# Performance Predictability of Divide and Conquer Skeletons

Fernando Saez  Marcela Printista

LIDIC
Universidad Nacional de San Luis.
Ejército de los Andes 950, San Luis, Argentina.
e-mail: {**bfsaez@unsl.edu.ar, mprinti@unsl.edu.ar**}

**Abstract**

Parallel divide and conquer computations, encompassing a wide variety of applications, can be modeled and encapsulated as a high level primitive called skeleton.

The paper deals with a skeleton designed for parallel divide and conquer algorithms that provide hypercubical communications among processes The paper also introduces an accurate timing model designed for prediction of proposed primitive. The timing analysis model presented here still characterizing the communication time through architecture parameters but introduces a few novelties. The proposal is to introduce different kinds of components to the analytical model by associating a performance constant for each specific conceptual block of the skeleton. The trace files obtained from the execution of the resulting code using the skeleton are used by lineal regression techniques giving us, among other information, the values of the parameters of those blocks. An extended example showing the relative accuracy of the proposed approach concludes the paper.

Keywords: Paralellism, Parallel Model, Skeleton, Timing Analysis, Divide and Conquer

## 1 INTRODUCTION

Traditionally, parallel programs are designed using low-level message passing libraries, such as PVM or MPI. Message passing provides the two key aspects of parallel programming: (1) synchronization of processes and (2) communications between processes. However, programmers still encountered difficulties because these interfaces force to deal with low-level details, and their functions are too complicated to use for a nonexpert parallel programmer.

Many attempts have been undertaken to hide parallelism behind some kind of abstraction in order to free the programmer from the burden of dealing with low level issues.

There is an alternative model of parallel programming that avoids communications and synchronization problems and restricts the form in which the parallel computation can be expressed. The model provides programming constructors: *skeletons*, that directly they correspond to frequent parallel patterns. The user expresses parallelism using a set of basic predefined forms with solution to the mapping and restructuring problems. We consider the motivations of this approach with more detail in the next section. The rest of the paper is organized as follows. The section 3 describes the Divide and Conquer Skeleton. The section 4 describes the performance predictability of a skeleton and in section 5 we discuss an instance model of hypercube divide and conquer skeleton. A case of study is presented in section 6. The results are showed in section 7 and the conclusions are presented in section 8.

## 2 MOTIVATION

An alternative to the parallel programming is to provide a set of high-level abstractions which provides support for the mostly used parallel paradigms. A programming paradigm is a class of algorithms that solve different problems but have the same control structure. Programming paradigms usually encapsulate information about useful data and communication patterns, and an interesting idea is to provide such abstractions in the form of programming templates or skeletons. In parallel context, the essence of this programming methodology is that all programs have a parallel component that implements a pattern or paradigm (provided by the skeletons) and a specific component of an application (in charge of the user). After the recognition of parallelizable parts and an identification of the appropriate algorithm, a lot of developing time is wasted on programming routines closely related to the paradigm and not the application itself. With the aid of a good set of efficiently programmed interaction routines and skeletons, the development time can be reduced significantly. The skeleton hides from the user the specific details of the implementation and allows the user to specify the computation in terms of an interface tailored to the paradigm.

To develop a specific application, the programmer/user chooses one or several skeletons, customizes them for the application and, finally, composes customized components together to obtain the executable target program. For example, we are familiar with concepts such as ”pipeline”, ”processors farms”, ”divide and conquer”, ”dynamic programming”, ”simulating annealing” and, more recently, those related with optimization problems. Before a new problem, we may try to formulate a solution in one of these well known styles. Since we already know how to implement the essential computational structure of each technique, it will only be necessary to introduce problem specific details to obtain a parallel version.

