Dependable Computing in Virtual Laboratories

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

M. Hallett
McGill Centre for Bioinformatics
McGill University
Montreal, Canada
hallett@cs.mcgill.ca

Abstract

Many scientific disciplines are shifting from in vitro *to* in silico *research as more physical processes and natural phenomena are examined 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 computations.*

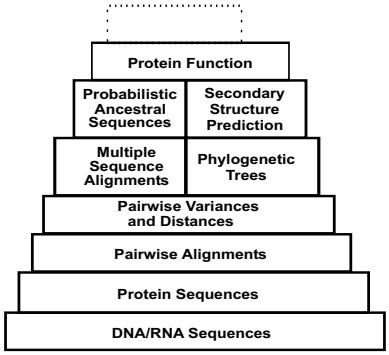

Figure 1. Tower of Information for Computational Biology

1. Introduction

Virtual laboratories are becoming increasingly pervasive now that the cost of storing observations is lower than the cost of making them [11]. Although these environments are typically associated with massive amounts of data [15, 20, 21], initial experiences have shown that data processing is more critical than data storage [6, 19].

To better illustrate the problem, consider the following example. One of the goals of the BioOpera project is to be able to build a software system capable of automatically predict the secondary structure of a protein from its primary aminoacid sequence alone. The idea is to construct a *tower of information* such as the one 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. The first step consists of locating genes in the raw DNA and translating these coding regions into amino acid sequences. The amino acid sequences 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 have occurred (and been accepted) in the genomes of the organisms over vast periods of time. This evolutionary history provides important clues towards the prediction of secondary structure [17]. In turn, these secondary structural units provide avenues for the prediction of the acutal structure (or shape) of the protein and knowledge of this three-dimensional configuration may lead to accurate assignments of the function of the protein.

In most existing virtual laboratories, storing, manipulating, and keeping track of computations like the tower of information 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. 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 [3] that has evolved into a programming [2] and runtime environment for cluster computing [14, 8] with the capability to define, execute, monitor and manage a broad range of large-scale, complex scientific computations.

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

In this paper, we will focus on sequence alignment [10, 9, 12]. Using a typical data set, e.g., Swiss-Prot vers. 38 (SP38[4]), the starting point is $80,000$ 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 [12]. 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 [16, 7]). 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.

3. BioOpera: a Virtual Laboratory for Bioinformatics

3.1. Process Design

A computation in BioOpera is represented as a *process*. BioOpera uses a language called *Opera Canonical Representation* (OCR) [14] 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). 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 *blocks* and *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.

In addition to these primitives, OCR also supports *exception handling*, *event handling*, and *spheres of atomicity* [1, 2]. The combination of these features allows the process designer to define sophisticated failure handlers as part of the process [14] (such as undo actions, alternative executions, and various forms of exception handling).

3.2. Architecture and Basic Functionality

BioOpera can be seen as a high-level distributed operating system managing the resources of a computer cluster. It

