Exploiting Nested Independent FORALL Loops in Distributed Memory Machines

J.A. González C. León C. Rodríguez F. Sande

I. Introduction

Since the early stages of parallel computing, one of the most common solutions to introduce parallelism has been to extend a sequential language with some sort of parallel version of the *for* construct, commonly denoted as *forall* construct. Although similar in syntax, these *forall* loops differ in their semantics and implementations. The High Performance Fortran (HPF) and OpenMP versions are, likely, among the most popular [8], [10]. This paper presents yet another *forall* loop extension for the C language. These extensions are aimed both for homogeneous distributed memory and shared memory architectures.

There are several differences with most current versions of HPF and OpenMP. One is that the parallel programming model introduced has associated a complexity model that allows the analysis and prediction of the performance. The other is that it allows and exploits any nested levels of parallelism, taking advantage of situations where there are several small nested loops: although each loop does not produce enough work to parallelize, their union suffices. Perhaps the most paradigmatic example of such family of algorithms are recursive divide and conquer algorithms. True, the last language specification of both HPF and OpenMP allow some form of nested parallelism. Nevertheless, most current implementation lack these features.

There is a third interesting feature of the model: it does not only integrates and extends the sequential programming model but also includes and expands the message passing programming model.

II. Syntax and Semantic

As OpenMP, the programming model being introduced, extends the classic sequential imperative paradigm with two new constructs: parallel sections and parallel loops. In contrast with OpenMP, the model aims for both distributed and shared memory architectures. The implementation on the last can be considered the "easy part" of the task.

A simplified version of the current syntax of parallel loops appears in figure 1. The programmer states that the different iterations i of the loop can be performed independently in parallel. The results of the execution of the i-th iteration are stored in the variables pointed by r_{1i}, r_{2i}, etc. Their sizes are respectively s_{1i}, s_{2i}, etc. To establish the semantic, let us imagine a machine composed of a number of infinite processors, each one with its own private memory and a network interface connecting them. We will call it the One-Thread Multiple-Processor Machine, abbreviated *OTMP*. The processors are organized in groups. At any time, the memory state of all the processors in the same group is identical. An *OTMP* computation assumes that they also have the same input data and the same program in memory. The initial group is composed of all the processors in the machine. Each processor is a *RAM* machine [1], the only difference among them is an internal register, containing an integer, the *NAME* (or *NUMBER*) of the processor. This register is not available to the programmer. When reaching the former *forall* loop, each processor decides in terms of its *NAME* the corresponding initialization of i and its subsequent value of the register *NAME*. Each independent thread $<$ $compound\_statement_i$ $>$ is executed by a subgroup. The simple equations performed by any processor *NAME* ruling the division process are:

forall(i= e_1; i<= e_2)
  result $(r_{1i}\, s_{1i}),\ (r_{2i}\, s_{2i}) \ldots (r_{mi}\, s_{mi})$
  $compound\_statement_i$

Fig. 1. *forall* simplified syntax

Number of iterations to do: $M = e_2 - e_1 + 1$ (1)

Iteration to do: $i = e_1 + NAME \,\%\, M$ (2)

for (j = 1; j < M; j++)

$$neighbour[j] = \Phi + (NAME + j) \,\%\, M \quad (3)$$

where Φ is given by: $\Phi = M \times (NAME \,/\, M)$

New value of register: $NAME = NAME \,/\, M$ (4)

These equations decide the mapping of processors to tasks, and how the exchange of results will take place. After the *forall* is finished, the processors recover their former value of *NAME*.

Each time a *forall* loop is executed, the memory of the group, up to that point contains exactly the same values. At such point the memory is divided

```
1  forall(i=1; i<=3)  result(ri+i, si[i]); {
2    ...
3    forall(j=0; j<=i)  result(rj+j, sj[j]); {
4      int a, b;
5      ...
6      if (i % 2 == 1)
7        if (j % 2 == 0)
8          send(j+1, a, sizeof(int));
9        else receive(j-1, b, sizeof(int));
10     ...
11   }
12   ...
13 }
```

Fig. 2. Two nested foralls

in two parts: the one that is going to be modified and the one that is not changed inside the loop. Variables in the last set are available inside the loop for reading. The others are partitioned among the new groups. The clause *result* next to the *forall* has the purpose to inform the compiler of the new "ownership" of the part of the memory that is going to be modified. It announces that the group performing thread i "owns" (and presumably is going to modify) the memory areas delimited by r_{1i} s_{1i}, r_{2i} $s_{2i}, \ldots$

To guarantee that after returning to the previous group, the processors in the father group have a consistent view of the memory and that they will behave as the same thread, it is necessary to provide the exchange among neighbors of the variables that were modified inside the *forall* loop.

