Implementing OpenMP for Clusters on Top of MPI*

Antonio J. Dorta[1], José M. Badía[2], Enrique S. Quintana[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 designed to extend OpenMP to distributed memory systems. Work in progress on the implementation of a compiler that translates `llc` code and targets distributed memory platforms is presented. Our approach generates code for communications directly on top of MPI. We present computational results for two different benchmark applications on a PC-cluster platform. The results reflect similar performances for the `llc` compiled version and an *ad-hoc* MPI implementation, even for applications with fine-grain parallelism.

Keywords: MPI, OpenMP, cluster computing, distributed memory, OpenMP compiler.

1 Introduction

The lack of general purpose high level parallel languages is a major drawback that limits the spread of High Performance Computing (HPC). There is a division between the users who have the needs of HPC techniques and the experts that design and develop the languages as, in general, the users do not have the skills necessary to exploit the tools involved in the development of the parallel applications. Any effort to narrow the gap between users and tools by providing higher level programming languages and increasing their simplicity of use is thus welcome.

MPI [1] is currently the most successful tool to develop parallel applications, due in part to its portability (to both shared and distributed memory architectures) and high performance. As an alternative to MPI, OpenMP [2] has emerged in the last years as the industry standard for shared memory programming. The OpenMP Application Programming Interface is based on a small set of compiler directives together with some library routines and environment variables.

* 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, contracts TIC2002-04498-C05-05 and TIC2002-04400-C03-03).

B. Di Martino et al. (Eds.): EuroPVM/MPI 2005, LNCS 3666, pp. 148–155, 2005.

One of the main drawbacks of MPI is that the development of parallel applications is highly time consuming as major code modifications are generally required. In other words, 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 computing: you can obtain the best performance but at the cost of quite a high development investment.

On the other hand, the fast expansion of OpenMP comes mainly from its simplicity. A first rough parallel version is easily built as no significative changes are required in the sequential implementation of the application. However, obtaining the best performance from an OpenMP program requires some specialized effort in tuning.

The increasing popularity of commodity clusters, justified by their better price/performance ratio, is at the source of the recent efforts to translate OpenMP codes to distributed memory (DM) architectures, even if that implies a minor loss of performance. Most of the projects to implement OpenMP in DM environments employ software distributed shared memory systems; see, e.g., [3,4,5]. Different approaches are developed by Huang *et al.* [6], who base their strategy for the translation in the use of Global Arrays, and Yonezawa *et al.* [7], using an *array section descriptor* called *quad* to implement their translation.

In this paper we present the language `llc`, designed to extend OpenMP to DM systems, and the `llc` compiler, `llCoMP`, which has been built on top of MPI. Our own approach is to generate code for communications directly on top of MPI, and therefore does not rely on a coherent shared memory mechanism. Through the use of two code examples, we show the experimental results obtained in a preliminary implementation of new constructs which have been incorporated into the language in order to deal with irregular memory access patterns. Compared to others, the main benefit of our approach is its simplicity. The use of direct generation of MPI code for the translation of the OpenMP directives conjugates the simplicity with a reasonable performance.

The remainder of the paper is organized as follows. In Section 2 we outline the computational model underlying our strategy. We next discuss some implementation details of the `llc` compiler in Section 3. Case studies and the experimental evaluation of a preliminary implementation of the new constructs are given in Section 4. Finally, we summarize a few concluding remarks and future work in Section 5.

2 The `llc` Computational Model

The `llCoMP` compiler is a source-to-source compiler implemented on top of MPI. It translates code annotated with `llc` directives into C code with explicit calls to MPI routines. The resulting program is then compiled using the native backend compiler, and properly linked with the MPI library. Whenever possible, the `llc` pragmas are compatible with their counterparts in OpenMP, so that minor changes are required to obtain a sequential, an MPI, or an OpenMP binary from the same source code.

The `llc` language follows the *One Thread is One Set of Processors* (*OTOSP*) computational model. Although the reader can refer to [8] for detailed information, a few comments are given next on the semantics of the model and the behaviour of the compiler.

The *OTOSP* model is a DM computational model where all the memory locations are private to each processor. A key concept of the model is that of *processor set*. At the beginning of the program (and also in the sequential parts of it), all processors available in the system belong to the same unique set. The processor sets follow a fork-join model of computation: the sets divide (fork) into subsets as a consequence of the execution of a *parallel construct*, and they join back together at the end of the execution of the construct. At any point of the code, all the processors belonging to the same set replicate the same computation; that is, they behave as a single thread of execution.

