Exploiting task and data parallelism

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

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

Abstract. Although task and data parallelism have been usually considered mutually exclusive approaches to parallel programming, many applications may benefit from mixing both forms of parallelism. In this work we tackle this issue using two benchmark applications implemented using OpenMP and `llc`.
The `llc` language is the result of our study in the opportunities to extend OpenMP to deal with multi-level parallelism and distributed memory architectures. As in OpenMP, in `llc` parallelism is expressed using compiler pragmas. The `llc` compiler produces code for both shared and distributed memory platforms. We present computational results for our experiments using a Sun E15000 and a Intel Pentium IV-based Beowulf cluster.

1 Introduction

The use of tools and languages that exploit high performance computers and resources is nowadays limited. The main reason for this limitation is the amount of knowledge and skills necessary to exploit these resources. The community of technicians and scientists having needs of high performance computing is not interested in learning new sophisticated tools or languages. If the parallel computing community wants to make the use of our techniques widespread, additional efforts must be made in the way of providing high level tools to the end users.

No doubt in recent times OpenMP [1] has become the dominant high level standard tool to develop parallel applications for shared memory architectures. The simplicity of the OpenMP model where a parallel program is built from its sequential counterpart by adding compiler pragmas is one of the main reasons for its success. Despite the broad triumph that OpenMP has reached during its life, some of its aspects still require attention, design and development efforts.

In this paper we investigate the advantage of mixing task and data parallelism. Data parallelism occurs when the same operation is applied to different data, while task parallelism appears when multiple independent code segments are run concurrently. Although many applications can benefit from both forms of parallelism [2], task and data parallelism have been usually considered mutually exclusive approaches to parallel programming.

Closely related to the ability to mix these two forms of parallelism is the capacity to exploit multiple levels of parallelism. Although the OpenMP specification allows some form of nested parallelism, most of the compiler implementations simply serialize any nested parallel region [3], [4], [5].

From our point of view, to identify and build a set of representative applications that constitute a benchmark to measure the quality of different compiler implementations is another aspect that requires the attention of the OpenMP community. While it is not difficult to find benchmarks that pay attention to other features in the language [6], [7] there is a need of benchmarks measuring the capacity to exploit multi-level parallelism or to mix task and data parallelism.

In this work we have implemented two applications that can benefit from mixing task and data parallelism and we use them to measure the performance obtained on two different platforms using OpenMP and `llc`.

`llc` is the result of our research in the opportunities to extend OpenMP to deal with multi-level parallelism and distributed memory architectures. As in OpenMP, in `llc` parallelism is expressed through the introduction of compiler pragmas in the code. Wherever it is possible our pragmas are compatible with those existing in OpenMP. For the `llc` experiences we use `llCoMP`, our `llc` prototype compiler. `llCoMP` is a source to source compiler that translates the source C code annotated with `llc` pragmas into C code augmented with calls to MPI [8] functions.

Instead of developing our test applications from scratch we have used two applications from the CMU task parallel suite [9]. This set of programs were collected as realistic example applications from diverse domains that take advantage of mixing task and data parallelism.

The remainder of the paper is organized as follows: In section 2 we summarize the `llc` language and its underlying computational model. In section 3 we describe the experiments that we have performed. The implementations and the computational results obtained are discussed in section 4 and section 5 offers conclusions and comments on future work.

2 The `llc` language

In `llc` the programmer has different ways of expressing the parallel execution of a body. In this work we will limit our attention to the `parallel for` construct.

The *OTOSP* model (*One Thread is One Set of Processors*) is the theoretical computational model underlying to `llc`. The model helps to understand how we map the parallel constructs of the language in real parallel architectures.

Let us imagine a machine composed of an *infinite* number of processors, each with its own private memory and a network interface connecting them. The processors are organized in *sets*. At any time, the memory state of all the processors in the same set is identical. An *OTOSP* computation assumes that all the processors in the same set have the *same input data* and *the same program in memory*. The only difference among the processors is an internal register, not

available to the programmer, containing an integer, the *NAME* (*number*) of the processor.

At the beginning of any computation, all the infinite processors in the machine are in the same set, they execute the same (sequential) thread and have identical values stored in their local memories. When this set of processors reaches a `parallel for`, the group executing it is divided in subgroups. Each processor in the set decides in terms of its *NAME* to which subgroup it is attached. When the execution of a `parallel for` finishes, all the processors in the subgroups executing it are joined again together to form the original set.

