

Chapter 4

Applications of Higher-Order Components

This chapter shows how HOCs are used in large-scale applications that require multiple computational resources. The introductory examples on HOCs in Chapter 2 and the HOC-Service Architecture (HOC-SA) in Chapter 3 were concerned with processing number series, string data and images or matrices, following quite simple rules, and the computational load was only created synthetically by copying input data for increasing its size and repeatedly running the same processes.

Contrary, this chapter shows two applications that tackle real-world problems and follow much more complex computational patterns:

(1) The first application, shown in Section 4.1, called *Clayworks*, deals with the simulation of the deformations resulting from the crash of clay objects moving in a virtual space [bG⁺09]. *Clayworks* is a software system which integrates collaborative real-time modeling and distributed computing. It addresses the challenge of developing a collaborative workspace with a seamless access to high-performance servers. *Clayworks* allows users to model virtual clay objects and to run compute-intensive deformation simulations for such objects crashing into each other. To integrate heterogeneous computational resources, modern grid middleware is adopted and users are provided with an intuitive graphical interface. The parallel computation of simulations is done using a specific, new HOC, called *Deformation-HOC*, as it can compute deformations of different kinds of material, expecting as its input objects, movement vectors, velocities and material characteristics. *Clayworks* is a representative of a large class of demanding systems which combine collaborative modeling with performance-critical computations, e.g., in engineering applications (car crash tests), biological evolution, or geophysical simulations.

(2) The second application, described in Section 4.2, deals with genome string analysis. Section 2.4 already introduced the basic string alignment computation as an example for adapting HOCs via code parameters. The application shown in this chapter makes use of the alignment computation, as one part of a genome analysis process which is more complex than the basic string alignment: users define the pre-processing of the application data (protein strings), a specific alignment procedure, and the postprocessing of the output data (rating matrices) for searching in large databases for non-trivial genome similarities, i.e., string permutations which may in-

clude non-linear rearrangements. Genome analysis in bioinformatics represents another demanding class of real-world applications, where Higher-Order Components can significantly reduce the developments costs. An extensible genome processing HOC, called Alignment-HOC is presented in this chapter and experimentally evaluated on the grid. During the experiments with the Alignment-HOC, genome similarities in the ProDom database were detected that were not known previously. The ProDom database is a standard source for biological research and it contains about 70 MB of genome data. The knowledge about the newly detected similarities helps to better understand protein evolution and the functionality of protein sequences.

Both applications have high demands for computing power due to the data masses they process and are, thus, typical candidates for distributing computations in the grid. In both applications, we focus on programming methodology.

This chapter starts with the introduction of *Clayworks* – the first large-scale application case study that addresses the challenge of integrating the different computation and communication requirements of tightly coupled collaborative work and loosely coupled HPC applications – in Section 4.1. The different communication technologies that *Clayworks* employs for connecting between a central server, the clients and the grid are addressed, as well as the challenges that arise when these technologies are integrated with each other. The distributed 3-tier architecture of *Clayworks*, is described and the Deformation-HOC and its parallel implementation on top of the Globus middleware are explained.

The Alignment-HOC – a HOC for bioinformatics – is discussed in Section 4.2. The section introduces specific genome similarities, called *circular permutations* (CPs), which can be detected using the Alignment-HOC. It is shown how users can modify the Alignment-HOC for performing, instead of a general genome analysis, a search for circular permuted proteins, including experiments with real biological databases.

Conclusions from using HOCs in large-scale applications are drawn in Section 4.3

4.1 Clayworks: A Collaborative Simulation Environment

With emerging grid computing technologies, it has become a broadly used concept to distribute computations over the Internet in a variety of applications for business, science and engineering, which are often summarized under the term *eScience*. According to John Taylor's definition [cTay02], *eScience* “is about global collaboration in key areas of science, and the next generation of infrastructure that will enable it.” Two important approaches in this area are: (a) *Computer-Supported Collaborative Work* (CSCW) environments which allow specialists to work together on a single project from different locations, and (b) *High-Performance Computing* (HPC) [bM⁺06].

The *Clayworks* system is designed as a distributed worksuite allowing to collaboratively model clay objects and execute deformation experiments with them.

In the modeling mode, several users concurrently model objects in a shared design workspace using virtual clay. Changes to objects are immediately shown at all user clients, which requires soft real-time communication and computation in order to provide a high level of responsiveness. In the simulation mode, clay objects are deformed when they crash into each other, which is simulated using a remotely located high-performance server.

In the CSCW part of Clayworks, several users, each connected via a graphical client to a shared workspace, model objects made of virtual clay:

- Users can create, delete, merge or modify the shape of objects, which will be immediately visualized at all connected clients.
- Objects can be grouped or locked for exclusive access, such that users can easily split up the work on complex objects among themselves and collaboratively create large scenarios involving a lot of objects.
- Objects can be saved to a database and reloaded later in other workspaces and projects, which allows to build a library of reusable objects.

In the simulation part, the computation of the deformations of moving clay objects is started on a multiprocessor remote server:

- Users assign a velocity and a direction vector to each object and define other simulation parameters.
- For each object, it can be defined whether the object is solid or should be deformable.
- Users can view the result of the simulation as a movie and freely move the camera and scale and rotate the scene.

Clayworks is a result of combining techniques from two active research areas: collaborative environments and HPC. Distributed collaborative applications allow experts from different locations to work together on a single project, while HPC, and especially the recent grid computing research, deals with accessing remote computational resources like processing power or storage space in a transparent way.

Deformation simulations among the moving objects can be executed, using the deformation algorithm described in [bDC04] as a basis. However, since the computation requires a lot of memory and processing time for larger scenarios, the algorithm was parallelized and embedded into a HOC. This setup enables the use of a server with a multiprocessor architecture for running the computations efficiently (see Section 4.1.2 for details).

