# A Case Study on Pattern-based Systems for High Performance Computational Biology

Weiguo Liu and Bertil Schmidt
School of Computer Engineering, Nanyang Technological University
liuweiguo@pmail.ntu.edu.sg, asbschmidt@ntu.edu.sg

## Abstract

*Computational biology research is now faced with the burgeoning number of genome data. The rigorous post-processing of this data requires an increased role for high performance computing (HPC). Because the development of HPC applications for computational biology problems is much more complex than the corresponding sequential applications, existing traditional programming techniques have demonstrated their inadequacy. Many high level programming techniques, such as skeleton and pattern based programming, have therefore been designed to provide users new ways to get HPC applications without much effort. However, most of them remain absent from the mainstream practice for computational biology. In this paper, we present a new parallel pattern-based system prototype for computational biology. The underlying programming techniques are based on generic programming, a programming technique suited for the generic representation of abstract concepts. This allows the system to be built in a generic way at application level and thus provides good extensibility and flexibility. We show how this system can be used to develop HPC applications for popular computational biology algorithms and lead to significant runtime savings on distributed memory architectures.*

## 1. Introduction

Many computational biology (CB) applications are compute intensive and suffer from long runtimes on sequential architectures. High performance computing (HPC) on PC clusters and computational grids can reduce this runtime significantly. However, because an HPC application is more complex than the corresponding sequential program, to realize this increase in speed some challenges must be overcome first and this daunting task usually falls on a few number of experts.

The development of HPC applications for CB problems can be greatly improved by adopting suitable programming techniques and tools. Recently, many high level technologies have been developed to facilitate the implementation of HPC applications. Among them, parallel design patterns have made a substantial impact on the mainstream practice in HPC programming [9]. Parallel patterns are based on sequential program design patterns used in object-oriented languages. Many parallel algorithms can be characterized and classified by their adherence to a number of generic patterns of computation and communication. By separating the communication structure of a parallel algorithm from the sequential application, parallel patterns allow for a rapid development of HPC applications. As an important difference from the usage of design-level patterns in the object-oriented domain, parallel patterns are often employed not only at the design level but also at the implementation level. That is, the design level patterns are often pre-implemented as reusable components. In the past decade, many parallel pattern-based systems have been developed to employ design patterns related concepts in the HPC domain. Some of the systems based on similar ideas include *Code* [6], *Frameworks* [25], *Enterprise*[23], *HeNCE* [7], *Tracs* [5], *DPnDP* [26], and $CO_2P_3S$ [4]. Unfortunately, most of these systems lack practical usability for the CB field because of the following reasons:

1. Most systems only provide a limited set of parallel patterns, such as pipeline and task farm [9, 18]. These patterns can not meet the requirements of most CB applications.
2. The components of existing systems are not expressed in a compact generic way. If new components are required, the users have to do tremendous work to implement these extensions.
3. They are not flexible enough for the user to reuse the components of the systems at application level.
4. No computational grid oriented pattern-based systems has been developed. With the increased availability of grid computing platforms, grid-enabling of pattern-based systems are of high importance.

| Algorithms | Applications | References |
|---|---|---|
| Smith-Waterman algorithm with linear and affine gap penalty | Genome local alignment | [16, 27] |
| Syntenic alignment | Generalized genome global alignment | |
| Smith-Waterman algorithm with general gap penalty | Genome local alignment | [12, 27] |
| Nussinov algorithm | RNA base pair maximization | |
| Viterbi algorithm | Gene sequence alignment using HMMs, Multiple sequence alignment | [12] |
| Double DP algorithm | Protein threading | [20] |
| Spliced alignment | Gene finding | [14] |
| Zuker algorithm | RNA secondary structure prediction | [32] |
| CYK algorithm | RNA secondary structure alignment | [12] |

**Table 1. Popular DP algorithms in computational biology.**

In this paper, we present a new parallel pattern-based system prototype and its application for the CB area. This system has been developed using generic programming techniques that are suited for the representation of abstract concepts [29]. Because the system is built in a generic way at the design and application level, it has good extensibility and flexibility. We demonstrate how this new system can be used to develop HPC applications with substantial performance gains for popular CB applications.

The rest of the paper is organized as follows: Section 2 describes the task partitioning and communication schemes for several popular CB algorithms. Section 3 presents the system overview. Section 4 shows performance results for several HPC applications on PC clusters and computational grids. Section 5 concludes this paper.