Let us denote the execution of body of the $i-th$ thread (*compound_statement*$_i$) by T_i. Each independent thread produces a set of results: r_{ij} is a pointer to the memory area containing the j-th result of T_i. The size of this result is s_{ij}. The interval $[r_{ij}, s_{ij}]$ denotes the addresses from the address pointed by r_{ij} and goes s_{ij} bytes. The semantic imposes two restrictions:

1. Given two different independent threads T_i and T_k and two different result items r_{ij} and r_{kt}, it holds:
$$[r_{ij}, s_{ij}] \bigcap [r_{kt}, s_{kt}] = \emptyset \ \forall i, j, k, t$$
2. For any thread T_i and any result j, all the memory space defined by $[r_{ij}, s_{ij}]$ has to be allocated previously to the execution of the thread body. This makes impossible the use of non-contiguous dynamic memory structures like lists or trees in the threads results.

The programmer has to be specially conscious of the first restriction: it is mandatory that the address of any memory cell written during the execution of T_i has to belong to one of the intervals in the list of results for the thread.

As an example, consider the code in figure 2. Initially, the infinite processors are in the same group, represented by the root of the tree in figure 3. All the processors are executing the same thread and have identical values stored in their local memory. Applying equations 1, 2, 3 and 4, the parallel loop in line 1 of figure 2 divides the group in three. After the execution of the loop, and to keep the coherence of the memory, each processor exchanges with its two neighbors the corresponding results. Thus, processor 0 in the group executes iteration `i=1` and sends `si[1]` bytes starting at the address pointed by `ri+1` to processors 1 and 2. Furthermore, it receives from processor 1 `si[2]` bytes and stores those bytes in its local address starting at the `ri+2`. Analogously, receives from processor 2 `si[3]` bytes starting at the address pointed by `ri+3`. The same exchange is repeated among the other corresponding triplets $((3,4,5),(6,7,8),\ldots)$. You can easily visualize the operation if you realize that any new nested *forall* creates/structures the current subgroup according as a *M-ary* hypercube, where M is the number of iterations in the parallel loop and the neighborhood relation is given by formula 3. Thus, this first *forall* produces "the face" of a ternary hypercubic dimension, where every corner has two neighbors.

The second and nested *forall* at line 3 requests for different number of threads in the different groups. The first group (`i = 1`) executes a loop of size 2, and so the group is divided following a binary hypercube. The loop for the second group (`i = 2`) is of size 3, and the processors in this subgroup are accordingly divided in a ternary dimension. The last group (`i = 3`) executes a *forall* of size 4, and consequently the group is partitioned in 4 subgroups. In this 4-ary dimension, each processor is connected with 3 neighbors in the other subgroups. Therefore, at the end of the nested compound statement, processor 17 will send `rj+1` to processors 14, 20 and 23 and will receive from them `rj+0`, `rj+2` and `rj+3`. The same will occur with any of the quartets involved.

The complexity $\mathcal{T}(\mathcal{P})$ of any *OTMP* program $\mathcal{P}$ can be computed in what refer to sequential ordinary constructs (*while, for, if, ...*) as in the RAM machine [1]. The cost of the *forall* loop is given by the recursive formula:

$$\mathcal{T}(forall) = A_0 + A_1 \times M + \max_{i=e_1}^{e_2} (\mathcal{T}(T_i)) + \sum_{i=e_1}^{e_2} (g \times N_i + L) \quad (5)$$

