Solving Parallel Problems by OTMP Model

F. Piccoli M. Printista
J.A. González C. León J.L. Roda C. Rodríguez F. Sande

Universidad Nacional de San Luis. Centro Superior de Informática.
Ejército de los Andes 950, San Luis, Argentina. Universidad de La Laguna, Tenerife, Spain.
e-mail: {**mpiccoli@unsl.edu.ar**}

Abstract

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 syntax, these *forall* loops differ in their semantics and implementations. The High Performance Fortran (HPF) and OpenMP versions are, likely, among the most popular. This paper presents yet another *forall* loop extension for the C language.

In this work, we introduce a parallel computation model: One Thread Multiple Processor Model (OTMP). This model proposes an abstract machine, a programming model and cost model. The programming model defines another *forall* loop construct, the theorical machine aims for both homogeneous shared and distributed memory computers, and the cost model allows the prediction of the performance of a program. OTMP does not only integrates and extends sequential programming, but also includes and expands the message passing programming model. The model allows and exploits any nested levels of parallelism, taking advantage of situations where there are several small nested loops.

Keywords: Computation Model, Abstract Machine, Programming Model, Cost Model.

1 Introduction

When compared parallel computing with that of sequential computing, the current situation is different. No single model of parallel computation has yet come to dominate developments in parallel computing in the way that von Neumann model has dominated sequential computing. The aims of parallel computing model are: to describe classes of architectures in simple and realistic terms and, to propose the design methodology of parallel algorithm. It provides an abstract view of both the technologies and applications. An abstract model defines such as the algorithms are designed and analyzed in the abstract model and coded in programming model.

The design of parallel programs requires fancy solutions that are not present in sequential programming. Thus, a designer of parallel applications is concerned with the problem of ensuring the correct behavior of all the processes that the program comprises.

Generally, a programming model describes a programmer's view of the parallel systems. It has a different meaning in other areas of computer science, but in particular, it often stands for complete semantics of a language. A good parallel programming model has to satisfy the next properties: Programmability, Reflectivity, Cost model, Architecture independence and flexibility[8].

In this paper, we present *OTMP* computation model, its abstract machine and, its parallel programming model, valid to distributed memory machines and shared memory machines.

Although, there are others similar models, HPF[6] and OpenMP[11], *OTMP* has several differences with their current versions.
The next section presents the theorical machine, the programming model: syntax and semantic, and the associated complexity model. Section three deals load balancing issues and mapping and scheduling policies used. Finally, the section four shows many examples and computational results.

2 OTMP Model

One of the most common solutions to introduce parallelism has been to extend a sequential language with a short set of parallel constructors. Such constructor are a version of the *for* construct, commonly denoted as *forall* construct. This model proposes another *forall* loop extension for the *C* language. These extensions are aimed both for homogeneous distributed memory and shared memory architectures.

This model has several differences with other model [6][11]. The most important are:

- The parallel programming model introduced has associated a complexity model that allows the analysis and prediction of the performance.
- 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. The recursive divide and conquer algorithms are the most paradigmatic example.
- It does not only integrates and extends the sequential programming model but also includes and expands the message passing programming model.

2.1 Abstract Machine

OTMP theorical computer is a machine composed of a number of infinite processors, each one with its own private memory and a network interface connecting them. 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, the *NAME* of the processor. This register is not available to the programmer. According to this definition, every real parallel machine is a OTMP machine.

2.2 Syntax and Semantic

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

The programmer with the parallel loops

```
forall(i= first; i<= last; (r[i],s[i]))
        compound_statement_i
```

states that the different iterations i of the loop can be performed independently in parallel. ($r[i]$,$s[i]$) is the memory area of the results of the i-th iteration, where $r[i]$ points to first positions and $s[i]$ is the size in bytes.

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:

$$\text{Number of iterations to do: } M = last - first + 1 \quad (1)$$

$$\text{Iteration to do: } i = first + NAME \,\%\, M \quad (2)$$

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

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

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

$$\text{New value of register: } NAME = NAME \,/\, M \quad (4)$$

These equations decide the mapping of processors to tasks, and how the exchange of results will take place. After the *for all* 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 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 last parameter in the *forall* has the purpose to inform 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 $(r[i], s[i])$. $r[i] + j$ is a pointer to the memory area containing the j-th result of the i-th thread.

