Dependable Computing in Virtual Laboratories

G. Alonso, W. Bausch, C. Pautasso, M. Hallett, A. Kahn
Dept. of Computer Science, Swiss Federal Institute of Technology (ETH)
ETH Zentrum, CH-8092 Zürich, Switzerland
{alonso,bausch,pautasso,hallett,kahn}@inf.ethz.ch

August 4, 2000

Abstract

Many scientific disciplines (e.g., biology, astrophysics, particle physics, earth sciences) are shifting from in vitro to in silico research as more physical processes and natural phenomena are simulated in a computer (in silico) instead of being observed (in vitro). In many of these virtual laboratories, the computations involved are very complex and long lived. Currently, users are required to manually handle almost all aspects of such computations, including their dependability. Not surprisingly, this is a major bottleneck and a significant source of inefficiencies. To address this issue, we have developed BioOpera, an extensible process support management system for virtual laboratories. In this paper, we briefly discuss the architecture and functionality of BioOpera and show how it can be used to efficiently manage long lived complex computations.

1 Introduction

In a virtual laboratory, science is made based on electronically stored data instead of on direct observations of natural phenomena. Such virtual laboratories are becoming increasingly pervasive now that the cost of storing observations is lower than the cost of making them [E00]. These environments are typically associated with massive amounts of data [ILGP96, WZS95, ZI94]. However, initial experiences have shown that data processing can be a more critical issue than data storage [BSG00, SKT00].

To get a better idea of the problem, consider the following example. One of the main goals of the BioOpera project at ETH-Zürich is to be able to build a software system capable of automatically predict the secondary structure of a protein given the recipe encoded in its DNA. This could be achieved, for instance, by starting with the raw DNA and applying a set of computational transformations to construct the tower of information depicted in Figure 1. This tower of information represents a sequence of derived data sets, each one of them with a higher information content than its predecessors. As a first step, genes are located in the raw DNA and these are translated into protein sequences. The proteins are then aligned (i.e., compared) with known proteins and various statistics are calculated. These intermediate results, in turn, are used to build a phylogenetic (or evolutionary) tree. The trees together with a multiple sequence alignment provide the biochemist with a historical perspective of which evolutionary events occurred in the genomes of the organisms over time. This information is important to produce an accurate secondary structure prediction. From here, the shape the protein is predicted and, from this shape, one may eventually deduce the function of the protein.

The problem with automating such a computation is that each step represents a non-trivial amount of computation (most of the steps are \(NP\)-complete and algorithms are yet to be developed for some of them). In addition, the algorithms and basic data sets used in each step are continuously evolving. Computing the whole tower of information is very expensive and may take several months.
when using a medium size cluster of computers (between 50 and 100 nodes). Thus, it makes sense to keep the results of each step so that it is not necessary to start from the beginning every time an algorithm changes. This requires one to keep track of which steps produced which data. It is also necessary to keep track of all the data interdependencies created during the analysis. Moreover, computations along the tower of information sometimes need to be disambiguated or corrected by changing appropriate input parameters. This forces the system to continually recompute the data with slightly different parameters in order to make sure that acceptable results are eventually produced. As a result, a virtual laboratory where such a computation runs requires a substantial degree of dependability. Otherwise, the cost and overhead of keeping the system running becomes prohibitive.

In most existing virtual laboratories, storing, manipulating, and keeping track of the computation is done manually through ad-hoc pieces of code. The data processing logic is typically written using conventional programming languages (e.g., C, Fortran) for the basic algorithms and collections of operating system scripts (mainly Perl scripts) as the glue between the different components. Such an approach leads to logic that is extremely difficult to modify and rather primitive, unsystematic methods for driving and monitoring computations. When considering that in some studies the tower of information will be built several thousand times, it becomes clear that better software support is needed and that an organized way to store and manage information about the entire procedure is critical to the success of the virtual experiment. Such support should include automated mechanisms to keep track of every step of the computation, record every dependency between algorithms and among data sets, checkpoint the computation at appropriate times in addition to the necessary procedures to automatically recover from failures and resume the computation after recovering from failures. Endeavors like the tower of information are only feasible with this functionality in place. In this paper we will concentrate on how to provide this basic functionality. The core of our efforts is BioOpera, a process support system for virtual laboratories in bioinformatics. BioOpera is based on Opera, a workflow-like middleware tool [AHST97] that has evolved into a programming [HA99] and runtime environment for cluster computing [H99] with the capability to define, execute, monitor and manage a broad range of large-scale, complex scientific computations. For reasons of space, we cannot elaborate on all aspects of BioOpera. In what follows, we briefly describe the architecture of BioOpera and show how it can be used to increase the dependability of virtual laboratories.