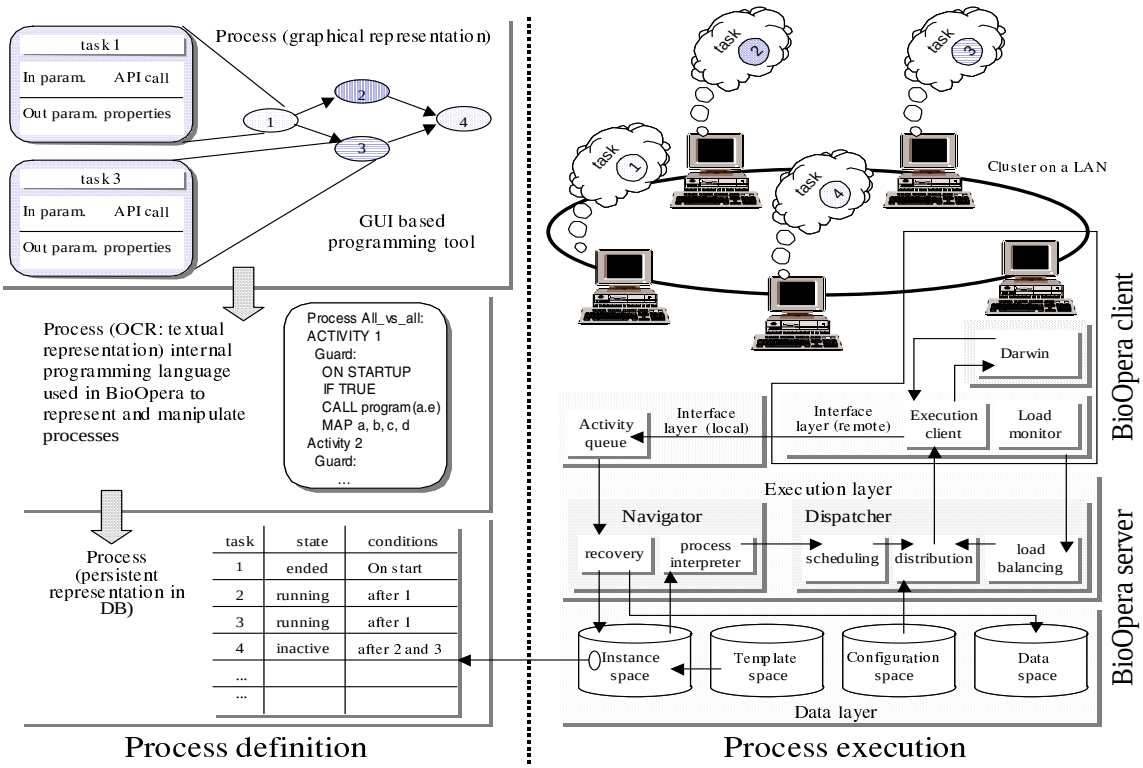

Figure 2. Architecture of Opera

has both a runtime component and a development environment (Figure 2).

The development environment allows users to specify BioOpera processes (eventually a library of processes and activities will be provided so that users can run predefined processes without having to define them themselves). The development environment encompasses three elements: *process creation*, *library management*, and *configuration management*. The configuration management allows users to specify the hardware and software 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 job placement decisions, 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. 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 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. 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 given control and looks for the next activities to execute.

Interaction with external applications takes place through specific interfaces or *wrappers*. For computational purposes, we are currently working exclusively with one software tool, Darwin [13], but any other such system could be used.

3.3. 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* [5] 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.

3.4. 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 has several advantages. Since the computation is outlined in the process, 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 (e.g., 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 re-schedule the processes accordingly.

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 [13] as our bioinformatics application. This software offers a dynamic programming local alignment algorithm which uses the GCB scoring matrices and an affine gap penalty [12, 18]. 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 re-start 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 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 [1]: 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.

[1] Care was taken to rule out redundant comparisons across different subprocesses.

