ABSTRACT

Though there have been substantial progress, the current status of parallel computing is still immature. No single model of parallel computation has yet come to dominate developments in parallel computing in the way that the von Neumann model has dominated sequential computing.

The design of parallel programs requires fancy solutions that are not present in sequential programming. There are specific parallel solutions for specific problems, but the ultimate goal is to find a general methodology at least for some classes of problems. We present a general methodology to derive efficient parallel divide and conquer algorithms for distributed memory machines using asynchronous groups of processors. The strategy allows the arbitrary division and coherent reunification of processor subsets, easing the opportunities of the underlying software to divide the network into independent sub networks. This way the impact of the sub network traffic in the rest of the network is minimized and consequently performance predictability is benefited. This methodology is formally defined introducing a computing model we have called One Thread Multiple Processor Model (OTMP). Its expressiveness is exemplified through two divide and conquer programs.

KEY WORDS

Division Function, Dynamic Polytope, Hypercubic Communication, Programming Model.

1 Introduction

The Bulk Synchronous Parallel (BSP) model is a generalization of the widely researched PRAM model that was initially proposed by Valiant [1][2]. The initial formulation of BSP considered the possibility of machine decomposition. Careful attention was paid to this feature, and the BSP library [3] standard document mention this, but in the absence of any clear solution it was decided to exclude it from the standard.

Although the need of division functions appears in a wide class of algorithms, there is no doubt Divide and Conquer algorithms [4] constitute a motivation for the introduction and formalization of group division functions [5].

The general scheme of a divide and conquer algorithm is presented in figure 1.

```
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 1. General frame for a parallel D&C algorithm

This code 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. 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 $r_1$.

In this paper, we present a scheme for the partitioning of processors (machine decomposition): polytope, and a parallel programming model: One Thread Multiple Processor Model (OTMP). In the OTMP model [6], processor sets are automatically divided through sentences called 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: the second section introduces a division function scheme. The third section presents 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 deals with load balancing issues and mapping and scheduling policies. Finally, the last section shows a OTMP algorithm and some important results.

2 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 in a processors subset 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 performed by processors subset. 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.

2.1 Dynamic Polytopes

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

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

A partnership relation in \( \Gamma \) is any correspondence

\[
G = (G_i)_{i \in \{0,\ldots,m-1\}} \quad \text{where}
\]

\[
G: Q \to \Pi_{i=0,\ldots,m-1} P(Q_i) \\
G_i: Q \to P(Q_i)
\]

where \( P(Q_j) \) the set of all subsets of set \( Q_j \).

If \( q \in G_j(q') \) we say that \( q \) is a partner of \( q' \) in the neighborhood defined by \( (\Gamma_i, (G_i))_{i \in \{0,\ldots,m-1\}} \). 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 \( (\Gamma, (G_i))_{i \in \{0,\ldots,m-1\}} \) is a neighborhood if any element in \( Q_j \) has a partner in \( Q_i \) (Exhaustivity), and if each processor \( q \) in a set \( Q_i \) receives the results of task \( T_j \) only from one of the processors in \( Q_i \) (Injectivity)[5].

A tree or hierarchy of neighborhoods \( H \) constitutes a Dynamic Polytope iff it holds that:

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

2. If node \( T \) is labeled with the neighborhood \( (\Gamma^{d}, (G_i^{d}))_{i \in \{0,\ldots,r-1\}} \) such that \( \Gamma^{d} = \{Q_0^{d}, \ldots, Q_{r-1}^{d}\} \) is a partition of \( Q^d \), then the children nodes of \( T \) are labeled with neighborhoods that partition the sets \( Q_i^{d} \) of \( \Gamma^{d} \). Eventually, some of the sets \( Q_i^{d} \) of \( \Gamma^{d} \) 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 \( (\Gamma, G) \) 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 \( \Gamma^0 = \{Q_0^0, \ldots, Q_{k-1}^0\} \) of the set \( Q = \{0, 1, \ldots, k^d - 1\} \) in \( k \) subsets \( Q^0 \). 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) \( i \in \{0, \ldots, d-1\} \) determines a partition \( \Gamma^i = \{Q_0^i, \ldots, Q_{k-1}^i\} \), where

\[
Q_s^i = \{n \in Q / \text{ith digit of } n \text{ is } s \} \quad \text{with } s \in \{0, \ldots, k-1\} \tag{1}
\]

Node \( q \) is connected in dimension \( i \in \{0, \ldots, d-1\} \) to nodes whose \( k \)-ary representation differs from \( q \) in the \( i \)-th \( k \)-ary digit. For any \( n \in Q \) let be \( {n_{d-1}, \ldots, n_0} \) the \( k \)-ary representation of \( n \), then:

\[
G^i: Q \to \Pi_{s=0,\ldots,k-1} P(Q_s^i) \\
G_s^i: Q \to Q_s^i \\
G_s^i(n_{d-1}, \ldots, n_0) = n_{d-1,\ldots,n_{i+1} = s, n_{i-1} = \ldots, n_0}
\]

Perceiving that in any dimension \( i \) of a \( k \)-ary hypercube the degree is \( |\Gamma^i| = k \).

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

![Figure 2. Dynamic Polytope](image_url)

Figure 2 represents a hierarchy of neighborhoods, \((\Gamma^d, (G_i^d))\) defined by the partitions \( \Gamma^d \) (different regions) and the partnership relations \( (G_i^d) \) (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.

When the hypercube is regular, 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 2. 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.

2.2 Translation Scheme

Let us go back to the general frame of a divide and conquer introduced in Figure 1. In general, each time a processor set is divided by the execution of a division function, a new dimension \((\Gamma_{\text{dim}}, G_{\text{dim}})\) 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 k-ary hypercube (equation 1). With such policy, the initial set of processors, \(Q\) is divided in two subsets \(\Gamma_{\text{dim}} = \{Q_0, Q_1\}\) 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:

\[
\frac{|Q_0|}{n_0} = \frac{|Q_1|}{n_1}
\]

where \(n_i\) is the number of leaves in the tree of calls produced by the call to \(pDC(x_1, r_1)\). 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 \(\frac{|Q_1|}{n_1}\).

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 \(Q_1\) get a copy of the result.

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 [7], 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.

