BioOpera
Managing large-scale computations in bioinformatics

Author(s):
Alonso, G.; Bausch, Win; Hallett, M.T.; Kahn, A.; Pautasso, Cesare

Publication Date:
2000

Permanent Link:
https://doi.org/10.3929/ethz-a-006654174

Rights / License:
In Copyright - Non-Commercial Use Permitted
Managing Large-Scale Computations in Bioinformatics

G. Alonso†, W. Bausch†, M. Hallett‡, A. Kahn†, C. Pautasso†,
†Information and Communication Systems (IKS)
‡Computational Biochemistry Research Group (CBRG)
	Tel. +41 1 632 7475, Fax. +41 1 632 11 72
Dept. of Computer Science, ETH Zürich, Switzerland
{alonso,bausch,hallett,kahn,pautasso}@inf.ethz.ch

Keywords: process support, large-scale computation, bioinformatics, all vs. all

Abstract

Large-scale computations are pervasive in bioinformatics due to the size of the datasets involved and the resource demands of the algorithms. These computations typically run for long periods of time and produce vast amounts of data. Currently, users are required to manage system resources, data, and the distribution of each process throughout the entire computation. Not surprisingly, this introduces a major bottleneck in the whole procedure.

This paper describes the architecture of BioOpera, a process management system that controls the distribution, execution, and monitoring of large-scale computations on clusters of computers. BioOpera can be used in heterogeneous computing environments and provides capabilities for specifying complex sequences of computations and tools for monitoring their execution. It can also persistently store all intermediate and final results produced. To demonstrate the power of BioOpera, we performed an all vs. all alignment of Swiss-Prot version 38. This computation, which requires every sequence to be aligned against every other sequence, involved several billion pairwise Smith-Waterman alignments and ran for approximately one month in the background of a heavily loaded cluster of machines. Our need to intervene on behalf of BioOpera was minimal and the entire computation required significantly less time to complete than previously performed manual efforts.

Introduction

Although research has produced better and more efficient algorithms for a variety of computational problems in bioinformatics, hardly any attention has been paid to the problem of how to use these algorithms on large amounts of data. For instance, how to align two protein sequences is a well-understood problem that can be solved in a variety of ways. However, how to perform several billion sequence alignments is quite a different matter. Such large-scale computations are pervasive in bioinformatics. The datasets involved e.g., sequence databases, structure databases, or complete genomes (Bairoch and Apweiler, 1999; Stoesser et al., 1999; Fletcher et al., 1996) tend to be extremely large. To make matters worse, algorithms for optimization problems such as alignment, gene finding, tree construction, physical mapping and sequence reconstruction frequently require large amounts of computing resources both in time and space. Furthermore, new data mining techniques promise new insights into molecular data but also promise unequalled computational burdens (see Marcotte et al., 1999) for a very recent example.

We offer the following example in order to better understand this problem. Given a set of nucleotide or peptide sequences, the standard “first step” into any inquiry of the evolution, structure, and ultimately function of these biomolecules is the alignment of each sequence in this set against every sequence in a large dataset such as EMBL (Stoesser et al., 1999) or Swiss-Prot (Bairoch and Apweiler, 1999). Such cross-comparisons lie at the heart of comparative genomics and the resulting alignments are used for a broad range of purposes (see Chervitz et al., 1998; Cannarozzi et al., 2000; Gonnet et al., 1992; Snel et al., 1999) as starting points. In this paper, we concentrate on the problem of aligning every entry in Swiss-Prot vers. 38 (SP38) against all other entries in this dataset - a self-comparison or all vs. all. The exact details of how each alignment is carried out and the data-structures used are beyond the scope of this paper. Suffice it to say, we use the Darwin system (Gonnet et al., 2000) as our example bioinformatics application. This software offers a dynamic programming local alignment algorithm which uses the GCB scoring matrices and an affine gap penalty (Gonnet et al., 1992; Smith and Waterman, 1981). Those pairs of sequences which exceed an extremely liberal similarity bound are refined using a computationally more expensive alignment algorithm that seeks to find the PAM distance maximizing similarity score. We refer the reader to the (CBRG, 2000) for further information and the actual results of the all vs. all.