## 2. Mapping of Computational Biology Applications onto Parallel Architectures

In very broad terms, we can identify two high-level categories of CB problems: those that can be addressed analytically, and those that cannot. These two categories are addressed in turn by two respective classes of algorithms: analytical solution approaches and heuristic approaches.

### 2.1. Task Partitioning Schemes for Parallel Dynamic Programming Algorithms

Dynamic programming (DP) is an important analytical solution approach in CB. Several popular DP algorithms are shown in Table 1.

DP views a problem as a set of interdependent sub-problems. It solves sub-problems and uses the result to solve larger sub-problems until the entire problem is solved. In general, the solution to a DP problem is expressed as a minimum (or maximum) of possible alternative solutions. Each of these alternative solutions is constructed by composing one or more sub-problems. If *r* represents the cost of a solution composed of sub-problems $x_1$, $x_2$,. . . ,$x_l$, then *r* can be written as:

$$r = g(f(x_1), f(x_2), \ldots, f(x_l)) \quad (1)$$

The function $g()$ in Eq. (1) is called the composition function, and its nature depends on the problem described. If the optimal solution to each problem is determined by composing optimal solutions to the sub-problems and selecting the minimum (or maximum), Eq. (1) is then said to be a dynamic programming formulation [19].

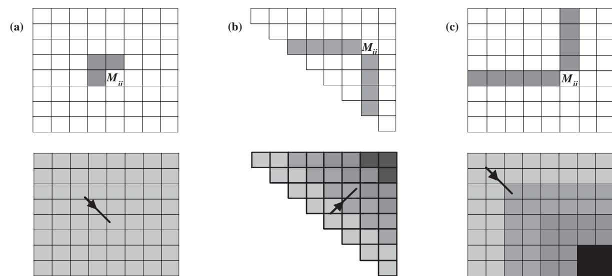


**Figure 1. The dependency relationship and the distribution of computational load density along the computation shift direction for regular DP algorithm (a) and irregular DP algorithms (b) and (c).**

DP algorithms can be divided into regular and irregular categories according to the dependency relationship of each cell on the matrix [13]. Figure 1 shows the dependency relationship and the change of computational load density for some DP algorithms. The change of load density is indicated by using increasingly blacking shades along the computation shift direction.

In order to balance the workload among processors for irregular DP algorithms, we introduce a tunable block-cyclic based task partitioning and communication scheme. The

concept is illustrated in Figure 2. The parameter *division* is used to implement a cyclic distribution of columns to processors. The parameter *rowwidth* is used to control the size of messages that processor $P_i$ sends to processor $P_{i+1}$. Increasing the number of cyclic divisions and decreasing the size of messages can lead to a better load balancing. Of course, doing this also increases the communication overhead. Thus, the choice of the parameter *division* and *rowwidth* is a trade-off between load balancing and communication time.

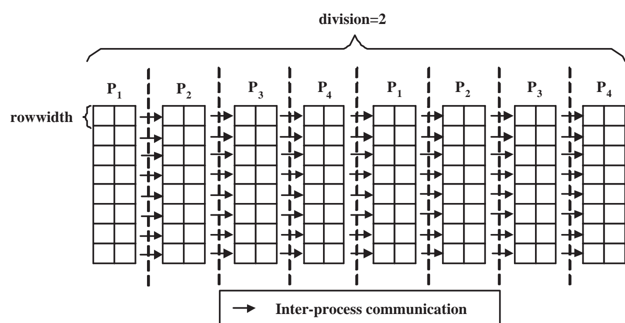


**Figure 2. The tunable block-cyclic partitioning and communication scheme for irregular DP algorithms**

## 2.2. Communication Schemes for Hierarchical Parallel Genetic Algorithms

If we can address a CB problem analytically, this generally means that we know enough about the structure of the search space to reliably guide a search towards the best solution. The more typical situation is that we do not have enough analytical knowledge to do this. Perhaps even more commonly, we can analyze the structure of the problem to a small extent, but not enough to be able to use the knowledge to reliably find the best solution. This type of CB problems fall into the heuristic category.

Genetic algorithms (GAs) are an efficient heuristic search method based on Darwinian evolution. They have powerful characteristics of robustness and flexibility to capture global solutions for complex problems. GAs have been widely used in CB, such as pairwise and multiple sequence alignment[21], protein structure prediction [22], microarray data clustering [10], and feature selection methods for in-silico drug design [31].