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 *for all* loop. Let us denote the execution of the body of the i-th thread (*compound_statement_i*) by T_i. The semantic imposes two restrictions:

1. Given two different independent threads T_i and T_k and two different result items $r[i]$ and $r[k]$, it holds:

$$[r[i], s[i]] \bigcap [r[k], s[k]] = \emptyset \ \forall i, k$$

2. For any thread T_i and any result j, all the memory space defined by $[r[i] + j, s[i]]$ has to be allocated previously to the execution of the thread body. This makes impossible the use of non-contiguous dynamic memory structures.

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 1.

```
1  forall(i=1; i<=3; (ri[i], si[i]))
2  { ...
3    forall(j=0; j<=i; (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 }
```

Figure 1: Two nested foralls

Initially, the infinite processors are in the same group, represented by the root of the tree in

figure 2. 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 1 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 its memory area, (ri[1],si[1]) to processors 1 and 2. Furthermore, it receives from processor 1 (ri[2],si[2]) and from processor 2 (ri[3], si[3]). The same exchange is repeated among the other corresponding triplets $((3,4,5),(6,7,8),\ldots)$. Every 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 *for all* produces "the face" of a ternary hypercubic dimension, where every corner has two neighbors. The second nested *forall* at line 3 requests for different number

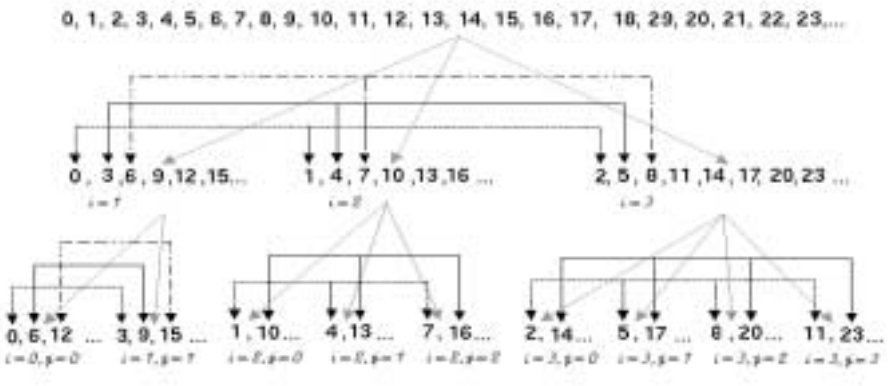


Figure 2: The mapping associated with the two nested *for all* in figure 1

of threads in the different groups. For examples, to the last group ($i = 3$) executes a *for all* 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 its data, rj[1], to processors 14, 20 and 23 and will receive from them their data rj[0], rj[2] and rj[3].

2.3 Others Clauses

The *OTMP* model admits several coherent extensions, in this section we present two: reduction clauses and global communication.

The reduction clauses have syntax and semantic similar to the above *forall*, its simplified syntax is

```
forall_R(i= first; i<= last; (r[i],s[i]); f_r[i]; (rr[i],sr[i]))
{
    compound_statement_i;
    f_r;
}
```

The *for all_R* works of the same way that *forall*, divides the processors, executes in parallel *compound_statement_i* over owns data and exchanges the results, r. When all is done, each processor reduces the results through *f_r*. The result of *f_r* is rr and it has size sr. The *f_r* can be any function, but it is mandatory that it has to be commutative and associative.