where T_i is the code of < *compound_statement*$_i$ >, M is the size of the loop

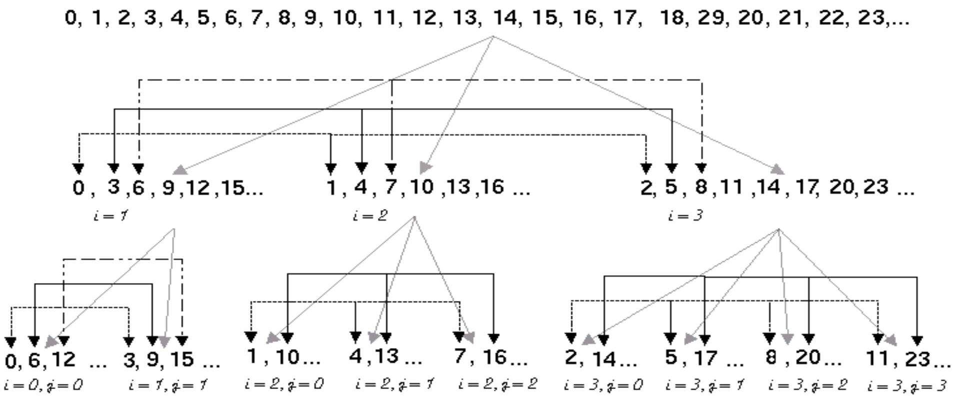

Fig. 3. The mapping associated with the two nested *forall* in figure 2

and N_i is the size of the message transferred in iteration i,

$$M = (e_2 - e_1 + 1) \qquad N_i = \sum_{k=1}^{m} s_{ki}$$

A_0 is the constant time invested computing formulas 1, 2 and 4. A_1 is the time spent finding the neighbors (formula 3). Constant g is the inverse of the bandwidth, and L is the startup latency.

A remarkably fact, is that the *OTMP* machine not only, generalizes the sequential RAM but also the Message Passing Programming Model. Lines 6-8 in figure 2 illustrate the idea. Each new *forall* "creates" a communicator (in the MPI sense). The execution of lines 7 and 8 in the fourth group (`i=3`) implies that thread `j=0` sends `a` to thread `j=1`. This operation carries that, at the same time that processor 2 sends its replica of "a" to processor 5, processor 14 sends its copy to processor 17 and so on. To summarize: any send or receive is executed by the infinite couples involved. Still, the two aforementioned constraints have to be true. Any variable modified inside the loop and non local to the loop has to be allocated before the loop and has to appear inside the return clause.

The *OTMP* model admits several coherent extensions, including reduction clauses (with syntax and semantic similar to the ones in HPF and OpenMP) and an equivalent to the OpenMP *section* construct. We will not treat them here, for sake of briefness, and since the emphasis of this paragraph is in the (simplified) model rather than in the implementation details.

III. Examples

The first problem to solve is to compute *tasks* matrix multiplications ($C^i = A^i \times B^i \quad i = 0, \ldots tasks-1$). Matrix A^i and B^i have dimensions $m \times m$. Figure 5 shows the algorithm. Although

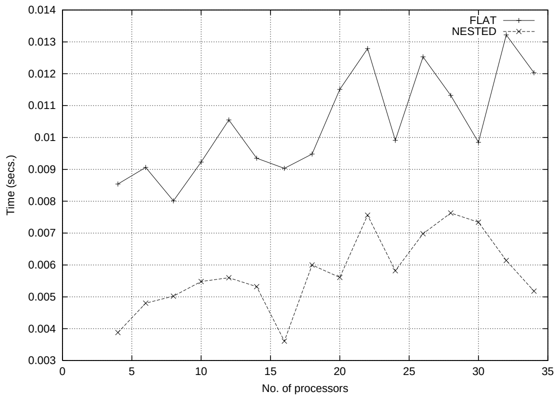

Fig. 4. Nested versus Flat parallelism. Cray T3E

```
1   forall(t = 0; t < tasks)
2   result(C[t], m * m) {
3     forall(i = 0; i < m)
4     result(C[t][i], m) {
5       for(j = 0; j < m; j++)
6         for(C[t][i][j] = 0.0, k = 0; k < n; k++)
7           (C[t][i][j] += A[t][i][k] * B[t][k][j];
8     }
9   }
```

Fig. 5. Exploiting 2 levels of parallelism

all the *for* loops are candidates to be converted to *forall* loops, we have considered two cases: the parallelization of only the loop in line 3 (labelled FLAT in the results) and the one showed in figure 5 where, additionally, the loop at line 1 is also converted to a *forall* (label NESTED).

