Generic Communication in Parallel Computation

F.Piccoli M.Printista
C.González

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

Abstract

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.

There are different solutions to each problem, but the question is to find one, that is general. One possibility is allowing the use of asynchronous groups of processors. We present a general methodology to derive efficient parallel *divide and conquer* algorithms. Algorithms belonging to this class allow the arbitrary division of the processor subsets, easing the opportunities of the underlying software to divide the network in independent sub networks, minimizing the impact of the traffic in the rest of the network in the predicted cost. This methodology is defined by *OTMP* model and its expressiveness is exemplified through three divide and conquer programs.

Keywords: Division Function, Dynamic Polytope, Hypercubic Communication, Programming 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 as the algorithms are designed and analyzed in the abstract model, and coded in programming model [7].

The Bulk Synchronous Parallel (BSP) model is a generalization of the widely researched PRAM model that was initially proposed by Valiant [9] [10] [15]. The initial formulation of BSP considered the possibility of machine decomposition. Careful attention was paid to this feature, and the BSP library [4] standard document mention this, but in the absence of any clear solution it was decided to exclude it from the standard. In this paper, we present a division function of processors(machine decomposition): *polytope*, and a parallel programming model: *One Thread Multiple Processor Model* ($OTMP$) that implements it. In the $OTMP$ model [13], processor sets is automatically divided through sentences belong programming models what we call *parallel clauses*. Furthermore of computation and remote memory accesses, processors can perform group operations implying all the processors in the set.

The remainder of the paper is organized as follows: section 2 introduces the needed proprieties that every division functions have to hold. The third section introduces a reduced and idealistic version of the model, considering the simplified case where the number of available processor is larger than the number of processors required by the algorithm. Additionally, the section introduces load balancing issues, mapping and scheduling policies, and the laws that drive the time-cost of program execution under the model. Finally, the last section shows several $OTMP$ algorithms and some important results.

2 Strategies for the Implementation of Division Functions

Although the need of division functions appears in a wide class of algorithms, there is no doubt Divide and Conquer algorithms [1] constitute a motivation for the introduction and formalization of division functions [5]. The $OTMP$ Computing Model makes easier the implementation of Divide and Conquer algorithms. The divide and conquer approach presented in Figure 1 to find the solution of a problem x proceeds by dividing x in subproblems x_0 and x_1 (function divide in line 6) and applying recursively the same resolution scheme. This recursive procedure ends when the subproblems are small enough, in which case, another procedure (conquer) is used to solve the problem.

```
1  procedure DC(x: Problem; r: Result);
2  begin
3    if trivial(x) then conquer(x, r)
4    else
5    begin
6      divide(x, x0, x1);
7      DC(x0,r0);
8      DC(x1,r1);
9      combine(r, r0, r1);
10   end;
11 end;
```

Figure 1: General frame for a D&C algorithm

These typical problems are good to solve in parallel. The calls in lines 7 and 8 can be done in parallel, the figure 2 shows how it is made.

```
1  procedure pDC(x: Problem; r: Result);
2  begin
3    if trivial(x) then conquer(x, r)
4    else
5    begin
6      divide(x, x0, x1);
7      PARALLEL(pDC(x0,r0), pDC(x1,r1));
8      combine(r, r0, r1);
9    end;
10 end;
```

Figure 2: General frame for a parallel D&C algorithm

The call to function *PARALLEL* in line 7 produces the parallel activation of two tasks (pDC) to solve each of the two subproblems in which the original problem has been divided. *PARALLEL* divides the actual set of leaf processors in

two subsets. Each of the subsets solve in parallel a subproblem x_i, and at the end of the division process, all the processors in the original set achieve the solution r combining the partial solutions r_0 and r_1. This procedure is applied recursively until there is only one processor in the leaf set. In this case, two sequential calls to pDC are made. Functions *trivial()*, *conquer()*, *divide()* and *combine()* in Figures 1 and 2 must be free of side-effects.

The code in Figure 2 is an example of the wide class of problems where the implementation of functions or sentences to divide the processors can be made efficiently using only local information to the set of processors executing the function.