Other OTMP clauses are the global communication clauses: *result* and *result_P*. Their function is to communicate partial results among every processors belong distinct groups. The

difference between *result* and *result_P* is, the first only communicates the same data to every processors in the other groups, and *result_P* communicates data to every processors in the other groups, but the data are different and depend of the receptor processor's group. The corresponding syntax is

```
result(i,data[i],s[i])
result_P(i,data_to[i],s_to[i],data_from[i], s_from[i])
```

where i identifies the task, T_i. In the first clauses, $data[i]$ points to the data of T_i and $s[i]$ is its size. In the second clauses, $data\_to[i]$ is a pointer to the memory area containing the data to send to T_i, and $s\_to[i]$ contains its size. $data\_from[i]$ and $s\_from[i]$ have the same function, but to received data of T_i. These clauses are similar to functions *gather* and *scatter* in standards library such as MPI [10], but one of differences is the neighborhood relation, it takes place every that the exchanging of data is necessary.

2.4 Cost Model

In this section, we analyze the cost of the *forall* loop. A similar analysis is required to other clauses.

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 *for all* loop is given by the recursive formula:

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

where T_i is the code of *compound_statement_i*, M is the size of the loop and N_i is the size of the message transferred in iteration i,

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

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. Assuming the rather common case in which the volume of communication of each iteration is roughly the same

$$N \approx N_i \approx N_j \quad \forall i \neq j = e_1, e_2$$

From formula 5 and from the fact that in current machines the scheduling and mapping time $A_0 + A_1 \times M$ is dominated by the communication time $\sum_{i=e_1}^{e_2}(g \times N_i + L)$, it follows the result that establishes when an independent *for* loop is worth to convert in a *forall*:

$$\mathcal{T}(forall) \leq \mathcal{T}(for) \Leftrightarrow M \times (g \times N) + L + \max_{i=e_1}^{e_2}\mathcal{T}(T_i) \ll \sum_{i=e_1}^{e_2}\mathcal{T}(T_i) \tag{6}$$

A remarkable fact is that the $OTMP$ machine not only generalizes the sequential RAM but also the Message Passing Programming Model. Lines 6-8 in figure 1 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: an ysend 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 inform it.

3 Mapping and Scheduling

Unfortunately, an infinite machine, such as that described in the previous section, is only an idealization. Real machines have a restricted number of processors. Each processor has an internal register *NUMPROCESSORS* where stores the number of available processors in its current group. Three different situations have to be considered in the execution of a *forall* construct with $M = e_2 - e_1 + 1$ threads:

- *NUMPROCESSORS* is equal to 1;
- M is larger or equal than *NUMPROCESSORS*;
- *NUMPROCESSORS* is larger than M;

The first case is trivial. There is only one processor that executes all the threads sequentially, and there is not opportunity to exploit any intrinsic parallelism.

The second case has been extensively studied as flat parallelism. The main problem that arises is the load balancing problem. To deal with it, several scheduling policies have been proposed [11]. Many assignment policies are also possible: *block*, *cyclic-block*, *guided* or *dynamic*.

The third case was studied in the previous section. But the fact that the number of available processors is larger than the number of threads introduces several additional problems: the first is load balancing. The second is that, not anymore, the groups are divided in subgroups of the same size.

If a measure of the work w_i per thread T_i is available, the processors distribution policy viewed in the previous section can be modified to guarantee an optimal mapping [3]. The syntax of the *forall* is revisited to include this feature:

```
forall(i= first; i<= last; w[i]; (r[i], s[i]))
    compound_statement_i
```

If there are not weights specification, the same workload is assumed for every task. The semantic is similar to that proposed in [2]. Therefore, the mapping is computed according to a policy similar to that sketched in [3]. The figure 3 shows the applied mapping algorithm. There is, however, the additional problem of establishing the neighborhood relation. This time the simple $one - to - (M-1)$ hypercubic relation of the former section does not hold. Instead, the hypercubic shape is distorted to a polytope holding the property that each processor in every group has one and only one incoming neighbor in any of the other groups.

```
1  for(i = 0; i< M; i++)
2     p_i = 1;
3  for( j = M+1 ; j< NUMPROCESSORS; j++)
4  {
5     Get p_k such as w_k/p_k ≥ max_i = 2,..., M w_i/p_i;
6     p_k= p_k + 1;
7  }
```

Figure 3: Optimal Mapping Algorithm

This algorithm satisfies with the postulate of optimal mapping.

