Parallel Cluster Labeling on a Network of Workstations[1]

Felipe Knop[2]

Department of Computer Sciences
Purdue University
West Lafayette, Indiana 47907, USA

Laboratório de Sistemas Integráveis
Departamento de Engenharia Eletrônica
Escola Politécnica da Universidade de São Paulo
São Paulo, SP, Brazil

E-mail: knop@cs.purdue.edu

Vernon Rego
Department of Computer Sciences
Purdue University
West Lafayette, Indiana 47907, USA
E-mail: rego@cs.purdue.edu

Resumo

Nos últimos anos, encorajados pelas rápidas estações de trabalho e também pelos sistemas de software que visam transformar aglomerados de estações em ambientes de programação paralela, redes de estações tem sido cada vez mais usadas como engenhos computacionais. Entretanto, redes de estações não são substitutos ideais para supercomputadores, devido à baixa capacidade de interconexão fornecida pelas atuais redes locais. Neste artigo, apresentamos *EcliPSe*, um sistema de software para processamento paralelo em ambiente heterogêneo para aplicações baseadas no paradigma de replicação. *EcliPSe* também suporta formas mais gerais de processamento paralelo basedo em troca de mensagens. Neste artigo, apresentamos o uso do sistema *EcliPSe* na paralelização de um algorítmo de rotulamento de aglomerados. O algorítmo foi projetado para usar algumas funcionalidades de *EcliPSe* para reduzir o tempo gasto em comunicação, o que torna o algorítmo adequado para uso em redes de estações. Descrevemos no artigo algumas medidas de desempenho obtidas na execução do programa paralelo.

Abstract

In recent years, encouraged by today's fast workstations and by software systems designed to transform workstation clusters into parallel programming environments, network of workstations have been increasingly used as computational engines. Networked workstations, however, are not ideal replacements for supercomputers, because of the low interconnection capacity provided by current local area networks. In this paper, we present an application using the *EcliPSe* toolkit, a system for replication-based parallel processing in heterogeneous environments. Although primarily designed for replicative applications (that generally do not require large amounts of communication), *EcliPSe* can be used for more general forms of message-passing parallel processing. We describe the use of the toolkit in the parallelization of a cluster labeling algorithm. The algorithm is designed so that it uses some of *EcliPSe*'s features to reduce communication overhead, making the algorithm suitable for execution on workstation clusters. Early performance results are described.

[1] Research supported in part by NATO-CRG900108, NSF CCR-9102331, ONR-9310233, and ARO-93G0045.

[2] PhD student. Research supported by CNPq-Brazil process number 260059/91.9.

1 Introduction

The last few years have witnessed a dramatic increase in the processing power available on networked workstations. Many scientific applications that previously required supercomputers can now be executed in a distributed manner by dividing the computation among heterogeneous workstations and using a local (or wide) area network as the communication substratum. This approach has been called "cluster computing", "metacomputing", "network-based computing", among other names [14]. Using networked workstations in distributed computing is very attractive, since these are usually much cheaper and more easily available than supercomputers. Also, the increase in available networked resources available has fostered the development of several software systems giving support for cluster computing, motivating scientists to move many compute-intensive programs to networked workstation environments. We mention PVM [13], HeNCE [1], Isis [2], MPI [4], and *EcliPSe* [10] as example of such software systems.

At the present time, heterogeneous workstation clusters are not ideal replacements for supercomputers, mainly because of their low interconnection bandwidth and reliability. Today's relatively low speed networks and communication protocols, not really designed to support cluster computing, yield long communication delays. Consequently, cluster computing applications are forced to use communication efficiently to avoid communication bottlenecks.

Keeping in mind the limitations of current network environments, we have designed the *EcliPSe* toolkit primarily to ease the task of parallelizing replication-based simulation applications (those where a simulation must be run several times to obtain confidence intervals for some desired parameters; see [5]). We describe in [7, 8] some performance experiments of the toolkit using replication-based applications. *EcliPSe*, however, also provides features that support more general forms of distributed computing. Here, we plan to demonstrate the utility of these features.

In this work we use *EcliPSe* to parallelize the execution of a *cluster labeling* algorithm [3, 6]. Nowadays, researchers investigate statistical mechanics of polymer solutions. Among the research questions are [9]: What shape does a long chain molecule take on when it is confined in something like a porous sandstone? How do polymers move through a membrane ? The first question is of interest to the oil industry and the second to pharmaceutical companies. The problem is appraoched by focusing on a linear chain that has a restricted interaction with the medium; that is, there are forbidden regions, and the chain is confined to other parts of the medium. This can be modeled by a self-avoiding walk on a randomly diluted lattice (see [11] for example), which is created by having a grid that is filled with probability p. A self-avoiding walk is one that moves from a site to a nearest neighbor, avoiding sites that were previously occupied and also sites that are not filled. One phase of the algorithm is the cluster labeling phase, where the connected components are identified. Similar models are used in applications ranging from the determination of the behavior of the DNA molecule to the determination of the electrical conductivity in composite materials.