```plaintext
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 processor 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 \(compound\_statement\) is executed by a subgroup. When the execution of a parallel clause finishes, every processors in every subsets are joined in the original set.

When forall loop 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 loop. Variables in the last set are available inside the loop for reading. The others are partitioned among the new groups. In other every moment, the memory of the group contains exactly the same values.

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 \((r[i], s[i])\). \(s[i] + j\) 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 (\(compound\_statement\omega\)) 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:
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.

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, $M = \text{last} - \text{first} + 1$, and each processors is mapping to subgroup $i$ subgroup, $i = \text{first} + NAME \% M$. 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:

$$\forall \text{forall} \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall \forall$$

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

\[
\text{neighbour}[j] = \Phi + (NAME + j) \% M
\]

where $\Phi$ 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 3 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 an $i+1$-ary hypercube.

1 forall(i=1; i<=3; (ri[i], si[i]))
2 { ...height=5cm,
3 \forall forall j=0; j<=i; (r[j][j], s[j][j]){
4 \forall forall i;
5 } ...
6 ...
7 }

Figure 3. Two nested foralls

3.1 Load Balancing

Unfortunately, the real scenario is different that the theoretical scenario. 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 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 and each subset will compute several tasks. This case has been extensively studied as flat parallelism.

The 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 [8]. The syntax of the forall is revised to include this feature:

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

If there are not weight specifications, the same work load is assumed for every task. Therefore, the mapping is computed according to a policy similar to that sketched in [6] [8]. There is, however, the additional problem of stabilize the neighborhood relation. This time the simple one – to – $(M - 1)$ hypercubic relation of the former section does not hold. The hypercubic shape is distorted to a polytope.

4 Examples

Two examples and some interesting results have been chosen to illustrate the use of the OTMP model: Matrix Multiplication and Large Integer Multiplication, but there are others. We use the current software system that consists of a C compiler and a run time library, built on top of MPI[9].