This section gives an overview of the target application and the design of Clayworks. The 3-tier architecture (Client, CSCW-server, parallel computation) designed for the Clayworks implementation is introduced. The main operations which can be performed by the users are explained and the integrated workflow of cooperatively modeling clay objects and simulating their deformation in Clayworks is discussed.

4.1.1 The 3-tier Architecture of Clayworks

The main challenge in the development of Clayworks was the integration of the CSCW and the HPC part, which in some sense have contradictory requirements: The CSCW part is a soft real-time system which requires timely communication and computation for a high responsiveness of the application. However, the computations for the modeling are not very intensive and can be executed locally on a standard, modern desktop PC. In contrast, the simulation algorithm, which iteratively moves and deforms the clay objects, needs one or multiple HPC hosts with a very high computational power, but has no real-time requirements for the communication.

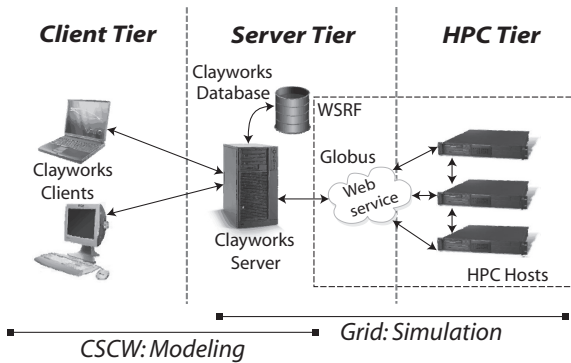


Figure 4.1 The 3-tier Architecture of Clayworks

In order to integrate these different requirements of the CSCW modeling and HPC in Clayworks, the 3-tier architecture shown in Fig. 4.1 was developed, allowing to build up each tier upon a different basis technology: the HPC tier is based on grid middleware, the server tier is based on RMI and JDBC [cSM06b], and the client tier is based on Java Swing. The server tier is in the middle, consisting of the Clayworks server and a database put in between the user clients and the HPC tier, thus decoupling the clients from the HPC host, i.e., users can store and reload 3D-objects without connecting to the HPC hosts. The use of a Web service for interconnecting the Clayworks server and the HPC host enables to flexibly exchange the HPC host according to the application requirements without affecting the client, as indicated by the dashed lines in the figure. This makes sense, when, e.g., instead of multiple compute nodes (e.g., of a cluster, as indicated in the figure) the parallel computations are outsourced to a single grid server, such as the SMP machine used in the experiments in Section 4.1.2.

The clients together with the server form the modeling part which is optimized for immediate communication, while HPC host(s) constitute the high-performance computing facility of Clayworks for the deformation simulations.

In the following, the three tiers and their respective functionality are briefly discussed:

Client Tier

The clients run at the users’ desktop computers and provide an integrated access to all functionality of Clayworks. Users can connect to a shared workspace on a Clayworks server, model objects and observe, as well as chat about the work of other users in real-time. The Clayworks client makes use of OpenGL via the Java 3D API

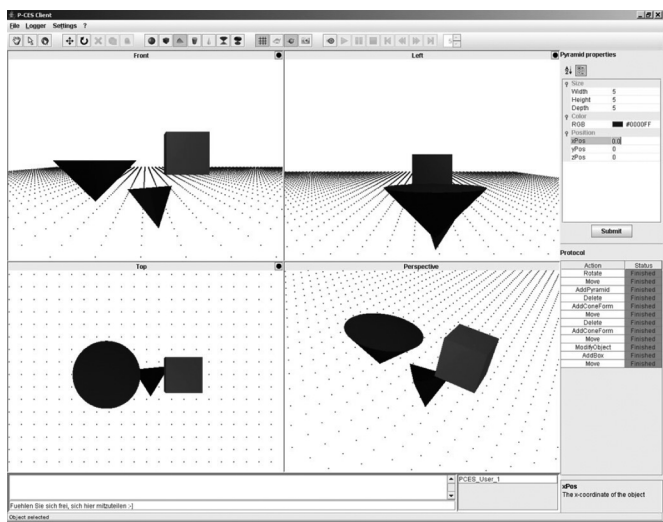


Figure 4.2 Client GUI

for the visualization of the workspace. It is important to notice that the clients are nothing more than visual terminals (shown in Fig. 4.2) which never perform any actions affecting data in the shared workspace locally.

The connection between the clients and the HOC is established via a Web service that is supplied with data and code parameters by sending it the corresponding identifiers. In the Clayworks application, a HOC is used for computing the simulation sequences; the identifiers which users send to it refer either to some material-specific transformation code (i.e., a standard, application-specific code parameter) or they refer to a 3D-object representation in a database.

Thus, the clients do not communicate the input data to the HOC directly in this application, but they communicate it to the Clayworks server, responsible for the synchronization of the data. The HOC independently downloads its input from the database.

The described decoupling between the Clients and HOC allows to maintain different representations of the workspace data (i.e., the coordinates and measures of

the 3D-objects) at the Clayworks server and the HPC hosts running the HOC in the grid. Section 4.1.2 describes the voxel-based representation maintained inside the HOC used by Clayworks. The representation on the Clayworks server is rather hierarchically than graphically oriented. Instead of a polygonal mesh, only the types of the 3D-objects (sphere, cone, etc.) and their connections are stored using a tree structure (the *octree* [cBI06] which is quite common in computer graphics), optimally for the communication of modification commands in the network: When an object is affected by a change triggered by a client, not all the coordinate data, but only the modification command needs to be exchanged over the network. The HOC and the HPC servers in the grid are not involved in such client-side interactions at all. This optimized data exchange methodology prevents effects of jitter or network latencies for the clients.