In total, SP38 contains 80,000 amino acid sequences. An all vs. all comparison requires approximately $3.2 \times 10^9$

Copyright © 2000, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.
individual pairwise alignments\(^1\). As an indication of what this implies, over the past 7 years, the CBRG has updated (and made public) the all vs. all comparison of Swiss-Prot versions, 27 (Gonnet et al., 1992). Current updates typically involve at most 10,000 new sequences and already require on the order of 3 to 4 months of computation (running in the background on a cluster of computers). During such computations the datasets, software, and workstations involved have to be painstakingly maintained. The onus for such chores lies on the users; they must partition the job into manageable smaller pieces, distribute the jobs to various machines in their computing environment, remove jobs from overloaded machines, restart jobs due to either system or software errors, and eventually coalesce results. Not surprisingly, such maintenance has historically been one of the greatest bottlenecks, becoming a dominant factor in the overall cost of performing such computations. In fact, this problem is not exclusive to bioinformatics - it appears in almost any scientific discipline where large-scale, heterogeneous computing is used (see (Meldanis et al., 1996; Ioannidis, 1996; Bonner et al., 1996) for examples).

From this experience, it has become clear that the way updates are currently performed is no longer viable. In this and in many other similar computational problems, it is necessary to automate the management of the computation. Amongst other items, it is necessary to implement mechanisms that automatically and transparently handle issues such as efficient scheduling of jobs, load balancing, tracking the progress 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 both intermediate and final results of the computation. In addition, a structured way to design such computations is needed. The use of C, JAVA or Perl scripts results in a very inflexible solution, since it is difficult to modify, port and update them. Ideally, a tool should be available which allows the user to express these computations in a robust, machine independent manner via a high-level programming language.

To address these issues, we looked into current prototypes developed by the Information and Communication Systems Group (IKS) to automate the management of distributed computations (Alonso et al., 1997). From this starting point, we developed BioOpera, a tool based on the notion of process. A process is a sequence of invocations of computer programs in a distributed and heterogeneous environment and the corresponding data exchanges between these programs. The notion of process can be identified with a meta-program (the process template) encoding a sequence of program invocations. At execution time this process template is used to create a process instance; this can be thought of as a runnable copy of the template. Herein, we say that a system which supports the definition of process templates and the execution/monitoring of process instances is a process support system. This, BioOpera is a process support system that can be used to define, execute, monitor and manage a broad range of large-scale, complex bioinformatic computations in a heterogeneous computing environment.

This paper describes the architecture and functionality of BioOpera. To demonstrate the potential of the system, we compute an all vs. all alignment of SP38 using a small cluster of heavily used machines. In what follows, we show how BioOpera helped to significantly reduce the computation time and automate the procedure to the point where minimal manual intervention was necessary. By providing such functionality, BioOpera can become a key tool in bioinformatics as it addresses some of the most critical aspects of managing large computations over clusters of computers. Although it is always possible to resort to large mainframes or specialized computers to perform this type of computations, not all bioinformaticians have access to these resources. As the complexity of the questions bioinformaticians ask grows and more groups join this research area, the notions of usability, persistence, control, and automation offered by BioOpera will become more and more relevant.

The paper is organized as follows. Section **Process Design in BioOpera** provides a concrete example of a BioOpera process via the all vs. all case study. Section **Architecture and Basic Functionality** describes the architecture of BioOpera and its basic functionality. Section **Experimental Results** gives the performance results obtained from computing the all vs. all of SP38. Finally, Section **Discussion** comments on various aspects of these results and a number of important issues related to future directions for BioOpera.