Because of their inherent parallelism, GAs are suitable candidates for mapping onto parallel and distributed architectures. Generally, from the point of view of basic communication structures, parallel GAs can be categorized into three main types [8]:

1. Global single-population master-slave GAs;
2. Single-population fine-grained GAs;
3. Multiple-population coarse-grained GAs.

In (1) there is a single panmictic population, but the evaluation of fitness is distributed among several processors. A single master processor does the supervision of the whole population and also does the selection. Slave processors receive the individuals that are recombined to create offsprings. Fine-grained GAs are suited for massively parallel computers and consist of one spatially-structured population. Selection and mating are restricted to a small neighborhood. Multiple-population GAs consist of several subpopulations which exchange individuals occasionally. This exchange of individuals is called migration. In this paper, we mainly focus on the multiple-population coarse-grained GAs since it is the most popular GA used in computational biology.

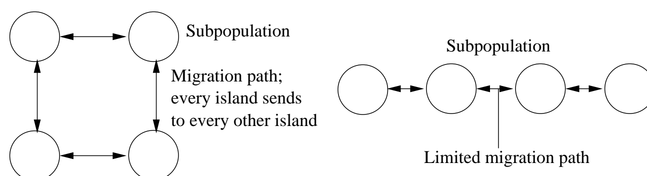


**Figure 3. Migration models for multiple population GAs: (a) The island model (b) the stepping stone model**

There are two popular approaches for modelling migration in multiple-population GAs: the island model and the stepping stone model. In the island model, individuals are allowed to be sent to any other subpopulations (see Figure 3a). It places no restrictions on where an individual may migrate. In the stepping stone model, migration is limited by allowing emigrants to move only to neighboring subpopulations (see Figure 3b). The stepping stone model reduces communication overhead by limiting the number of destinations to which emigrants may travel, and thereby limiting the number of messages. The island model allows more freedom, and in some ways represents a better model of nature. However, there is significantly more communication overhead and delay when implementing such a model [30].

A few researchers have tried to combine two of the methods to parallelize GAs, producing hierarchical parallel GAs (HPGAs). HPGAs combine the benefits of its components, and it promises better performance than any of them alone [8]. Figure 4 shows two communication architectures for HPGAs. The grid architecture considered in

this paper consists of a cluster of PC clusters. Computing resources located in geographically distributed sites are shared in this environment. The grid infrastructures, such as Globus Toolkits [1], are installed on the head nodes of these resources. Inside each site, the resource is managed by the local resource management system. A typical characteristic of this computational grid architecture is the large gap between the fast connection inside a cluster and the slow connection between clusters. Thus applications designed for uniform speed interconnections will lead to performance degradation on the computational grid environment. In order to map HPGAs onto computational grids efficiently, the high level part of an HPGA is mapped onto the grid layer and the low level part is mapped onto the cluster layer (see Figure 5). Different MPI libraries are linked in the two layers. In the grid layer, MPICH-G2 [17] based processes run on the head nodes. They migrate the subpopulations in the local cluster to other clusters within the grid environment. Inside the cluster, MPICH-P4 [2] is used to transfer data between different nodes. Figures 4 and 5 show that HPGAs are very suitable for mapping onto the computational grid architecture. Subpopulations can be exchanged between the grid layer and the cluster layer by reading and writing to two shared memory blocks on the head node of each cluster.

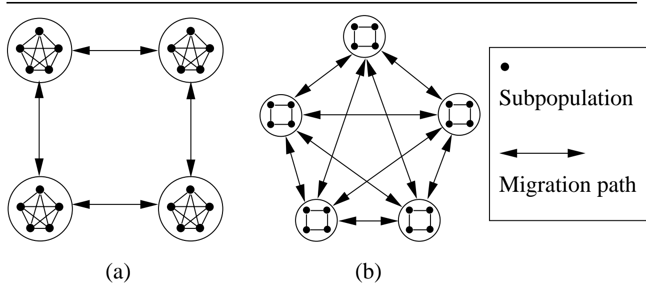


**Figure 4. (a) Hierarchical parallel GA with the stepping stone model at the higher level and the island model at the lower level, (b) hierarchical parallel GA with the island model at the higher level and the stepping stone model at the lower level.**

In this section, we have analyzed the characteristics of some popular CB algorithms. In order to map these algorithms onto parallel architectures efficiently, we have provided the corresponding task partitioning methods and communication schemes . In practice, in order to develop HPC applications conveniently using the described partitioning and communication schemes, we need to integrate them into a system in a generic way. In the next section, we will demonstrate the implementation details of a generic pattern-