Server Tier

This middle tier realizes the functionality between clients and the HPC host(s), which is not feasible to be run on the HPC host(s). In particular, current Web services-based middleware provides no possibility to implement real-time interactions of users; therefore, all real-time communication of the collaborative modeling is handled by the Clayworks server. The database, residing at the Clayworks server, allows to reuse objects and to recover a workspace in case of a server failure. Since objects and workspaces are persistently stored on the server side, users can work asynchronously and leave or join a session at any time. Once connected, all changes to the 3D-objects performed by any user are communicated to all users synchronously. Distributed consistency is guaranteed using a remote command queue which makes a replicated representation of the object data unnecessary [bM⁺06]. Principally, the object synchronization is implemented by centralizing the control over all objects at the server: before any command takes effect on the client side, it is communicated via RMI to the Clayworks server which locks all objects until the most recent changes on them have been committed and communicated back to all clients, similar to a 2-phase commit in a distributed database system [aGR93].

For simulations on the HPC host, the server prepares datasets for the remote computation; it starts and monitors the progress of the computations performed by the remote HPC host and finally returns the result to the clients.

HPC Tier

The HPC tier of Clayworks runs a parallel implementation of the clay deformation algorithm [bDC04] used for the simulation. This is realized using a HOC called Deformation-HOC: the Deformation-HOC allows its user to specify only material characteristics while it shields the user from the parallel algorithm (and its realization as a grid component) used to compute a simulation. The result is transferred back to the clients as a sequence of single images (see Fig. 4.3 for an example).

Users pick a default material definition offered by the Deformation-HOC implementation (e.g., the one for clay or solid objects) or define a new material, by specifying a material-specific density shift operation, which is a sequential step in the algorithm.



Figure 4.3 Screenshots of a Simulation Sequence

When users request the computation of a simulation, the Clayworks server commits all relevant data to the database (object coordinates and object measures, and additional simulation information, like the objects' velocities and movement direction vectors) and returns the corresponding database keys. These keys are sent to the Web service which connects the clients to the HOC which is used for computing the simulation in the grid. The deformation algorithm then iteratively moves and deforms the objects, resulting in a sequence of single simulation “screens”. Upon completion of the simulation, i.e., when all objects have spent their kinetic energy for movement and deformation, the server downloads the simulation results from the Web service and sends them to the clients. To reduce the size of the transferred data, the ZLIB-compression utilities from the `java.util.zip` package are employed. Due to the low number of differences between subsequent pictures in an animation, the size of the result files, which must be transferred over the network, can typically be less than a hundredth of their original size in average, enabling a quick transfer between the Clayworks database and the HOC.

4.1.2 The Deformation-HOC for Parallel Simulations

The simulation algorithm employed by Clayworks can be used for the parallel implementation of a broad class of spacial simulations. It is not only suitable for crash tests but for computing realistic presentations of any scene wherein an arbitrary material is deformed, e.g., the diffusion of gases or the freezing of fluids. The algorithm exhibits a recurrent control flow structure. Therefore, it is a typical candidate for a reusable implementation as a HOC. This section presents the concept and implementation of the Deformation HOC, which was developed for this purpose.

The algorithm implemented by the Deformation-HOC in Fig. 4.4 represents an enhanced version of the algorithm from [BDC04]. The HOC runs the algorithm in parallel on multiple partitions of the object data and, in contrast to [BDC04], it does not require the presence of a solid tool object, i.e., all objects in the virtual universe can be built of clay.

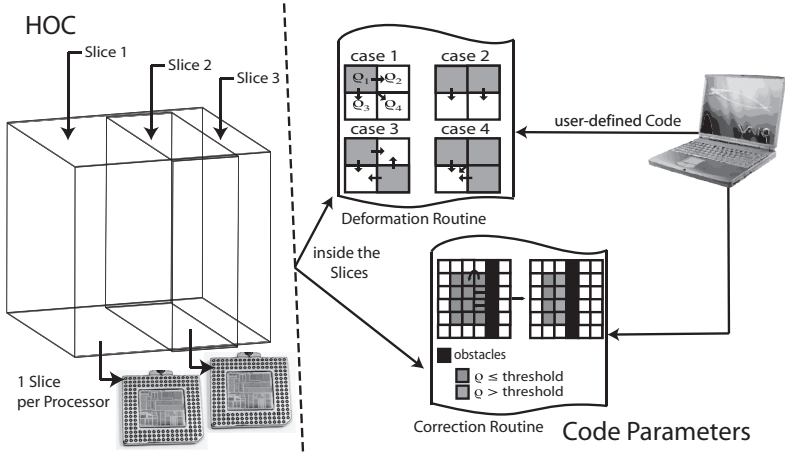


Figure 4.4 Computation Schema of the Deformation-HOC

Data Representation in the Deformation-HOC

Specifically for the Deformation-HOC, there is, besides the octree-based *polygonal* representation which the users collaboratively work on, another *voxel-based* data representation in a three-dimensional grid. Each voxel represents a cell in the simulation space as a data record holding its coordinates and attributes such as the material density, etc.

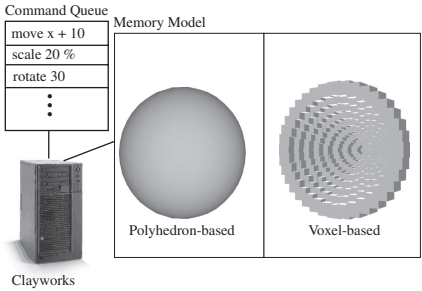


Figure 4.5 Two Object Representations on the Clayworks Server

Figure 4.5 shows a sphere-shaped object in the two different formats: the polygonal version is shown left and the coarser presentation on the right illustrates the voxel-based version. In the cut-away view of the voxel-based version, it can be seen that only the visible external part of the object is covered by voxels while its inside is unfilled, thus reducing the data size of this format. The polygonal representation is used in the modeling part and allows to build the objects in an intuitive way by

defining vertices and faces. The voxel-based representation is required by the parallel simulation algorithm in the Deformation-HOC.

Upon the start of the simulation, the Clayworks server converts the polygonal objects into the discrete voxel-based representation using the algorithm described in [bH⁺98]. This initialization step forms only a very small fraction of the simulation computation. It is therefore performed directly on Clayworks server. This way, only a single representation of the object data is required on the remote HPC host(s) in the grid which perform(s) the largest part of the computations in parallel. Upon completion of the parallel computations, the Clayworks server runs the *Marching Cubes* algorithm [bLC87] for re-polygonalizing the objects on each scene of the simulation sequence.