### Process Design in BioOpera

The main difficulty involved with supporting large-scale computations over a cluster of machines lies in finding an appropriate representation for describing the computation. In essence, this description consists of two levels: the code for the various algorithms (e.g., sequence alignment, tree construction) that are applied to the data, and the code that drives the application of the algorithms to the data in the cluster of machines. Historically, such a division between the applications and the software controlling the applications has not been made. Often, programming languages (e.g., C, JAVA) and collections of operating system scripts (e.g., Perl, Python) have been used together in an ad hoc fashion to meet the immediate needs of the application at hand. Such an approach leads to code that is extremely difficult to modify and rather primitive, unstructured methods for driving and monitoring computations.

A computation in BioOpera is represented as a process. A process, as explained above, can be seen as

\(^1\)Certainly, datastructures such as the suffix tree, which we do use here, reduce this number significantly.
an annotated directed graph where the nodes represent tasks to complete and the arcs represent the control/data flow between these tasks.

More formally, a process consists of a set of tasks and a set of data objects. Tasks can be activities, blocks, or subprocesses. The data objects store the input and output data for tasks and are used to pass information around the system. Figure 1 depicts a simplified version of the all vs. all process as it is implemented within BioOpera. Activities (rectangles in Figure 1) are the basic execution steps; each activity has an external binding that specifies a program to be executed (not shown in the figure). This information is used by the runtime system to execute external applications or to instruct users to perform certain actions. In the simplest form, a process consists of only activities, control and data flow descriptions. 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 1); the figure shows the activation condition for the control connector (\(\text{queue file}\)) 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 is also possible between tasks and between processes (shown in Figure 1 as thin connecting lines). 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 1). 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. The input and output data structures of the process itself are part of the whiteboard; this acts as the global data area for the process.

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 language constructs: for, do until, while, for, and for parallel processing (Alignment in Figure 1 is a block). 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 modifications of a running process by offering the ability to change its subprocesses.

We now briefly describe each task in the all vs. all process depicted in Figure 1.

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 SP38 and our queue file contains the list of entries \(E \subseteq [1 : N]\) where \(N = 8 \times 10^4\) for SP38. The purpose of the queue file is two-fold. First, it is used by the following two tasks to control the degree of parallelism during execution. Second, the indexing provided by the queue file allows BioOpera to discard ill-behaving sequences and smoothly re-start computation when failures occur.

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

Task “Preprocessing” is responsible for preparing the data for parallel execution by creating a partition \(P = \{P_1, \ldots, P_n\}\) of the entries \(E\) in the queue file. Members of \(P\) are not necessarily of equal size. Each such \(P_i\) is treated as a single unit throughout the computation. That is, the input to each instance of the Alignment block consists of one \(P_i\) and this block is responsible for aligning each \(p \in P_i\) against all other entries of SP38. The \(n\) members of the partition \(P\) may be computed in parallel by instances of Alignment block. The degree of parallelism and level of granularity is controlled by our choice of size for the members of \(P\) (see Section Parallelism for a more complete discussion of this topic). Future versions of BioOpera will produce partitions of \(E\) automatically by considering the load balancing and scheduling strategies in place for the particular computation.

Block “Alignment” receives as input one \(P_i\) and is responsible for aligning each entry of \(p \in P_i\) against all other entries of SP38 so as to filter out the uninteresting ones. All computation within one instance of the Block Alignment is carried out sequentially in two steps. First, a Fixed PAM alignment performs a pairwise alignment at a fixed PAM distance of 250 (see (CBRG, 2000), and (Cannarozzi et al., 2000) for more details). Second, any of these alignments passing a (liberal) similarity score are refined in task PAM-param refinement. Here a computationally more expensive but more informative alignment algorithm is used. This algorithm returns an estimation of PAM distance which maximizes the similarity score of the alignment. The result of each instance of this block is stored as a separate dataset \(\mathcal{R}_i\).

Task “Merge by Entry #” merges the output files created by the previous step into one master file containing all significant alignments between sequences in SP38. The contents of this file are sorted according to the entry number in the original database.