## 3 ABSTRACTION PRIMITIVE: DIVIDE AND CONQUER SKELETON

The Divide and Conquer approach ($DC$) finds the solution of a problem $x$ by dividing $x$ in subproblems $x_0$ and $x_1$. This procedure is applied recursively to solve a problem where subproblems are smaller versions of the original problem. In this typical structure, the two subproblems can be done in parallel (Fig. 1). Infinite recursion is prevented using a predicate $trivial$. If this predicate returns $TRUE$, the function $conquer$ is applied to solve the problem directly without any further division. At the ending of the procedure, the function $combine$ is used for merge the subsolutions in a general solution.

```
1  procedure DC(p: Problem, r: Result)
2  begin
3    if trivial(p) then conquer(p, r);
4    else
5    begin
6      divide(p, p0, p1);
7      do in parallel (DC(p0,r0), DC(p1,r1) );
9      combine(r, r0, r1);
10   end;
11 end;
```

Figure 1: Parallel $D\&C$ approach

From the experience obtained in the programming skeletal, especially in the design of different skeletons [7, 5], we have implemented a versatile parallel *Divide and Conquer* skeleton [6] .

The skeleton is written in C and we chose MPI (message passing interface) to avoid introduce new sintaxis.

The prototype for skeleton $DC\_Call$ is as follows:

```
void DC_Call(typeDC Type, int Weight, mInteraction IM,
             TPF_trivial Itrivial, TPF_conquer Iconquer,
             TPF_divide Idivide, TPF_combine Icombine,
             TPF_secuencial Isecuencial,
             TypeN *In,int SizeBufferIn, int SizeDataTypeIn,
             TypeN *Out, int SizeBufferOut,
             int SizeDataTypeOut, MPI_Comm comm)
```

The parameters number in the call to the skeleton $DC\_Call$, may look a little complex, but this long parameter list allows substantial flexibility, which will bring benefits in different domains. The first parameter (an enumerate type) specifies the type of algorithm to be used, which will depend of the specific problem to solve. In this work, we explore Hypercube Divide and Conquer, $HDC$. This type provides a structure with hypercubical communications among processes. It generates, recursively, a binary tree of groups of processes whose leaves consist of only one process. The number of processes in each branch is halved at each level and its interactions within a level occur between pairs of processes which will have the same rank (in distinct groups) at the next level down.

There are other types of $DC$ algorithms, such as Classical Divide and Conquer, Divide and Conquer with Embarrassing Divisibility and Divide and Conquer with Trivial Combine Operation. In the first type of algorithm, the input is presented in only one processor. When the computation reaches the division phase, the processor will communicate the subproblems to the available processors and it will continue with its subproblem. In the Divide and Conquer with Embarrassing Divisibility type, the communication is not necessary. This structure is common in many algorithms whose data are totally distributed. Examples are the scalar product and the balanced matrix multiplication. In the third type, once the final point of recursion is reached, the leave processes will have the solution to the total problem in a distributed way. For example, mergesort has a trivial divide operation.

The body of the routines $trivial$, $sequential$, $divide$ and $combine$ are described by functions and they will need to be implemented by the user.

# 4 PERFORMANCE PREDICTABILITY

A computational model is an abstract computing tool used to reason about computation. Algorithm complexity depends on the computational model in which the algorithm is defined. Some models are necessarily elaborate and include a large number of parameters. There are other models of complexity like $LogP$ or $BSP$, characterizing the performance of distributed machines through a few architecture parameters but they incur in a considerable loss of accuracy [3].

The advantage of using skeleton is the availability of a formal framework for reasoning about programs. In addition, cost measure can be associated with skeletons, thus enabling performance considerations. The timing analysis model presented here still characterizing the communication time through architecture parameters but introduces a few novelties.

The proposal is to introduce different kinds of components to the analytical model by associating a performance constant for each specific conceptual block of the skeleton.

The proposed parallel computational model considers a cluster made up by a set of $P$ processing elements and memories connected through a network. The computation in the model involves three kinds of components:

- Common pattern of paradigm: sequence of local operations needed to implement the paradigm.
- User functions of paradigm: sequence of operations on local data needed to implement the application.

- Communications: exchange of data among two or more processes in one or more processors.

Some of these components can be dropped in some type of skeleton, therefore, a skeleton model is characterized by the way it does each of the three previous components.

In conclusion, the model is characterized by the tuple:

$$(\Upsilon_{sk}, \Upsilon_{f_1}, ..., \Upsilon_{f_n}, g, l, P) \tag{1}$$

The conceptual blocks $\Upsilon_{f_i}$ will depend on the specific application and, in this case, they will be obtained by means of a description provided by the designer of the application.