The Multithreaded Deformation-HOC Implementation

In the following description of the Deformation-HOC, it is assumed that the component is implemented as a multithreaded Java program. This way, the Deformation-HOC was implemented for the experiments described below and a single SMP-server that maps Java threads to different processors was used as the HPC hosting platform. However, many alternative implementations of this component are possible (e.g., for a cluster or a network of workstations) and can be connected to Clayworks using the same Web service.

In the presented version, the Deformation-HOC can also be used to compute multiple independent simulations on multiple distributed grid servers simultaneously, but each single simulation is then computed using one dedicated server.

The Internal Multilayer Process

Internally, the Deformation-HOC performs parallel computations which are invisible to the users: the voxel-based representation of the objects is held in a 3-dimensional array, the *cubic voxel universe*. Each element of this array is an instance of the `Voxel`-class, holding the two attributes `density` and `velocity`. For each voxel, the `density` attribute stores the density of the clay at the voxel. All voxels have the same constant volume, which depends on the granularity of the cubic voxel universe, which must be specified before the simulation starts. Before the start and after the completion of the algorithm, the clay density of all voxels ranges between 0 and 1, but during the computation, values above 1 are possible. Voxels with a density value above 1 are called *congested*. The impact of congested cells on the computation is discussed below. The second attribute of the `Voxel`-class, `velocity`, is a vector describing the movements taking place in the universe. These movements are always linear shifts, triggered by the Clayworks users.

In the Deformation-HOC, the clay deformation algorithm [bDC04] is extended by simulating energy reduction, which terminates the deformation process in a natural way without requiring the user to stop it.

The Deformation-HOC performs a 3-layer traversal of the simulation space (i.e., three passes). In each single traversal, all voxels are visited once and new density values are computed for them. During the first layer traversal, the shifting of voxels is computed. As long as the position of a voxel after a shift does not collide with the position of another voxel, there is no clash and the direction of the velocity vector belonging to this voxel is preserved. The magnitude of the vector is altered by multiplying it with an energy reduction factor α in each traversal. To find a good assignment for the energy reduction α , multiple different factors were tested, including simple linear ones. Empirically, it was verified that the reduction of energy during a shift is simulated very realistically when the exponential function is used to compute the energy reduction factor α , as follows:

$$\alpha := \exp\left(\frac{-1.0 * \rho_1}{\rho_2}\right) \quad (4.1)$$

where ρ_1 and ρ_2 are the density values of the clashing voxels. When voxels clash into each other, a new velocity vector is computed using the formula

$$\delta := \frac{1 - \alpha}{2} \delta_1 + \frac{1 + \alpha}{2} \delta_2 \quad (4.2)$$

where δ_1 and δ_2 are the original velocity vectors of the clashing voxels. α adheres to definition (4.1) reflecting the intensity of the clash, which is reduced proportionally to the reduction of energy.

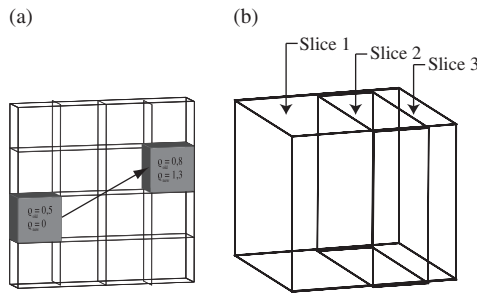


Figure 4.6 Density Shift and Sliced Partitioning

The computations in this layer are performed as a simple vector addition, allowing for a density congestion in the resulting voxels (see Fig. 4.1.2, where a shift between two voxels results in a density above the example threshold 1). The purpose of the successive layers is to adapt the results of the first layer, such that congested voxels distribute their density overplus among their neighbors.

Each layer of the deformation algorithm is parallel in nature, as the cubic voxel universe can be partitioned into sub-cuboids (slices) as shown in Fig. 4.1.2, and each layer can be applied in parallel to the slices, while only the ordering of the layers

must be preserved. The partitioning depends on the distribution of the clay objects specified by the user. Synchronization and communication is required whenever an operation inside one slice affects voxels in another slice. The irregularity of this partitioning is due to the load-balancing: smaller slices (slices 2 and 3 in the figure) contain more congested voxels, such that the number of dependences between slices is approximately balanced.

Code Parameters of the Deformation-HOC

In the *Deformation-HOC*, only the procedures during density shifts in the first layer and the procedures to correct overfull voxels in the remaining layers are application-specific. Therefore, the *Deformation-HOC* takes one customizing parameter which carries a serialized Java class.

```

1:  public interface DeformationParameter {
2:      public boolean isSolid(int objectID);
3:      public double getMaxDensity( );
4:      public double powTrans(int distance, int power);
5:      public double densTrans(int layerNr, double dens); }

```

Figure 4.7 Java Interface for a Deformation-HOC Code Parameter

This class must be accessible using the interface shown in Fig. 4.7. The methods in this interface are as follows:

`isSolid` (line 2): returns, for a given `objectID`, whether the respective object is built of a solid material and therefore not affected by the deformation process;

`getMaxDensity` (line 3): determines when a voxel is overfull. When a value above 1 is specified, less corrections must be performed, speeding up the higher-layer computations, but the accuracy of the simulation is reduced this way;

`powTrans` (line 4): converts kinetic energy into deformation energy, as it happens during the deformation process;

`densTrans` (line 5): is used to distribute density among neighboring voxels in layers > 1 , i.e., within a correction.