The paper is organized as follows. Section 2 presents a typical virtual experiment. Section 3 briefly describes BioOpera. Section 4 shows how the virtual experiment can be represented in BioOpera notation. Section 5 discusses the experimental results. Section 6 concludes the paper.
2 Motivation and Problem Statement

For the purposes of this paper, we will focus on the problem of sequence alignment, a cornerstone of the tower of information. This problem lies at the heart of comparative genomics and its results are used for a broad range of purposes [Ch98, CHNZ00, GCB92, SBM99]. The datasets involved (e.g., EMBL [S99] or Swiss-Prot [BA99]) tend to be large. For instance, Swiss-Prot vers. 38 (SP38) contains 80,000 amino acid sequences. Aligning every entry in SP38 against all other entries in this dataset – a self-comparison or all vs. all – requires approximately $3.2 \cdot 10^9$ individual pairwise alignments (certain optimizations can be used to reduce this figure). As an indication of what this implies, over the past 7 years, the Computational Biochemistry Research Group of ETH Zürich has updated (and made public) the all vs. all comparison of Swiss-Prot vers. 27 [GCB92]. Current updates typically involve at most 10,000 new sequences and require 3 to 4 months of computation on a cluster of 16 dual processor nodes. During such computations, the datasets, software, and workstations involved need to be painstakingly maintained: nodes fail, algorithms fail, data entries need to be discarded, jobs restarted, results coalesced, and so on. Today, this is done manually due to the lack of appropriate tools. Future experiments such as the tower of information are orders of magnitude more complex than the all vs. all. Clearly, adequate tools are needed for these efforts to be viable.

The most basic aspect of such tools is to be able to dependably run computations for months at a time with minimal user intervention. This requires to automatically and transparently handle issues such as efficient scheduling of jobs, load balancing, tracking of progress and results of the computation, recovery from system errors and machine crashes, access to intermediate results as they are computed, automatic accounting of statistics concerning computing time, and a systematic method for storing all necessary meta-data.

The first step towards providing this functionality involves finding an appropriate representation for the computation. We have chosen the notion of process, similar to that used in workflow management systems (although the final implementation is rather different since workflow tools are not entirely adequate for virtual laboratories [MVW96, BSR96]). A process is an annotated directed graph where the nodes represent tasks and the arcs represent the control/data flow between these tasks. Thus, the notion of process allows one to capture sequences of invocations of computer programs in a distributed and heterogeneous environment and the corresponding data exchanges between these programs. From here, the process can be encoded in such a way so as to allow its efficient storage in a database. Once in a database, this information is persistent, allowing us to both automatically manage the computation and increase its dependability. This is the approach we have taken in BioOpera.

It must be noted that there are several runtime environments and tools for high performance computing over clusters [Bu99] that address several of these issues. In fact, some of the ideas below have been inspired by some existing cluster and grid computing systems. However, these are low level tools, some of them being simply a set of libraries, that work at an entirely different level of granularity than BioOpera. To our knowledge, none of these tools provides the type of support for long-lived computations offered by BioOpera. In particular, the change in programming paradigm proposed by BioOpera (processes), and the support for persistently storing all relevant information about the computations are unique characteristics of BioOpera and, in our view, its main contributions.

3 BioOpera: a Virtual Laboratory for Bioinformatics

3.1 Process Design

A long lived computation in BioOpera is represented as a process. A process is a coarse-granule representation of the different steps of a computation. BioOpera uses a language called Opera
Canonical Representation (OCR) [H99] to describe processes. In OCR, a process consists of a set of tasks and a set of data objects. Tasks can be activities, blocks, or subprocesses, thereby allowing modular design and reuse (e.g., the tower information is built as a process where every step is a subprocess; for those steps where alternative algorithms are needed, the subprocess can be divided into further subprocesses). The data objects store the input and output data for the tasks and are used to pass information around a process.

As an example, Figure 3 depicts a simplified version of the all vs. all process as it is implemented within BioOpera. Activities (rectangles in Figure 3) are the basic execution steps. They correspond to stand alone programs or systems that can be relied upon to complete one of the computational steps of the process. Each activity has an external binding that specifies the program to be invoked (not shown in the figure). This information is used by the runtime system to launch external applications. Control flow inside a process is based on control connectors which, formally, are annotated arcs \((T_S, T_T, C_{Act})\), where \(T_S\) is the source task, \(T_T\) is the target task, and \(C_{Act}\) is an activation condition (bold connecting lines in Figure 3; the figure shows the activation condition for the control connector (!queue_file) between tasks user input and queue generation). Each activation condition (or activator) defines an execution order between two tasks and is capable of restricting the execution of its target task based on the state of data objects, thereby allowing conditional branching and parallel execution. Data flow between tasks or processes is also represented with the help of connectors (shown in Figure 3 as thin connecting lines). In the simplest form, a process consists of only activities, control and data flow dependencies. Each task has an input data structure storing its input parameters and an output data structure storing any return values (represented as cylinders in Figure 3). The input parameters of a task can be bound to data items in the global data area of the process (the whiteboard) or in output structures of other tasks. When a task starts, these bindings are analyzed and the necessary values are passed to the task. After the successful execution of a task, a mapping phase transfers data from its output structure to the global data area or to other tasks.