Task “Merge by PAM Distance” separates the results into various files according to PAM distance estimations.
Architecture and Basic Functionality

Currently, users must express their computations directly in the BioOpera specific language called Opera Canonical Representation (OCR) (Hagen, 1999). This has several drawbacks. First, it is unrealistic to assume that a novice user could quickly master the details of such a low-level language. Most bioinformations are familiar with neither programming languages nor distributed computation! Second, working with OCR directly does not separate the definition of processes from the definition of system configuration. The former tells BioOpera what is to be done; the latter describes where it is to be done. This is an important level of abstraction as the computation changes much more often than the environment. We are currently developing a graphic user interface (GUI) for BioOpera. Via drag and drop operations, the user will be able to design their process without the need to know the OCR programming language and it will separate the notions of process definition from system configuration. The specification created in the GUI will be compiled automatically into the OCR.

Beyond the primitives implicit in the discussion of the preceding section, OCR supports advanced programming constructs such as exception handling, event handling, and spheres of atomicity (Hagen and Alonso, 1998; Hagen and Alonso, 1999). These constructs were included in order to provide a more reliable computing environment by allowing users to specify how the system should respond when failures occur. The simplest way of handling a faulty process is to undo all of its effects. This is called a process abort and effectively cancels the process by deleting all data produced up until that point. If necessary, a more complex recovery mechanism has been developed that allows one to specify more sophisticated failure reactions (Hagen, 1999).

Once an image of the process template (i.e. a process instance) is stored in the database, the process can be executed. Note that there are several “spaces” within the database: the template space contains process templates, the instance space contains processes currently executing, the configuration space contains the information related to system configuration, and the data space contains all processes already executed along with references to external datasets (see Figure 2).

During execution, a process expressed in OCR 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 network (i.e. a machine in the computing environment) and a particular application (i.e. software). If the choice of assignment is not unique, the node is determined by the scheduling and load balancing policy in use (Figure 2). 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. 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. A special 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.

Interaction with external applications takes place through specific interfaces or wrappers. For computational purposes, we are working right now exclusively with Darwin (Gonnet et al., 2000) but any other system could be used. In particular, we are currently studying how to incorporate several existing data visualization tools into BioOpera.

**Advanced Functionality**

Of all the features BioOpera provides, three of them deserve special attention: parallel execution, monitoring, and load balancing. We discuss each in turn below.

**Parallelism**

Parallelization is mandatory for bioinformatic processes given the size of the datasets involved (again, consider the all vs. all). The concept of process as a programming paradigm well supports certain types of parallelism.

In a data parallelization approach, the input data $E$ is partitioned into sets to be computed in parallel. Let $\mathcal{P} = \{P_1, \ldots, P_n\}$ denote this partition. We call the task which is responsible for computing the entire set $E$ a parallel task. This parallel task is subdivided into $n$ subtasks defined by $\mathcal{P}$. We call one such subtask a task execution unit (TEU). Essentially, each TEU is assigned a machine and a member $P_i$ of $\mathcal{P}$ by BioOpera. It...
is responsible for sequentially running the appropriate application on each member $p \in P_i$.

The first issue here is to decide how large each $P_i$ should be. That is, we must decide our level of granularity. If the granularity is too fine (there are many $P_i$), then the system will incur excessive overhead due to an unnecessary high number of invocations of the scheduling, load balancing, and merging algorithms. In the all vs. all 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. Our experience suggests each partition should contain significantly more than a single one vs. all.

At the same time, the granularity should be chosen to be fine enough to allow BioOpera enough flexibility to effectively schedule and balance jobs. For example, workstations are heavily loaded during the day but remain idle over night. If the granularity were set too coarse, a task requiring an entire day to complete might be sent to an idle machine in the evening, successfully compute the entire night, but remain idle the entire day when the owner of the workstation returns. Clearly, it would be better to split this task into two smaller units, submit the first to the idle machine over night, and submit the second to a less-burdened machine in the morning.