Figure 4.8 presents a closer view at the parameter code (familiar from Fig. 4.4). The `DeformationParameter`, used to customize the *Deformation-HOC*, is composed of a deformation routine (Fig. 4.8, left) and a correction routine (Fig. 4.8, right). The *Deformation-HOC* always tries to move mass out of a congested voxel, as shown for case 1 (in Fig. 4.8, left) first, and, if not possible (depending on the density values of the neighboring voxels) choosing, as an alternative, case 2, case 3 or case 4 (in this order), since, this way, mass movements result in as little new congested voxels as possible. The user-defined `powTrans` conversion is called to compute the density updates for each voxel. Figure 4.8, right, shows how a single correction layer works, i.e., the process which is repeated until no congested voxels remain.

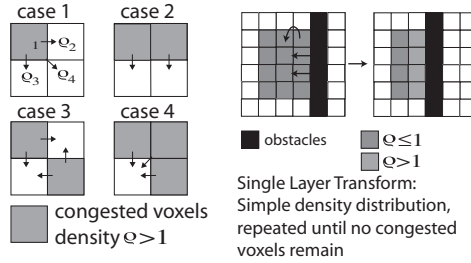


Figure 4.8 Modifying the Density Shift Inside the Deformation-HOC

If no `DeformationParameter` parameter is specified by the user, the Deformation-HOC uses the `DefaultDeformation` class, wherein, e.g., the `powTrans`-method is defined using the exponential function, as shown in lines 2–3 of Fig. 4.9.

```

1: double powTrans(int distance, int collisionPower) {
2:   return Math.exp( (-1.0 / (collisionPower * 2) )
3:     * (distance * 2); }

```

Figure 4.9 Default `powTrans`-method for Clay Deformation

Also the other methods from the above interface are defined following the definitions (4.1) and (4.2) in Section 4.1.2; i.e., as long as no different `DeformationParameter` is specified, the *Deformation-HOC* computes the deformations for objects of a clay-like material.

Writing a specific `DeformationParameter` for a different application requires some knowledge about the physical properties of the material which is simulated. When, e.g., a steel brick is simulated, the required `DeformationParameter` is derived from the `DefaultDeformation-parameter` and the methods `isSolid` and `powTrans` are overridden, such that `isSolid` returns `true` and `powTrans` and `densTrans` methods return constants, depending on the weight of the brick.

The *Deformation-HOC* facilitates code reuse and customization. Its full implementation comprises approximately 600 lines of Java code plus 300 lines of XML code for the WSRF support. When a new `DeformationParameter` is derived from the `DefaultDeformation-parameter` for different simulations, a programmer using the *Deformation-HOC* has to write no more than 20–30 lines of Java code.

Experiments with the Deformation-HOC

Some experiments were conducted using a SunFire 880 SMP server with 8 UltraSparc-III processors running at 1200 MHz.

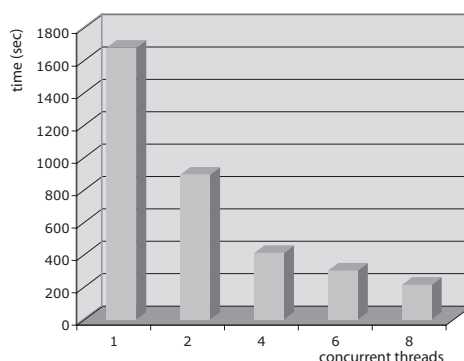


Figure 4.10 Experimental Results

Figure 4.10 shows the average runtime for computing a deformation of two clay cubes (see Fig. 4.3). As can be seen, there is an almost linear speedup for up to 4 concurrent thread. For a higher number of concurrent threads similar results can be expected when also the problem size (i.e., the number of object in the simulation space) is increased. For 8 threads, e.g., a speedup of 7.2 was measured, for a “good” case, i.e., for a scene wherein the objects were evenly distributed.

In the Clayworks example the HOC processes data which is produced by multiple users who concurrently manipulate objects. The following section presents an example where the input data is taken from different databases, selected by the users. Since the databases can be quite large, the data masses the HOC has to cope with are considerably higher than in the Clayworks application.

4.2 Protein Sequence Analysis with HOCs

This section deals with another real-world grid application, the analysis of large amounts of genome data, and introduces, for this purpose, the Alignment-HOC. The Alignment-HOC offers the distributed implementation of a generic alignment algorithm and allows users to plug in their own code, which makes it easy to study genome similarities in different databases or to run a specific genome processing algorithm on a biological database (e.g., 3D structure prediction).

4.2.1 The Alignment Problem in Bioinformatics

Genome processing algorithms which are used for sequence alignment or protein structure prediction typically compute one or more result matrices and have the time complexity of $\mathcal{O}(n^2)$ or higher for sequence length n [bNW70, bSW81]. Working on a large amount of data sequentially quickly causes an unreasonably long runtime.

Therefore, the calculation power of multiple networked computers is indispensable to run pairwise analyzes on entire databases with a typical size of almost 100 MB. Depending on algorithm and database, a genome analysis can take several months of calculation time on a standard computer. Unfortunately, most of the available software for Genome processing software is developed for single PCs [bW⁺05] (or homogeneous multiprocessor clusters [bK⁺02, bS⁺04]). Porting such software to the grid requires additional time and redundant re-implementations of the same or similar software, distracting the programmer from developing new algorithms which are interesting for biologists.

Section 2.4 presented a solution for adapting the Farm-HOC to compute sequence alignments for single genome pairs in parallel on the grid. While sequence alignment is fundamental to genome processing applications, most genome analyses are more complex and require, besides the processing of a multitude of genome sequence pairs, e.g., a preprocessing of the sequences and a postprocessing of the output. This section presents a new HOC, specifically developed for genome processing. While the user is free to arrange custom pre- and postprocessing operations of the input, this HOC always computes an alignment and is, therefore, called Alignment-HOC. The purpose of the code parameters of this HOC is to customize the Alignment-HOC for different kinds of similarity detections. This is demonstrated in the following by a case study of searching so-called *circular permutations* in genome databases. Using an appropriate post-processing parameter (called *traceback*), the Alignment-HOC is also optimized for improving its reliability.