Associated with these components there are cost functions. The function $\tau_{sk}$ corresponds to the time invested in computation by the skeleton (implementation of paradigm), the function $\tau_{Exc}$ is the analytical communication model and it plays an important role in prediction of the execution time of parallel applications on parallel machine and functions $\tau_{f_i}$ are associated to the cost of each specific function involved in the particular skeleton (user functions). The cost $\Omega$ of a skeleton is given by:

$$\Omega_{SK,P} = F(\tau_{sk}, \tau_{f_1}, ..., \tau_{f_n}, \tau_{Exc}) \tag{2}$$

The values of $\tau_{sk}, \tau_{f_i}$ and $\tau_{Exc}$ typically depend on the number of processors $P$ and on the problem size $SIZE$.

The time $\tau_{Exc}$ predicts the time invested in communications. In addition to $P$ and $SIZE$, it depends on the value of two determinants parameters to evaluate computational capacity of a parallel machine. In skeletons like Hypercube Divide and Conquer ($HDC$) where many sources are continuously sending data to many processors, we use an average data full throughput $g_{full}$ and a latency $l$, both dependent on $P$. In this case the communication bottleneck is the transfer capacity of the network.

In skeleton $DC\_Call$, $\tau_{Exc}(K, g_{(full,P)}, l_P)$ represents the data exchange phase, and it can be formuled by means of the following equation:

$$\tau_{Exc}(K, g_{(full,P)}, l_P) = t_{send}(K, g_{(full,P)}, l_P) + t_{recv}(K, g_{(full,P)}, l_P)$$

where $t_{send}$ and $t_{recv}$ respectively denote the time to send and receive a block containing $K$ contiguous data units. Our communication model is linear, representanting the communication time by a linear function of the message size:

$$\begin{aligned} t_{send}(K, g_{(full,P)}, l_P) &= (l_P + K * g_{(full,P)}) \\ t_{recv}(K, g_{(full,P)}, l_P) &= (l_P + K * g_{(full,P)}) \end{aligned}$$

The communication time can be summarized as follows:

$$\tau_{Exc}(K, g_{(full,P)}, l_P) = 2 * ((l_P + K * g_{(full,P)}))$$

The parameters needed to design the model are not only architecture dependent $(P, l, g)$ but also it must be reflect skeletal characteristics ($\Upsilon_{sk}$, $\Upsilon_{f_i}$). The dependence of the architecture allowed in the cost functions is in the coefficients defining each particular function. Thus, once the analysis for a given architecture has been completed, the predictions for a new architecture can be obtained replacing in the formulas the function coefficients.

# 5 PERFORMANCE PREDICTABILITY OF HYPERCUBE DIVIDE AND CONQUER SKELETON

Next, we present and verify an accurate timing parallel model of computation developed to analyze and to predict performance of $HDC$ algorithms using the skeleton $DC\_Call$.

In order to achieve an instance model of $HDC$, we need to describe the cost functions. As we have previously seen, a divide and conquer problem needs to define five specific functions: $trivial$, $conquer$, $divide$, $combine$ and $sequential$. We do not include costs associated with tasks $trivial$ and $conquer$ since they do not make a significant contribution to execution time when $SIZE$ much bigger than $P$.

The parallel time of an $HDC$ algorithm with input size $SIZE$ can be formulated as a function of five parameters:

$$\Omega_{HDC,P}(SIZE) = F(\tau_{HDC}, \tau_{Div}, \tau_{Comb}, \tau_{Seq}, \tau_{Exc}) \tag{3}$$

The communication time of a cluster of $P$ processors can be described by machine-dependent parameters: $g$, the time needed to send one data word into the communication network, or to receive one word, in the asymptotic situation of continuous message traffic and $l$, the latency or startup cost. They are well known in parallel models area, and they are easy to obtain from any cluster. So, associated to the stage of communication between partners there is a cost function $\tau_{Exc}(SIZE, g, l, P)$ which gives us a time prediction invested for communications in terms of number of processors ($P$) in the work group and the lengths of messages involved.

In this case, $\tau_{HDC}$ is considered zero because $\Upsilon_{HDC}$ is insignificant compared to other components affecting the overall. Next, we need to find a different proportionality constant (cost) for each specific function.