When different processors (sub-)sets join into a single set at the end of a *parallel construct*, *partner processors* exchange the contents of the memory areas they have modified inside the *parallel construct*. The replication of computations performed by processors in the same set, together with the communication of modified memory areas at the end of the *parallel construct*, are the mechanisms used in *OTOSP* to guarantee a coherent image of the memory.

Although there are different kinds of *parallel constructs* implemented in the language, in this paper we focus on the `parallel for` construct.

3 The `llCoMP` Compiler

The simplicity of the *OTOSP* model greatly eases its implementation on DM systems. In this section we expose the translation of parallel loops performed by `llCoMP`. For each of the codes in the NAS Parallel Benchmark [9] (columns of the table), Table 1 indicates the number of occurrences of the directive in the corresponding row. All the directives in Table 1 can be assigned a semantic under the premises of the *OTOSP* model. Using this semantic, the directives and also the data scope attribute clauses associated with them can be implemented using MPI on a DM system. For example, let us consider the implementation

Table 1. Directives in the NAS Parallel Benchmark codes

	BT	CG	EP	FT	IS	LU	MG	SP
`parallel`	2	2	1	2	2	3	5	2
`for`	54	21	1	6	1	29	11	70
`parallel for`		3					1	
`master`	2	2	1	10	4	2	1	2
`single`		12		5		2	10	
`critical`			1	1	1	1	1	
`barrier`				1	2	3	1	3
`flush`						6		
`threadprivate`			1					

of a `parallel` directive: since all the processors are running in parallel at the beginning of a computation, in our model the `parallel` directive requires no translation.

Shared memory variables (in the OpenMP sense) need special care, though. Specifically, any shared variable in the left-hand side of an assignment statement inside a `parallel loop` should be annotated with an `llc result` or `nc_result` clause. Both clauses are employed to notify the compiler of a region of the memory that is potentially modifiable by the set of processors which execute the loop. Their syntax is similar: the first parameter is a pointer to the memory region (`addr`), the second one is the size of that region (`size`), and the third parameter, only present in `nc_result`, is the name of the variable holding that memory region. Directive `result` is used when all the memory addresses in the range `[addr, addr+size]` are (potentially) modified by the processor set. This is the case, for example, when adjacent positions in a vector are modified. If there are write accesses to non-contiguous memory regions inside the parallel loop, these should be notified with the `nc_result` clause.

`llCoMP` uses *Memory Descriptors* (MD) to guarantee the consistency of memory at the end of the execution of a *parallel construct*. MD are data structures based on queues which hold the necessary information about memory regions modified by a processor set. The basic information holded in MD are pairs (*address*, *size*) that characterize a memory region. Prior to their communication to other processor sets, these memory regions (pairs) are compacted in order to minimize the communication overhead. In most of the cases, the communication pattern involved in the translation of a `result` or `nc_result` is an *all-to-all* pattern. The post-processing performed by a processor receiving a MD is straightforward: it writes the bytes received in the address annotated in the MD. In section 4 we present an experiment that has been designed to evaluate the overhead introduced in the management of MDs.

In [8] we presented several examples of code with the `result` directive. In this paper we focus in the implementation of non-contiguous memory access patterns, the most recent feature incorporated into `llCoMP`.

```
1  #pragma omp parallel for private(ptr, temp, k, j )
2  for (i=0; i<Blks->size1; i++) {
3    ptr = Blks->ptr[i];
4    temp = 0.0;
5    k = index1_coordinate(ptr); // First element in i-th row
6    for (j=0; j<elements_in_vector_coordinate(Blks, i); j++) {
7      temp += value_coordinate(ptr) *
8              x[index2_coordinate(ptr)*incx];
9      inc_coordinate(ptr);
10   }
11   #pragma llc nc_result(&y[k*incy], 1, y)
12   y[k*incy] += alpha * temp;
13 }
```

Listing 1. A parallelization of the USMV operation

In particular, consider the code in Listing 1, which shows the parallelization using `llc` of the main loop of the *sparse matrix-vector product* operation $y = y + \alpha Ax$, where x and y are both vectors and A is a sparse matrix (this is known as operation USMV in the Level-2 sparse BLAS). Matrix elements are stored using a rowwise coordinate format, but we also store pointers to the first element on each row in vector `ptr`. In the code, each iteration of the external loop in line 2 performs a dot product between a row of the sparse matrix and vector `x`, producing one element of the solution vector `y`. The code uses three C macros (`index1_coordinate(ptr)`, `index2_coordinate(ptr)` and `value_coordinate(ptr)`) in order to access to the row index, column index and value of an element of the sparse matrix pointed by `ptr`. A fourth macro, namely `inc_coordinate`, moves the pointer to the next element in the same row. Values `incx` and `incy` allow the code to access to vectors `x` and `y` with strides different from 1.