2.1 A Division Function Scheme

The underlying idea in our proposal for the implementation of division functions is the establishment of a relation among processors in the different sets produced by the division. Each processor q in a processor set Q_i produced by the division settles a partnership relation with one or more processors in the other subsets. This partnership relation determines the communication of the results produced by the parallel task T_i performed by processor set Q_i. The structure of divisions produced by the division functions and the partnership relation among processors give place to communication patterns among processors that are topologically similar to a hypercube. The number of divisions produced determines the dimension while the degree in each dimension is the number of parallel tasks (i.e. the number of subsets) created by the division function. Similarly with what occurs in a conventional k-ary hypercube a dimension divides the set in k subsets communicated through the dimension. Nevertheless, opposite subsets in a dimension may have not the same cardinality. We have named the resulting structures Dynamic Polytopes. The following paragraph, formally introduces this concept in order to settle the conditions that guarantee the correctness of the translation of the division functions.

2.2 Dynamic Polytopes

Let be $\Gamma = Q_0, .., Q_{m-1}$ a partition of a set Q. We will name complementary sets of Q_i to the sets Q_j with $j \neq i$. Let be $P(A)$ the set of all subsets of set A.

For any $q \in Q_i$ we will name $G_j(q) \subseteq Q_j$ to the set of processors in Q_j to which processor q will send its results.

A partnership relation in Γ is any correspondence $G = (G_i)_{i \in \{0,...,m-1\}}$ where

$$G : Q \rightarrow \Pi_{i=0,...,m-1} P(Qi)$$
$$G_j : Q \rightarrow P(Q_j)$$

In a conventional binary hypercube, the neighbor or partner of a node in a fixed dimension is unique, while in a partnership relation G a node may have more than one partner in one dimension. This is the reason why functions G_j take their values in $P(Q_j)$ instead of Q_j.

We will say that the pair $(\Gamma, (G_i)_{i \in \{0,...,m-1\}})$ is a neighborhood if the following conditions are fulfilled:

- *Exhaustivity:* for any $i, j \in \{0, ..., m-1\}$, any element in Q_j has a partner in Q_i:

  $$\forall i, j \in \{0, ..., m-1\} : \bigcup_{q \in Q_i} G_j(q) = Q_j$$

  This condition guarantees that, in the algorithm to be presented in Figure 5, any processor q in Q_j receive the result of the execution of task T_i performed by the processors in Q_i.

- *Injectivity:* $\forall i, j \in \{0, ..., m-1\}, i \neq j$

  $$\forall q, q' \in Q_j, q \neq q' \; it \; holds \; G_i(q) \bigcap G_i(q') = \emptyset$$

  This condition imposes that each processor q in a set Q_i receives the results of task T_j only from one of the processors in Q_j.

If $q \in G_j(q')$ we say that q is a partner of q' in the neighborhood defined by $(\Gamma, (G_i)_{i \in \{0,...,m-1\}})$.

A tree or hierarchy of neighborhoods H constitute a *Dynamic Polytope* iff it holds that:

1. The root of the tree is the *trivial neighborhood* (Γ, G) where $\Gamma = Q$ and G is the identity function.

2. If node T is labeled with the neighborhood $(\Gamma^d, (G_i^d) i \in \{0, ..., r-1\})$ such that $\Gamma^d = \{Q_0^d, .., Q_{r-1}^d\}$ is a partition of Q^d, then the children nodes of T are labeled with neighborhoods that partition the sets Q_i^d of Γ^d. Eventually, some of the sets Q_i^d of Γ^d may remain without division (in which case they are leaves in the tree).

We will name dimension of H to the depth of the neighborhood tree. The nodes at the same level d of the hierarchy H constitute what we will call a dimension of the *Dynamic Polytope*. A dimension is generically designated by the value of its level minus one, and therefore, the first trivial dimension (Γ, G) is dimension (-1).

If k is a natural number, a k-ary d-dimensional hypercube is a graph with $p = k^d$ nodes. Dimension 0 defines a partition $\Gamma^0 = \{Q_0^0, ..., Q_{k-1}^0\}$ of the set $Q = \{0, 1, ...k^d - 1\}$ in k subsets Q_s^0. Nodes whose least significant k-ary digit is s belong to Q_s^0. In general, each dimension (in the classical sense of the term) $i \in \{0..d-1\}$ determines a partition $\Gamma^i = \{Q_0^i, ..., Q_{k-1}^i\}$, where

$$Q_s^i = \{n \in Q / \ i^{th} \ digit \ of \ n \ is \ s\} \ with \ s \in \{0..k-1\} \quad (1)$$

Node q is connected in dimension $i \in \{0..d-1\}$ to nodes whose k-ary representation differs from q in the i^{th} k-ary digit. For any $n \in Q$ let be $n_{d-1}...n_o$ the k-ary representation of n, then:

$$G^i : Q \to \Pi s = 0, .., k-1 Q_s^i$$
$$G_s^i : Q \to Q_s^i$$
$$G_s^i(n_{d-1}...n_0) = n_{d-1}...n_{i+1} s n_{i-1}...n_0$$

Observe that in any dimension i of a k-ary hypercube the degree is $\mid \Gamma^i \mid = k$.