The time $\Omega_{HDC,P}(SIZE)$ taken by $P$ processors using the skeleton $DC\_Call$ is recursively defined by the formula:

$$\begin{aligned}\Omega_{HDC,P}(SIZE) &= \max_{i=1,\ldots,P}\{\tau_{Div,i}(SIZE)\} + \max_{i=1,\ldots,P}\{\Omega_{HDC,P}(\frac{SIZE}{2})\} + \\ &\quad \max_{i=1,\ldots,P}\{\tau_{Exc,i}(K_0, g, l)\} + \max_{i=1,\ldots,P}\{\tau_{Comb,i}(SIZE)\}\end{aligned}$$

$\Omega_{HDC}$ can be derived by successive substitution:

$$\begin{aligned}\Omega_{HDC,P}(\frac{SIZE}{2}) &= \max_{i=1,\ldots,P}\{\tau_{Div,i}(\frac{SIZE}{2})\} + \max_{i=1,\ldots,P}\{\Omega_{HDC,P}(\frac{SIZE}{4})\} + \\ &\quad \max_{i=1,\ldots,P}\{\tau_{Exc,i}(K_1, g, l)\} + \max_{i=1,\ldots,P}\{\tau_{Comb,i}(\frac{SIZE}{2})\}\end{aligned}$$

$$\begin{aligned}\Omega_{HDC,P}(\frac{SIZE}{4}) &= \max_{i=1,\ldots,P}\{\tau_{Div,i}(\frac{SIZE}{4})\} + \max_{i=1,\ldots,P}\{\Omega_{HDC,P}(\frac{SIZE}{8})\} + \\ &\quad \max_{i=1,\ldots,P}\{\tau_{Exc,i}(K_2, g, l)\} + \max_{i=1,\ldots,P}\{\tau_{Comb,i}(\frac{SIZE}{4})\}\end{aligned}$$

And so on. When processors finish the recursion, the time is given by:

$$\begin{aligned}\Omega_{HDC,P}(\frac{SIZE}{2^{(\log P)-1}}) &= \max_{i=1,\ldots,P}\{\tau_{Div,i}(\frac{SIZE}{2^{(\log P)-1}})\} + \max_{i=1,\ldots,P}\{\Omega_{HDC,P}(\frac{SIZE}{2^{\log P}})\} + \\ &\quad \max_{i=1,\ldots,P}\{\tau_{Exc,i}(K_{(\log P)-1}, g, l)\} + \max_{i=1,\ldots,P}\{\tau_{Comb,i}(\frac{SIZE}{2^{(\log P)-1}})\}\end{aligned}$$

After $\log P$ division steps, the algorithm resolves each subproblem in a sequential form:

$$\Omega_{HDC,P}(\frac{SIZE}{2^{\log P}}) = \max_{i=1,\ldots,P}\{\tau_{Seq,i}(\frac{SIZE}{2^{\log P}})\}$$

The cost of an algorithm is simply the sum of the costs of its components:

$$\begin{aligned}\Omega_{HDC,P}(SIZE) &= \sum_{j=0}^{(\log P)-1} \max_{i=1,\ldots,P}\{\tau_{Div,i}(\frac{SIZE}{2^j})\} + \sum_{j=0}^{(\log P)-1} \max_{i=1,\ldots,P}\{\tau_{Comb,i}(\frac{SIZE}{2^j})\} \\ &+ \sum_{j=0}^{(\log P)-1} \max_{i=1,\ldots,P}\{\tau_{Exc,i}(k_j,g,l)\} + \max_{i=1,\ldots,P}\{\tau_{Seq,i}(\frac{SIZE}{2^{\log P}})\}\end{aligned}$$

### 5.1 Measuring Communication Performance

Different proposals can be used to determine $\tau_{Exc}$. For example, it would be valid to take as $\tau_{Exc}$ the empirical set of linear by pieces functions obtained from Abandahs [2] or Arruabarrenas [4] studies, where latency and bandwidth are considered depending on the communication pattern. On other hand, it would be valid to evaluate the effective latency and throughput of message transmissions using a classical round-trip test (also called a ping-pong test) between two machines. The ping-pong test measures the time needed for a message of a given length to go from one machine to the other and to come back immediately.