BioOpera must perform various bookkeeping chores in order to support this type of parallelism. In particular, it has to create $n$ input datasets (a one to many function) from the dataset $E$. Each step of the process must then be applied to each of these $n$ datasets (many to many function), resulting in $n$ output datasets. A final stage merges the resultant datasets back to a single dataset (many to one function).

A search space parallelization approach is best illustrated with a simple example. The Phylogenetic Tree Problem involves searching an exponentially large space of trees for the tree(s) requiring the [most likely, fewest number, most agreeing] set of evolutionary events. The task here is to find the optimally scoring tree under one of these criteria. With such a parallelization approach, BioOpera would partition the space of candidate trees (i.e. the search space) into sets, distribute these sets to available machines for execution (scoring), and eventually report the best tree(s) found. As before, BioOpera needs to do some bookkeeping in order to map input from a variety of sources into a single target. This is currently not supported, but it will be incorporated during our next development phase.

Finally, an application parallelization approach involves modifying the application a task invokes (i.e. parallelizing the algorithm in the application). The modified application is applied to the dataset as a whole and the results are merged upon successful completion. Such an approach is classical parallel algorithms research.

**Monitoring**

Monitoring is an important aspect in long duration computations as it enables one to keep track of the state of the computation and influence the outcome if necessary. Most information pertaining to a process 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 which, in turn, allows BioOpera to react to changes in the computing environment and, of course, provide a very complete view of the computation. The most important reactions here include the ability to load balance and schedule according to machine usage and availability, to restart computations smoothly when failures occur, and to avoid inconsistencies in the output data after failures.

The monitor also shows dynamically how far our computation has progressed and (eventually) it will be capable of making predictions for the amount of time remaining. Besides these passive abilities, the monitor allows us to actively influence computation as the user can start, stop, abort, re-start, and change input parameters during each step of the computation.

**Scheduling and Load Balancing**

Scheduling and load balancing are necessary in any system offering parallelism. Once a task has been appropriately partitioned, it must be distributed over the available machines. The monitoring capabilities of BioOpera provide sufficient information to make intelligent choices concerning where to send each task. Currently, the system 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 program execution client (PEC) compares the last recorded load at a machine with the current load at the machine. 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 whilst 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 load curve generated by the base line sampling strategy, which corresponds to a constant sampling interval of 10 seconds.

**Experimental Results**

**Hardware Environment**

For our experiments, we use a combination of PCs and Unix workstations. 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* and identify each node as *linneus 1, linneus 2*, etc. The second 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 cluster*. In the first part of the experiment, both clusters shared a file system whereas in the second part, the linneus cluster had a private file system. This change was due to problems during the computation.

**Experiment Description**

We have performed two different sets of tests. The first experiment was performed in order to find the optimal granularity level for parallelization. The results of this experiment shaped how the *all vs. all* process was executed in the second test and also helped to identify problems BioOpera would have over the course of a long computation. This first experiment consisted of several versions of the *all vs. all* process using a smaller database (500 entries) and different granularity levels. The second experiment was the computation of the *all vs. all* on SP3s. Of course, our objective here was to measure, in a meaningful way, how effective BioOpera was at managing the large-scale computation. We introduce our measures below.

We note that the interface between BioOpera and Darwin was also developed during these tests. The problems encountered during the first phase of testing helped us to define and implement a set of APIs (Application Program Interfaces) to Darwin so that it can be used by external applications. An important aspect of this new set of APIs is that it has been made BioOpera aware: the computations do not execute blindly; they first perform a number of checks. Essentially, these checks include the determination of whether a computation has previously been executed, whether it executed to completion or only partial completion, and a control flow logic to make use of the recovery procedures provided by BioOpera.