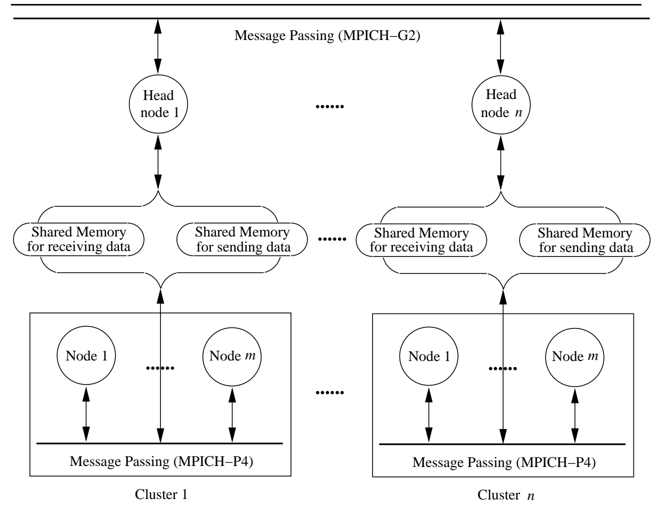


**Figure 5. Communication schemes for HPGAs on the computational grid environment**

based system prototype for the described algorithms.

## 3. The Generic Pattern-based System Prototype

In this section, we will demonstrate the framework of a generic pattern-based HPC system prototype for the two categories of CB algorithms presented in Section 2. The underlying programming technique is based on generic programming which is a program design technique that deals with finding abstract representations of algorithms, data structures, and other software concepts [29]. It can use both the traditional objected-oriented techniques (such as inheritance and virtual functions) and templates. Because of its good flexibility, extensibility and security, generic programming techniques are very suitable for the development of pattern based system. The STL (the Standard Template Library, which later was adapted and incorporated into the C++ standard library) and Janus [15] are two examples of generic programming applications.

### 3.1. System Overview

The code in Figure 6 shows the general structure of the system framework. Concrete parallel applications can be implemented by extending and instantiating the template parameters (lines 1, 2, 3, 4) of `GenericPattern`. These parameters encapsulate the abstract structure and behavior of a set of parallel patterns in an application independent manner. The parameter `AlgorithmIni` initializes some parameters and defines the data structure of the communication message. `SequentialComp` processes the sequential computation. `ParallelCommu` consists of the

```
template<class datatype,
         class AlgorithmIni,
         class SequentialComp,
         class ParallelCommu,
class GenericPattern{
    void HpcComputing(){
        AlgorithmIni::PreProcess();
        while the cyclic loop is not
        completed{
    /*For parallel DP applications,
    SequentialComp is invoked in the
    following method*/
            ParallelCommu::Launch();

    /*For parallel GAs applications,
    ParallelCommu is invoked in the
    following method*/
            SequentialComp::Launch();
        }
        AlgorithmIni::PostProcess();
    }
};
```

**Figure 6. The structure of class template** `GenericPattern`**.**

communication behavior between all processors participating in the parallel computation. By defining these three parameters, the parallel part (`ParallelCommu`) is separated from the sequential application parts (`AlgorithmIni` and `SequentialComp`). Thus, both parts of a parallel application can evolve independently. This allows for the rapid prototyping of parallel applications, and permits users to experiment with alternative communication structures conveniently. Currently, we have integrated the tunable block-cyclic partitioning method for DP algorithms into the system. The system also supports two dimensional and three dimensional applications with nested recurrence relationships. As to parallel GAs, the island and stepping stone communication models have been pre-implemented.

### 3.2. Generic Pattern-based System: Extensibility and Flexibility

An important aspect of our system is the generic representation for a set of patterns, i.e. a generic pattern. With this generic pattern, we mainly focus on the extensibility of the framework rather than how many limited patterns it can support. In order to achieve a good extensibility, the template parameters of `GenericPattern` are also defined as class templates. Thus, the user can extend the generic pattern by specifying these application-independent templates. Different specialization will lead to different implementation strategies for a concrete parallel application.