Larger processes are structured using one of two nesting constructs: blocks or subprocesses. A block is a named group of tasks. The scope of the block name is the process in which it is defined. Blocks are used for modular process design and to implement specialized parallel processing language constructs (Alignment in Figure 3 is a block). In particular, blocks are used to implement unconventional branching in the flow of control (e.g., executing an algorithm on every section of a grid, resulting in an activity being applied in parallel to every section of the grid). Subprocesses are processes which are used as components in other processes. A subprocess can be seen as a reference to a process inside of another process. Like blocks, they allow the hierarchical structuring of complex processes. Late binding - the subprocess is instantiated only when it is started - allows dynamic modification of a running process by offering the ability to change its subprocesses. Late binding is also used on the data objects to allow dynamic modification of the flow of data.

In addition to these primitives, OCR supports advanced programming constructs such as exception handling, event handling, and spheres of atomicity [HA98, HA99]. These constructs provide a reliable computing environment and make OCR a complete programming language. The combination of these features allows the process designer to define sophisticated failure handlers as part of the process [H99] (such as undo actions, alternative executions, and various forms of exception handling).

### 3.2 Architecture and Basic Functionality

BioOpera functions to a large extent like a high-level distributed operating system managing processes and the resources of a computer cluster. It has both a runtime component and a development environment associated to it (Figure 2).

The development environment allows users to specify BioOpera processes via a GUI that does not require programming expertise (most of the configuration information required can be automat-
Process (OCR: textual representation) internal programming language used in BioOpera to represent and manipulate processes

Process (graphical representation) internal programming language used in BioOpera to represent and manipulate processes

Process (graphical representation) internal programming language used in BioOpera to represent and manipulate processes

Process (persistent representation in DB) internal programming language used in BioOpera to represent and manipulate processes

Process definition

Process execution

Figure 2: Architecture of Opera

Additionally extracted from the system and used as default; eventually a library of processes and activities will be provided so that users can run predefined processes without having to define them themselves. This GUI is currently being developed and encompasses three elements: process creation, library management, and configuration management. The configuration management allows users to specify the hardware and software characteristics of the computer infrastructure to be used in the execution of a process (IP addresses, type of OS, CPU specifications, etc.). This information will be used by the runtime component to make placement decisions for jobs to be executed, for load balancing, and to deal with failed nodes. The library management element allows the definition of the runtime aspects of activities: program to be invoked, input, output, where it runs, how to pass arguments, and so on. The library management element has been designed to allow users with more computer knowledge to prepare pre-packaged activities for those users with less computer knowledge. The idea is to eventually form a library of such activities that can be distributed with BioOpera\(^1\). Finally, the process creation element will allow users to create processes by simply selecting activities from the library management element, combining them as individual activities or as part of blocks, and specifying the flow of control and data among them. Process creation will be graphical, with a compiler in charge of translating the graph into the proper OCR code.

All this information is stored in one of the corresponding data spaces in BioOpera: the template space contains process templates (processes as defined by the user), the instance space contains processes currently executing, the configuration space contains the information related to system configuration, and the data space contains historical information about all processes already executed, along with references to external datasets created by the latter.

During execution, a process instance is persistent both in terms of the data and the state of the execution. This allows BioOpera to resume execution of processes after failures occur without losing already completed work. The fact that the process state is persistently stored in a database

\(^1\)In bioinformatics, there is a set of tools that are widely accepted and used by many researchers. This set of common tools will be incorporated as pre-packaged activities within BioOpera.
also offers significant advantages for monitoring and querying purposes. From the instance space, process execution is controlled by the *navigator*. In this sense, OCR acts as a persistent scripting language interpreted by the navigator. Once the navigator decides which step(s) to execute next, the information is passed to the *dispatcher* which, in turn, schedules the task and associates it with a processing node in the cluster and a particular application. If the choice of assignment is not unique, the node is determined by the scheduling and load balancing policy in use. The dispatcher then contacts the *program execution client* (PEC); this is a small software component present at each node responsible for running application programs on behalf of the BioOpera server. The PEC is responsible for invoking the application program that will complete the computational step. It is written in Java and, thus, it is platform independent, allowing BioOpera to work with heterogeneous nodes. This client also performs additional activities like monitoring the load at the node and reporting failures to the BioOpera server. When applications complete a task, results are returned via the PEC to the activity queue at the server. The recovery module reads this data and updates the database so as to keep track of all events that have occurred. Afterwards, the navigator is then given control and looks for the next activities to execute. This procedure is similar to that used in workflow management systems except for blocks, where the navigator may execute complex application logic to decide how many activities need to be executed in parallel and to determine the input for each of them.