To evaluate the results of the experiments performed, we have used several criteria. First, each activity is considered a *black box*, that is, we measure the time it took to complete a given computational step, $A_i$, by looking at how long it was active on a given CPU (CPU time: $CPU(A_i)$) and also by comparing its starting and finishing time (REAL time: $REAL(A_i)$). For any given activity, it is always the case that $REAL(A_i) \geq CPU(A_i)$. The difference between these two figures is a good indicator of the load at a processor. Let $\Omega$ denote the execution time of the entire *all vs. all* process and let $A_1, A_2, \ldots$ denote the length of each activity executed during the computation of $\Omega$. We calculate the following:

- The **CPU time** of a process (the number of CPU cycles need to complete the process) is the CPU time it took to execute all of its activities. This value indicates how much resources a process demands.
  \[
  CPU(\Omega) = \sum_{A_i \in \Omega} CPU(A_i)
  \]

- The **REAL time** of a process is the sum of the REAL time of all its activities. When compared with the CPU time, this value indicates how long the process was waiting for computational resources to become available (either because of failures or because they are being used by other users) and the CPU time taken to complete it. As is the case for each individual activity, the difference between the REAL time and the CPU time is a measure of the load in the clusters. That is, it is a measure of how much the process had to share the resources with other jobs in the system. It is always the case that $REAL(\Omega) \geq CPU(\Omega)$.
  \[
  REAL(\Omega) = \sum_{A_i \in \Omega} REAL(A_i)
  \]

- The **WALL time** measures the absolute time it takes for the process to complete as the difference between its starting and finishing times - one looks to the clock on the wall when the process starts and when the process finishes. The WALL time depends heavily on the amount of parallelism achieved - essentially, the more parallelism, the less the WALL time. In addition, the WALL time is also a good indicator for the overhead introduced by BioOpera. This is the time needed to schedule the activities and navigate through the process description.

- The **average speedup factor** indicates the gain in processing time due to the parallelization of activities. Note that if there is no parallelism, the REAL and the WALL time would be similar (the differences are due to the overhead created by BioOpera). Also note that this measure also includes factors related to the load of the cluster (since the REAL and WALL time include this parameter).
  \[
  speedup(\Omega) = \frac{REAL(\Omega)}{WALL(\Omega)}
  \]

- The **average speedup performance** relates the average speedup factor to the maximum possible speedup factor ($speedup_{max}$). The latter can be approximated
by the total number of processors available. The value then becomes a lower bound for speedup performance.

\[ \text{perf}(\Omega) = \frac{\text{speedup}(\Omega)}{\text{speedup}_{\text{max}}} \]

\[ \text{perf}(\Omega) \approx \frac{\text{speedup}(\Omega)}{\#\text{processors}} \]

- Finally, the relation between the CPU and the REAL time with respect to the number of activities in the process \(|\Omega|\) gives a rough approximation of the time needed per activity and provides a set of values that can be used to better compare results.

\[ \overline{CPU}(A) = \frac{\text{CPU}(\Omega)}{|\Omega|} \]

\[ \overline{REAL}(A) = \frac{\text{REAL}(\Omega)}{|\Omega|} \]

The CPU, REAL, and WALL times have been extracted directly from the server logs whilst all other values are to be considered as derived data.

**Testing the Granularity Level**

As a first step towards determining optimal granularity levels for later use during the full *all vs. all*, we carefully analyzed the performance (i.e., CPU, REAL, 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 cluster in an exclusive mode i.e. no other users or jobs were allowed to use the machines. This allows a better control of the overhead and provides purer measurements.

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). Table 1 shows our results. Figure 3 charts these values using a logarithmic scale along the x-axis.