Our resulting parallel cluster labeling program makes use of a workstation cluster. We evaluate our effort in regard to performance and ease of parallelization.

The remainder of this paper is organized as follows. Section 2 presents the specification of the cluster labeling problem and a sequential solution. Section 3 describes the *EcliPSe* toolkit. In section 4 we describe the approach taken by our parallel solution to the cluster labeling problem, and also the use of *EcliPSe* features in the implementation of the program. Section 5 presents the results of initial performance measurements, and Section 6 concludes the paper.

2 Problem Specification

2.1 The Cluster Labeling Problem

Let M be a $N \times N$ 2-D mesh, where each element (site) $M[i, j]$ $(0 \leq i, j < N)$ can be either "full" or "empty". Element $M[i, j]$ is a *neighbor* of element $M[k, l]$ iff

$$\begin{array}{l} (k = i \text{ and } l = j + 1) \quad \text{or} \\ (k = i \text{ and } l = j - 1) \quad \text{or} \\ (k = i + 1 \text{ and } l = j) \quad \text{or} \\ (k = i - 1 \text{ and } l = j) \end{array}$$

(that is, no "diagonal neighbors" are allowed, although the proposed algorithm can accommodate such generalizations)

Site $M[c, d]$ is *reachable* from site $M[a, b]$ if there exists a sequence of "full" sites

$$S_0 = M[a, b], S_1, \ldots, S_n = M[c, d]$$

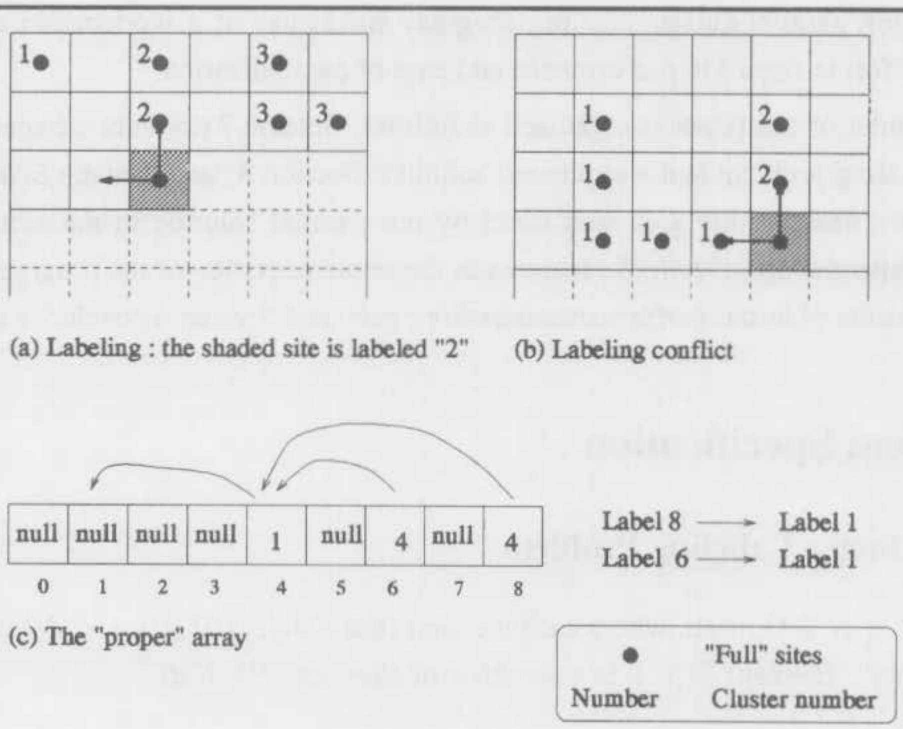
such that S_{i+1} is a neighbor of S_i.

A *cluster* $C(i, j)$ is defined as a set containing $M[i, j]$ and all elements $M[k, l]$ such that $M[k, l] \in C(i, j)$ iff $M[k, l]$ is reachable from $M[i, j]$. The task of the cluster labeling algorithm is to identify (label) each cluster occurring in M with a unique integer id.

2.2 Sequential Solution

We describe the sequential solution presented in [6].