4.2.2 Circular Permutations of DNA

Circular permuted protein sequences (CPs) occur in a number of protein families [bW⁺05] and can be found in all large databases of protein data. Their linear order may be quite different, but the 3-dimensional structure of their resulting protein and its biological functionality are often the same.

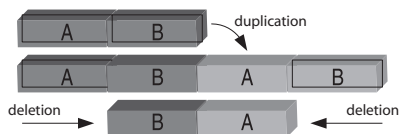


Figure 4.11 Possible Development of a Circular Permutation

In Fig. 4.11, *A* and *B* are arbitrary subsequences of a protein sequence. The figure shows an example of one possible development of circular permutations, by doubling the original sequence and inserting afterwards new start and end codons (tri-nucleotide codes that define the begin and end of a gene expression [bJe99]). Such a circular permuted sequence consists of two parts from the original sequence,

but in a different order. Another way of CP development is assembling two proteins from existing sequence fragments [bBu02].

The problem with circular permuted sequences is caused by the non-linear rearrangement of the amino acid order. Different from other mutations like insertions or deletions of single amino acids in the linear sequence string, a CP shifts the beginning of the sequence to the end, thus, radically changing the original linear order of the sequence. Standard alignment algorithms will not detect a significant similarity between the original and the circular permuted sequence (shown in Fig. 4.12), although the tertiary structures (the 3-dimensional folding of the amino acid chain) of the resulting proteins may be nearly the same.

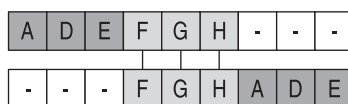


Figure 4.12 Standard Alignment of a Circular Permuted Sequence

Similarities between circular mutated sequences and their related sequences are usually not marked in sequence databases, making it difficult for a scientist to work with such sequences. In order to find these similarities, the whole database must be processed by a specialized algorithm adjusted for the non-linear sequence rearrangement of CPs.

4.2.3 The Alignment-HOC and its Code Parameters

The Alignment-HOC (see Fig. 4.13) was developed for genome analyses like, e.g., the detection of circular permutations (as explained above). Actually, the Alignment-HOC can be used for detecting CPs (i.e., sequences which are highly similar after the CP has been made undone) for different kinds of input: either for sequences of protein domains (domains are functional units in a protein consisting of up to several hundred amino acids) or for the underlying amino acid sequences. Because domains consist of many amino acids, the domain sequences of a protein are significantly shorter than its amino acid string. This reduces the complexity of calculations performed on domain data but also reduces the amount of protein information available for the similarity detection.

Both CP detection algorithms have their advantages and disadvantages in different applications. Therefore, the user can vary between both versions via the code parameters of the Alignment-HOC. Switching between both versions (via a Web service, as explained in Chapter 3) only affects the operations that differ when either domains or amino acid strings are being processed, while the main component code is reused in any application.

The Alignment-HOC has three code parameters.

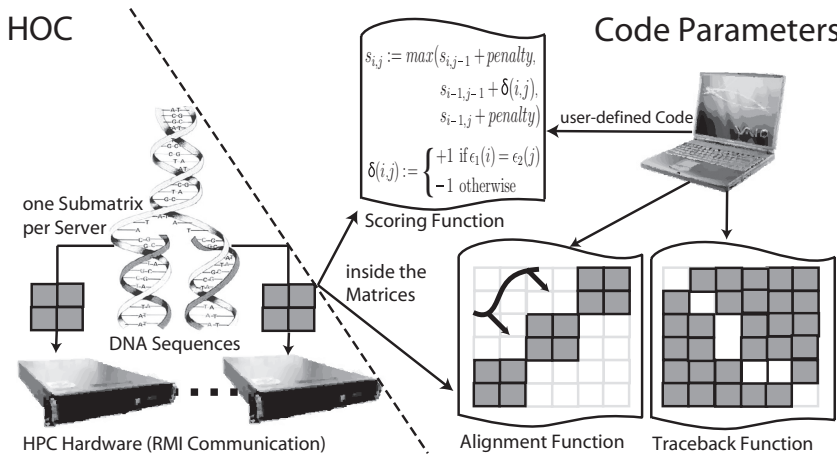


Figure 4.13 Computation Schema of the Alignment-HOC for DNA Similarity Detection

Parameter 1: Scoring Function

Via this parameter users can modify the scoring function, i.e., the operations used to compute a single element in a scoring matrix using the same *Binder*-interface as shown in Fig. 2.31 (Section 2.4.2). In this matrix, each element rates differences between pairs of protein residues (i.e., elements of the genome code) and is applied to the two subsequences, delimited by the matrix elements' indices. In most cases, the scoring function complies with the default definition for a Needleman Wunsch Alignment [bNW70] (shown in Section 2.4.2). However, the Alignment-HOC is more flexible than the simple Wavefront-HOC (i.e., the adapted Farm-HOC from Chapter 2). The Alignment-HOC does not only allow its users to vary gap penalties [bH⁺90] but also a different scoring schema can be implemented, e.g., according to the lengths and frequencies of collective insertions and deletions. Such a scoring schema can lead to a more accurate alignment, since the evolution of genomes involves more complex processes than point mutations (i.e., a change of a single base into another base [bCa07]).