4 Examples

Many examples have been chosen to illustrate the use of the *OTMP* model: Matrix Multiplication, Fast Fourier Transform, QuickSort and Parallel Sort by Regular Sampling. The results are presented for several machines and we use the current software system that consists of a C compiler and a run time library, built on top of MPI.

4.1 Matrix Multiplication

The 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 respectively dimensions $m \times q_i$ and $q_i \times m$. Therefore, the product $A^i \times B^i$ takes a number of operations *w[i]* proportional to $m^2 \times q_i$. Figure 4 shows the algorithm. Variables A, B and C are arrays of pointers to the matrices. The loop in line 1 deals with the different matrices, the loops in lines 5 and 7 traverse the rows and columns and finally, the innermost loop in line 8 produces the dot product of the current row and column. Although all the *for* loops are candidates to be converted to *forall* loops, we will focus on two cases: the parallelization of only the loop in line 5 (labeled FLAT in the results, figure 5) and the one shown in figure 4 where additionally, the loop at line 1 is also converted to a *forall* (label NESTED in figure 5). This example illustrates one of the common situation where you can take advantage

```
1   forall(i = 0; i < tasks; w[i]; (C[i], m * m))
2  {
3    q = ...;
4    Ci = C+i; Ai = A+i; Bi = B+i;
5     forall(h = 0; h < m; (Ci[h], m))
6    {
7      for(j = 0; j < m; j++)
8        for(r = &Ci[h][j], *r=0.0, k=0; k<q; k++)
9          *r += Ai[h][k] * Bi[k][j]
10   }
11 }
```

Figure 4: Exploiting 2 levels of parallelism

of nested parallelism: when neither the *inner* loop (lines 5-10) nor the external loop (line 1) have enough work to have a satisfactory speedup, but the combination of both does. We will denote by $SP_R(\mathcal{A})$ the speedup of an algorithm $\mathcal{A}$ with R processors and by $\mathcal{T}_P(\mathcal{A})$ the time spent executing algorithm $\mathcal{A}$ on P processors. Let us also simplify to the case when all the inner tasks take the same time, i.e. $q_i = m$. Under this assumptions, the previous statement can be rewritten more precisely:

Lemma 1

Let be

$tasks < P, \quad SP_P(inner) < SP_{P/tasks}(inner)$ *and* $\quad tasks \times (g \times m^2) + L \leq \mathcal{T}_{P/tasks}(inner)$

then

$$SP_P(FLAT) < SP_P(NESTED)$$

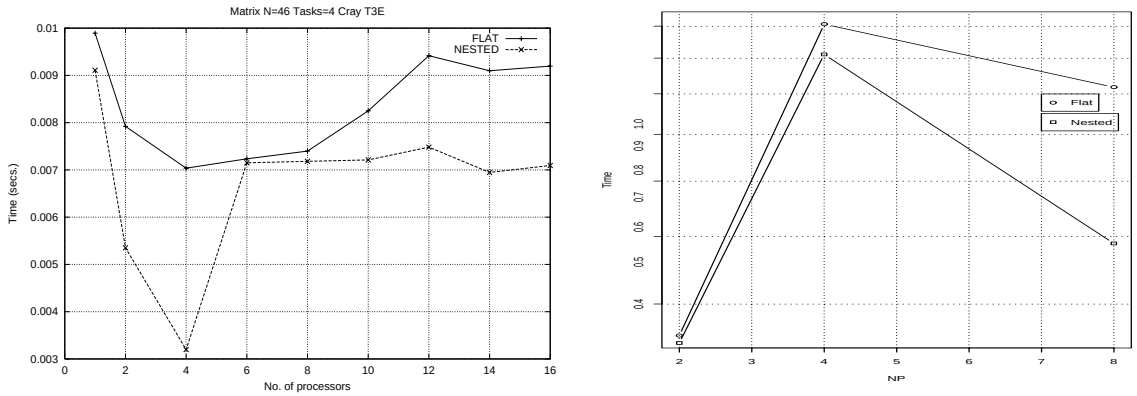
The figure 5 shows the expressed before for this example. Both matrix have dimensions 45×45. The number of task is 8, and to when the number of processors is 2 and 4, there are processor virtualization. In all case and to different architectures, the nested parallelism works well.