One intuitive solution to the problem is to traverse M row by row in increasing order, assigning to $M[i, j]$ the same label as $M[i - 1, j]$ if both positions are "full", or the same label as $M[i, j - 1]$ if both $M[i, j]$ and $M[i, j - 1]$ are "full", as Figure 1(a) shows. It may be possible, however, that the two neighbors examined are both "full" and have *different* labels, as shown in Figure 1(b). This situation occurs when two clusters initially thought as being different are actually the same. The solution is to *re-label* one of the two clusters and to mark the new site



(a) Labeling : the shaded site is labeled "2" (b) Labeling conflict

(c) The "proper" array

Figure 1: Sequential cluster-labeling algorithm

as belonging to the other cluster. In the example of Figure 1(b), the shaded position is assigned label 1, and the elements previously belonging to cluster 2 are moved to cluster 1. To implement the relabeling phase efficiently (avoiding having to search for all elements whose labels must be changed) an array named `proper` is created and initialized to *null*. When elements of cluster x are moved to cluster y, we execute the command

```
proper[x] <- y
```

In a second traversal of M, we update the label of each $M[i,j]$ by recursively accessing array `proper` until we find `proper[x]` is *null*, as Figure 1(c) shows. Then we set the label of $M[i,j]$ to be x.

2.3 Evaluating the Algorithm

To evaluate the cluster-labeling algorithm experimentally, we must first create a sample 2-D mesh. This task is accomplished by randomly filling each site with "full" or "empty", with a given probability p of having each site filled with "full". Different values of p result in the algorithm (sequential and parallel) having different performance.

We measure the execution time of the algorithm as $(t_B - t_E)$, where t_B is the time at which mesh initialization begins, and t_E is the time at which all cluster labeling ends. To measure the parallel execution time, the same definition applies, but we must be careful to wait for all the machines to finish their tasks.

3 The *EcliPSe* Toolkit

In this section we examine the execution environment where the parallel cluster-computing algorithm is implemented: the *EcliPSe* toolkit. *EcliPSe* is a cluster computing software system for heterogeneous networked processors. It has been created primarily to parallelize replicative applications, being able to obtain excellent performance for such applications. The system's design was guided by the following simple goals:

simplicity: little work should be required in modifying a sequential application so that it can be replicated and executed on several processors in a coordinated manner.

flexibility: although structured for replicative simulations, the system should also allow general data-parallel computations with interprocess communication.

portability: a distributed application should execute on a variety of architectures, including multiprocessors and workstations on wide area networks.

scalability: mechanisms that inhibit serializing bottlenecks should be provided, so that applications can scale well to run on a large number of processors.

fault tolerance: programs should be able to recover from machine crashes and other failures that affect long-running applications.

The present version of *EcliPSe*, an improvement over the prototype presented in [10, 12], has already been used for production applications, such as the work described in [11].

3.1 Structure

A sequential application requires only minimal changes in order to utilize the power of *EcliPSe*. This generally entails insertion of *EcliPSe* primitives in the original source code with some (usually trivial) rearrangement of the code. Also, the user is required to provide a file containing the names of the machines to be used, usually (though not necessarily) "idle" workstations. The end result is a run consisting of a set of concurrently executing *computational processes* coordinated by one or more *monitor* processes. An *EcliPSe* program must contain the following components:

Computation code. This is code that is run by each of the *computational processes*, being responsible for most of the "actual work" that the application performs. The computation code usually requires (input) data from and returns (result) data to a monitor process.

Monitor function(s). These are functions executed by *monitor processes*. A monitor process is responsible for coordinating the computation done by a set of computational processes, generating data for and collecting data from these processes, and finally determining when the computation should be terminated.

Declarations. Each type of data item that is exchanged between monitors and computational processes must be declared. By declaring data types, the user informs *EcliPSe* about data characteristics. Declarations provide the added advantage of making explicit the flow of information between monitors and computational processes.

3.1.1 Declarations

EcliPSe declarations are handled by a special preprocessor, which allows the user to make the declarations using a "C-like" syntax. For example, suppose that computational processes produce an array of 10 double precision numbers for the monitor. The declaration of this data item is done as follows:

```
eclipse_decls {
        double   type_result[10];
}
```

The `eclipse_decls` block defines the region that the preprocessor is supposed to act on. The preprocessor declares an *integer* variable called `type_result` that, at run time, will contain a handle used in all subsequent *EcliPSe* calls that refer to the double precision array (analogous to the notion of "file descriptor" in the UNIX systems). Therefore, when an array of 10 double precision numbers is to be sent to the monitor, only the data type handle and a pointer to the data must be provided. Data is then transmitted in a machine-independent format.

3.1.2 Computation code primitives

The basic primitives available to the computational processes are simple: `request_data` obtains data from a monitor, and `put_stat` sends data to a monitor. Both take as parameters an (integer) type handle obtained in the declarations part and a pointer to the data to be transferred. In general, if an application's sequential code is already available, changing it to work with *EcliPSe* is a simple task. It suffices to (a) replace the data input code (sometimes obtained from the keyboard or from a file) by the corresponding `request_data` primitives, and (b) replace the collection of result and statistics by the corresponding `put_stat` primitives.

The functionality of the original user code that was replaced by `request_data` and `put_stat` primitives as described above is moved to the monitor function.