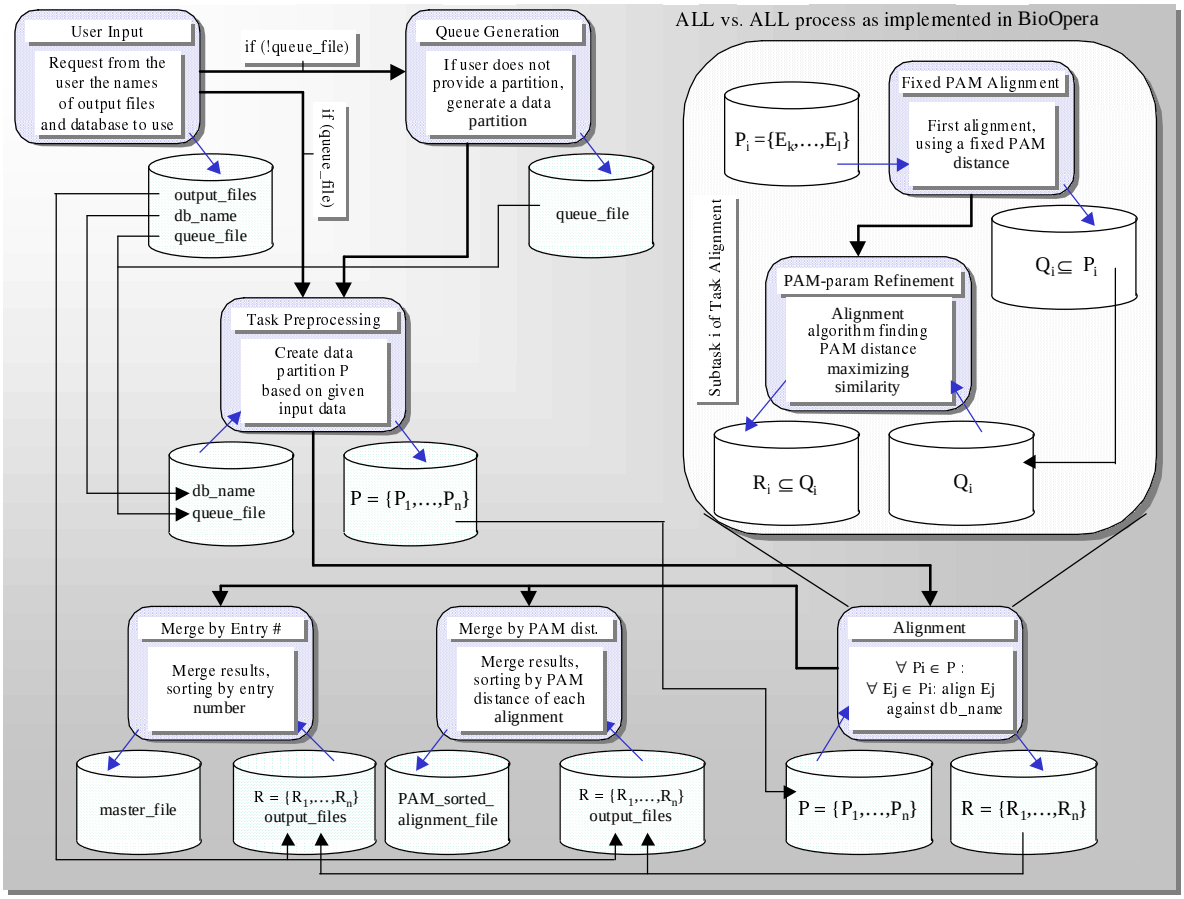

Figure 3. The all vs. all process as implemented in BioOpera

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 measure 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.

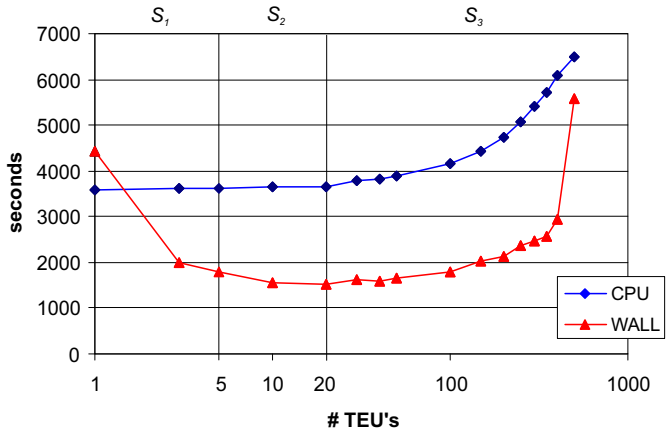

Figure 4. Effects of the granularity level.

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

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.

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

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

To determine the optimal granularity levels in the *all vs. all* computation, we analyzed the overall performance (i.e. CPU and WALL times) for varying numbers of task execution units (TEUs). The dataset used for this purpose consisted of 500 entries from SP38. These experiments were run on the *ik-sun* cluster with no other users or jobs in the cluster. The number of TEUs for these experiments varied between $n = 1$ (no parallelization), and $n = 500$ (each of the 500 *one vs. all*s 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 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 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-sun* machines were disabled towards the end of the computation. All activities were run with lowest priority. The computation lasted from the 17^{th} December 1999 until the 25^{th} of January 2000. Note that the final two days of computation are not represented in Figure 5.