The timming model presented characterizing the communication time through architecture parameters and message size. It assumes a linear by pieces behaviour in the message size of the functions in $\tau_{Exc}$. However, this behaviour can be non-linear in the number $P$ of processors (i.e. broadcast usually have a logarithmic factor in $P$).

To obtain the architecture parameters, we used micro-benchmarks [1]. they were run in a dedicated fashion, i.e. no other user programs or unnecessary system services were allowed to run during the benchmarking. We considered two micro-benchmarks to measure; (1) The worst case $l$ (latency) and (2) The worst case $g_{full}$, measured for data full all-to-all collective communication (throughput). Each one of them was calculated for two different methods; (1) Let the message size vary and (2) Let the number of messages vary.

The Figure 2 shows the parameters for the cluster $LIDIC$. The cluster consisting of 14 networked nodes, each one a Pentium IV of 3.2 GHz and 1 GB of Ram. The nodes are connected together by Ethernet segments and a Switch Linksys srw $2024$ of 1 $GB$. The base software on cluster include a Debian etch SO, and MPICH 2 1.0.6. In both graphics, the $y$-value corresponds to the time it takes for a single integer (32-bit word) to be delivered. Note the latency increases much faster than the throughput decreases. Also note that the scale of the y-axes differ a factor $1000$. One may conclude that latency becomes the most important factor when the number of processors grows.

Next sections exemplify the use of the model to predict the time spent by a $HDC$ algorithm.

## 6 CASE STUDY: CONNECTED COMPONENT

The problem of connected component is usually considered one of the most elementary graph problems. It try to find all connected components of an undirected graph $G = (V, E)$ of $|V| = N$ nodes

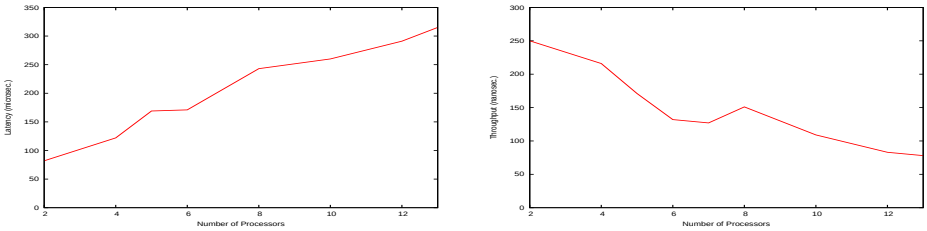


Figure 2: Latency and Throughput on Cluster LIDIC

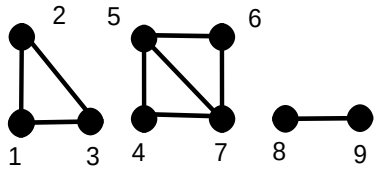


Figure 3: Example of a graph

and $|E| = M$ edges. The connected components of $G$ are the sets of nodes such that all nodes in each set are mutually connected (reachable by some path), and no two nodes in different sets are connected.

As a case of study we propose a parallel divide and conquer algorithm to identify subgraphs of the graph in which the nodes of the subgraph are connected.

We use a simple *edges list* to represent the graph. If $G = (V, E)$ is a graph, its edges list is a structure $X_{Mx2}$, where each edge $e_i = (u, v) \in E$, $u < v$, is stored in $X[i]$, and $X[i, 0] = u$ y $X[i, 1] = v$. The algorithm takes as input the edges list of a graph $G$ ($N$ nodes and $M$ edges) stored in a generic structure $VectorInput$. The procedure keeps for each node in the graph, all other nodes which it has an edge to (the *adjacency list* of a node). The algorithm produces as output a vector $VectorOutput$, where each element $VectorOutput[j]$ for $1 \leq j \leq N$ will store the root of the connected component for $v_j$.

The following main program shows the connected components algorithm using the skeleton $DC\_Call$:

```
Algorithm ParallelConnectedComponent()
   initPar(&VectorInput,&VectorOutput,N,M);
   DC_Call(HDC, 2, &trivial, &conquer, &divide, &combine, &secuencial, MPI_COMM_WORLD,
           &VectorInput, M, sizeof(edge), &VectorOutput, N, sizeof(int));
```

The first argument is used to specify the $D\&C$ type. In this case, it is a Hypercube Divide and Conquer algorithm. The second parameter indicates the ramification factor. For $HDC$, $Weight$ must be equal to 2 (the number of activities generated in the division phase). The next five arguments are used to introduce the specific code of application (user functions).