It is easy to see that $(\Gamma^i, (G_s^i)_{s \in \{0,...,k-1\}})$ obey the conditions that characterize the neighborhood concept in any dimension i.

Figure 3(a) shows a 2-dimensional ternary hypercube. This hierarchy of neighborhoods could be created by two nested calls to a division function. In this example, the first call has created three parallel tasks, and the second recursive call has divided again each subset in three new subsets. Each node in the tree represents a neighborhood, while processor subsets assigned to each task are represented by the shadowed nodes. The lines represent the partnership relations. Solid lines correspond to dimension zero while dotted lines correspond to dimension one. Observe that in this example, each node has two neighbors in each dimension. Each node has two solid and two dotted edges. Notice that the definition of Dynamic Polytope Dimension we have provided matches with the classical concept of dimension in a k-ary hypercube.

Figure 3(b) represents a hierarchy of neighborhoods, $(\Gamma^d, (G_s^d))$ defined by the partitions Γ^d (different regions) and the partnership relations (G_s^d) (edges connecting processor nodes) for a 3-dimensional Dynamic Polytope.

Solid lines correspond to the first dimension (first level in the hierarchy), fine dotted lines to the second dimension and coarse dotted lines to the third dimension. In the example of the figure, the nodes are always divided in two subsets, although not necessarily of the same size. This polytope would be produced by a binary division function if the amount of processors assigned to each task varies.

Precisely speaking, accepting $\mid Q_o^i \mid \geq \mid Q_1^i \mid$ and that $\mid Q_o^i \mid$ is multiple of $\mid Q_1^i \mid$, partnership functions G_i are given by the following expressions:

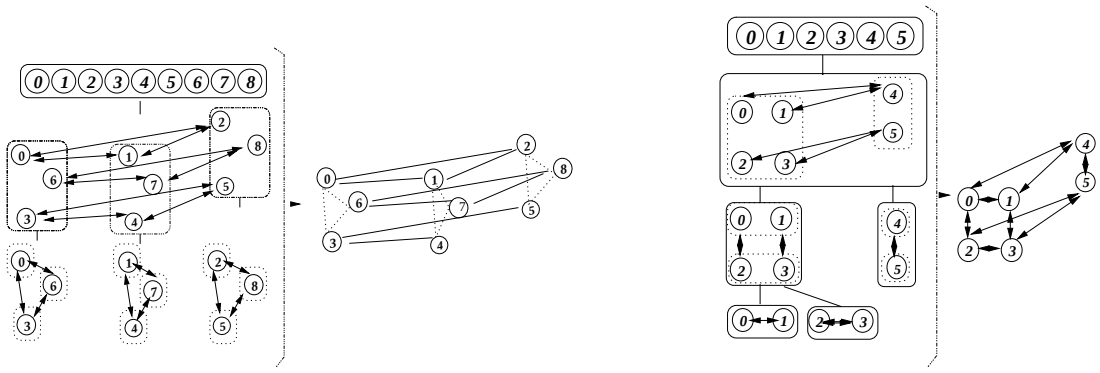
$$G_1^i(n) = \{n - first(Q_0^i)) / \mid Q_1^i \mid + first(Q_1^i) \ for \ n \in Q_0^i\}$$
$$G_0^i(n) = \{j / such \ that \ (j - first(Q_0^i)), \mid Q_1^i \mid + first(Q_1^i) = n\} \quad (2)$$

and analogously for the other case $\mid Q_1^i \mid \geq \mid Q_0^i \mid$.

As we can observe in Figure 3(a) for the case of a regular Hypercube, fixing a dimension, there is only one partner for each node in any complementary set. This is not true for the general case of a Dynamic Polytope, Figure 3(b). For example, for the first dimension, both nodes 4 and 5 have two partners (partners(4)=0, 1; partners(5)=2, 3) in the complementary set.