When the dimension of the matrices and the number of tasks are small, neither the inner nor the outer loop provide by themselves enough work to get advantage of the available number of processors. However, as figure 4 shows, the combination of the two nested loops provides enough work for them. When the size of the problems to solve is

```
for(t = 0; t < tasks; t++) {
  #pragma omp parallel for default(none)
                    private(i, j, k)
                    shared(t, A, B, C, n, m)
  for(i = 0; i < m; i++) {
    for(j = 0; j < m; j++)
      for(C[t][i][j] = 0.0, k = 0; k < n; k++)
        C[t][i][j] += A[t][i][k] * B[t][k][j];
  }
}
```

Fig. 6. OpenMP implementation of flat parallelism.

```
omp_set_nested(1);
#pragma omp parallel for default(none)
                 private(t, i, j, k)
                 shared(A, B, C, tasks, n, m)
for(t = 0; t < tasks; t++) {
  #pragma omp parallel for shared(A, B, n, m)
                       firstprivate(C)
  for(i = 0; i < m; i++) {
    for(j = 0; j < m; j++)
      for(C[t][i][j] = 0.0, k = 0; k < n; k++)
        C[t][i][j] += A[t][i][k] * B[t][k][j];
  }
}
```

Fig. 7. Nested parallelism in OpenMp

large, the speedup reached is linear as it shows figure 8. This behaviour is also observed in the Sun HPC 6500. Figures 6 and 7 present an OpenMP standard compliant implementation of this example. Exploitation of both flat and nested parallelism are considered again.

Figure 9 compares the flat case implementation in standard OpenMP and OTMP. As it could be expected, the implementation on native threads is slightly more efficient than the OTMP version [7]. Current OpenMP compilers do not support nested parallelism as it is shown in Figure 10 (label OpenMP). As we have stated previously, the implementation of the OTMP model on a shared memory architecture is not a difficult task. Figure 10 also compares the OTMP model implementation on native threads (label OpenMP-OTMP) and using MPI (MPI-OTMP). The use of native programming model justifies the better performance.

In the next examples we will concentrate in the more interesting case where the nested loops are small and, according to the complexity analysis, there are few opportunities for parallelization.

The *OTMP* code in figure 11 implements the Fast Fourier Transform (FFT) algorithm developed by Tukey and Cooley [4]. Parameter $W_k = e^{-2\pi ik/N}$ contains the powers of the N-th root of the unity. The computational results for such implementation on a Cray T3E appear in Figure 12. The speedup curve behaves according to what a complexity analysis following formula 5 predicts: a logarithmic increase in the speedup.

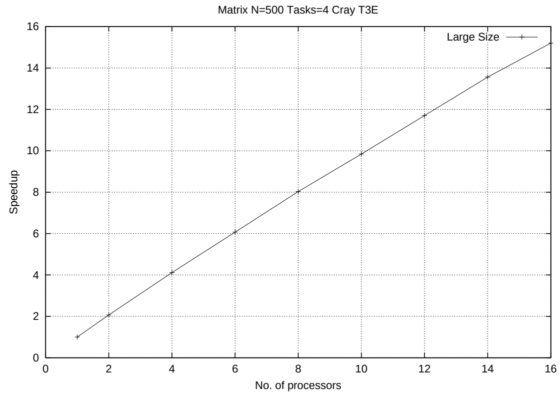

Fig. 8. Speedup reached for large size problems

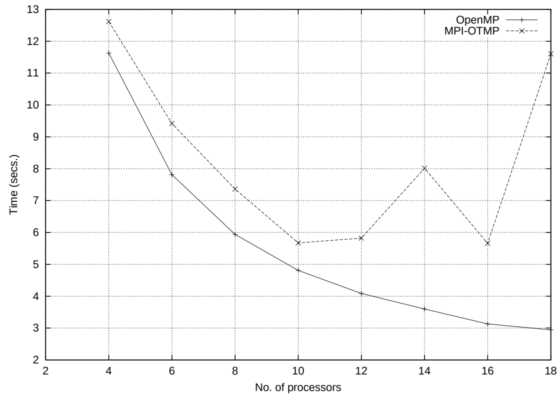

Fig. 9. OTMP vs. OpenMP Flat parallelism on the Sun HPC 6500

The third example is the well-known quicksort algorithm [6]. This and the next example are specially interesting, since the size of the generated sub-vectors are used as weights for the *forall*. Remember that, depending on the goodness of the pivot chosen, the new subproblems may have rather different weights.