3.1.3 Monitor function

All code related to file I/O and to the collection of statistics is placed inside the monitor function, which is executed by a *monitor process.* If a sequential application is being ported to *EcliPSe*, writing the monitor generally means moving the above functionality from the computation code into the monitor. Calls to `produce_data` (counterpart to `request_data`) and `collect_stat` (counterpart to `put_stat`) are inserted when needed. Thus, if the original code read an input parameter from a file, the *EcliPSe* code will have the file read operation and a call to `produce_data` in the monitor function, and a call to `request_data` in the computational process.

It is worth noting that it is always up to the monitor to decide whether the computation must be terminated. The computational processes must stay in an infinite loop, always working "on demand". Being the only process with some global notion of the computation, the monitor is the only process capable of deciding when it is time to terminate the computation.

3.2 General features

Without mechanisms for efficient data transfers between computational processes and monitor processes, it is possible for a monitor process to become a bottleneck for a distributed computation. *EcliPSe* provides a set of control mechanisms that prevent such serializing bottlenecks and network clogs from occurring. These include:

Granularity control. With a small change in a data type declaration, the user may specify a "grain size" to be used for that type. As a result, subsequent `put_stat` calls buffer data instead of sending data directly to a monitor. When the number of buffered data items reaches "grain size", the buffered data is sent in a single message. This helps reduce network usage and also decreases overhead at monitors and at computational processes.

Multiple monitors. If a monitor is being overworked due to high incoming traffic, then the incoming workload can be distributed among several monitor processes. This is accomplished by coding additional monitor functions for the different workloads, specifying their names in the declarations, and indicating (again, in the declarations) which data types are to be associated with which monitors. Code for the computational processes need not change.

Tree-combining. If a monitor were to receive data directly from a large number of computational processes, the amount of incoming traffic and resulting combining work could make the monitor a bottleneck for the entire computation. This can happen, for example, when the monitor averages results it receives from computational processes. To prevent such a bottleneck from occurring, *EcliPSe* allows processes to be organized in a virtual tree structure, with a user-defined topology and the monitor as the root. Each computational process transparently sends data to its parent in the tree, instead of sending data directly to the root. Each such parent *combines* its own results with the results it receives from all of its children in the tree, applying the same operation that the monitor process would have applied had the tree-combining scheme not been used. As a result, the monitor at the root only needs to combine data it receives from its own children.

The user may choose to employ the tree-combining scheme by declaring a data type to be a "combining type" and by specifying its corresponding combining operation. The latter may be either a user-written function or one of the standard combining operations provided by *EcliPSe* (i.e., averaging, summation, concatenation, and others). No change is required for the computational processes.

Data-diffusing. The virtual tree structure described above can also be used to speed up the distribution of data from a monitor to each computational process. Instead of sending data directly to each computational process, a monitor only needs to use the `produce_data_diffuse` primitive on an array of data items to be distributed. Data is then "diffused" down the tree, with each computational process receiving a data item. The process behaves like a tree-combining process in reverse. The same primitive can also be used for an efficient data broadcast. As with tree-combining, the declaration of a data type must be changed to indicate use of data-diffusing. Code for the computational processes need not change.

4 Parallel Cluster Labeling

4.1 Outline

The basic idea of the proposed parallel cluster labeling algorithm is to divide the mesh M into n row-wise strips (n is the number of processors being used) and assign each strip to a different processor. Each processor then executes the sequential algorithm outlined in section 2.2 on its strip. This simple approach leads to incorrect labeling, since actual clusters may span more than one strip. Therefore it is clear that processors must communicate to allow multi-strip clusters to be correctly labeled.

One re-labeling alternative is to make each processor numbered $2i+1$ send its strip to processor $2i$ ($i = 0, 1, \ldots n/2$), allowing the latter to re-label the clusters in both strips. The whole procedure is repeated $\log_2 n$ times in a tree-combining fashion, resulting in the whole mesh M being correctly labeled. As it turns out, this approach is clearly impractical for a workstation cluster, because of the amount of data that must flow over the network for large meshes. This scheme also requires one processor to hold the final mesh, which may not be possible for large meshes.

Another alternative, which we adopt, involves the transmission of *boundary rows* rather than of the whole strip. When processor i sends its last row to processor $i+1$, the latter can build a *mapping table*, as shown in Figure 2.

