79	9.37	13.20
16	7.41	9.02	8.61	9.35	12.71	13.62	9.56	13.76

Table 1. Sequential time and speed-up obtained on the SM platforms for Variants 1 and 2 of the `SYRK` operation for both OpenMP and `llc`. For the Bull NovaScale 6320 (Bull), m=10000 and k=7000. For the SGI Altix 250 (SGI), m=6000 and k=3000.

Figure 1 shows the speed-up obtained on the hybrid system. Again the second variant exhibits a better performance, and a maximum speed-up slightly above 25 is attained using 36 processors.

5 Conclusions and Future Work

`llc` is an language based on C that, given a sequential code annotated with directives and using the `llCoMP` translator-compiler, produces MPI parallel code. `llc` combines the high productivity in code development of OpenMP with the high performance and the portability of MPI.

In this paper we have evaluated the performance of the *Task Queues* implementation in `llc` using FLAME codes for the `SYRK` operation. We have shown that the `llc` directives facilitate optimization and tuning. The additional complexity introduced in the `llc` version with respect to the OpenMP version is clearly paid off by the portability of the code. The performance achieved with our approach is comparable to that obtained using OpenMP. Taking into account the smaller effort to develop codes using `llc` compared with a direct MPI implementation, we conclude that `llc` is appropriate to implement some classes of parallel applications.

Work in progress concerning this topic includes the following:

- To study other variants and parallelization options for the `SYRK` operation, such as using two tasks per iteration or splitting the `while` loop.
- To study other FLAME operations. We are currently working on the matrix-vector product.
- To apply our approach to other scientific and engineering applications.
- To extend the computational results to other machines and architectures.

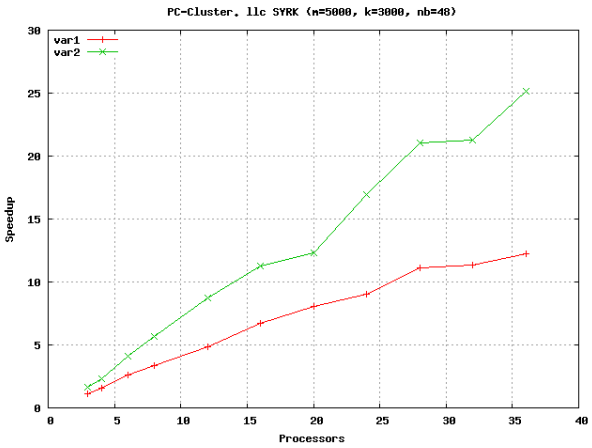

Fig. 1. Speed-up on the hybrid system for Variants 1 and 2 of the `SYRK` operation parallelized using `llc`. On this system, m=5000 and k=3000.

References

1. Message Passing Interface Forum, MPI: A Message-Passing Interface Standard, University of Tennessee, Knoxville, TN, 1995, `http://www.mpi-forum.org/`.
2. OpenMP Architecture Review Board, OpenMP Application Program Interface v. 2.5 (May 2005).
3. A. J. Dorta, J. A. González, C. Rodríguez, F. de Sande, llc: A parallel skeletal language, Parallel Processing Letters 13 (3) (2003) 437–448.
4. A. J. Dorta, P. Lopez, F. de Sande, Basic skeletons in `llc`, Parallel Computing 32 (7–8) (2006) 491–506.
5. A. J. Dorta, J. M. Badía, E. S. Quintana, F. de Sande, Implementing OpenMP for clusters on top of MPI, in: Proc. of the 12th European PVM/MPI Users' Group Meeting, Vol. 3666 of LNCS, Springer-Verlag, Sorrento, Italy, 2005, pp. 148–155.
6. S. Shah, G. Haab, P. Petersen, J. Throop, Flexible control structures for parallelism in OpenMP, Concurrency: Practice and Experience 12 (12) (2000) 1219–1239.
7. P. Bientinesi, J. A. Gunnels, M. E. Myers, E. S. Quintana-Ortí, R. A. van de Geijn, The science of deriving dense linear algebra algorithms, ACM Trans. on Mathematical Software 31 (1) (2005) 1–26.
8. C. L. Lawson, R. J. Hanson, D. R. Kincaid, F. T. Krogh, Basic linear algebra subprograms for fortran usage., ACM Trans. Math. Softw. 5 (3) (1979) 308–323.
9. G. H. Golub, C. F. Van Loan, Matrix Computations, 3rd Edition, Johns Hopkins University Press, Baltimore, MD, 1996.
10. F. Van Zee, P. Bientinesi, T. M. Low, R. A. van de Geijn, Scalable parallelization of FLAME code via the workqueuing model, ACM Trans. on Mathematical Software To appear. Electronically available at `http://www.cs.utexas.edu/users/flame/pubs.html`.