2.3 Translation Scheme

Lets go back to the general frame of a divide and conquer introduced in Figure 2. In general, each time a processor set is divided by the execution of a division function, a new dimension (Γ^{dim}, G^{dim}) is created. The creation of the neighborhood can be accomplished in time proportional to the number of tasks demanded (degree) using, for example, the policy described for



(a) Ternary Hypercube (b) Dynamic Polytope

Figure 3: Types of Hypercubes

a k-ary hypercube (equation 1). With such policy, the initial set of processors, Q is divided in two subsets $\Gamma^{dim} = \{Q_0, Q_1\}$ of equal size using constant time. Processors in Q_0 perform the task $pDC(x_0, r_0)$ and those in Q_1 execute $pDC(x_1, r_1)$. However, the optimal partition minimizing load unbalance is the one satisfying:

$$\frac{\mid Q_0 \mid}{n_0} = \frac{\mid Q_1 \mid}{n_1}$$

where n_i is the number of leaves in the tree of calls produced by the call to $pDC(x_i, r_i)$. We propose the use of division functions having as additional parameters weights, w_i, either provided by the user or by an heuristic function that estimate the quotient $\frac{|Q_i|}{n_i}$.

Each of the processors q in Q_0 will hold in variable r_0 the result of $pDC(x_0, r_0)$. At the end of the execution of the parallel tasks, each processor $q \in Q_0$ sends the result r_0 to its partner processors in $G_1(q)$. Symmetrically, each processor $q' \in Q_1$ sends r_1 to its partners in $G_0(q')$.

Since the injectivity conditions of the dynamic polytope holds, only one message is received by each processor. On the other side, the exhaustivity condition guarantees that at the end all processors in Q_i get a copy of the result $rl - i$.

The cost of the code in lines 10-26 is dominated by the communication time:

$$D * max\{\mid r_0 \mid *max\ q(\mid G_1(q) \mid), \mid r_1 \mid *max\ q(\mid G_0(q) \mid\}$$

where $\mid r_i \mid$ represents the size of the results, $\mid G_i(q) \mid$ denotes the cardinal of $G_i(q)$ and D is a constant. The partnership relation G that maximizes the communication balance would be any that minimizes the number of partners each processor has. A solution is the policy exposed in equation 2 also used by the division functions of $OTMP$ model.

The time invested in a call to $pDC(x, r)$ obeys the recursive expression:

$$\Phi(pDC(x,r),N) = W(divide(x,x_0,x_1)) + W(parallel) + \\ + max\{\Phi(pDC(x_0,r_0),N_0), \Phi(pDC(x_1,r_1),N_1)\} + L + \\ + g * max\{\mid r_0 \mid *max\ q(\mid G_1(q) \mid), \mid r_1 \mid *max\ q(\mid G_0(q) \mid))\}$$

where $N_i = \mid Q_i \mid$ and $N = \mid Q \mid$.

3 *OTMP* Model

The *OTMP* (*One Thread Multiple Processors*) model, enables to express parallelism over any machine, sequential or parallel computer, adding only parallel clauses to sequential code. The *OTMP* model being introduced, extends the classic sequential imperative paradigm with new constructs: parallel loops and global communication. In contrast with *OpenMP* [12], the model works fine 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 the next code. 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 memory area ($r[i]$,$s[i]$), where $r[i]$ points to first positions and $s[i]$ is the size in bytes.

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

How does *forall* work? 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. The processors are organized in *sets* or *groups*. At any time, the memory state of every processors in the same group is identical. An *OTMP* computation assumes that all 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, *NAME*, containing the name or number of the processor in the group.

When any computation begins, every infinite processors in the machine belong to same group, they execute the same thread and have identical values stored in their local memories. When all processors in the set reaching the former *forall* loop, the set is divided in subsets and each processor decides in terms of its *NAME* to which subset it will belong. In how many subgroups will be divide the original group depends of the particular parallel clause. Each independent thread $compound\_statement_i$ is executed by a subgroup. When the execution of a parallel clause finishes, every processors in every subsets are joined in the original set.

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 the processors in the father group have a consistent view of the memory, after returning to the previous group, it is necessary the exchanging of the variables that were modified inside the *forall* loop among neighbors. 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.

Summaring, to have the same memory at the end of the parallel clauses, every local memories of subsets have to be communicated. The communication among processors is sorted, 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 memory area to be modified.