Using this mapping table, processor $i+1$ can re-label all its clusters. Unfortunately this

Figure 2: Example of mapping table

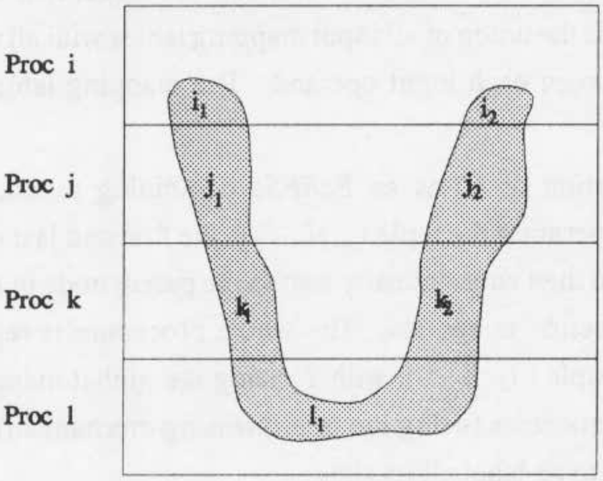


Figure 3: Example of cluster that spans over more than one processor

re-labeling using only information local from processors i and $i + 1$ is not enough to guarantee correct labeling. This is demonstrated in Figure 3.

In the example of Figure 3, it is only after gathering information from processors i, j, k, and l that we find that clusters i_1 and i_2 (for example) are actually part of the same cluster. It is clear that an accurate mapping table can only be obtained after examining *all* processors.

The main idea of the proposed algorithm is to use the tree-combining mechanism to build and combine mapping tables rather than whole strips, therefore dramatically reducing the amount of information that must flow between processors. Implementing tree-combining is made reasonably simple by using *EcliPSe*'s tree-combining mechanism: only the combining function must be written; all communication is handled internally by *EcliPSe*.

The combining operator $\otimes$ must be designed so that its result has the same type as each of its operands. To satisfy this condition, the operands and the result are all defined to be the tuple t

$$(f, l, T)$$

where f is the first row of the operand, l is the last row, and T is the mapping table.

Applying the combining operator $\otimes$ over operands $t_1, t_2, \ldots t_k$ results in $t_r = t_1 \otimes t_2 \otimes \ldots \otimes t_k$, where $t_r = (f_r, l_r, t_r)$ is defined as:

$$\begin{aligned} f_r &= f_1 \\ l_r &= l_k \\ T_r &= (\bigcup_{i=1\ldots k} T_i) \cup (\bigcup_{i=1\ldots k-1} T(l_i, f_{i+1})) \end{aligned}$$

provided the inputs correspond to *contiguous segments*. The combined first row is the first row of the first segment (operand). The combined last row is the last row of the last operand. The combined mapping table is the union of all input mapping tables with all mapping tables resulting from the boundaries between each input operand. The mapping table is further discussed in section 4.2.

With the implementation of $\otimes$ as an *EcliPSe* combining function, it remains for each processor i the task of generating the tuple $(f_i, l_i, T_\emptyset)$: the first and last of its rows and an empty mapping table. Tuples are then automatically sent to the parent node in the virtual tree topology, where the combining function is applied. The whole procedure is repeated, and the monitor process finally receives tuple (f_1, l_n, T), with T being the global mapping table. The monitor must broadcast T to all processes (using the tree-diffusing mechanism), and each of them uses the global mapping table to re-label all its sites.

The implementation of the algorithm using *EcliPSe* presents an added benefit: the use of heterogeneous machines is allowed, since all information is transmitted over the network in a network-independent format.

Figure 4 shows an example of a cluster found by the parallel cluster labeling program in a 256 × 256 mesh using 4 processes.

4.2 Implementation of the Combining Operator

After initially building a mapping table (Figure 2), we must change it into a format that allows us to identify all clusters that are actually part of other clusters. The initial mapping table obtained from the boundary between adjacent processors a and b is nothing but a sequence of pairs (a_i, b_i), where a_i is the cluster number of site $M_a[0, i]$ (first row) and b_i is the cluster number of site $M_b[(N/n) - 1, i]$ (last row), provided both sites are "full". We may assume that $a_i > b_i$, since initially the cluster numbers assigned by each processor are disjoint.