4.1 Matrix Multiplication

The problem to solve is to compute $tasks$ matrix multiplications $(C^i = A^i \times B^i ; i = 0, \ldots, tasks - 1)$ [10]. 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 and the one shown in figure 4 where additionally, the loop at line 1 is also converted to a forall. This 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 $SP_{P}(A)$ the speedup of an algorithm $A$ with $R$ processors and by $T_P(A)$ the time spent executing algorithm $A$ on $P$ processors.

The figure 5 shows the corresponding results of the algorithm showed in 4, the parallelization of only the loop

\[
[r[i], s[i]] \cap [r[k], s[k]] = \emptyset \forall i, k
\]
forall (i = 0; i < tasks; w[i]; (C[i], m * m))
{
    q = ...;
    Ci = C+i; Ai = A+i; Bi = B+i;
    forall (h = 0; h < m; (Ci[h], m))
    for(j = 0; j < m; j++)
        for(r = &Ci[h][j], *r=0.0, k=0; k<q; k++)
            *r += Ai[h][k] * Bi[k][j]
}

Figure 4. Exploiting 2 levels of parallelism

in line 5 is labeled FLAT and the nested parallelization is label NESTED. Both matrix have dimensions 45 x 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.

4.2 Large Integer Multiplication

A wide variety of large number crop up in mathematics, some are contrived, but others arise in proofs. Large decimal numbers beginning with 10^9. It is possible to perform different operations with these numbers, in particular the multiplication[11].

The multiplication of two n-digits numbers(n > 10^9) can be resolved by an algorithm with a complexity of O(n^2). This is a divide and conquer problem. The multiplication of two n-digits numbers is a recursive process, the multiplication is equal to the combination of four multiplication of 2^2 digits. The next expression shows the procedure to resolve z = x*y. If x = x_n x_{n-1} ... x_1 x_0  e  y = y_n y_{n-1} ... x_1 x_0 then

\[ z = x \cdot y = 10^n(a \cdot c) + 10^{n-2}((a \cdot d) + (b \cdot c)) + (b \cdot d) \]  

where a = x_n x_{n-1} ... x_0, b = x_{n-1} x_{n-2} ... x_0, c = y_n y_{n-1} ... y_0 and d = y_{n-1} y_{n-2} ... y_0.

Figure 6 shows the OTMP algorithm. The PPEL applies the methodology expressed in 2. The input problems are two n-digits numbers. The parallel recursive function receives the initial and end position of each number, in1, in2, f1 y f2, and returns the result in R.

```c
1 void PPEL (int in1, int f1, int in2, int f2, short *R) {
2     int s1 = f1 - in1, s2 = f2 - in2;
3     int i, l;
4     int s[2], f1[4], f2[4], i1[4], i2[4];
5     short *a;
6     ...
7     if ((s1 < sizeof(short))) /*The trivial case*/
8         ConvertAtoI(in1, f1, in2, f2, *k, *j);
9         k *= j;
10        ConvertItoA(k, R);
11     else /*In other case*/
12         a = calloc (4*M_BYTES, sizeof(short));
13         for (l = 0; l < 4; l++) res[l] = a + l*M_BYTES;
14         data_division(i1, i2, f1, f2, s1);
15         forall (i = 0; i < 4; r = (i*M_BYTES), M_BYTES * sizeof(short))
16             PPEL(i1[i], f1[i], i2[i], f2[i], (a + i*M_BYTES));
17         combination(a, R);
18         free(a);
19     }
20 }
```

Figure 6. OTMP Large Integer Multiplication

The figure 7 shows the obtained speed up by the OTMP algorithm in the resolution of large integer multiplication. The numbers have 4KB-digits and y 8KB-digits.

Figure 7. SpeedUp of Large Integer Multiplication

Moreover, when the size of the problems to solve is large, the speedup grows.

5 Conclusion and Future Work

In this work we have focused our attention in the efficient implementation of this functions applied to a class of al-
algorithms: divide and conquer algorithms, and propose a programming model that implement it, OTMP model.

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 proposed partnership functions that guarantee not only the correct behavior of the algorithm but also its efficiency.

OTMP has several differences with most current versions of OpenMP. One 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. The other is every clauses of parallel iteration proposed by OTMP programming model implement the hypercubic division.

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. However, the combination of every its properties makes worth the research and development of tools oriented to this model.

References