A direct parallelization of the code can be obtained having into account that different dot products are fully independent. Therefore, a `parallel for` directive is used in line 1 to indicate that the set of processors executing the loop in line 2 has to fork to execute the loop. The `llc` specific directive `nc_result` in line 11 indicates to the compiler that the value of the *y[k*incy]* element has to be "annotated".

4 Experimental Results

The experiments reported in this section were obtained on a cluster composed of 32 Intel Pentium Xeon processors running at 2.4 GHz, with 1 GByte of RAM memory each, and connected through a Myrinet switch. The operating system of the cluster was Debian Sarge Testing Linux. We used the MPICH [10] implementation on top of the vendor's communication library GM-1.6.3.

In order to evaluate the performance of the `llCoMP` translation we have used two benchmarks: the *sparse matrix-vector product* USMV, and a Molecular Dynamics (MD) simulation code [11]. These benchmarks were selected because they are composed of irregular, non-contiguous accesses to memory, and also because they are simple codes representative for a much larger class. Besides, the USMV operation is a common operation in sparse linear algebra, extremely useful in a a vast amount of applications arising, among many others, in VLSI design, structural mechanics and engineering, computational chemistry, and electromagnetics.

Using MPI and `llCoMP` we developed two parallel versions of the USMV code. Consider first the parallelization using MPI. For simplicity, we assume vectors x and y to be both replicated. With A distributed by rows, the matrix-vector product is performed as a series of inner products which can proceed in parallel. An *all-to-all* communication is required at the final stage to replicate the result y onto all nodes. On the other hand, using `llc` to implement the product, we parallelize the external for loop, so that each thread deals with a group of inner products (see Listing 1). As all threads share vectors x and y, it is not necessary to perform any additional gathering of partial results in this case.

Table 2. Execution time (in secs.) for the *ad hoc* MPI vs. `llCoMP` parallel versions of the USMV code. Problem sizes of 30000 and 40000 are employed with sparsity degrees of 1% and 2%.

	ad hoc MPI				llCoMP			
	30000		40000		30000		40000	
#Proc.	1%	2%	1%	2%	1%	2%	1%	2%
SEQ	11.09	21.55	26.67	45.16	11.09	21.55	26.67	45.16
2	5.85	11.99	11.59	30.11	8.24	15.15	21.70	34.08
4	3.64	6.65	8.26	16.72	4.85	8.18	10.34	21.34
8	2.23	3.33	4.44	11.39	2.93	4.23	6.42	10.57
16	1.62	2.13	2.86	6.76	1.86	2.68	3.48	6.57
24	1.45	1.82	2.43	6.15	1.80	2.32	3.29	6.23
32	1.27	1.55	2.00	5.25	1.94	2.40	3.02	4.35

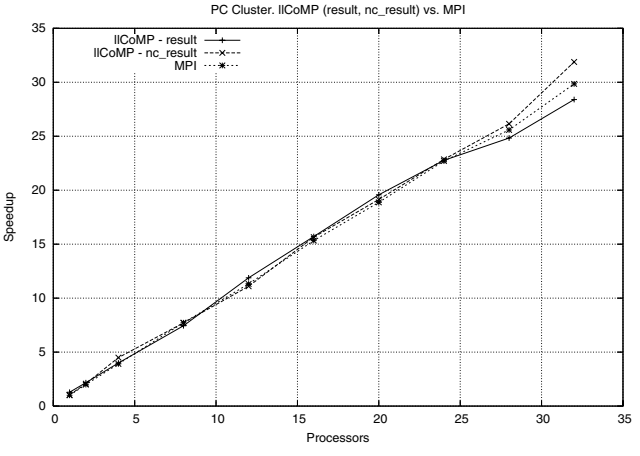

Fig. 1. MPI and `llCoMP` (`result` and `nc_result`) speedups for the MD code

Table 2 compares the accumulated execution time of 100 runs for the USMV code using an *ad hoc* MPI implementation and the translation produced by `llCoMP`. The executions correspond to square sparse random matrices of dimensions 30000, 40000 and sparsity factors of 1% and 2%.