Moreover, when the size of the problems to solve is large, the speedup grows to be quasi-lineal. We check that the speedup reached is linear as it shows figure 6. 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.

4.2 Fast FourierTransform

The Discrete Fourier transform is widely used in solving problems in science and engineering. It is defined as

$$A(j) = \frac{1}{N} \sum_{k=0}^{N-1} a(k) e^{-2\pi i k j / N} \tag{7}$$



(a) Cra yT3E (b) Origin 2000

Figure 5: Nested versus Flat parallelism

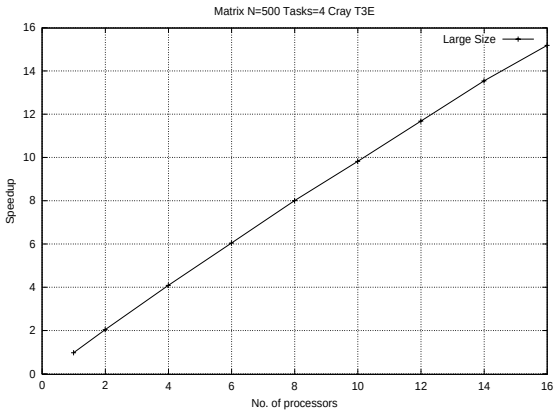


Figure 6: Speedup reached for large size problems

The *OTMP* code in figure 7 implements the Fast Fourier Transform (FFT) algorithm developed by Tukey and Cooley [4]. P arameter$W_k = e^{-2\pi ik/N}$ contains the pow ers of the$N$-th root of the unity.

The figure 8 shows the computational results for such implementation on a SGI Origin 2000 and CRAY T3E . The speedup curve behav es according to what a complexity analysis following formula 5 predicts. Moreover, when the size of problem is small, the communications influence on the performance.

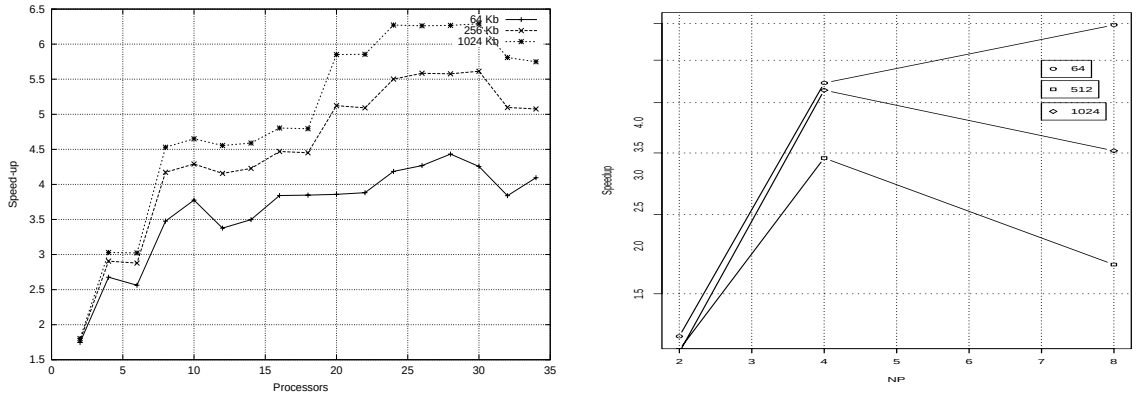
4.3 QuickSort

The w ell-knowvn quicksort algorithm [5] is a divide-and-conquer sorting method. As suc h, it is amenable to a nested parallel implementation. This example is specially in teresting, since the size of the generated sub-vectors are used as w eights for the *forall*, see figure 9, line 14. Remember that, depending on the goodness of the pivot c hosen, the new subproblems may hav e rather different weights.