Parameter 2: Alignment Function

This parameter allows the user to control the course of actions in a genome processing application using the Alignment-HOC. For this purpose, users may implement the interface shown in Fig. 4.14. If a standard Needleman Wunsch Alignment should

```

1: public interface Aligner extends Serializable {
2:   public char[] preprocess(char[] sequence);
3:   public char[][] align( char[] seq1, char[] seq2,
4:                         boolean postprocess ); }

```

Figure 4.14 Aligner-Interface for the Alignment Function

be computed, there is no need to care about these methods, but the Alignment-HOC will run the default implementation (based on the JAligner library [cAM07]). In a custom alignment, users may convert the input sequences in the `preprocess`-method (line 2), e.g., for filtering out information that is not relevant in an application or an on-the-fly translation from DNA to amino acids or vice versa. The `align`-method allows users to adapt the computation of the scoring matrix, e.g., for optimizing it for a particular grid platform type: as shown by experiments in Chapter 2, computing a single scoring matrix using the wavefront pattern scales well on shared-memory servers, while computing multiple independent matrices in parallel is preferable in a distributed-memory network. The `postprocess`-flag (line 4) is used to enable/disable a postprocessing, defined via the `traceback` function which is supplied to the HOC as its third code parameter.

```
1: public interface Traceback extends Serializable {
2:   public int[] traceback( char[] directionMatrix,
3:                         int width, int height); }
```

Figure 4.15 Traceback-Interface for the Traceback Function

Parameter 3: Traceback Function

Via this parameter, the user can specify how a traceback path is detected, i.e., a path following high values in the scoring matrix which starts at the bottom/right side of the matrix. Traceback paths are encoded as flat arrays of coordinates, as obvious from the interface shown in Fig. 4.15. The default traceback results in one main path that always starts in the most bottom right element (the *global* score which rates the similarity of the sequences over their total length) and runs in a more or less diagonal manner through the whole scoring matrix. Typically, there are variations in the diagonal path, since the compared sequences have point mutations which are reflected by an uneven distribution of high scores in the matrix. Figure 4.16 (left) shows a traceback running straight along the middle diagonal, indicating that the compared sequences consist of the two equal chains *A* and *B*. Via the traceback function, Alignment-HOC users can modify the processing of the scoring matrix resulting from a sequence alignment to perform different kinds of genome similarity detections.

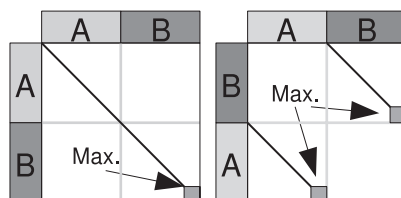


Figure 4.16 Standard Alignment (*left*) and CP Detection Algorithm (*right*)

4.2.4 Using an Alternative Traceback

The right part of Fig. 4.16 shows what the Alignment-HOC does in the traceback step when we specify the traceback code parameter for CP detection (as described in Section 4.2.2) instead of the default traceback. The CP code parameter is included in the Code Service of the HOC-SA, i.e., users can run this kind of genome processing by simply selecting this alternative traceback parameter and are not required to write new code for this application.

In case of a circular permutation between two compared sequences, the first part of each analyzed sequence is strongly similar to the last part of the other sequence (or the parts are even equal, as in the depicted example). Instead of one main traceback path, an alignment between two circular permuted sequences has two traceback paths starting from two *local* maximum scores, one in the bottommost row and the other one in the rightmost column of the scoring matrix (marked in the figure with ‘Max’). If such two maxima cannot be located, since there is almost no variation in the values on the outermost matrix row and column, the sequences are not circular permuted and no further testing is necessary.

When the traceback paths run (almost) along shifted diagonals of the matrix, as in the figure, a CP is detected. The traceback code parameter for detecting CPs tests for the matrix whether the criterion *traceback paths run along shifted diagonals* is fulfilled as follows: since shifted diagonals pass three quadrants of the matrix and never intersect each other, it starts from the two maxima, follows the high scores and counts the number of quadrants passed on these paths. The test is positive if there is no intersection (i.e., a common element) and both paths pass three quadrants.

4.2.5 Optimizations of the Alignment-HOC

As compared to the many other implementations of sequence alignment (including the Wavefront-HOC in Chapter 2), the Alignment-HOC offers some optimizations which were specifically developed to increase the efficiency of CP detection using the Alignment-HOC. The following paragraphs show how these optimizations work and how users can benefit from them by selecting the corresponding code in the Code Service, without dealing with the implementation themselves.

The Alignment-HOC stores in the main memory only the part of the alignment data that is necessary for CP detection. Instead of working on both doubled sequences at the same time, the algorithm processes half the sequences separately and, thus, when a matrix quadrant is computed, only the relevant half of the input is loaded. Moreover, not all matrix element values are relevant in the application, but only the matrix elements which are crucial for choosing the traceback direction, i.e., for each row of the matrix, only the position of the maximum element is stored, as suggested in [bGo82]. The Alignment-HOC allows to apply these application-specific optimizations without reducing the component’s standard functionality. All optimized code is placed inside the code parameter for the alignment function (parameter 2, see Section 4.2.3) which users can replace by the default step on demand.

Doubled Input for Increased Sensitivity

To avoid false-positive detections of CPs, both compared genome sequences should be doubled. before the scoring matrix is calculated. After the duplication the traceback results in extended paths.

Figure 4.17 shows how this path extension is used to increase the sensitivity of CP detection: the lengths of the four line segments which the traceback paths mark by their intersections with the inner borders of the matrix quadrants are compared. Only if the corresponding lengths (i.e., the segments of a single line, such as α and β on the inner vertical border) have nearly the same ratio, the CP test is positive. Doubling the sequences obviously results in a matrix that is four times larger, but experiments justify the higher computation costs: tests without doubling the input [bW⁺05] delivered too many false-positive results.

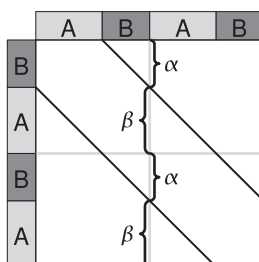


Figure 4.17 Increased Detection Sensitivity

Noise Reduction for Increased Accuracy

The Alignment-HOC can process domains and also amino acid sequences. Because of the high number of different domains that occur in a domain sequence (there are several thousands of known domain strings in biological databases), a match, i.e., the occurrence of equal input elements, in the scoring matrix is relatively rare when domain data is aligned. In the case of aligning amino acid sequences, the probability of a match is 1:20, since only 20 different amino acids exist. A high number of matches in the scoring matrix produces noise, i.e., variations in the distribution of high scores, which makes the detection of the two characteristic shifted diagonals that indicate a CP quite difficult.