Interaction with external applications takes place through specific interfaces or *wrappers*. For computational purposes, we are currently working exclusively with one software tool, Darwin [GHKB00], but any other such system could be used.

### 3.3 Parallelism

A crucial aspect of BioOpera is the support for applications typical of high performance computing environments. Although the process-based programming paradigm is quite different from conventional message passing or shared memory approaches, it is still possible to support parallelism as part of OCR. We are currently developing several block constructs adapted to concrete parallelization approaches. In this we benefit from the architecture of BioOpera and the structure of OCR, both being easily extensible [H99].

For reasons of space, we concentrate here only on the techniques used in the *all vs. all*. In this case, we use a special block construct called a *parallel task* that takes as input list of data objects and produces as output another list of data objects. A parallel task has a single activity (or a subprocess) associated to it. Once the flow of control reaches the parallel task, an instance of the activity will be created for each element in the input array and all these instances can be executed in parallel. The parallel task concludes when all the instances have concluded. This approach has the advantage that the degree of parallelism can be determined at runtime by producing a longer or shorter list as input (this list can be produced by another activity).

More formally, the input data $E$, consisting of a set of $N$ elements, is partitioned into $n$ subsets. Let $\mathcal{P} = \{ P_1, \ldots, P_n \}$ denote this partition. The task which is responsible for computing the entire set $E$ is a *parallel task*. This parallel task is subdivided into the $n$ tasks defined by $\mathcal{P}$. We call one such task a *task execution unit* (TEU). Given this setting, the crucial issue is to decide how large each $P_i$ should be (the *level of granularity*). If the granularity is too fine ($n$ is large), system will incur excessive overhead. In the *all vs. all* process, for example, we could partition at the level of the individual pairwise alignment, which results in $O(N^2)$ activities. Each such alignment requires a fraction of a second to compute but a few seconds to schedule, distribute, initiate, and merge. Clearly, such an approach is wasteful. A slightly better but still not optimal approach would dictate that we partition at the level of the *one vs. all*, of which there are $N$ in total. Now, each TEU would be responsible for computing the alignment between one sequence and every other sequence in SP38.

To choose between these different alternatives, we performed several experiments that have
allowed us to optimize the partitioning strategy. In addition to optimizing for performance purposes, the granularity should be chosen fine enough to enable BioOpera to effectively schedule and balance jobs. In particular, since checkpointing is done for complete activities, smaller activities result in less work lost when failures occur. Since we intend to reuse this process many times, it pays off to invest time in optimizing it as much as possible. This is likely to be true of most processes designed in BioOpera. In this particular case, our tests suggest each partition should contain significantly more than a single one vs. all (see Section 5.3).

3.4 Monitoring, Scheduling and Load Sharing

Monitoring is an important aspect in long lived computations as it allows to keep track of the state of the computation and influence its outcome, if necessary. Most information pertaining to a process and its execution environment is stored persistently by BioOpera. Beyond task start times, task finish times and task failures, the system also stores information regarding the load in each node, node availability, node failure, node capacity, and other relevant information regarding the state of the computing environment. All together, this information allows the creation of an awareness model [BGS+99] which, in turn, allows BioOpera to react to changes in the computing environment and provides a very complete view of the computation. This information is then used to share the load between the different nodes, to schedule the computation according to machine usage and availability, to resume the execution of the computation smoothly when failures occur, and to avoid inconsistencies in the output data after failures.

For instance, BioOpera examines the workload of the available machines using an adaptive monitoring technique. At the heart of this technique lies the idea that processors which display a constant workload over a long period of time do not have to be monitored as closely as processors having a variable workload. First, the local program execution client (PEC) compares the last recorded load with the current load at that node. If the change falls below some predetermined cut-off level, the interval before the next sampling is increased. Otherwise, the interval is decreased. Second, the PEC notifies the BioOpera server of changes in load only if the amount of change has increased/decreased beyond a second predetermined cut-off level. Experiments have shown that this scheme helps to considerably reduce the sampling and network overheads while preserving a highly accurate view of the load in the cluster. Our simulations have shown that an adaptive strategy discarding 70% of the samples before they are sent to the BioOpera server, induces an average 0.1 error per sample when we compare the load curve as seen by the server to the actual load curve.

In addition, the monitor allows users to actively influence the computation as the user can start, stop, abort, re-start, and change input parameters during each step of the computation. Once BioOpera is linked to the appropriate visualization tools (which are process-specific and can be incorporated as user triggered activities), users will be able to check intermediate results and change or eliminate them if necessary. This is a very useful feature as there are many data sets for which the algorithms available do not produce a reasonable output.

3.5 Planning and Dealing with Outages

BioOpera has been designed as a tool to help managing long-lived computations. This implies that, in addition to the functionality discussed, it also needs to provide support for planning ahead. This is a key feature when running virtual experiments that may last months and, therefore, are likely to encounter many different situations (in addition to failures): the need to upgrade software and hardware, the replacement of nodes, changes in storage devices, and so forth.