<table>
<thead>
<tr>
<th># TEUs</th>
<th>CPU</th>
<th>REAL</th>
<th>WALL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3596</td>
<td>4418</td>
<td>4421</td>
</tr>
<tr>
<td>3</td>
<td>3631</td>
<td>4597</td>
<td>2011</td>
</tr>
<tr>
<td>5</td>
<td>3629</td>
<td>4728</td>
<td>1790</td>
</tr>
<tr>
<td>10</td>
<td>3648</td>
<td>4612</td>
<td>1545</td>
</tr>
<tr>
<td>20</td>
<td>3661</td>
<td>4679</td>
<td>1506</td>
</tr>
<tr>
<td>30</td>
<td>3774</td>
<td>4946</td>
<td>1613</td>
</tr>
<tr>
<td>40</td>
<td>3898</td>
<td>4973</td>
<td>1594</td>
</tr>
<tr>
<td>50</td>
<td>3996</td>
<td>5234</td>
<td>1653</td>
</tr>
<tr>
<td>100</td>
<td>4157</td>
<td>5612</td>
<td>1798</td>
</tr>
<tr>
<td>150</td>
<td>4440</td>
<td>5837</td>
<td>2033</td>
</tr>
<tr>
<td>200</td>
<td>4749</td>
<td>6238</td>
<td>2145</td>
</tr>
<tr>
<td>250</td>
<td>5067</td>
<td>6634</td>
<td>2381</td>
</tr>
<tr>
<td>300</td>
<td>5386</td>
<td>7142</td>
<td>2455</td>
</tr>
<tr>
<td>350</td>
<td>5703</td>
<td>7637</td>
<td>2573</td>
</tr>
<tr>
<td>400</td>
<td>6086</td>
<td>8414</td>
<td>2955</td>
</tr>
<tr>
<td>500</td>
<td>6495</td>
<td>8816</td>
<td>5592</td>
</tr>
</tbody>
</table>

Table 1: Execution times (in sec.) for the 500 vs. 500 with different granularity levels

These results indicate several key points. Not surprisingly, Table 1 shows that the REAL and WALL time are almost identical with a single TEU. This is due to the fact that BioOpera introduces very little overhead when compared to the time to compute the actual alignments. Note also that the 1 TEU level gives the best CPU time but one of the worst WALL times. This can be better seen in Figure 3 where the WALL time decreases as the number of TEUs increase and the CPU time increases as the number of TEUs increase.

At the other extreme where the number of TEUs is 500, the CPU time is almost twice that the 1 TEU level. This is due to the overhead incurred from Darwin initialization stages and reading/writing to the disk and BioOpera overhead incurred from the scheduling and managing of the jobs. This is also reflected in the WALL time.

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 need to split Figure 3 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 adds significantly to the the overall CPU, REAL, 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 behav-
ior 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 course, this phenomenon can become quite large. One can compare the TEUs with pieces of wood of unequal size that have to be stored into several equally sized containers. If there are only as many pieces as we have containers, it might not be feasible to make the pieces fit exactly into the containers. However, if we cut up the pieces into smaller ones of unequal size, then there is a better chance that they can be evenly stored.

A granularity level of 20 implies that each TEU performs approximately $5\%$ ($6,250$) of the total number of individual pairwise alignments ($300$). 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.

The All vs. All for SP38

The all vs. all experiment was performed on both the lk and linneus clusters with some machines dedicated to certain tasks. In particular, the slower lk cluster was responsible for the refinement stages. All activities were run in nice mode with lowest priority. The computation lasted from the 17th December until the 25th of January. Note that the finally two days of computation are not represented in Figure 4.

![Figure 4](image)

Table 2: Performance of the All vs. All on SP38

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$CPU(\Omega)$</td>
<td>44d</td>
<td>07h</td>
</tr>
<tr>
<td>$REAL(\Omega)$</td>
<td>67d</td>
<td>21h</td>
</tr>
<tr>
<td>$WALL(\Omega)$</td>
<td>38d</td>
<td>21h</td>
</tr>
<tr>
<td>$CPU(A)$</td>
<td>0d</td>
<td>20h</td>
</tr>
<tr>
<td>$REAL(A)$</td>
<td>1d</td>
<td>7h</td>
</tr>
<tr>
<td>$speedup(\Omega)$</td>
<td>17.27</td>
<td></td>
</tr>
<tr>
<td>$perf(\Omega)$</td>
<td>0.48</td>
<td></td>
</tr>
</tbody>
</table>

The overall figures are shown in Table 2. The bottom-line here is that the entire process only required 38 days (WALL time). Previous manual efforts required significantly more time (on the order of months) and computed significantly less (mere updates of earlier version of Swiss-Prot). We also add that the cluster was heavily utilized by other users (with not-so-nice processes!) over this same period. However, we were required to manually intervene only a small number of times. Table 2, Figure 4 and the remaining discussion document this experience.