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 re-

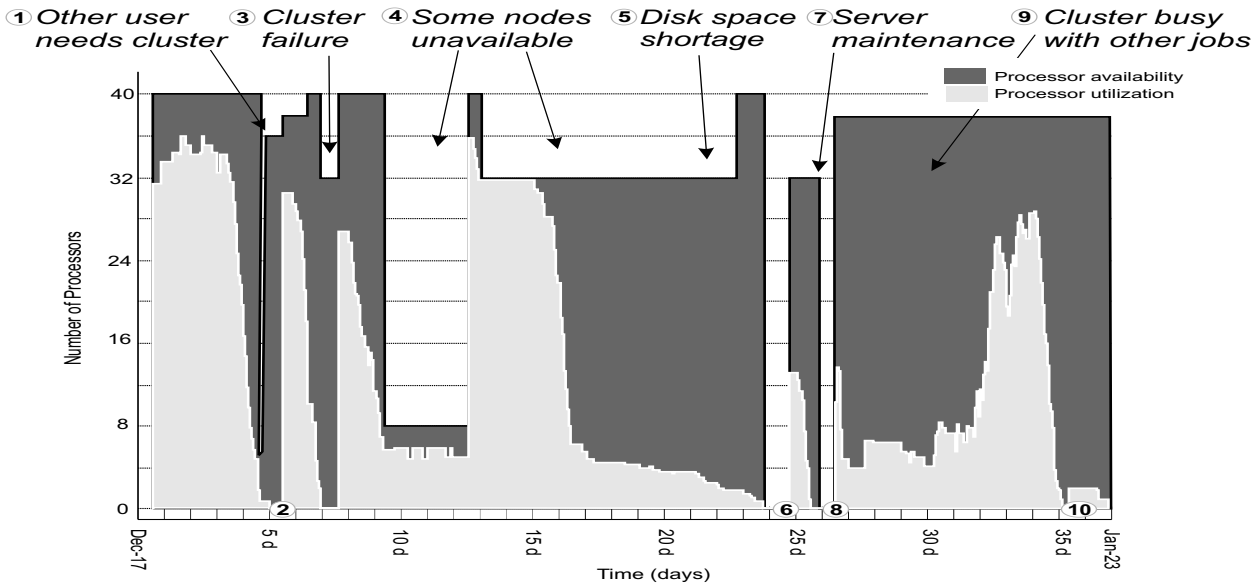

Figure 5. Lifecycle of the all vs. all (first run).

	Shared cluster	*Non-shared cluster*
Max. # of CPUs	40	16
$CPU(\Omega)$	441d 7h 11m	400d 2h 36m
$WALL(\Omega)$	38d 21h 30m	49d 20h 46m
$\overline{CPU(A)}$	0d 20h 41m	0d 18h 39m

Table 1. Results for the two experiments

sults 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.

Regarding the ability of BioOpera to automate the overall procedure, the history of this first computation is summarized in Figure 5. BioOpera jobs were run in *nice* mode, giving priority to the other users. Figure 5 contains a number of *event indicators* that 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). Events 2, 6, 8 and 10 indicate BioOpera server shutdowns, which caused the running process to terminate. After restart, BioOpera automatically resumed the computation from where it stopped at server shutdown. 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 since no manual intervention was necessary to deal with system or activity failures.

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 31^{th} May until the 27^{th} July 2000. As shown in Figure 6 the operating system configuration changed, since from day 35 a second processor was added to each node. The results show that BioOpera is 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 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 immediately took advantage of the available CPU power.

6. Conclusions

The experimental results demonstrate that BioOpera is 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. BioOpera 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

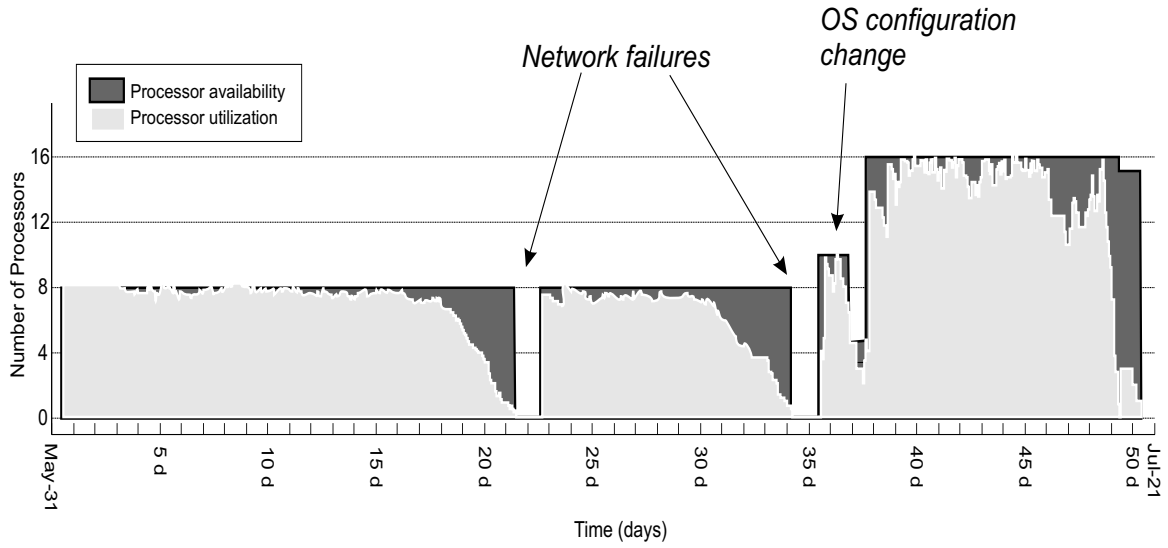

Figure 6. Lifecycle of the all vs. all (second run).

to recompute processes as data or algorithms change. The ability to suspend and resume a process is also profusely used to check intermediate results and correct intermediate data. Given these results and the feedback obtained from bioinformaticians, we feel BioOpera has a good potential as a tool for Virtual Laboratories and opens up very interesting and challenging research directions.