In *OTMP*, the fundamental clause is the parallel iteration *forall*. This parallel iteration is a general loop, the number of independent task is determined by the number of iteration needed when the loop is sequential.

A *forall* structures the current group according as a *M-ary* hypercube, where M is the number of iterations in the parallel loop, $M = last - first + 1$, and each processors is mapping to subgroup i subgroup, $i = first + NAME \ \% \ M$. Then, a *forall* produces "the face" of a *M-ary* hypercubic dimension, where every corner has a neighbor. The neighborhood relationship is given by the formula

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

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

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

The OTMP's clauses can be nested and, in consequence, generate multiple level parallelism. Multiple level parallelism enables the generation of work from different simultaneously executing threads. The figure 4 shows two level of parallelism, the outer *forall* divides the processors group in three subgroups. The second nested *forall* at line 3 requests for different number of threads in the different groups. This nested *forall* structures the current subgroup according as a *i+1-ary* hypercube,

```
1 forall(i=1; i<=3; (ri[i], si[i]))
2 { ...
3   forall(j=0; j<=i; (rj[j], sj[j])){
4     f(j);
5   }
6   ...
7 }
```

Figure 4: Two nested foralls

The *OTMP forall* can be extended. An coherent extension is the reduction clause. This clause has syntax and semantic similar to the above $forall\_R$, but its difference is: when all is done, each processor reduces the results through f_r. The result of f_r has to belong to the memory that is going to be modified in the clause. 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 belonging to distinct groups. These clauses are similar to functions *gather* and *scatter* in standards library such as MPI [11], but they differing the established communications, in the neighborhood relationship. Every clauses apply dynamic polytopic communication.

3.1 Load Balancing

Unfortunately, the real scenario is different that the theoretical scenario. In the last one, there are always processors available to resolve some task in parallel, the theoretical machine has infinity processors. The situation is so different when the parallel machine is a real machine, when a set of processors reaches a parallel clause, two situations are possibles: the number of processors, *NUMPROCESSORS*, is larger or is smaller than the number of tasks. If there are more task than available processors, the set of processors is divided in as subset such as processors exist (each subset has only one processor) and each subset will compute several tasks. This case has been extensively studied as flat parallelism. The main problem that arises is the load balancing problem.

Other situation, there are more processors than tasks to make was detailed in previous section. When the number of available processors is larger than the number of threads or tasks, it 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 established in the previous section can be modified to guarantee an optimal mapping [3]. The syntax of the *forall* is revised to include this feature:

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

If there are not weight specifications, the same work load 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] [13]. There is, however, the additional problem of stablish 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.

4 Examples

Many examples have been chosen to illustrate the use of the *OTMP* model: Matrix Multiplication, QuickSort and Parallel Sort by Regular Sampling. Finally, we show some interesting results. The results are obtained over 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 ($C^i = A^i \times B^i \quad i = 0, \ldots tasks - 1$) [14]. 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 5 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 and the one shown in figure 5 where additionally, the loop at line 1 is also converted to a *forall*. This

```
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 5: Exploiting 2 levels of parallelism

example illustrates one of the common situation where you can take advantage 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.

4.2 QuickSort

The well-known quicksort algorithm (QS) [6] is a divide-and-conquer sorting method. As such, it is amenable to a nested parallel implementation. This example is specially interesting, since accept several solutions applying *OTMP* model and different parallelism paradigms. One of these is showed in figure 6. It shows how solve QS using the reduction clause: *forall_R*. The reduction function is *merge* of list. This function satisfies whit the necessary condition to reduction function, it is commutative and associative.