Figure 10 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 1MB integers.

```
1  void llcFFT(Complex *A,Complex *a,Complex *W,
       unsigned N,unsigned stride,Complex *D){
2    Complex *B, *C, Aux, *pW;
3    unsigned i, n;
4
5    if(N == 1) {
6      A[0].re = a[0].re;
7      A[0].im = a[0].im;
8    }
9    else {
10     n = (N >> 1);
11     forall(i = 0; i <= 1;(D+i*n , n))
12    {
13     llcFFT(D+i*n,a+i*stride,W,n,stride<<1,A+i*n);
14    }
15     B = D;            /* Combination phase */
16     C = D + n;
17     for(i = 0,pW = W; i<n; i++,pW += stride) {
18       Aux.re = pW->re*C[i].re - pW->im*C[i].im;
19       Aux.im = pW->re*C[i].im + pW->im*C[i].re;
20       A[i].re = B[i].re + Aux.re;
21       A[i].im = B[i].im + Aux.im;
22       A[i+n].re = B[i].re - Aux.re;
23       A[i+n].im = B[i].im - Aux.im;
22     }
25   }
26 }
```

Figure 7: The *OTMP* implementation of the FFT



(a) Cra yT3E (b) Origin 2000

Figure 8: FFT. Different sizes.

4.4 Parallel Sorting b yRegular Sampling

Parallel Sorting by Regular Sampling($PSRS$) is an parallel sorting algorithm proposed by Li et all[9]. It is an example of simple and synchronous algorithm, and has been shown effective for a wide variet y of MIMD architecture.

PSRS w orkswith p process and assumes that the input list has n unsorting elements. It arises in four synchronous phases:

- *Phase 1*
  Each of the p processors uses the sequential quicksort algorithm to sort a local $\lceil \frac{n}{p} \rceil$ elements. The n elements no wform p independent lists. Each processor selects, among its sorted

```
1  void qs(int *v, int first, int last) {
2    int w[2];        /* weight vector */
3    int pos_f[2],pos_l[2]; /* first and last of each partition*/
4    int pivot, i, j;
5
6    select(v, first, last, *pivot);
7    i=first; j=last;
8    partition(v, &i, &j, pivot);
9      /* setting each subproblems and weight vector*/
10   w[0]=(j-first+1);  w[1]=(last-i+1);
11   pos_f[0]=first; pos_f[1]=i;
12   pos_l[0]=j;     pos_l[1]=last;
13
14    forall(i=0; i<=1;  w[i]; (v+pos_f[i],w[i]*sizeof(int)))
15      qs(v, pos_[i],pos_l[i]);
16 }
```

Figure 9: The *OTMP* implementation main of QuickSort

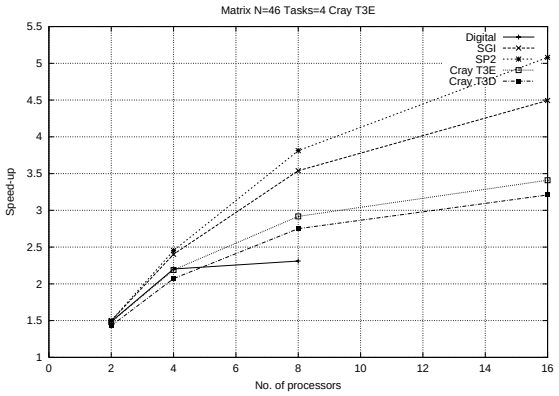


Figure 10: QuickSort. Size of the vector: 1M

data, a sorted list as regular sample.

- *Phase 2*
  One processor gathers and sorts every local samples. Finally, it selects $p-1$ pivots value from list of regular samples. At this point, each processors knows the final pivots list and partitions its sorted data into p disjoint pieces, the pivot values are the separators between the pieces.

- *Phase 3*
  Each processor i keeps its ith partition and assigns the jth partition to processor j. Each processor keeps one partition and reassigns $p-1$ other partitions to other processors.

- *Phase 4*
  Each processor merges its p partition into a single list. The concatenation of every processor's lists obtains the final sorted list with n elements.

OTMP algorithm to *PSRS* is shown in figure 11. To solve the problem, it uses the global communication constructors, line 14 and 23. This algorithm is different to original propose, every processes have every samples and don't need extra communications to know the final pivots.