Figure 4.18 shows the optimization in the Alignment-HOC that helps to avoid the described similarity noise by rating the matches. Instead of noise suppression in the result matrix, the Alignment-HOC reduces noise while calculating the scoring matrix. Whenever input elements match, a different value than the scoring function output is assigned to the corresponding matrix element, e.g., score 1 for a simple match, score 10 for a double match (i.e., matching elements plus a match in the upper left neighbour cell), score 20 for a triple match and score 50 for four

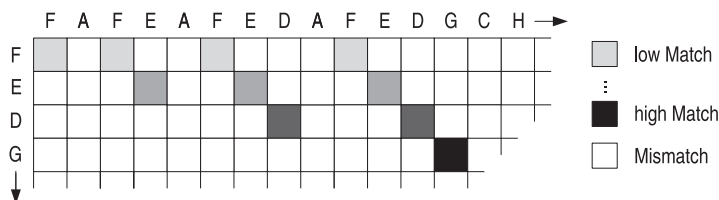


Figure 4.18 Reducing Noise by Using Dynamic Match Values

successive matches on the main diagonal (i.e., top left to bottom right). Thus, the rating counts neighboring matches for rating the degree of a match. The exact values chosen for the rating schema are arbitrary, except for the condition that they range approximately within the scoring function's codomain to effectively suppress low similarities in the aligned sequences. This simple noise reduction schema has been experimentally proven to satisfy the demands of CP detection, while other genome similarity analyses require more sophisticated filtering methods [bH+02] including, e.g., wavelet transform (Chapter 3).

4.2.6 Experiments with the Alignment-HOC

Using the Alignment-HOC, the protein database ProDom was scanned [bL⁺07] and genome similarities were found that were not known previously, since earlier projects never processed the full database.

ProDom Version	Raspodom	Alignment-HOC
2003.1	36	129(40)
2004.1	93	850(192)

Table 4.1 Number of Detected CPs in Different Databases

Table 4.1 shows the number of CPs detected by the Alignment-HOC and, in turn, by a program from a related research project (called *Raspodom* [cW⁺03]) for two versions of the ProDom database. The numbers in parentheses are those CPs where two or more protein domains are involved in the sequence rearrangement. In these sequences, the detected CPs span at least two functional parts of the protein data and, thus, deliver strong evidence that the sequences are closely related.

Two of the newly detected CPs are shown as dot plots in Fig. 4.19. In each dot plot, the two characteristic shifted diagonals (see Section 4.2.3) are observable (compare Fig. 4.16). The new CP, shown in the dot plot on the left, compares the sequences '*Peptide synthetase MBTF*' (O05819) and '*FxbB*' (O85019). Both are parts of mycobacteria genomes, but only the function of '*FxbB*' has been known in the Swiss-Prot database as part of an adenosine monophosphate (AMP) binding enzyme family. The relationship between '*Peptide synthetase MBTF*' and '*FxbB*' of being circular permutations of each other leads to the hypothesis that '*Peptide synthetase MBTF*' is also part of the same AMP binding enzyme family with a similar function.

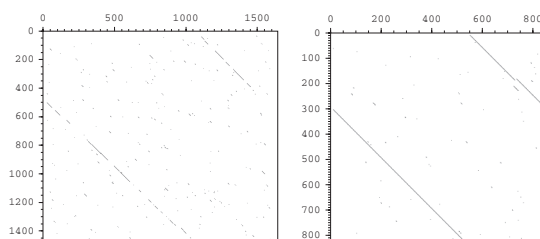


Figure 4.19 Dot Plots of the Newly Detected Circular Permuted Sequences

The dot plot in the right part of the figure compares the two nameless sequences Q69616 and Q9YPV1. Both are listed in the Swiss-Prot database as polymerases, but only Q9YPV1 is a known DNA-polymerase. The newly detected circular permutation between the two sequences indicates that Q9YPV1 is also a polymerase interacting with DNA.

4.3 Conclusions from Using HOCs in Large-Scale applications

In this chapter, it was shown by two practical examples, how HOCs help programmers porting real-world applications to the grid. The first example, Clayworks, is a CSCW application that combines an interactive multi-user environment with high-performance computing. The employed HOC, the Deformation-HOC, is used to outsource the compute-intensive simulation process to the grid. The second example is the genome sequence processing via the Alignment-HOC.

The Alignment-HOC is able to handle the pairwise processing of hundreds of megabytes of data (as present in total genome databases) by distributing the computations. The calculation power offered by the Alignment-HOC makes it possible to keep up with the exponentially growing amounts of data in genome sequence databases used in biological data analysis applications. The development of new problem-specific code parameters can be easily done because only simple Java interfaces have to be implemented. The data distribution is transparent to the user.

The main contribution of the Clayworks worksuite is that it tightly integrates collaborative modeling with a grid component: The Deformation-HOC which computes the deformation simulations in parallel. Thus, end users can collaborate and easily access high-performance grid servers in a transparent way.

When compared to other distributed problem solving environments like CUMULVS [bW⁺06] or NetSolve/GridSolve [bS⁺05], Clayworks' novel feature is the support for tightly-coupled, synchronous collaboration with soft real-time deadlines. The 3-tier architecture of Clayworks satisfies the different requirements of collaborative modeling and of HPC. The real-time requirements are combined with the HPC infrastructure in a transparent way for the end users. Furthermore, the transformation of the clay objects from polygonal to voxel-based representation and vice

versa allows to use the representation best suitable for visualization and computation, respectively.

The concept of seamless integration of remote HPC servers into application software can be expanded into novel areas beyond traditional industrial and scientific applications. In order to make such software suitable for the mass market and non-experts in the area of computing, the access to HPC resources has to be made as transparent as possible. As shown in this chapter, HOCs are a promising step into this direction.

Higher-Order Components for Grid Programming

Making Grids More Usable

Dünnweber, J.; Gorlatch, S.

2009, XIII, 186 p., Hardcover

ISBN: 978-3-642-00840-5