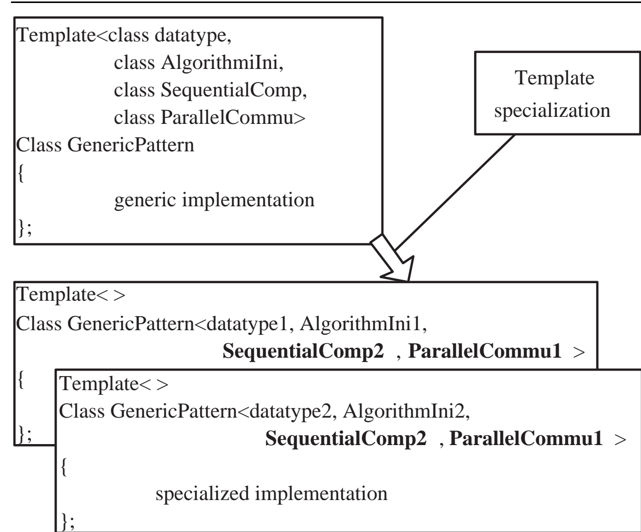


**Figure 7. Extend the generic pattern to implement application-dependent parallel programs.**

Figure 7 shows how to use the generic pattern to develop parallel programs for new algorithms. The user only needs to specify the relevant template parameters according to the characters of the algorithm. In generic programming techniques, this extension is also called template specialization. Our framework provides users with some predefined, efficient and reusable class templates for parallel and distributed computations, thus relieving the user of the need of rebuilding all the error prone template parameters common in parallel and distributed program code. The user only needs to provide the sequential application specific code while the system supplies the necessary parallel code and vice versa. From Figure 7 we can also find that each template parameter is defined independently from other parameters. Yet different template parameters can interact with each other via standard interfaces. Consequently, the system has a good flexibility. For instance in Figure 7, two algorithms share the same sequential and parallel characters. Thus we can entirely reuse the overall design of `SequentialComp2` and `ParallelCommu1` to develop another algorithm. The user can therefore reuse the existing components to develop new applications in a flexible way.

## 4. Performance Evaluations

The system prototype is presently implemented using standard C++ and MPI library provided by MPICH 1.2.5 and MPICH-G2. We have used it to develop high performance applications for some DP algorithms in computational biology and the genetic algorithm for protein folding simulations with the HP lattice model [28].

Two test beds are used in our experiments. One is an Alpha cluster which comprises eight ES45 nodes. Each node contains four Alpha-EV68 1GHz processors with 1GB RAM and 2 MB L2 cache for each processor. All the nodes are connected with each other by a Gbit/sec quadrics switch. Another test bed is a heterogeneous cluster environment. It contains four high-performance clusters. These clusters contain 8 Intel Xeon 2.6 GHz, 8 Intel Xeon 3.0 GHz, 8 Intel Pentium 731MHz and 8 Intel Itanium 733MHz respectively. A 1Gbit/sec Myrinet switch connects each cluster internally and a 100Mbit/sec Ethernet switch is used as an inter-cluster connection. Globus Toolkit™ is installed on the head node of each cluster. Ganglia and Globus MDS [1] are used to collect and present resource information for users.

### 4.1. Experimental Results for Parallel Dynamic Programming Algorithms

We have run the parallel DP applications on the Alpha cluster. Figure 8 shows the best speedups for different number of processors when *division* is set from 1 to 100 and *rowwidth* is set from 5 to 50. It is important to note that these applications are implemented using different methods. The linear space method is used to reduce the RAM needed by the Smith-Waterman algorithm (with linear gap penalty and affine gap penalty) and the Syntenic alignment algorithm for long sequences. Similar space-saving methods are used for the three dimensional applications such as the spliced alignment algorithm. As for the Smith-Waterman algorithm with general gap penalty and the Nussinov algorithm, we store and compute the whole matrix. Notice the super linear speedups are observed in several applications. This is because of the effects due to better caching are different according to different implementation methods for specific applications.