The quicksort in line 17 can be replaced by a merge of $NUMPROCESSORS$ sorted list of samples.

```
1  void PSRS(int i, int *v, int size, int *V, int *size_V){
2   int j, k;
3   int *Samples, sample_st;   /*Samples vector and stride in v to obtain samples*/
4   int *Pivots, pivot_st;     /*Pivots vector and stride in Samples to obtain pivots*/
5   int *to_task;              /*First list to each process*/
6   int *res_o_task, *from_size; /*List of received data from other process and its size*/
7
8   int v_Local; /*Data list of local process*/
9
10  /* ================== Phase 1 =================*/
11  quicksort(v, size);
12  for ((k=0,j=i*NTASK); k<NTASK; (k+=sample_st,j++))
13      Samples[j]= v[k];
14   result(i,Samples+(i*NTASK), NTASK*sizeof(int));
15
16  /* ================== Phase 2 ================*/
17  quicksort(Samples, NTASK*NTASK);
18  for(k=0,j=1;j<=(NTASK-1);k++,j++)
19      Pivots[k]=Samples[(j*NTASK + pivot_st)-1];
20  divide_list(v, Pivots, to_task);
21
22  /* ================== Phase 3 ================*/
23   result_P(i,v,to_task,res_o_task,from_size);
24
25  /* ================== Phase 4 ===============*/
26  V= merge_list(v_Local, res_o_task, from_size, size_V);
27 }
```

Figure 11: The *OTMP* implementation of the PSRS

The figure 12 sho wsthe **main** function. The problem is solved b y $NUMPROCESSORS$ tasks. Each task w orksin parallel o v er $\frac{SIZE}{NUMPROCESSORS}$ data. V is the output v ector, it is sorted. After the $forall$ not is necessary an ycommunications, ev ery processes ha v ethe total sorted vector.

```
1    void main(argc, argv){
2      int i, size, new_size;
3      int *v, *V;
4
5      size= SIZE/NUMPROCESSORS;
6      v = calloc(size, sizeof(int));
7      V = calloc(SIZE, sizeof(int));
8
9      initialize(v, size);
10     forall(i=0; i<=NUMPROCESSORS-1; (V[i],new_size[i]))
11         PSRS(i,v,size[i],(V[i]),&(new_size[i]))
12   }
```

Figure 12: The *OTMP* main function of PSRS

The *OTMP PSRS* is an algorithm easy to understand and follow. The computational results are not reported because we are taking and analyzing their.

5 Conclusions and Future Work

OTMP is a computation model offers: simplicity, easiness of programming, improv ement of the performance (does not in troduce an y o v erheadfor the sequential case), portability, and a cost model associated. In addition to these characteristics, the model extend not only the sequential but the MPI programming model.

OTMP has several differences with most current versions of HPF and OpenMP. One is that

the parallel programming model introduced has a complexity model that allows the analysis and prediction of performance. The other is that it allows 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.

We have a prototype for the model. Many issues can be optimized. Even incorporating these improvements, the gains for distributed memory machines will never be equivalent to what can be obtained using raw MPI. Results obtained prove the feasibility of exploiting nested parallelism with the model. However, the combination of every its properties makes worth the research and development of tools oriented to this model.

Acknowledgments
We wish to thank the Universidad Nacional de San Luis and the ANPCYT from which we receive continuous support. Also to the european centers: EPCC, CIEMAT, CEPBA and CESCA.

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

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

[7] Keller, J. Kesler, C. Larsson, J.. Practical PRAM programming. John Wiley & Sons inc. (2001)

[8] Leopold, C. Paralleland Distributed Computing: A survey of models, paradigms, and approaches. John Wiley & Sons inc. (2001)

[9] Li, X. Lu, P. Schaeffer, J. Shillington, J. Wong, P. Shi, H.. On the Versatility of Parallel Sorting by Regular Sampling. Tech. Report TR 91-06. University of Alberta, Edmonton, Alberta, Canada. (1992)

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

[11] OpenMP Architecture Review Board: OpenMP Specifications: FORTRAN 2.0. http://www.openmp.org/specs/
mp-documents/fspec20.ps (2000).