Figure 13 presents the speed-ups on a digital Alpha Server, an Origin 2000, an IBM SP2, a CRAY T3E and a CRAY T3D. The size of the problem was 1M integers.

The QuickHull algorithm [5] constitutes our fourth example. The algorithm shares a few similarities with its namesake, QuickSort: QuickHull is also recursive and each recursive step partitions data into several groups.

Figure 14 presents the speedups for different platforms: A digital Alpha Server, a Hitachi, an Origin 2000, an IBM SP2, a CRAY T3E and a CRAY T3D. The size of the problem was 2M points.

IV. Conclusions

In this work we have introduced the OTMP programming model. Among other features, the model

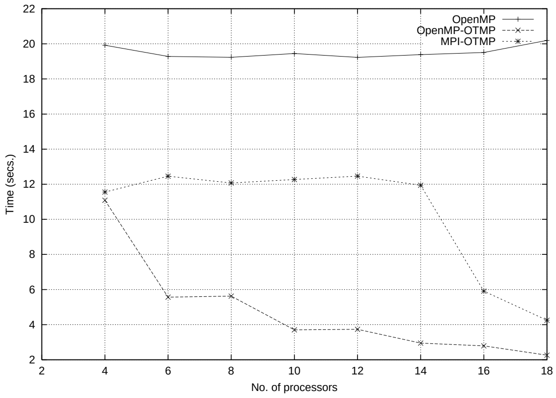

Fig. 10. Nested parallelism on the Sun HPC 6500

```
void llcFFT(Complex *A, Complex *a, Complex *W,
            unsigned N, unsigned stride,
            Complex *D) {
  Complex *B, *C, Aux, *pW;
  unsigned i, n;

  if(N == 1) {
    A[0].re = a[0].re;
    A[0].im = a[0].im;
  }
  else {
    n = (N >> 1);
     forall(i = 0; i <= 1)
     result(D+i*n , n) {
      llcFFT(D+i*n, a+i*stride, W, n,
             stride<<1, A+i*n);
    }
    B = D;             /* Combination phase */
    C = D + n;
    for(i = 0,pW = W; i<n; i++,pW += stride) {
      Aux.re = pW->re*C[i].re - pW->im*C[i].im;
      Aux.im = pW->re*C[i].im + pW->im*C[i].re;
      A[i].re = B[i].re + Aux.re;
      A[i].im = B[i].im + Aux.im;
      A[i+n].re = B[i].re - Aux.re;
      A[i+n].im = B[i].im - Aux.im;
    }
  }
}
```