Before the execution of a `parallel for`, the memory of the set of processors contains exactly the same values. When a `parallel for` is executed, the memory is divided in two parts: the one that is going to be modified and the one that is not changed inside the body of the construct. Variables in the latter part are available inside the body for reading. The others are partitioned among the new subgroups.

To guarantee that after returning to the previous group, the processors in the parent set have a consistent view of the memory and that they will behave as the same thread, some mechanism must be provided. When a subgroup is created, each processor in the subgroup sets up a *partnership* relation with one or more processors in the other subgroups. When sets are merged together, *partner* processors exchange the memory areas they have modified inside the `parallel for`.

The *OTOSP* model allows *parallel constructs* to be nested arbitrarily: each time a set of processors reaches a *parallel construct*, the set is divided, and this division process can be applied recursively.

The equations determining the formation of the subgroups in the division process are simple enough to allow an efficient implementation. Also the communication patterns between *partner* processors in the subgroups can be optimized. For details about these aspects, please refer to [10].

```
1  int ffts(complex *a, int *brt, complex *w) {
2    int i;

4    #pragma omp parallel for private(i)
5    #pragma llc result (&a[i * n], n)
6    for (i = 0; i < n; ++i)
7      fft(&a[i * n], brt, w);
8    return 0;
9  }
```

Listing 1. A parallel loop in `llc`

Let us use the simple code in listing 1 to study how `llc` implements the ideas of the *OTOSP* model. The `parallel for` at line 4 indicates that the iterations of the for loop at line 6 are independent, and consequently can be split among the processors in the current set.

In the *OTOSP* theoretical model there are always processors available to deal with a *parallel construct* because the 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 loop iterations (tasks) 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 on each. 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.

As we can see in listing 1 `llc` extends OpenMP syntax and semantics to deal with distributed memory computing. Whenever an OpenMP pragma with the required semantics exist we use it (note the use of the `omp` and `llc` prefixes in the pragmas). In fact, the clause `private` in line 4 is kept only for compatibility with OpenMP, since it is not required: all the storages are private in the *OTOSP* model.

The clause `result` in line 5 is associated with the `parallel for` in the previous line. 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 memory area starting at `&a[i * n]` and whose size is `n` is owned (and potentially could be modified) by the set of processors performing thread i.

For additional details concerning the `llc` language, its compiler and the *OTOSP* model please refer to [5] and [11].

3 Experiments

We decided to use the CMU task parallel suite [9] as target for our experiments because these programs share some positive features for our interests: they can benefit from a mix of task and data parallelism, they are representative in different application domains, have a manageable size, they are simple and finally, the source code is available.

```
for (i = 0; i < ITERS; i++) {
  T1();
  T2();
  ...
  TN();
}
```

Listing 2. General form of the codes

The general form of the programs in the CMU task parallel suite appears in listing 2. An outer loop performs several iterations over different input data sets. The body of the loop consists of calls to tasks routines that typically include data-parallel statements. Depending on the application, there may be dependences across iterations of the outer loop and among the tasks subroutines.

For the experiments we have chosen two of the five test programs present in the suite: the 1D fast Fourier transform (FFT) and the Multibaseline stereo imaging code. Complete C sources for these implementations can be obtained from [11].

3.1 The FFT algorithm

The discrete Fourier transform (DFT) is an important tool with plenty of applications in diverse fields of science and engineering. A DFT is a matrix-vector product $y = F_n x$ where x and y are complex vectors and $F_n = (f_{pq})$ is an $n \times n$ matrix such that $f_{pq} = w_n^{pq}$, with $w_n = cos(2\pi/n) - i\ sin(2\pi/n) = e^{-2\pi i/n}$ and $i = \sqrt{-1}$.

A fast Fourier transform is an efficient algorithm that exploits the structure in F_n to compute the DFT product in $O(n \log n)$ time. Higher dimensional FFTs can also be defined. The interested reader may refer to [12] for a detailed description of different FFT algorithms.

```
1   for(k = 0; k < ITERS; k++) {
2     dgen(xin, N);
3     tpose(xin, aux, N);
4     ffts(aux, brt, w, N, LOGN, N / 2);
5     scale(aux, v, N);
6     tpose(aux, xin, N);
7     ffts(xin, brt, w, N, LOGN, N / 2);
8     tpose(xin, aux, N);
9     chkmat(aux, N);
10  }
```

Listing 3. Bailey's "6-step" FFT implementation

The 1D FFT algorithm used in the CMU suite is the Bailey's "6-step" algorithm [13] and the code in listing 3 shows the basic steps.