In this regard, BioOpera exploits the functionality discussed above beyond any existing system. Since the computation is outlined in the process and its status is well known at all times, it is possible to determine what would happen when a given node is taken off-line. This gives system
administrators a very powerful tool to perform upgrades and changes to the system as the computation proceeds while minimizing the impact of the outages. Thanks to the dynamic scheduling and load balancing mechanisms, BioOpera is capable of working with a system that shrinks or grows in size dynamically. It is even possible (thanks to the late binding approach) to replace activities initially intended to run in a given node with alternative activities running on a different node (even with a different OS). In our experiments, BioOpera successfully coped with failures in the entire cluster, with complete network outages, with hardware upgrades of all nodes in the cluster (from one to two processors), and with user driven interruptions of the computation (to let other users utilize the cluster). In all cases, the computation was successfully resumed with minimal or no human intervention. The fact that all the necessary information is well organized and stored in a database opens up the opportunity to create very sophisticated automatic tools for system and computation administration that go well beyond anything available today. For instance, if many processes are being run, a system administrator could ask the system which processes will be affected if a node or set of nodes is taken off-line. BioOpera will then use the configuration information and the process structure to determine whether alternatives exist and will then reschedule the processes accordingly, notifying the administrator of the processes that will stop, how far in their execution these processes are, their priority (if any), and so forth.

It needs to be emphasized that this type of functionality will play a crucial role in virtual laboratories where increasingly complex concurrent computations will be run over increasingly large clusters. We believe the approach pioneered by BioOpera, process based programming with a database to store the necessary meta-data, is the only way to cope with such environments.

4 All vs. All in BioOpera

In order to better understand the experiments carried out and to illustrate the use of processes in BioOpera, in this section we present the all vs. all process (Figure 3). Recall that an all vs. all is a self-comparison of all entries in a certain dataset. The result of the computation will be the set of all sequence pairs whose similarity scores reach a user-defined threshold, along with some information about the characteristics of the pairs. In the following, we will call such a sequence pair a match. The exact details of how each activity is computed and the data-structures used are beyond the scope of this paper. Suffice it to say, we use the Darwin system [GHKB00] as our bioinformatics application. This software offers a dynamic programming local alignment algorithm which uses the GCB scoring matrices and an affine gap penalty [GCB92, SW81]. Except for the two initial tasks in the process, all other tasks are executed as Darwin programs. That is, when an task needs to be executed, BioOpera contacts Darwin at the appropriate machine and instructs it to execute a particular algorithm on a particular set of inputs. The tasks in the all vs. all process depicted in Figure 3 are as follows.

Task “User Input” queries the user for the input parameters to the all vs. all process. These parameters consist of a dataset, a so-called queue file and the location where results should be stored. Here our dataset is SwissProt v38 and our queue file contains the list of entry indexes $E \subseteq [1, \ldots, N]$ into the dataset where $N = 8 \cdot 10^4$ for SP38. The purpose of the queue file is twofold. First, the indexing provided by the queue file allows BioOpera to discard ill-behaving sequences and smoothly restart computation when failures occur, since only the dataset entries listed in the queue file take part in the comparison. Second, it is used by the two succeeding tasks to control the degree of parallelism during execution. The queue file is an optional input parameter, its absence or presence determines which of the two possible successor tasks will be executed after this task finished.

Task “Queue Generation” produces a queue file consisting of the complete list of entries in SP38 $E = [1, \ldots, N]$, if no queue file is provided by the user.

Task “Preprocessing” is responsible for preparing the data for parallel execution by creating a partition $\mathcal{P} = \{P_1, \ldots, P_n\}$ of the entries $E$ in the queue file. This will be the input to the next
Figure 3: The all vs. all process as implemented in BioOpera

Task, which is a parallel task.

Block “Alignment” is a parallel task where the internal activity corresponds to a subprocess. When started, \( n \) of these subprocesses will be run, where subprocess \( i \) computes the alignment for each entry in \( P_i \) against SP38.\(^2\) First, a fixed PAM alignment performs a pairwise alignment using a fast but inaccurate algorithm, the set of matches found by the latter being \( Q_i \). Second, every match in \( Q_i \) is refined in task PAM-param refinement by recalculating the corresponding alignment using computationally more expensive but more informative algorithm. We call the resulting set of matches \( R_i \). When all \( n \) subprocesses running within the parallel task finish, all datasets \( R_i \) are assembled into \( \mathcal{R} = \{R_1, \ldots, R_n\} \) which forms the result of the parallel task.

Task “Merge by Entry #” merges the set of files \( \mathcal{R} \), output of the Alignment Block, into one master file. The contents of this file are sorted according to the entry number in the original database.

Task “Merge by PAM Distance” sorts the matches in \( \mathcal{R} \) into various files according to PAM distance estimations.