Here, we give a little example to show how the parallel algorithm works when using four processors. Consider the graph as shown in the Figure 3, where $|E| = M = 9$ and $|V| = N = 9$.

The edges list $< (1, 2), (2, 3), (1, 3), (4, 5), (5, 6), (6, 7), (4, 7), (5, 7), (8, 9) >$ is stored in *VectorInput*.

The procedure *divide* divides the edges list in two of about $n/2$ edges each. Each of which is then solved by applying the same approach recursively. Once we have reached a base case (*trivial*), a function *conquer* is applied to find the roots of the connected components of its subedges list.

In the above example, *VectorOutput* associated to processor $P_i$ would then look like:

- $P_1 : < 1, 1, 1, 4, 5, 6, 7, 8, 9 >$

- $P_2 : < 1, 2, 3, 4, 4, 4, 7, 8, 9 >$
- $P_3 : < 1, 2, 3, 4, 5, 4, 4, 8, 9 >$
- $P_4 : < 1, 2, 3, 4, 5, 6, 5, 8, 8 >$

Where $VectorOutput$ associated $P_i$ contains the roots of the connected components of its subedges partition. For example, in the processor $P_1$, the node $1$, $2$ and $3$ in the graph $G$ belong to the same subgraph with root in node $1$.

Then, the function *combine* can just merge this sublist with its level partner to produce a new root vector of connected components. After the first *merge*, the *VectorOutputs* will be as follows:

- $P_1 : < 1, 1, 1, 4, 4, 4, 7, 8, 9 >$
- $P_2 : < 1, 1, 1, 4, 4, 4, 7, 8, 9 >$
- $P_3 : < 1, 2, 3, 4, 4, 4, 4, 8, 8 >$
- $P_4 : < 1, 2, 3, 4, 4, 4, 4, 8, 8 >$

After the last *merge*, the *VectorOutputs* will be as follows:

- $P_1 : < 1, 1, 1, 4, 4, 4, 4, 8, 8 >$
- $P_2 : < 1, 1, 1, 4, 4, 4, 4, 8, 8 >$
- $P_3 : < 1, 1, 1, 4, 4, 4, 4, 8, 8 >$
- $P_4 : < 1, 1, 1, 4, 4, 4, 4, 8, 8 >$

The list identifies the root of each connected component. It can be seen that the root of a component is the member node with lowest visitation index.

# 7 MEASURING OF D&C FUNCTIONS OF CONNECTED COMPONENT

To predict the time of $HDC$ algorithms is necessary to estimate the execution time of each function implemented by the user ($divide$, $combine$ and $sequential$) on the cluster.

The algorithm takes as input two parameters: the number of nodes ($N$) and edges ($M$). Each parameter affects in some way to the specific function. In the function $divide$, the number of edges of the graph plays a fundamental role to estimate the cost. On the other hand, the number of nodes determines the behavior for function $combine$. The function $sequential$ is affected by both variables.

Multivariate statistics refers to a group of inferential techniques that have been developed to handle situations where sets of variables are involved as predictors of performance. We make use of statistical analysis for determining model coefficients.

A model relating the experimental execution time of function $divide$ to a set of independent variables is: $T_{Div} = Div_0 + Div_1 * M$ where $1$ and $M$ are the basis functions of the model and $Div_i$ will be estimated by the parameter estimation algorithm. Linear *least-squares models (LSQ)* estimate the coefficients $Div_0$ and $Div_1$ to minimize the squared sum of errors between predicted and experimental values of function $divide$. The execution time model of function $combine$ is obtained in a similar way: $T_{Comb} = Comb_0 + Comb_1 * N$, where $Comb_0$ and $Comb_1$ are their unknown coefficients.

Each data in the trace file is used like input for regression techniques to find coefficients that achieve the best approximation to the real function.