The fine-grain parallelism present in the USMV code and the small amount of computations performed by this operation are at the source of the limited speed-up reported in the experiment. Not surprisingly, coarse-grain algorithms are the best scenario to achieve high performance for the translations provided by `llCoMP`. Nevertheless, if we compare the results obtained from an *ad hoc* program using MPI with those produced by the `llc` variant, we can expect this overhead to be compensated in some situations by the much smaller effort invested in the development of the parallel code.

The source code for the MD code written in OpenMP can be obtained from the *OpenMP Source Code Repository* [12,13]; translation of this code using `llc` is straight-forward. With this experiment our aim is to evaluate the overhead introduced by `llCoMP` in the management of non-regular memory access patterns. The MD code exhibits a regular memory access pattern, but as the `nc_result` clause is a general case of `result`, we have implemented it using both clauses. Figure 1 shows the speedup achieved by the MD code for three different implementations: and *ad hoc* MPI implementation and two different `llc` implementations using `result` and `nc_result`. We observe an almost linear behaviour for all the implementations. For this particular code, no relevant differences are appreciated when using regular and non-regular memory access patterns.

5 Conclusions and Future Work

We believe that preserving the sequential semantics of the programs is a major key to achieve the objective of alleviating the difficulties in the development of parallel applications. Surely the extension of the OpenMP programming paradigm to the DM case is a desirable goal. At the present time the technology and the ideas are not mature enough as to show a clear path to the solution of the problem and, in this line, our own approach does not intend to compete with other authors' work. We show that a compiler working under the premises of the *OTOSP* computational model and using direct generation of MPI code for communications can produce acceptable results, even in the case of fine-grain parallel algorithms.

Work in progress in our project includes the following issues:

- To unburden the final user of the specification of memory regions to be communicated (using the `result` clauses).
- To explore the potential sources of improvement for the compiler prototype.
- To collect OpenMP applications suitable to be targeted by the `llCoMP` compiler.

Acknowledgments

We wish to thank the anonymous reviewers for their suggestions on how to improve the paper.

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, electronically available at `http://www.openmp.org/drupal/mp-documents/spec25.pdf` (May 2005).

3. S.-J. Min, A. Basumallik, R. Eigenmann, Supporting realistic OpenMP applications on a commodity cluster of workstations, in: Proc. of of WOMPAT 2003, Workshop on OpenMP Applications and Tools, Toronto, Canada, 2003, pp. 170–179.
4. M. Sato, H. Harada, A. Hasegawa, Cluster-enabled OpenMP: An OpenMP compiler for the SCASH software distributed shared memory system., Scientific Programming, Special Issue: OpenMP 9 (2-3) (2001) 123–130.
5. Y. C. Hu, H. Lu, A. L. Cox, W. Zwaenepoel, OpenMP for Networks of SMPs, Journal of Parallel and Distributed Computing 60 (12) (2000) 1512–1530.
6. L. Huang, B. Chapman, Z. Liu, Towards a more efficient implementation of openmp for clusters via translation to global arrays, Tech. Rep. UH-CS-04-05, Department of Computer Science, Univeristy of Houston, electronically available at `http://www.cs.uh.edu/docs/preprint/2004_11_15.pdf` (dec 2004).
7. N. Yonezawa, K. Wada, T. Ogura, Quaver: OpenMP compiler for clusters based on array section descriptor, in: Proc. of the 23rd IASTED International Multi-Conference Parallel and Distributed Computing and Networks, IASTED /Acta Press, Innsbruck, Austria, 2005, pp. 234–239, electronically available at `http://www.actapress.com/Abstract.aspx?paperId=6530`.
8. A. J. Dorta, J. A. González, C. Rodríguez, F. de Sande, llc: A parallel skeletal language, Parallel Processing Letters 13 (3) (2003) 437–448.
9. D. H. Bailey et al., The NAS parallel benchmarks, Technical Report RNR-94-007, NASA Ames Research Center, Moffett Field, CA, USA, electronically available at `http://www.nas.nasa.gov/News/Techreports/1994/PDF/RNR-94-007.pdf` (Oct. 1994).
10. W. Gropp, E. Lusk, N. Doss, A. Skjellum, A high-performance, portable implementation of the message passing interface standard, Parallel Computing 22 (6) (1996) 789–828.
11. W. C. Swope, H. C. Andersen, P. H. Berens, 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 (1982) 637–649.
12. OmpSCR OpenMP Source Code Repository `http://www.pcg.ull.es/ompscr/` and `http://ompscr.sf.net`.
13. A. J. Dorta, A. González-Escribano, C. Rodríguez, F. de Sande, The OpenMP source code repository, in: Proc. of the 13th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP 2005), Lugano, Switzerland, 2005, pp. 244–250.