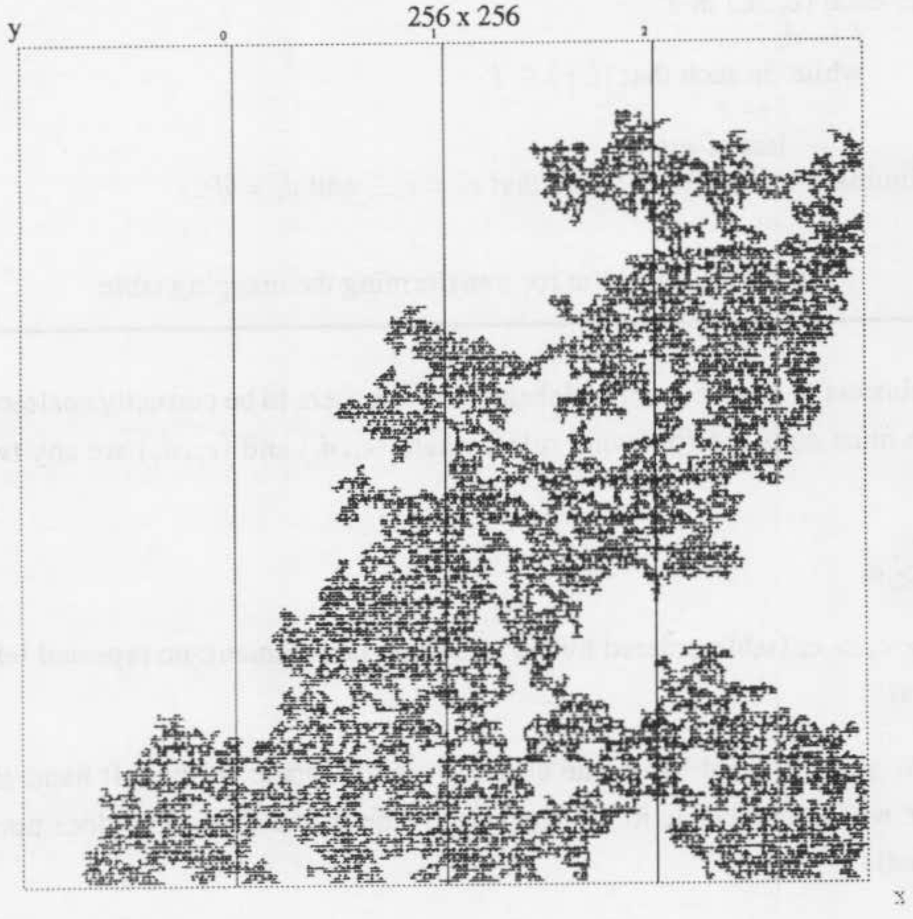


Figure 4: Example of cluster found by the program

```
table_transform(table $T$)        $T$ has entries of type $(c_i, d_i)$
{
    Sort $T$ by increasing $c_i$
    For each $(c_i, d_i)$ in $T$
        if $c_i = c_{i-1}$
            $(c_i, d_i) \leftarrow (\max(d_{i-1}, d_i), \min(d_{i-1}, d_i))$
            if $c_i \neq d_i$
                re-insert($(c_i, d_i)$, $T$)        (keep $T$ sorted)
            else
                no action    (originally $(c_i, d_i) = (c_{i-1}, d_{i-1})$; this entry will be
                 later eliminated)
    For each $(c_i, d_i)$ in $T$
        $l \leftarrow d_i$
        while $\exists r$ such that $(l, r) \in T$
            $l \leftarrow r$
        $d_i \leftarrow$ last of such $r$
    Eliminate entries $(c_i, d_i)$ such that $c_i = c_{i-1}$ and $d_i = d_{i-1}$
}
```

Figure 5: Algorithm for transforming the mapping table

To allow clusters to be effectively relabeled, forcing them to be correctly coalesced, the final mapping table must obey the following rules, where (c_i, d_i) and (c_j, d_j) are any two elements of the table:

1. $\forall i: \ c_i > d_i$
2. $j > i \Rightarrow c_j > c_i$ (table ordered by the left-hand-side element; no repeated left-hand-side elements)
3. $\forall i, j : \ d_i \neq c_j$ (a right-hand-side element cannot appear in the left hand side; that is, a cluster whose number is in the right hand side of the final table does not need to be re-labeled)

In the final mapping table, it is possible that for some i we have c_i and d_j belonging to the same processor, meaning that the final table identifies clusters that one processor previously computed as different clusters as actually being parts of the same cluster (such as clusters i_1 and i_2 in Figure 3).

The algorithm used to transform the initial mapping table into the final mapping table is outlined in Figure 5. The algorithm's worst case execution time is $\mathcal{O}(|T|^2)$.

We may combine mapping tables by using essentially the same algorithm used to transform an initial mapping table obtained from two boundaries. The tables to be combined only need to be concatenated, with the algorithm described in Figure 5 applied to the resulting table.

Besides allowing easy detection of repeated entries, having the mapping table sorted also helps the last phase of the parallel cluster labeling, where the final table is broadcast to each processor and re-labeling takes place. Upon receiving the final table composed of pairs (c_i, d_i), each processor determines elements f and l in the table such that the numbers $c_f, c_{f+1}, \ldots, c_l$ are in the range of cluster numbers initially assigned to this processor. Then the final re-labeling is done by traversing the whole mesh and replacing cluster number x of $M[i, j]$ by d_k if $x = c_k \in \{c_f, c_{f+1}, \ldots, c_l\}$. The search in $\{c_f, c_{f+1}, \ldots, c_l\}$ is performed using binary search.

5 Experiments

The parallel version of the cluster labeling algorithm was implemented with the help of the *EcliPSe* system. We report the results of initial experiments conducted on a network of SUN SparcStations 5. Up to 33 machines were used in the experiments, each of them having 32Mb of main memory and 120Mb of "swap" space. The communication time (as measured by the UNIX `traceroute` command working with 40-byte packets) between the machines is 3 ms. The machines were lightly loaded during the experiments.