First, we note that the speedup factor $speedup(\Omega)$ shows we used less than half of the available computing power. Under optimal conditions, the entire process would have required only 19 days to complete.

Figure 4 also contains a number of event indicators that will be used to refer to particular phases of the execution. The dashed line indicates how many processors were actually available and ranges between 0 and 40. This variation is due to network failures, system maintenance, and software upgrades. The dark grey area indicates the number of processors that were doing some work that was eventually aborted (because of failures or crashes). The light grey area indicates the processors that had activities assigned to them but that were not performing any computation (because other users were using the clusters or because of problems with storage). The medium grey area represents those processors that had activities assigned to them and were actually computing. The processors that are available but are not used are processors BioOpera considered to be overloaded and, therefore, were not included in the scheduling decisions.

As indicated by the figure, the actual computing time is a small fraction of the total WALL time. However, we believe this result accurately reflects 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. Several of these failures are due to BioOpera and several can be attributed to the system.

The most significant BioOpera related issue was suboptimal scheduling/load balancing decisions. Several times 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 combat this would be to have BioOpera abort the TEU and re-schedule it elsewhere. However, the effectiveness of this strategy greatly depends on the utilization patterns in the cluster. As was the case here, if the foreign user tends to use all of the machines, such a strategy will perform worse than if we had simply left the TEU to run. If however the user tends to use only a subset of the processors, it should show marked improvements in overall run times. Future versions of the system will offer users both of these alternatives.

We briefly discuss each encountered system problem below.

**Event 1** Here another user requested exclusive access to the cluster. We directed BioOpera to stop
sending new TEUs. Existing TEUs were allowed to complete.

**Event (2)** This is the sole occurrence of a BioOpera server “crash” due to a communication protocol problem and lack of robustness on the side of the BioOpera communication server. This problem has since been corrected. Once the problem was resolved, BioOpera smoothly resumed computing where it left off.

**Event (3)** A failure in the cluster terminated all BioOpera clients and Darwin programs. The BioOpera server correctly classified all active jobs as “failed” but could not re-start them since the clients were not running. This problem was solved by specifying at the operating system level that the BioOpera client should be started at boot time.

**Event (4)** Again, several failures in the cluster. Manual intervention was necessary to restart the BioOpera client.

**Event (5)** At this point we ran out of disk space. All active TEUs slowed down considerably (producing the gap between the light and medium grey curves). The increase in failed activities (dark grey area) is due to the fact that they were manually terminated until the storage space problem was resolved.

**Event (6)** The process had to be stopped to modify the process specification to write to the location of the new disk. Future versions of BioOpera will better separate the configuration information from the process specification in order to avoid the need for such intervention.

**Event (7)** The computer running the BioOpera server was shut down for maintenance. All active TEUs were registered (correctly) as failed.

**Event (8)** The BioOpera server is re-started smoothly.

**Event (9)** 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 failures.

**Event (10)** Due to a problem with the application,
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.

We feel that these results are very positive, especially considering that BioOpera is only at a prototype stage. All of the errors discussed here were easily corrected in each of the 7 manual interventions2.

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.

Conclusions
The results of the experiment with the all vs. all for SP38 show the BioOpera is already a useful tool for managing large-scale computations in bioinformatics. BioOpera is still a prototype but it is rapidly growing and becoming more functionally complete. Future extensions include a graphic user interface for process definition and monitoring, a more complete awareness model, and improvements to the architecture to support process replication, migration and backup.

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 (Hallett and Lagergren, 1999; Hallett and Lagergren, 1999b). 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

2 As an added bonus, BioOpera proved to be Y2K complaint: the transition from 1999 to 2000 was one of the most active periods.