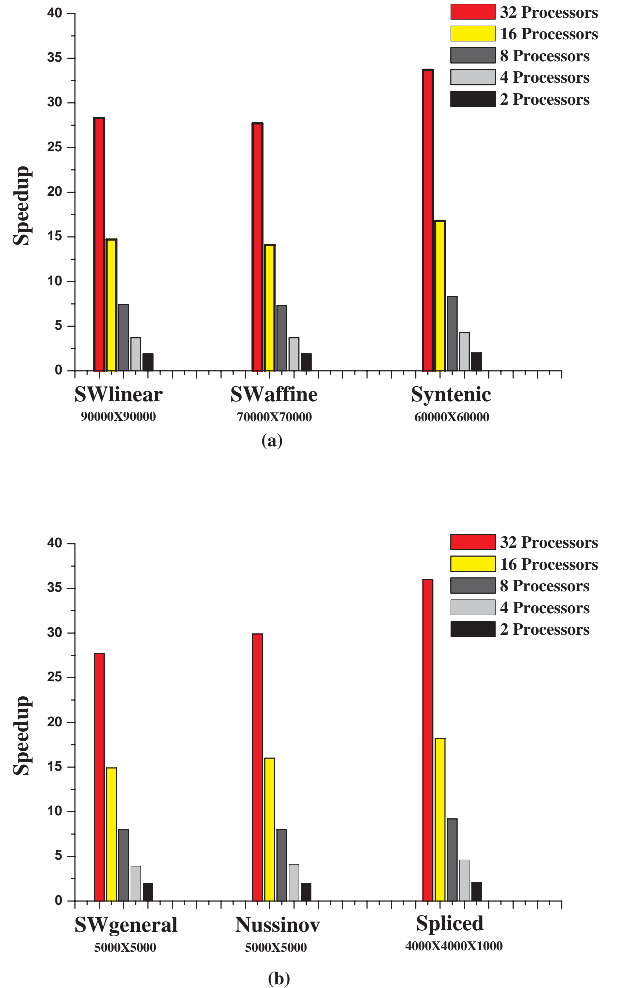


**Figure 8. Speedups for some regular (a) and irregular (b) DP algorithms with corresponding matrix size in computational biology.**

### 4.2. Experimental Results for Parallel Genetic Algorithms

Computing the speedup of a parallel algorithm is a well-accepted way of measuring its efficiency. According to the conventional definition, the speedup of a parallel GA can be defined as the ratio of the execution time of the best sequential algorithm, $T_S$, and the execution time of the parallel program, $T_P$ [3]. $T_P$ can be described as the sum of the time used by one subpopulation in the computation ($T_{COMP}$) and the time it used to communicate information to its neighbors ($T_{COMM}$), so we can compute the speedup as:

$$S_P = \frac{T_S}{T_P} = \frac{T_S}{T_{COMP} + T_{COMM}} \qquad (2)$$

Although speedup is very common in the deterministic parallel algorithms field, in the GA community the topic of parallel speedups has raised significant controversy. The main reason is that the execution time of the serial and parallel GAs are compared without considering the quality of the solutions found in each case. Because of the limited

number of individuals and the inherited selection, the sequential GA has much more tendency to be trapped in a local minimum, without enough genetic diversity to help itself out. Communicating individuals between different evolutions in parallel GAs can help in keeping the genetic diversity of the population, thus greatly reducing the probability to be trapped into local minimum. Assuming the GA is seeking the maximum of some fixed real-valued function, the parallel GA will have an unfair advantage to work out the result much faster. In fact, many researchers have achieved super-linear speedups when using a parallel GA. Shonkwiler [24] has proven this theoretically and provided the following formula to compute the speedup for parallel GAs:

$$S_P = P \times S^{P-1} \tag{3}$$

In Eq. (3) he introduced an acceleration factor *S* that can explain the super-linear speedup ($S > 1$), *P* is the number of processors. It is also shown that for the "deceptive" problem where the time to reach the goal can be infinite, very large speedups are possible.

Eq. (3) provides us a precise way to explain and predict the speedups for parallel GAs. However, the computation of the acceleration factor *S* is too complex. And because for any time *T*, there is a non-zero probability that the algorithm will take time *T* to work out the final result, in order to get the expected speedups by (3), the user may wait a very long time *T* [24]. This is not feasible in practice. During an experiment, if the time to reach the goal tends to be infinite, we think the algorithm is trapped into local minimum. We therefore define the *hit rate* of a running parallel GA as the percentage of the number of outcomes achieved in finite time relative to the total number of experiments. With the concept of *hit rate*, we introduce an approximate way to compute the speedups for parallel GAs:

$$S_P = P \times (k \times \frac{H_P}{H_S}) \tag{4}$$

where $H_S$ and $H_P$ are the hit rates for the sequential and parallel GAs respectively. *k* is an algorithm-dependent constant.

The following sequence with the length 64 has been used in the experiments: *hhhhhhhhhhhhphphpphhpphhpphpphhpphhpphpphhpphhpphphphhhhhhhhhhhh*. This sequence is based on the hydrophobic-hydrophilic (HP) model [11]. The HP model reduces a protein instance to a string of *h*'s and *p*'s that represents the pattern of hydrophobicity in the protein's amino acid sequence.