In the first experiment, we measured execution time as the number of machines used were increased. In the second experiment, we evaluated the influence of parameter p (probability of each site in the mesh being "full") on the program's execution time. For both experiments we used a 4096 × 4096 mesh.

5.1 Scalability

The program's execution time was measured as we increased the number of machines used. Throughout this experiment the value of p was fixed at 0.5. Figure 6 shows the result. The number reported on the x abscissa refers to the number of computational processes used; an additional process was used as the monitor. To help us get a feeling for the program's performance, execution time was measured at two points in the program: after the monitor receives and combines the mapping tables ("collecting mapping tables") and after all processes have received the final table and re-labeled its sites ("complete").

It was not possible to obtain the execution time for only one computational process because of the prohibitively large memory space required by the program (this actually highlights one important motivation for parallelizing the algorithm).

For 2 and 4 processes, the execution time suffers because of the lack of physical memory to label the mesh. Measurements made at particular points of the program indicated that more than 70% of the execution time was being spent paging in or out. For 8 and 16 processes, memory is no longer a serious problem, because of the smaller sub-mesh size assigned to each process. For 32 processes, however, communication starts playing a bigger role: not only does the computation-to-communication ratio decrease, but the increased number of process boundaries forces a growth in the number of elements of the mapping tables. We believe, however, that

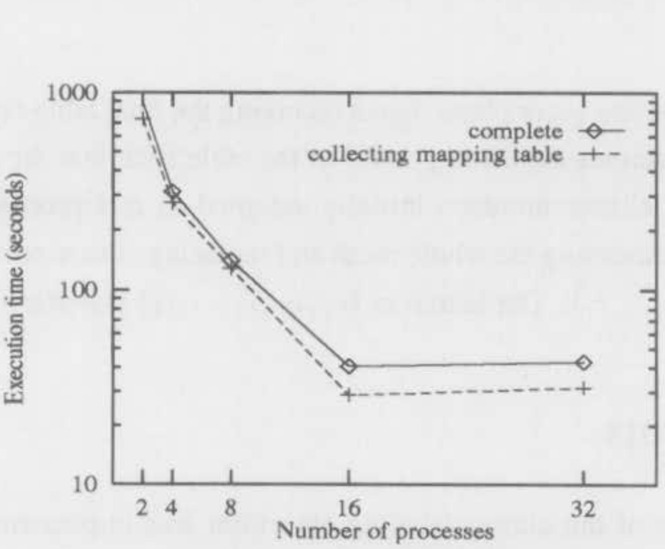


Figure 6: Execution times as we increase the number of processes

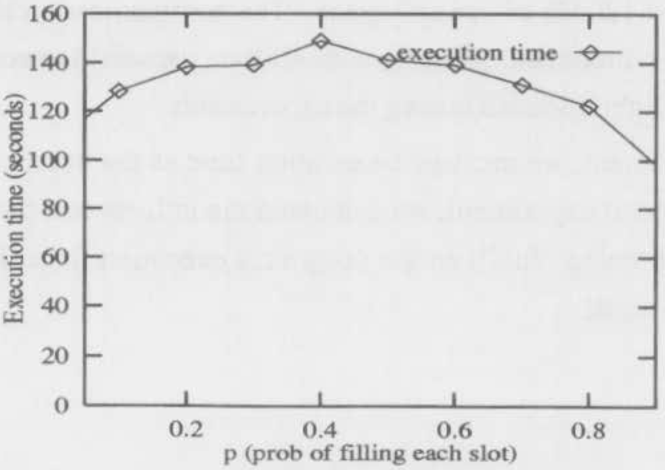


Figure 7: Execution time as a function of p

increasing the size of the problem might still result in better speedup for the 32-process case. Also, as we use a larger number of processes, tree-combining becomes crucial, and the tree topology used may create a significant impact on execution time.

The lackluster performance of the 32-process case highlights one problem in using a network of workstations as a computational engine: communication is several orders of magnitude slower than computation. A significant amount of effort is required in some applications to be able to extract reasonable speedup in a cluster computing environment.

5.2 Sensitivity to variation in p

In this experiment, we kept the number of computational processes fixed at 8 and varied the probability p of each site in the mesh being "full". Figure 7 shows the results.