```
1  void qs(int *v, int first, int last) {
2    int i, j, S;
3    int *pos_f,*pos_l,*s, *R;
4
5    S= last-first+1;
6    s[0]=(S/NUMPROCESSORS)+(S%NUMPROCESSORS);
7    pos_f[0]=0;
8    for (i=1;i<NUMPROCESSORS;i++){
9        s[i] = (S/NUMPROCESSORS);
10       pos_f[i]=s[i-1]+pos_f[i-1];
11       pos_l[i-1]=pos_f[i]-1;
12   };
13   pos_l[NUMPROCESSORS-1]=Size-1;
14    forall_R(i=0; i<NUMPROCESSORS;
                    (v+pos_f[i]),s[i]);(R,S))){
15      QSseq(v,pos_f[i],pos_l[i]);
16      merge(R,S,v+pos_f[i],s[i]);
17  }
18  memcpy (v, R, S*sizeof(int));
19 }
```

Figure 6: QS_{Reduc} *OTMP*

In this particular solution, we don't use nested parallelism, the problems is divided in *NUMPROCESSORS* subtask.

4.3 Parallel Sorting by Regular Sampling

Parallel Sorting by Regular Sampling($PSRS$) is a parallel sorting algorithm proposed by Li et all [8]. It is an example of simple and synchronous algorithm, and has been shown effective for a wide variety of MIMD architecture.

PSRS works with p processes and assumes that the input list has n unsorting elements. It arises in four synchronous phases:

- *Phase 1*
  Each process applies the sequential quicksort algorithm over local elements and selects a sorted list of sample among these sorted data.

- *Phase 2*
  A process gathers, sorts every samples and, finally, selects $p-1$ pivots over samples and sends them to others processes. Each process divides its sorted data into p disjoint pieces. Each pivot values is the separators between the pieces.

- *Phase 3*
  Each process i keeps its i^{th} partition and assigns the j^{th} partition to processor j.

- *Phase 4*
  Each process merges its p partition into a single list.

OTMP algorithm to *PSRS* is shown in figure 7. To solve the problem, it uses the global communication constructors, line 14 and 23. This algorithm is different to the original propose, every processes have every samples and don't need extra communications to know the final pivots.

```
1  void PSRS(int i,int *v,int s,int *V,int *s_V){
2   int j, k;
3   int *Samples, sample_st;
4   int *Pivots, pivot_st;
5   int *to_task;
6   int *res_o_task, *from_s;
7
8   int v_Loc;      /*Data list of local process*/
9
10  /* ================== Phase 1 ==================*/
11  quicksort(v, s);
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  Samples= merge(Samples, 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_s);
24
25  /* ================== Phase 4 ===============*/
26  V= merge_list(v_Loc, res_o_task, from_s, s_V);
27 }
```

Figure 7: The *OTMP* implementation of the PSRS

The figure 8 shows the **main** function. The problem is solved by *NUMPROCESSORS* processes or tasks. Each task works in parallel over $\frac{SIZE}{NUMPROCESSORS}$ data. V is the output vector, it is sorted. After the $forall$ is not necessary any data combination, every processes have the total sorted vector.

The *OTMP PSRS* is an algorithm easy to understand and follow. This algorithm is an examples of *flat parallelism*, even if there are nesting of clauses, *forall* and the global communication *result* and *result_P*.