Table 2 shows the performance measurements for different numbers of processors on the Alpha cluster. Because GAs are stochastic procedures, we have done more than twenty measurements for each configuration. The total population size is set to be 768. For the mutation stage and the crossover stage, the cooling scheme starts with *C* = 2 and is cooled by a factor of 0.99 for every 4 generations. The communication frequency is set to be every 5 generations, exchanging 10% subpopulations.

| **Number of Processors** | **1** | **4** | **8** | **16** | **32** |
|---|---|---|---|---|---|
| **Island Model** | | | | | |
| **Hit Rate** | 25% | 60% | 42.9% | 50% | 40% |
| **Speedups predicted by Eq. (4)** | \ | 9.6 | 13.7 | 32 | 51.2 |
| **Experimental Speedups** | \ | 8.66 | 14.4 | 32.4 | 54.7 |
| **Stepping Stone Model** | | | | | |
| **Hit Rate** | 25% | 45% | 52% | 44% | 36% |
| **Speedups predicted by Eq. (4)** | \ | 7.2 | 16.6 | 28.2 | 46.1 |
| **Experimental Speedups** | \ | 7.5 | 13.3 | 30.1 | 49.6 |

**Table 2. Performance measurements for the general PGA on the Alpha cluster.**

| **Number of Processors** | **1** | **4** | **8** | **16** | **32** |
|---|---|---|---|---|---|
| **Grid level: Island Model; Cluster level: Island Model** | | | | | |
| **Hit Rate** | 25% | 53.8% | 64% | 44% | 37.5% |
| **Speedups predicted by Eq. (4)** | \ | 8.6 | 20.5 | 28.2 | 48 |
| **Experimental Speedups** | \ | 8.3 | 15.6 | 31.2 | 52.1 |
| **Grid level: Island Model; Cluster level: Stepping Stone Model** | | | | | |
| **Hit Rate** | 25% | 52% | 60% | 40% | 32% |
| **Speedups predicted by Eq. (4)** | \ | 8.3 | 19.2 | 25.6 | 41 |
| **Experimental Speedups** | \ | 8.52 | 14.7 | 29.7 | 44.3 |
| **Grid level: Stepping Stone Model; Cluster level: Stepping Stone Model** | | | | | |
| **Hit Rate** | 25% | 42.3% | 46.2% | 39.2% | 32% |
| **Speedups predicted by Eq. (4)** | \ | 6.8 | 14.8 | 25.1 | 41 |
| **Experimental Speedups** | \ | 5.8 | 11.8 | 26.2 | 39.8 |
| **Grid level: Stepping Stone Model; Cluster level: Island Model** | | | | | |
| **Hit Rate** | 25% | 37.5% | 45% | 38.7% | 36% |
| **Speedups predicted by Eq. (4)** | \ | 6 | 14.4 | 24.8 | 46.1 |
| **Experimental Speedups** | \ | 6.7 | 10.7 | 30.2 | 49.8 |

**Table 3. Performance measurements for the HPGA on the heterogeneous cluster environment.**

Table 3 shows the performance measurements for the protein folding simulations using the HPGA on the hierarchical grid environment. The processors used are distributed evenly on the four clusters. The island model and the stepping stone model are used on the grid level and the cluster level respectively. From Table 3 we can see that the hierarchical communication architecture in Figure 5 for HPGAs can be efficiently applied to a hierarchical grid environment. From Table 2 and Table 3 we can also find that the speedups predicted by Eq. (4) are very close to the experimental speedups.

## 5. Conclusions

In this paper we have presented a new generic pattern-based system prototype for the development of high performance computational biology applications. We have identified common communication patterns by analyzing the

characteristics of popular CB algorithms. By integrating these pattern in a generic way, our system provides good extensibility and flexibility. Moreover, we have integrated the two level communication schemes for grid computing into the system. So, it can support two level hierarchical applications, such as HPGAs, on the computational grid. Experiments show that our system can be used to develop high performance applications with substantial performance gains on both PC clusters and computational grid environments.

The exponential growth of genome data demands even more parallel and distributed solutions in the future. As algorithms favored by biologists are not fixed, programmable parallel environments are eagerly required to speed up these tasks. Our future work includes identifying more CB applications that can benefit from our system. Moreover, we will develop a tool to facilitate the easy integration of new patterns.

## References


[1] Globus project: http://www.globus.org/.

[2] http://www-unix.mcs.anl.gov/mpi/mpich/.

[3] G. Almasi and A. Gottlieb. *Highly Parallel Computing*. Benjamin/Cummings Publishing Company, 1994.

[4] J. Anvik, J. Schaeffer, D. Szafron, and K. Tan. Why not use a pattern-based parallel programming system. In *EURO-PAR'2003*, LNCS 2790, 2003.