The function $sequential$ has a cost of $O(2N + M)$. To build an adjacency list, it requires $N$ operations, then it makes a $DFS$ (Depth-First Search) algorithm to find the connected components.

| $SIZE$ | $T_{Div}$ | $T_{Comb}$ |
|---|---|---|
| 100 | 0.000003 | 0.000029 |
| 500 | 0.000013 | 0.000130 |
| 1000 | 0.000021 | 0.000274 |
| 2000 | 0.000042 | 0.000609 |
| 4000 | 0.000080 | 0.001279 |
| 8000 | 0.0000157 | 0.002374 |
| 16000 | 0.000313 | 0.004199 |
| 32000 | 0.000623 | 0.006922 |
| ... | ... | ... |

Table 1: Trace file of experimental execution times of functions $divide$ and $combine$

From our algorithm, $DFS$ with adjacency list requires time proportional to $O(N + M)$. The table 2 shows a sample of invested time for function $sequential$. We can see it is linear in the size of the structure. The values of $Seq_0$, $Seq_1$ and $Seq_2$ will be obtained from regression techniques and they conform the coefficients of $T_Seq = Seq_0 + Seq_1 * N + Seq_2 * M$.

| $N$ | $M$ | $T_{Seq}$ | $N$ | $M$ | $T_{Seq}$ |
|---|---|---|---|---|---|
| 4000 | 4000 | 0.000960 | 16000 | 4000 | 0.001182 |
| 4000 | 8000 | 0.01804 | 16000 | 8000 | 0.002264 |
| 4000 | 16000 | 0.003624 | 16000 | 16000 | 0.004386 |
| 4000 | 32000 | 0.008119 | 16000 | 32000 | 0.009350 |
| 8000 | 4000 | 0.001065 | 32000 | 4000 | 0.001411 |
| 8000 | 8000 | 0.002029 | 32000 | 8000 | 0.002537 |
| 8000 | 16000 | 0.003962 | 32000 | 16000 | 0.004929 |
| 8000 | 32000 | 0.008445 | 32000 | 32000 | 0.010441 |

Table 2: Trace file of real execution times of function $sequential$

The table 3 shows the estimated values for approximation functions $divide$, $combine$ and $sequential$. In all cases, the coefficients of determination exceed $95\%$.

| | | | | |
|---|---|---|---|---|
| $division$ | $Div_0$ | $Div_1$ | | $(R^2)$ |
| | $1.94e^{-08}$ | $2.25e^{-06}$ | | $99\%$ |
| $combine$ | $Comb_0$ | $Comb_1$ | | $(R^2)$ |
| | $2.19e^{-07}$ | $2.32e^{-03}$ | | $95\%$ |
| $sequential$ | $Seq_0$ | $Seq_1$ | $Seq_2$ | $(R^2)$ |
| | 0 | $1.35e^{-07}$ | $4.60e^{-07}$ | $98\%$ |

Table 3: Numerical value of coefficients

**Results**

We performed initial experiments. The table 4 shows the execution time for several problem sizes using 8 processors.

To predict performance of some instances, the prediction model $\Omega_{HDC,P}(Size)$ was resolved using models and coefficients shown in section $5$. Like an example, we use the model to predict the

| $P$ | $N$ | $M$ | $T_{Par}$ |
|---|---|---|---|
| 8 | 1024000 | 512000 | 1.719770 |
| 8 | 1024000 | 1024000 | 1.844182 |
| 8 | 1024000 | 2048000 | 2.102299 |
| 8 | 2048000 | 512000 | 3.002905 |
| 8 | 2048000 | 1024000 | 3,128890 |
| 8 | 2048000 | 2048000 | 3.504908 |

Table 4: Times to find connected component using parallel algorithm

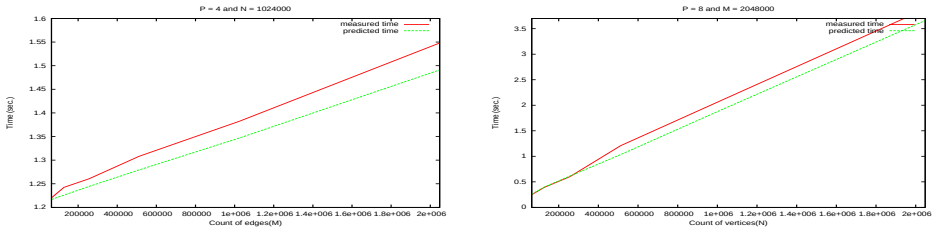


Figure 4: Measured Time vs Predicted Time for different configurations.

execution time to find connected components for a graph $G = (V, E)$ with $|V| = N = 1024K$ y $|E| = M = 2048K$ using $8$ processors.