5 Experimental Results

We have performed three different experiments. The first aimed at finding the optimal granularity level for parallelization. It consisted of several versions of the all vs. all process using a smaller database (500 entries) and different granularity levels. The other two experiments were the computation of the all vs. all using SP38. One was run on a cluster shared with other users, and the other was performed on a cluster exclusively managed by BioOpera. The objectives were to mea-
sure the effectiveness of BioOpera at managing large-scale computations and to test the system's long term stability, as well as its ability to cope with changes in the hardware configuration and various failures. The failures observed were not injected but part of the everyday operation of the systems.

### 5.1 Hardware Environment

The experiments were performed using a combination of PCs and UNIX workstations linked by an ordinary Ethernet 100Mbit network. The main cluster is comprised of 16 two-processor PCs (500 Mhz, 512 MB main memory) running Red Hat Linux v2.2.12 and 1 Sun SparcStation with 6 CPUs (336 Mhz, 3072 MB main memory) running Solaris. We refer to this cluster as the *linneus cluster*. Another cluster is a set of 5 Sun Ultra 5 (269 MHz, 192 MB main memory) running Solaris. We refer to this cluster as the *ik-sun cluster*. Finally, the *ik-linux cluster* is a group of 8 two-processor PCs (600 Mhz, 512 MB main memory) running Red Hat Linux v2.2.14.

### 5.2 Measurements

To evaluate the results of the experiments, several criteria were used. First, for each activity $A_i$, we measured the time it took to complete by looking at how long it was active on a given CPU (CPU time: $CPU(A_i)$). Let $A_1, A_2, \ldots$ denote each activity executed during the process and let $\Omega = \{A_1, A_2, \ldots\}$ denote the entire process. The CPU time of a process is the CPU time it took to execute all of its activities.

$$CPU(\Omega) = \sum_{A_i \in \Omega} CPU(A_i)$$

The WALL time measures the absolute time it takes for the process to complete as the difference between its starting and finishing times. The WALL time depends heavily on the amount of parallelism achieved. In addition, this measurement provides a good indicator of the effectiveness of BioOpera as a basic tool for virtual laboratories.

Finally, the relation between the CPU time with respect to the number of activities in the process ($|\Omega|$) gives a rough approximation of the time needed per activity and provides an intuition about the average recovery time.

$$CPU(A) = \frac{CPU(\Omega)}{|\Omega|}$$

We also measured BioOpera's ability to recover from failures and the time of manual interventions required to keep the computation going.

### 5.3 Determining the Optimal Granularity Level

As a first step towards determining optimal granularity levels for later use during the full all vs. all computation, we carefully analyzed the performance (i.e. CPU and WALL times) when allowing the number of task execution units (TEUs) to vary. As alluded to previously, our dataset consisted of 500 entries from SP38. These experiments were run on the *ik-sun* cluster in an exclusive mode, i.e. no other users or jobs were allowed to use the machines. The number of TEUs for these experiments varied between $n = 1$ (no parallelization), and $n = 500$ (each of the 500 one vs. all is parallelized). Figure 4 shows our results.

These results indicate several key points. Not surprisingly, the 1 TEU scenario gives the best CPU time but one of the worst WALL times, due to the lack of parallelism. As the chart shows from left to right, the CPU time increases, while the WALL time first decreases and then, with more than 20 TEUs, again increases. At the other extreme, the number of TEUs being 500, the CPU time has almost doubled. This is due to the overhead incurred from Darwin initialization.
Figure 4: Impact of the granularity level (# of TEUs) on CPU and WALL times (seconds) for the 500 vs. 500 on the *ik-sun* cluster.

stages, which are repeated 500 times. The increased WALL time also reflects the extra overhead due to BioOpera scheduling and executing an increased number of activities.

Our results indicate that the optimal choice for the granularity is 20 TEUs. This is somewhat counter-intuitive since one might be tempted to conclude that the optimal would coincide with the number of available CPUs, which is in this case 5. To explain this, we split the chart in Figure 4 into three segments $S_1 = [1, 5]$, $S_2 = [5, 20]$, and $S_3 = [20, 500]$. The explanation for the downward curve for WALL time in $S_1$ is straightforward: as more TEUs are added, more parallelism can be achieved. The CPU time increases only slightly since the difference in overhead between 1 and 5 TEUs is marginal. For $S_3$, the explanation is also straightforward. When the granularity becomes exceedingly fine, the number of alignments per TEU becomes exceedingly small. Therefore, the overhead both from BioOpera and Darwin significantly increases the the overall CPU and WALL times. The explanation for $S_2$ is somewhat more difficult. One would expect that optimal granularity level to be 5, to coincide with the number of processors, and not our observed 20. Clearly, the overhead of starting and stopping Darwin should not be the dominant factor in this discrepancy. The explanation for the observed behavior lies in a well-known scheduling phenomenon. Since TEUs may differ in size slightly and since the CPU time for TEUs will always differ, tasks which require all previous tasks to complete (e.g. the final merging task in our *all vs. all* process) will not be executed until this “longest” TEU is completed. Hence, the WALL time will be significantly affected. If the granularity is too coarse, this phenomenon can become quite large.