If we consider the size of the input signal as the product of two numbers, $n_1 \times n_2$, then the 1D FFT can be computed using smaller independent 1D FFTs. The input vector can be reshaped as an $n_1 \times n_2$ matrix with n_1 being the width with sequential memory accesses and n_2 being the height with non-sequential accesses (in listing 3, $n_1 = n_2 = N$). Then the basic structure of Bailey's "6-step" FFT algorithm is:

1. Transpose the input data set, considered as a $n_1 \times n_2$ complex matrix, into a $n_2 \times n_1$ matrix (line 3 in listing 3).
2. Perform n_1 individual n_2-point one dimensional FFTs on the resulting $n_2 \times n_1$ matrix (line 4 in the code).
3. Multiply the resulting $n_2 \times n_1$ complex matrix A_{ij} by $e^{\pm 2\pi ijk/n}$ (line 5).
4. Transpose the resulting $n_2 \times n_1$ matrix into a $n_1 \times n_2$ matrix (line 6).
5. Perform n_2 individual n_1-point one dimensional FFTs on the resulting $n_1 \times n_2$ matrix (line 7).
6. Transpose the resulting $n_1 \times n_2$ complex matrix into a $n_2 \times n_1$ matrix (line 8).

The 1D FFT in the CMU benchmark iteratively computes a certain number ($ITERS$) of "6-step" FFTs on synthetic input data set (generated by `dgen` in line 2). Therefore, the input parameters for this program are the number of iterations and the size (N) of the input signal.

3.2 The Multibaseline stereo imaging code

Binocular stereo vision is a method used to obtain three-dimensional information of a scene. Stereo vision is computationally intensive and the spatially repetitive nature of the method lends itself to parallelization. The main drawback of stereo is the problem with image point correspondence. The trade-off between accuracy (which is aided by a wide baseline, or separation between the cameras) and ease of correspondence has been mitigated using multiple cameras or camera locations. This approach has been named multibaseline stereo.

```
1 for(count = 0; count < ITERS; count++) {
2   getdata(ref, m1, m2, curbesterr, curbestdisp);
3   for(curdisp = 0; curdisp < 16; curdisp++) {
4     gendiffimg(ref, m1, m2, diffimg, curdisp);
5     updatedispimg(diffimg, curbesterr, curbestdisp, curdisp);
6   }
7   testdata(curbestdisp);
8   dumpimg(curbestdisp);
9 }
```

Listing 4. The multibaseline stereo algorithm

The implementation of the multibaseline stereo code in the CMU suite [14] is an adaptation from a previous data-parallel implementation [15] that provides better accuracy through the use of three cameras. Listing 4 shows the source code for the algorithm. The input to the algorithm consists of three $m \times n$ images. Two of them are *match images* and the third is a *reference image*. For each of 16 disparities (loop in line 3) the first match image is shifted by d pixels, the second is shifted by $2d$ pixels. Function `getdata` in listing 4 generates synthetic images and perform these shiftings. A *difference image* is formed by computing the sum of squared differences between the corresponding pixels of the reference and the shifted match images (function `gendiffimg` in line 4). Next, an *error image* is formed by replacing each pixel in the difference image with the sum of the pixels in a surrounding 13×13 window. A *disparity image* is then formed by finding, for each pixel, the disparity that minimizes the error. Finally, the depth of each pixel is displayed as a simple function of its disparity. Function `updatedispimg` in listing 4 sums the difference image over the moving window and updates the current best error and best disparity images.

The input parameters for the multibaseline stereo benchmark are the size of the images ($m \times n$) and the number of iterations performed ($ITERS$).

4 Implementations and computational results

The implementations of the FFT algorithm using both OpenMP and `llc` have been quite straightforward. We have used `pragma parallel for` to parallelize the external for loop and the same construction wherever it was possible in the task subroutines. In the case of the FFT we turned into parallel the `tpose`, `ffts` and `scale` routines. Listing 1 contains the code for the `ffts` routine in `llc`.