For small p, the mesh is sparsely populated with only a few clusters. Also the mapping tables tend to be small, since the number of elements in an initial mapping table is in average p^2N (N is the mesh's dimension), if we ignore the number of repeated entries. As p increases,

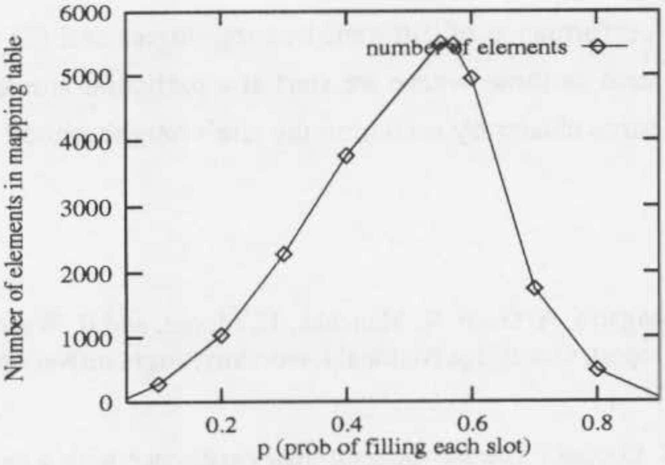


Figure 8: Number of elements of the final mapping table as we vary p

the number of clusters increases, as does the number of elements in a mapping table. This latter number, besides making combining more expensive, causes the last part of the algorithm (where each strip is re-labeled based on the final mapping table) to take longer.

When p reaches 0.6, however, the clusters tend to coalesce and contain a much larger number of elements, and the number of distinct clusters falls drastically. Moreover, the mapping tables start having a large number of repeated entries, which prompted us to eliminate repeated entries also in the beginning of the algorithm of Figure 5. The result is that the execution time actually *decreases* as p increases past 0.6.

Figure 8 shows the number of elements in the final mapping table as p varies. This number is bound to change if we alter the number of processes, mesh size, or simply the random number seeds.

6 Conclusions

Today's typical cluster computer environment is composed of fast workstations (such as the SUN 5s used in the experiments reported in this paper) interconnected with relatively slow networks (ethernet for our case). Although it is reasonable to expect networks to be faster in the future, workstation speeds keep improving even more rapidly. This means that parallel algorithms must be often modified for this environment, even if at an expense in terms of computation.

In this paper, we presented an algorithm for parallel cluster labeling and its implementation on a network of workstations. The algorithm minimizes the amount of data exchanged between processes by making them transmit only the information needed for the mapping table. By using *EcliPSe*'s tree-combining mechanism, we were able to parallelize combination of mapping tables. The use of *EcliPSe* has also alleviated the programming effort required, easing the task of exchanging data between processes.

Among the issues related to parallel cluster labeling that we intend to investigate, we mention

(1) evaluation of the performance of different tree topologies and (2) use of different cluster labeling approaches, such as those where we start at a particular site and then attempt to find other elements of the same cluster by searching the site's neighborhood.

References

[1] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, K. Moore, and R. Wade. HeNCE: a user's guide (draft). Technical report, Oak Ridge National Laboratory, Argonne National Laboratory, November 1991.

[2] K. Birman and R. Cooper. The Isis project: real experience with a fault tolerant programming system. *Operating Systems Review*, pages 103–107, 1991.

[3] H. Gould and J. Tobochnik. *An introduction to computer simulation methods – applications to physical systems*. Addison Wesley, 1988. part 2.

[4] B. Gropp and E. Lusk. A test implementation of the MPI draft message-passing standard. Technical report, Mathematics and Computer Science Division, Argonne National Laboratory, 1992.

[5] P. Heidelberger. Discrete event simulations and parallel processing: statistical properties. *SIAM Journal on Scientific and Statistical Computing*, 9(6):1114–1132, November 1988.

[6] J. Hoshen and R. Kopelman. Percolation and cluster distribution. Phys. Rev., B14(3438), 1976.

[7] F. Knop, E. Mascarenhas, V. Rego, and V. Sunderam. An introduction to fault tolerant parallel simulation with EcliPSe. In *Winter Simulation Conference*, pages 700–707, December 1994.

[8] F. Knop, V. Rego, and V. Sunderam. *EcliPSe*: A system for fault-tolerant replicative computations. In *Proceedings of the IEEE/USP International Symposium on High-Performance Computing*, March 1994.

[9] H. Nakanishi, V. Rego, and V. Sunderam. On the effectiveness of superconcurrent computations on heterogeneous networks. *Journal of Parallel and Distributed Computing*, 24:177–190, 1995.

[10] V. J. Rego and V. S. Sunderam. Experiments in Concurrent Stochastic Simulation: The EcliPSe Paradigm. *Journal of Parallel and Distributed Computing*, 14(1):66–84, January 1992.

[11] M.D. Rintoul, J. Moon, and H. Nakanishi. Statistics of self-avoiding walks on randomly diluted lattice. *(to appear)* Phys. Rev. E, July 1994.

[12] V. S. Sunderam and V. J. Rego. *EcliPSe*: A system for High Performance Concurrent Simulation. *Software-Practice and Experience*, 21(11):1189–1219, 1991.

[13] V.S. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4), December 1990.

[14] L.H. Turcotte. A survey of software environments for exploiting networked computing resources. Technical report, Engineering Research Center for Computational Field Simulation, Mississippi State University, June 1993.