To estimate the time spent in communication, we instantiate $t_{Exc}$ with the number of words (32-bit) to communicate by each recursion level and the architecture parameters ($l$ and $g$). For this problem, each algorithm recursion level always communicates the whole root's vector, this is $1024K$ of 32-bits words.

$$\begin{aligned}\sum_{j=0}^{2} \max_{i=1,\ldots,8} \{\tau_{Exc,i}(1024K, 1.51 * 10^{-7}, 2.43 * 10^{-4})\} &= 3 * 2 * (0.000243 + 1024K * 1.51 * 10^{-7}) \\ &= 3 * 2 * (0.000243 + 0.154624) = 0.929202\end{aligned}$$

The predicted time to solve the connected components of a graph $G$ is the sum of the time invested for each component function:

$$\Omega_{HDC,8}(1024K, 2048K) = 1.916492$$

In this particular prediction the error was $8,83\%$. (Predicted = 1.916492 versus Observed= 2.102299, see table 4). Errors are basically due to the lack of accuracy for the communication component.

The figure 4 shows a comparison between predicted time and the traces obtained for several configurations. The results were very near to the expectable behaviour.

## 8 CONCLUSIONS AND FUTURE WORK

In this paper, we first described the skeleton $DC\_Call$. It provides high-level abstraction for programming divide and conquer algorithms, easing the expression of parallelism, communication, and

synchronization. The results obtained using the skeleton described in this paper prove it to be suitable to handle the class of problems parallel divide and conquer. The application programmers benefit from proposed skeleton, which hides much of the complexity of managing parallel divide and conquer algorithms. We believe that preserving the semantics of the sequential divide and conquer program is a key point to achieve the objective to alleviating the difficulties in the development of parallel applications.

Besides this, the paper also presents and verifies an accurate timing model to predict the performance of the proposed primitive on a clusters of processors. The parameters needed to design the model are not only architecture dependent $(P, l, g)$ but also it must be reflect skeletal characteristics. The dependence of the architecture allowed in the cost functions is in the coefficients defining each particular function of the skeleton $DC\_Call$. Thus, once the analysis for a given architecture has been completed, the predictions for a new architecture can be obtained replacing in the formulas the function coefficients. We used statistical analysis for determining analytical model coefficients. The multivariate techniques described here are particularly applicable because of the large number of samples the system allows to obtain. In this way, the programmers are provided with a high level primitive whose different implementations have a well-understood behaviour and predictable efficiency.

Future work should concentrate in describing other abstraction primitives and to extend the skeleton's portability to support both shared and distributed memory architectures.

## ACKNOWLEDGMENTS

We wish to thank the Universidad Nacional de San Luis, the ANPCYT and the CONICET from which we receive continuous support.

## REFERENCES

[1] Mpiedupack 1.0. available at http://www.math.uu.nl/people/bisseling/edupack/mpiedupack1.0.tar.

[2] Davidson E.S. Abandah, G.A. *Modeling the Communication Performance of the IBM SP2*. Proc. 10th IPPS., 1996.

[3] Printista A.M. *Modelos de Predicci'on en Computaci'on Paralela*. Thesis of Magister submitted to the Universidad Nacional del Sur., 2001.

[4] Arruabarrena A. Beivide R. Gregorio J.A. Arruabarrena, J.M. *Assesing the Performance of the New IBM-SP2 Communication Subsystem.* IEEE Parallel and Distributed Technology. pp 12-22., 1996.

[5] C. Rodriguez Len F. Piccoli, M. Printista. *Dynamic Hypercubic Parallel Computations*. Proceeding (466) Parallel and Distributed Computing and Systems, 2005.

[6] M. Printista F.D. Saez. *Programacin Paralela Esqueletal*. XIII Congreso Argentino de Ciencias de la Computacin, Corrientes and Resistencia, Argentina, October 2007.

[7] M. Printista J.G. Zanabria, F. Piccoli. *Hypercubic Comunications in MPI.* Tesis submited for UNSL, 2005.