References

[1] G. Alonso and C. Hagen. Flexible exception handling in the opera process support system. In *18th International Conference on Distributed Computing Systems (ICDCS), Amsterdam, The Netherlands*, 1998.

[2] G. Alonso and C. Hagen. Beyond the black box: Event-based inter-process communication in process support systems. In *19th International Conference on Distributed Computing Systems (ICDCS), Austin, Texas, USA*, 1999.

[3] G. Alonso, C. Hagen, H.-J. Schek, and M. Tresch. Distributed processing over stand-alone systems and applications. In *Proceedings of the 23rd International Conference on Very Large Data Bases, Athens, Greece, 26–29 August 1997*, pages 575–579, 1997. Available at http://www.inf.ethz.ch/department/IS/iks-/publications.html.

[4] A. Bairoch and R. Apweiler. The SWISS-PROT protein sequence data bank and its supplement trEMBL. *Nucleic Acids Research*, 27:49–54, 1999.

[5] D. Baker, D. Georgakopoulos, H. Schuster, A. R. Cassandra, and A. Cichocki. Providing customized process and situation awareness in the collaboration management infrastructure. *CoopIS*, pages 79–91, 1999.

[6] T. Barclay, J. Gray, and D. Slutz. Microsoft TerraServer: a spatial data warehouse. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data: May 16–18, 2000, Dallas, Texas*, 2000.

[7] A. Bonner, A. Shrufi, and S. Rozen. LabFlow-1: A database benchmark for high-throughput workflow management. In *Proceedings of the 5th Int. Conference on Extending Database Technology (EDBT96), Avignon, France*, 3 1996.

[8] R. Buyya. *High Performance Cluster Computing, Volume 1 and 2.* Prentice-Hall, 1999.

[9] G. Cannarozzi, M. Hallett, J. Norberg, and X. Zhou. A cross-comparison of a large gene dataset. *Bioinformatics*, 16:654–655, 2000.

[10] S. Chervitz. Comparison of the complete protein sets of worm and yeast: orthology and divergence. *Science*, 282:2022–2028, 1998.

[11] Mining the digital skies. *The Economist*, 1.06.2000.

[12] G. Gonnet, M. Cohen, and S. Benner. An exhaustive matching of the entire protein sequence database. *Science*, 256:1443–1445, 1992.

[13] G. Gonnet, M. Hallett, C. Korostensky, and L. Bernardin. Darwin version 2.0: An interpreted computer language for the biosciences. *Bioinformatics*, 16:101–103, 2000.

[14] C. Hagen. *A Generic Kernel for Reliable Process Support*. PhD thesis, Dissertation ETH Nr. 13114, 1999.

[15] Y. E. Ioannidis, M. Livny, S. Gupta, and N. Ponnekanti. ZOO: A desktop experiment management environment. In *Proceedings of the 22nd international Conference on Very Large Data Bases, September 3–6, 1996, Mumbai (Bombay), India*, pages 274–285, 1996.

[16] J. Meidanis, G. Vossen, and M. Weske. Using workflow management in DNA sequencing. In *Proceedings of the 1st International Conference on Cooperative Information Systems (CoopIS96) Brussels, Belgium*, 6 1996.

[17] B. Rost and C. Sander. 3^{rd} generation prediction of secondary structure. In D. M. Webster, editor, *Predicting Secondary Structure*. Humana Press, 1998.

[18] T. Smith and M. Waterman. Identification of common molecular subsequences. *J. Mol. Biol.*, 147:195–197, 1981.

[19] A. S. Szalay, P. Z. Kunszt, A. Thakar, J. Gray, D. Slutz, and R. J. Brunner. Designing and mining multi-terabyte astronomy archives: the Sloan Digital Sky Survey. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data: May 14–19, 2000, Dallas, Texas*, volume 29(2), 2000.

[20] J. T. L. Wang, K. Zhang, and D. Shasha. Pattern matching and pattern discovery in scientific, program, and document databases. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data: May 23–25, 1995, San Jose, California*, volume 24(2), 1995.

[21] M. Zemankova and Y. E. Ioannidis. Scientific databases — state of the art and future directions. In *20th International Conference on Very Large Data Bases, September 12–15, 1994, Santiago, Chile*, pages 752–753, 1994.