Fig. 11. The *OTMP* implementation of the FFT

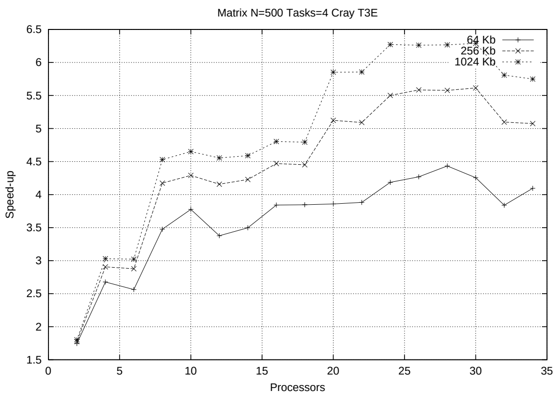

Fig. 12. FFT. Cray T3E. Different sizes

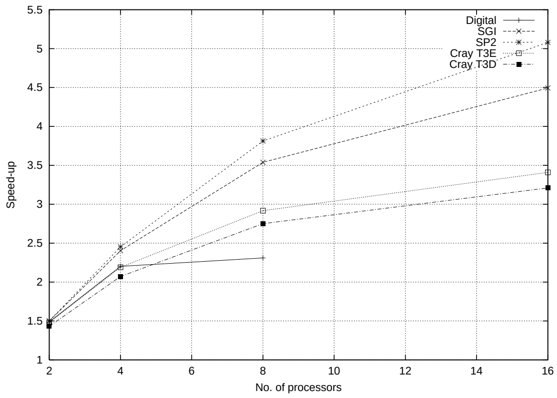

Fig. 13. QuickSort. Size of the vector: 1M

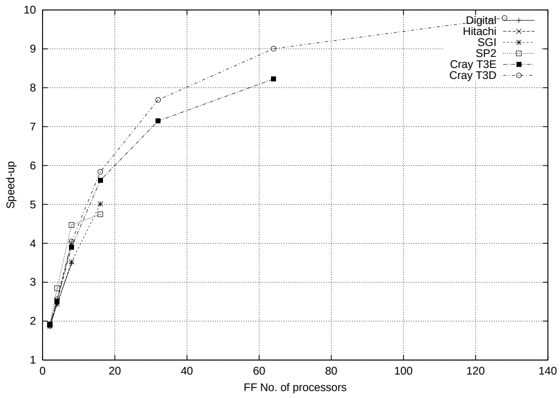

Fig. 14. QuickHull. Number of points: 2M

allows the exploitation of nested parallelism. We have presented a compiler prototype for this model, built on top of MPI. Results obtained both in shared and distributed memory architectures prove the feasibility of exploiting nested parallelism with the model.

There are many sources to improve the compiler prototype. Even incorporating these improvements, it is difficult to achieve the performance obtained using the native programming model (MPI, OpenMP) for the target architecture. However, the combination of the following aspects,

- the OTMP constitutes a proposal for exploiting nested parallelism,
- the model guarantees the portability to any platform,
- its easiness of programming,
- the improving obtained in performance,
- the fact that it does not introduce any overhead for the sequential case,
- the existence of an accurate performance prediction model,
- its benefit with respect to the OpenMP approach for shared memory machines and therefore, its portability to these kind of architec-

tures and
- the fact that it does not only extend the sequential but the MPI programming model

makes worth the research and development of tools oriented to this model.

Acknowledgments We wish to thank to the Edinburgh Parallel Computing Centre (EPCC), to the Centro de Investigaciones Energéticas, Medioambientales y Tecnológicas (CIEMAT), Centre Europeu de Parallelisme de Barcelona (CEPBA) and Centre de Supercomputació de Catalunya (CESCA). During this research, Dr. de Sande was on leave at the EPCC with a grant from the European Community (contract No HPRI-CT-1999-00026). This research also benefits from the support of Secretaría de Estado de Universidades e Investigación, SEUI, project MaLLBa, TIC1999-0754-C03.

References

[1] Aho, A. V. Hopcroft J. E. and Ullman J. D.: The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, Massachusetts, (1974).

[2] Ayguade E., Martorell X., Labarta J., Gonzalez M. and Navarro N. Exploiting Multiple Levels of Parallelism in OpenMP: A Case Study Proc. of the 1999 International Conference on Parallel Processing, Aizu (Japan), September 1999.

[3] Blikberg R., Sørevik T.. Nested parallelism: Allocation of processors to tasks and OpenMP implementation. Proceedings of The Second European Workshop on OpenMP (EWOMP 2000). Edinburgh, Scotland, UK. 2000

[4] Cooley, J. W. and Tukey, J. W.: An algorithm for the machine calculation of complex Fourier series, Mathematics of Computation, **19**, 90, (1965) 297–301.

[5] Eddy, W.: A New Convex Hull Algorithm for Planar Sets, ACM Transactions on Mathematical Software 3(4), (1977) 398–403.

[6] Hoare, C. A. R.: Quicksort, Computer Journal, 5(1), (1962) 10–15.

[7] Gonzalez J.A., Leon C., Piccoli F., Printista M., Roda J.L., Rodriguez C., Sande, F. Towards Standard Nested Parallelism. Proceedings of the 7th EuroPVM/MPI Users Grooup Meeting. Springer-Verlag LNCS, 1908, pp. 96-103. 2000

[8] High Performance Fortran Forum: High Performance Fortran Language Specification. Version 2.0 http://dacnet.rice.edu/Depts/CRPC/HPFF/versions/hpf2/hpf-v20/index.html (1997)

[9] MPI Forum: MPI-2: Extensions to the Message-Passing Interface, http://www.mpi-forum.org/docs/mpi-20.ps.Z (1997).

[10] OpenMP Architecture Review Board: OpenMP C and C++ application program interface v. 1.0 - October. http://www.openmp.org/specs/mp-documents/cspec10.ps (1998).