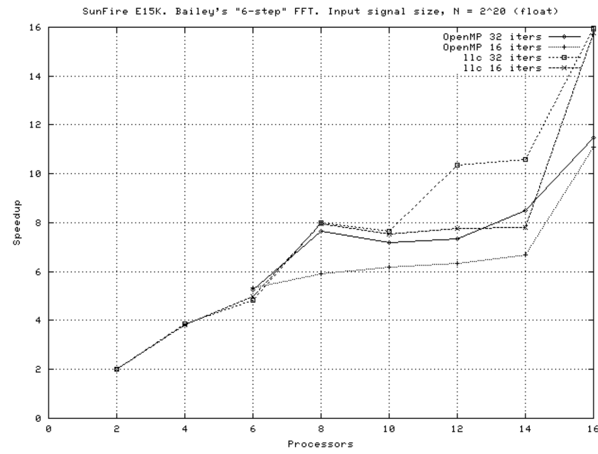

Fig. 1. Computational results for the FFT algorithm

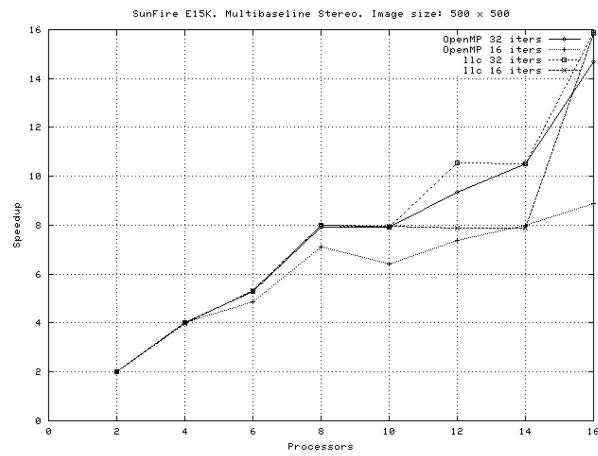

Fig. 2. Computational results for the multibaseline stereo algorithm

As it was explained in section 2, the `llc pragma result` in line 5 ensures a coherent view of the memory areas that have been modified by each processor subgroup and it is unnecessary in the OpenMP implementation. In fact, OpenMP compilers ignore it.

For the stereo algorithm, again we used `pragma parallel for` in the external loop (line 1 in listing 4) and `gendiffimg` is the only task subroutine that has been parallelized using `llc`. At first glance it is appealing to parallelism also the for loop in line 3, but data dependences in the computation of the best *disparity image* prevent from this parallelization in `llc` (each iteration of the loop depends on the previous one).

In the case of OpenMP we have parallelized not only the loops in lines 1 and 3 and the `gendiffimg` routine, but also we have been able to exploit additional levels of parallelism in the `updatedispimg` function, using locks to deal with the data dependences. Nevertheless, most OpenMP compilers ignore nested parallel regions, and therefore they only take advantage of the parallel for loop in line 1. To be fair in the comparison with `llc` we have checked that parallelizing in OpenMP only the inner loop in line 3 delivers poorer performance than exploiting parallelism in the outer level.

The experiments have been performed using two different platforms: a SMP SunFire E15000 with 900 MHz UltraSPARC-III processors and a Intel Pentium IV-based Beowulf cluster using a Myrinet interconnection network (1.2 Gigabit/sec).

In the SunFire we used the Forte C Developer compiler [16] with level 3 optimizations both for OpenMP and `llc` (MPI) versions. For the beowulf cluster we used `gcc` and MPICH `mpicc` compilers with the same optimization level.

Figures 1 and 2 show respectively the results obtained for the FFT and multibaseline stereo algorithms in the SunFire E15K. The figures show the results for both OpenMP and `llc` implementations. For the FFT we used $N = 2^{20}$ (`float`) as input vector size with 16 and 32 iterations ($ITERS$). For the multibaseline stereo, we used the same number of iterations and the size of the images was 500×500 pixels (`int`).

The performance is similar for both implementations. The results show that while `llc` takes advantage of multi-level parallelism, and it reaches linear speedup, the Sun compiler does not.

Finally, figure 3 present the results for the `llc` implementations in the Intel Beowulf cluster. The layered shape in this figure can be explained by the processor division behavior in `llc`. The performance is similar for both algorithms and it is almost linear.

5 Conclusions and future work

Two applications from the CMU task parallel suite have been ported to C to be implemented using `llc`. We have used these codes to show that mixing task and data parallelism is a valuable property in the way of achieving good performance.