```
1 void main(argc, argv){
2   initialization phase;
3
4   forall(i=0; i<=NUMPROCESSORS-1; (V[i],new_size[i]))
5   PSRS(i,v,size[i],(V[i]),&(new_size[i]))
6 }
```

Figure 8: The *OTMP* main function of PSRS

4.4 Some Computation Results

The figure 9 shows the corresponding results of the algorithm showed in 5, the parallelization of only the loop in line 5 is labeled FLAT and the nested parallelization is label NESTED. 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 better.

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 10.

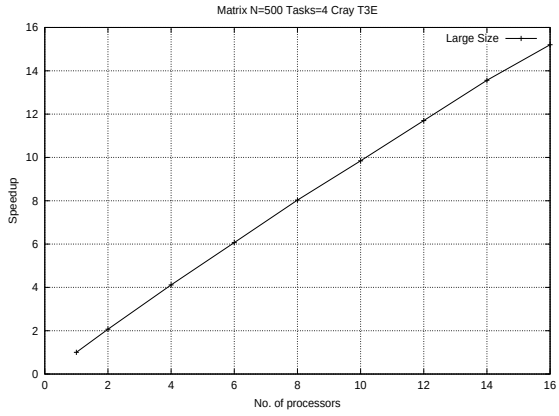
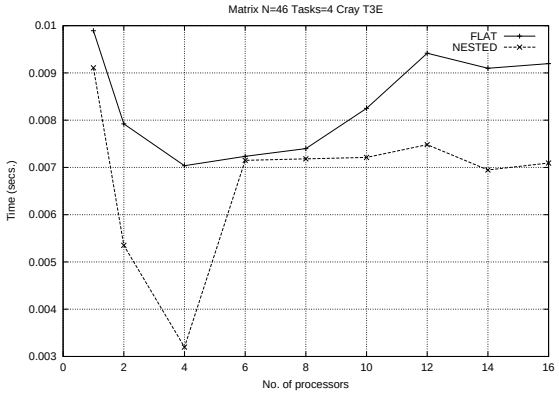


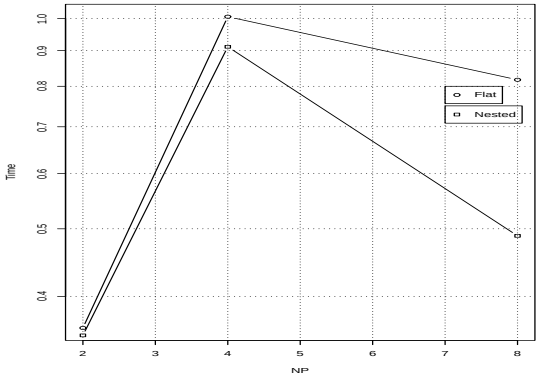
Figure 10: Speedup reached for large size problems

5 Conclusions and Future Work

In this work we introduce the *OTMP* model whose main elements are division functions. In this work we have focused our attention in the efficient implementation of this functions applied to a class of algorithms: divide and conquer algorithms. Every algorithms resolved by *OTMP* have principal characteristic, all processors involved in the computation have identical memory state, both the initial input data and the solution



(a) CrayT3E



(b) Origin 2000

Figure 9: Nested versus Flat parallelism

have to be stored in all the processors. This fact limits the maximum speedup achievable. Since the data have to be stored in all the processors, the size of the data constitute a lower bound of the time of any parallel algorithm.

In our proposal for the implementation of division functions, we have managed two main aspects: how many processors to attach to each of the parallel tasks created by the function, and the design of the partnership relation among the processors. The cardinality of the subsets created influences the workload of the algorithm, while the partnership relations act on the efficiency of the communications. The formalization of these factors led us to the concept of Dynamic Polytope.

We have stated the conditions for the partnership relations to ensure the correctness of the implementation of any division function. For the case divide and conquer algorithms, we have proposed partnership functions that guarantee not only the correct behavior of the algorithm but also its efficiency.

The hypercubic division function is implemented in every clauses of parallel iteration proposed by *OTMP* programming model. These clauses allow the arbitrary mapping of the processors in the division phase, easing the group work and minimizing the impact of the communication in the performance of application.

OTMP has several differences with most current versions of *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 to exploit 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.

The *OTMP* model has different characteristics: guarantees the portability to any platform, the implementation benefits strongly from the simplicity of the model, the quality of results is good in shared and distributed memory architecture, and does not only extend the sequential but the *MPI* programming model.

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 re-

ceive 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] Bonorden, O., Huppelshauser, N., Juurlink, B., Rieping, I.. *PUB library, Release 6.0 - User guide and function reference.* University of Paderbon, Germany. (1998)

[5] Gonzalez,J.A., Leon,C., Piccoli,M.F., Printista, M., Roda, J.L., Rodríguez, C., Sande, F.. *Groups in Bulk Synchronous Parallel Computing*. Euromicro Workshop on Parallel and distributed Processing. IEEE. Rodas, Grecia. Pp 224-251. (January 2000)

[6] Hoare, C. A. R.. *Quicksort.* Computer Journal. Vol 5 number 1. Pp 10-15. (1962)

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

[8] 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)

[9] McColl, W.F.. *General purpose parallel computing*. Gibbons and Spirakis. Pp: 337-391.(1993)

[10] Mccoll, W.F.. *BSP Programming*. DIMACS Series in Discrete Mathematics and Theorical Computer Science. (May 1994).

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

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

[13] Piccoli, F., Printista, M., González, J., Rodriguez Leon, C., Rodriguez, G., de Sande, F. *OTMP: A Parallel Programming Model*. International Conference on Computer Science, Software Enginering, Information Technology, e-Business, and Applications 2003(CSITeA'03). ISBN: 0-9742059-0-7. Rio de Janeiro, Brasil. Pp 218-223.(June 2003)

[14] Quinn, M.. *Parallel Computing. Theory and Practice*. Second Edition. McGraw-Hill Inc. (1994)

[15] Valiant, L.G.. *A Bridging Model for Parallel Computation*. Communications of the ACM. Vol. 33, number 8. Pp 103-111. (1990)