A granularity level of 20 implies that each *TEU* performs approximately 5% (6,250) of the total number of individual pairwise alignments, ($\binom{500}{2}$). If we extrapolate these results to the full *all vs. all* of SP38, we would use a granularity level of 3200. However, since the dataset is larger (containing 80,000 entries), the initialization cost per TEU is also much larger. Hence, we set our level of granularity to 512, a multiple of the number of processors available. This figure lies in the equivalent $S_2$ segment for an *all vs. all* with 80,000 entries and, thus, should be close to the optimal.

### 5.4 The *All vs. All* (shared cluster)

In the first run of the *all vs. all* experiment, we tried to test the ability of BioOpera to cope with the everyday changes that take place on a shared cluster. We used the *ik-sun* (only two nodes) and *linneus* clusters, with some machines dedicated to certain tasks. In particular, the slower *ik-sun* cluster was responsible for the refinement stages. As already stated above, both clusters shared a
storage device in the first part of the experiment. Due to problems with this device, we had to switch to a storage device accessible from the Linneus cluster only, thus implying that the ik-sum machines were disabled towards the end of the computation. All activities were run with lowest priority. The computation lasted from the 17th December 1999 until the 25th of January 2000. Note that the final two days of computation are not represented in Figure 5.

![Table 1](image)

<table>
<thead>
<tr>
<th></th>
<th>Shared cluster</th>
<th>Non-shared cluster</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max. # of CPUs</td>
<td>40</td>
<td>16</td>
</tr>
<tr>
<td>CPU(Ω)</td>
<td>441d 7h 11m</td>
<td>400d 2h 36m</td>
</tr>
<tr>
<td>WALL(Ω)</td>
<td>38d 21h 30m</td>
<td>49d 20h 46m</td>
</tr>
<tr>
<td>CPU(A)</td>
<td>0d 20h 41m</td>
<td>0d 18h 39m</td>
</tr>
</tbody>
</table>

Table 1: Performance of the all vs. all on SP38 for the two experiments

The overall performance figures are shown in the shared cluster column of Table 1. From the point of view of a virtual laboratory, the most relevant conclusion from these results is that the entire process required only 38 days (WALL time), using up to 40 processors in the first run, and 49 days using up to 16 processors in the second run (see details in Section 5.5). Previous manual efforts required significantly more time (on the order of months) and computed significantly less (mere updates of earlier version of SwissProt). This already proves the benefits of using a system like BioOpera. In addition, the fact that the process has been made explicit makes it very easy to reuse and modify as needed.

Regarding the ability of BioOpera to automate the overall procedure, the history of this first computation is summarized in Figure 5. Since the cluster was shared with other users, BioOpera jobs were run in nice mode, giving priority to the other users (who, at some times, utilized the cluster very heavily). Figure 5 contains a number of event indicators that will be used to refer to particular phases of the execution. The flat line (dark area) indicates how many processors were actually available at each point in time and ranges between 0 and 40. This variation is due to network failures, system maintenance, and software upgrades. The rugged line (light area) indicates the number of processors that were actually computing BioOpera jobs.

As indicated by the figure, the actual computing time is a small fraction of the total WALL time. This is due to either heavy utilization of the cluster by other users (events 1 and 9), or because of problems during the execution (event 5, where the process ran out of storage space). We believe these results accurately reflect what happens in a typical shared computational environment. We also stress that a major goal of the experiment was to test the ability of BioOpera to sustain the computation for a long period of time in spite of problems and require little manual attention. In this, BioOpera was quite successful. In this regard, the major events observed can be grouped into several categories.

In a first group are those events where the process was manually stopped (events 1, 5 and 6). Event 1 occurred when another user requested exclusive access to the cluster. We manually suspended the execution of the process (letting ongoing jobs finish but not starting new ones) and resumed once the cluster was freed. This cannot be considered a failure but it is a typical operation in a shared environment and allowed us to test the ability of BioOpera to suspend and resume executions smoothly. In event 5, the process ran out of disk space. This could have been noted and corrected earlier but nobody was monitoring the system at the time. Once this was realized, the process was manually stopped to solve the storage problem and resumed once it was done (event 6).

A second group of events are the real system failures. Event 2 is the sole occurrence of a BioOpera server crash. This was due to a communication protocol problem that has since been corrected. When the BioOpera server fails, ongoing processes are stopped. When the server recovers, they are automatically resumed as it happened here. Events 3 and 4 were massive hardware
failures in the cluster. The BioOpera server correctly classified all active jobs as failed and resumed execution once the problem was corrected. In this run, some manual intervention was necessary to restart the clients. This has now been automated and does not require manual intervention any more. In event 7, the computer running the BioOpera server was shut down for maintenance. The BioOpera server re-started smoothly once maintenance was completed (event 8). In event 10, and due to a software problem in the algorithms used, two of the last TEUs failed to report their results to the BioOpera server. In order to ensure that all was correctly computed, the process was re-started and BioOpera immediately re-scheduled the TEUs that then completed successfully.