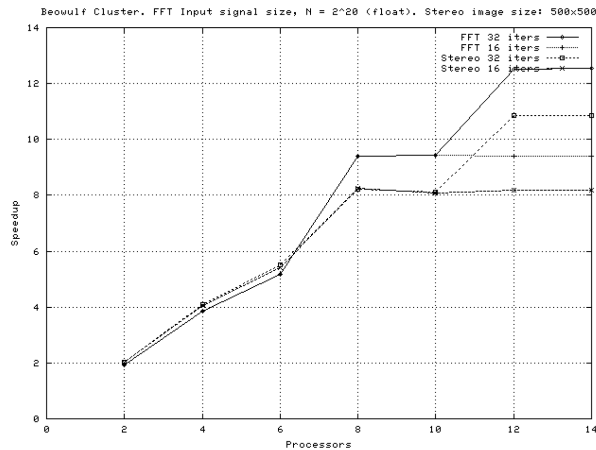

Fig. 3. Computational results in the Intel Beowulf cluster

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.

In the near future we plan to measure the performance of our benchmarks using different compilers, particularly those supporting nested parallelism.

We also project to try alternate parallel implementations of these test codes. OpenMP and `llc` source versions allow some improvements if we produce different codes for each alternative. In this first approach we have addressed the simplest way trying to imitate a non-expert user, and preserving the sequential semantic of the codes.

We plan to continue implementing the remaining CMU task parallel suite applications using `llc` and to carry out the corresponding experiments.

We think that setting up a common freely accessible and non-proprietary standard benchmark suite should be a positive step for the development of OpenMP. This suite should be accepted by researchers in the field as a way for promoting discussion and interchange of results for different issues that require attention. In this direction we think that the FFT and multibaseline stereo algorithms are two candidates to be considered.

Acknowledgements

The authors want to acknowledge the support of the European Commission through grant number HPRI-CT-1999-00026 (the TRACS Programme at EPCC). This work has been partially supported by the EC (FEDER) and the Spanish MCyT (Plan Nacional de I+D+I, contracts TIC2002-04498-C05-05 and TIC2002-04400-C03-03).

References

1. OpenMP Architecture Review Board. *OpenMP C/C++ Application Program Interface*, March 2002. Electronically available at http://www.openmp.org/specs/mp-documents/cspec20.pdf.
2. Soumen Chakrabarti, James Demmel, and Katherine A. Yelick. Modeling the benefits of mixed data and task parallelism. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 74–83, 1995.
3. Marc Gonzalez, Jose Oliver, Xavier Martorell, Eduard Ayguade, Jesus Labarta, and Nacho Navarro. OpenMP extensions for thread groups and their run-time support. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, volume 2017 of *LNCS*, pages 324–338, 2001.
4. 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.
5. 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.
6. H. Jin, M. Frumkin, and J. Yan. The OpenMP implementation of NAS parallel benchmarks and its performance. Technical Report NAS-99-011, NASA Ames Research Center, October 1999. Electronically available at http://www.nas.nasa.gov/Research/Reports/Techreports/1999/PDF/nas-99-011.pdf.
7. Vishal Aslot, Max Domeika, Rudolf Eigenmann, Greg Gaertner, Wesley B. Jones, and Bodo Parady. SPEComp: A new benchmark suite for measuring parallel computer performance. In *Proc. of of WOMPAT 2001, Workshop on OpenMP Applications and Tools*, volume 2104 of *LNCS*, pages 1–10, West Lafayette, IN, USA, 2001. http://link.springer-ny.com/link/service/series/0558/tocs/t2104.htm.
8. Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. University of Tennessee, Knoxville, TN, June 1995. http://www.mpi-forum.org/.
9. P. Dinda, T. Gross, D. O' Hallaron, E. Segall, J. Stichnoth, J. Subhlok, J. Webb, and B. Yang. The CMU task parallel program suite. Technical Report CMU-CS-94-131, School of Computer Science, Carnegie Mellon University, March 1994.
10. 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.
11. Francisco de Sande and Antonio J. Dorta. *llc Project web site*, 2002. http://nereida.deioc.ull.es/~llCoMP/.
12. Charles F. Van Loan. *Computational Frameworks for the Fast Fourier Transform*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1992.
13. David H. Bailey. FFTs in external or hierarchical memory. *The Journal of Supercomputing*, 4(1):23–35, March 1990.
14. M. Okutomi and T. Kanade. A multiple-baseline stereo. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(4):353–63, 1993.
15. Jon A. Webb. Implementation and performance of fast parallel multi-baseline stereo vision. In *Proceedings of the Computer Architectures for Machine Perception*, pages 232–240, New Orleans, LA, USA, Dec 1993.
16. Sun Microsystems. Forte Developer 7: C User's Guide. http://docs.sun.com/db/doc/816-2454.