[5] A. Bartoli, P. Corsini, G. Dini, and C. Prete. Graphical design of distributed applications through reusable components. *IEEE Parallel Distrib. Technol*, 3, 1995.

[6] J. Browne, M. Azam, and S. Sobek. Code: A unified approach to parallel programming. *IEEE Software*, pages 10–18, 1989.

[7] J. Browne, S. Hyder, J. Dongarra, K. Moore, and P. Newton. Visual programming and debugging for parallel computing. *IEEE Parallel Distrib. Technol*, 3, 1995.

[8] E. Cantu-Paz. *A Survey of Parallel Genetic Algorithms*, 1997.

[9] M. Cole. Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389–406, 2004.

[10] J. Deutsch. Evolutionary algorithms for finding optimal gene sets in microarray prediction. *Bioinformatics*, 19(1):45–52, 2003.

[11] K. Dill, S. Bromberg, K. Yue, K. Fiebig, D. Yee, P. Thomas, and H. Chan. Principles of protein folding: A perspective from simple exact models. *Protein Science*, 4:561–602, 1995.

[12] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis–Probabilistic Models of Protein and Nucleic Acids*. Cambridge University Press, 1998.

[13] Z. Galil and K. Park. Dynamic programming with convexity, concavity and sparsity. *Theoretical Computer Science*, 92:49–76, 1992.

[14] M. Gelfand, A. Mironov, and P. A. Pevzner. Gene recognition via spliced sequence alignment. *Proc. Natl. Acad. Sci*, 93:9061–9066, 1996.

[15] J. Gerlach. Generic programming of parallel application with janus. *Parallel Processing Letters*, 12(2):175–190, 2002.

[16] X. Huang and K. M. Chao. A generalized global alignment algorithm. *Bioinformatics*, 19(2):228–233, 2003.

[17] N. Karonis, B. Toonen, and I. Foster. Mpich-g2: A grid-enabled implementation of the message passing interface. *Journal of Parallel and Distributed Computing*, 63(5), 2003.

[18] H. Kuchen. A skeleton library. In *EURO-PAR'2002*, LNCS 2400, 2002.

[19] V. Kumar, A. Grama, A. Gupa, and G. Karypis. *Introduction to Parallel Computing*. The Benjamin-Cummings Publishing Company Inc, 1994.

[20] D. Mount. *Bioinformatics-Sequence and Genome Analysis*. Cold Spring Harbor Laboratory Press, 2001.

[21] C. Notredame and D. Higgins. Saga: Sequence alignment by genetic algorithm. *Nucleic Acid Research*, 24:1515–1524, 1996.

[22] J. Pedersen and J. Moult. Genetic algorithms for protein structure prediction. *Curr Opin Struct Biol*, 6(2):227–231, 1996.

[23] J. Schaeffer, D. Szafron, G. Lobe, and I. Parsons. The enterprise model for developing distributed applications. *IEEE Parallel Distrib. Technol*, 1:85–96, 1993.

[24] R. Shonkwiler. Parallel genetic algorithms. In *5$^{th}$ International Conference on Genetic Algorithms*, 1992.

[25] A. Singh, J. Schaeffer, and M. Green. A template-based tool for building applications in a multi-computer network environment. *Parallel Computing*, pages 461–466, 1989.

[26] S. Siu and A. Singh. Design patterns for parallel computing using a network of processors. In *6$^{th}$ IEEE International Symposium on High Performance Distributed Computing*, pages 293–304, 1997.

[27] T. Smith and M. Waterman. Identification of common subsequences. *Journal of Molecular Biology*, pages 195–197, 1981.

[28] R. Unger and J. Moult. Genetic algorithms for protein folding simulations. *Journal of Molecular Biology*, 1993.

[29] D. Vandevoorde and N. Josuttis. *C++ Template: The Complete Guide*. Addison Wesley, 2002.

[30] B. Wilkinson and M. Allen. *Parallel Programming-Techniques and Applications Using Networked Workstations and Parallel Computers*. Pearson Education, Inc, 1999.

[31] L. Xue and J. Bajorath. Molecular descriptors for effective classification of biologically active compounds based on principal component analysis identified by a genetic algorithm. *Journal of Chemical Information and Computer Sciences*, 40(3):801–809, 2000.

[32] M. Zuker and P. Stiegler. Optimal computer folding of large rna sequences using thermodynamics and auxiliary information. *Nucleic Acids Research*, 9, 1981.