Event 9 did not require any type of reaction from BioOpera. It only represents a period of time when many higher priority jobs were sent to the cluster causing progress to slow. File system instability caused the rate of failed TEUs to increase slightly. BioOpera successfully re-started all failed activities.

From these results, we can conclude that BioOpera successfully automated the computation, thereby greatly speeding up the overall procedure and making it much easier to handle. Of the 10 events observed during the entire computation, events 1 and 9 will always occur in any system and BioOpera offers the necessary functionality to suspend and resume computations smoothly. Event 10 is a problem of the underlying software used in the computations and beyond the control of BioOpera. All other events can now be masked by BioOpera so that no manual intervention is necessary to deal with system or activity failures. This is a significant improvement over the current state of the art in virtual laboratories.

Related to this experiment, we also noted some limitations of BioOpera. In particular, there were some problems with scheduling/load balancing decisions. Unlike cluster or grid computing
systems that work at the operating system level and do not resort to external applications for computing, BioOpera cannot migrate jobs once they are started in a computer. This means that, once a scheduling decision has been made and the job has started, there is not much that can be done aside of killing the job and starting it somewhere else. This is important because in this experiment there were several cases when BioOpera sent TEUs to processors that had available cycles. By the time the TEU was initiated, other users had started jobs with higher priority. Event 9 is a good example of this. One strategy to solve this problem would be to have BioOpera abort the affected TEU and re-schedule it elsewhere, other strategies are also being investigated [SW98]. However, the effectiveness of this strategy greatly depends on the utilization patterns in the cluster. As was the case here, if the non-BioOpera user tends to fill all machines, such a strategy will perform worse than if BioOpera had simply left the TEU where it was. If however the user tends to use only a subset of the processors, the kill and restart strategy may help to improve the WALL time. Future versions of the system will also support this load balancing strategy.

5.5 The All vs. All (non-shared cluster)

The first experiment proved that BioOpera can run long-lived computations while coping with the heterogeneity and continuous changes of shared clusters. In a second experiment we wanted to test the stability of the BioOpera system by itself by running the same computation on a non-shared cluster. This second experiment was run on the *ik-linux* cluster from the 31st May 2000 until the 27th July 2000. As shown in Figure 6 the operating system configuration changed, since from day 35 a second processor was added to each node, and BioOpera was able to take advantage of this.

The second experiment proved that BioOpera is indeed quite stable and can effectively use all available resources. In this run, there were only three events of interest. The first two were planned network outages that required to suspend the execution of the process. The third event, which did not require manual intervention at all, was an upgrade in the cluster that made an extra processor available for each machine. Figure 6 clearly shows how once the number of processors doubled, BioOpera took advantage of the available CPU power immediately. In terms of performance, the results of the second run validate and corroborate the results of the first run.
6 Conclusions

The results of the experiment with the all vs. all for SP38 show the BioOpera has fulfilled its main objective: to be able to dependably run month long computations with minimal user intervention. This is a significant step towards providing the software infrastructure needed in virtual laboratories. Although for reasons of space this has not been discussed, BioOpera not only provides a dependable environment in which to define and execute long lived computations. It also has additional functionality that is very important in virtual laboratories. For instance, lineage tracking is done automatically and all dependencies are persistently recorded. This makes it possible for the system to recompute processes as data inputs or algorithms change. The ability to suspend and resume a process is also profusely used to check intermediate results and correct data that the algorithms used have not treated correctly. Similarly, by developing the processes and running them together with bioinformaticians, BioOpera has been very finely tuned to the needs of the users both in terms of interfaces and APIs. Thus, we feel that these results are very positive, especially considering that BioOpera is only at a prototype stage.

As part of future work, we intend to provide a backup architecture for the BioOpera server so that if a server fails or requires maintenance, the backup can assume control and continue execution smoothly. We are also developing new processes. In particular, we have begun a gene prediction package. As each new genome is made available, the process will apply several existing and new gene finding algorithms to the raw DNA dataset. We have also developed several data mining approaches for molecular sequence data based on some of the ideas in [HL99a, HL99b]. Furthermore, we are now experimenting with a search space parallelization approach for the Phylogenetic Tree Problem with maximum likelihood scoring. The ability to easily perform distributed computation will increase significantly the size of the trees we are currently able to compute under this computationally expensive scoring scheme.

Lastly, we have begun to couple BioOpera with a number of tools for viewing the results of the computations. Besides allowing us to use the computed information more easily, it will allow us to disseminate this information to the general public more quickly.

References


