
Oceanus: A Distributed Web-based Framework for Execution of Genetic Algorithms

Jie Chi

Purdue eBusiness Research Center (PERC)
Purdue University
West Lafayette, IN 47907

Alok Chaturvedi

PERC
Purdue University
West Lafayette, IN 47907

Ananth Grama

Computer Science Dept.
Purdue University
West Lafayette, IN 47907

Shailendra Mehta

PERC
Purdue University
West Lafayette, IN 47907

Abstract

The world wide web infrastructure can provide tremendous memory and computing resources with its millions of interconnected workstations. In this paper, we present a software framework and associated distributed computing algorithms for using the web infrastructure for genetic programming. We develop a framework that can be easily adapted to a variety of applications using well-defined interfaces. We demonstrate its superior performance in the context of the Traveling Salesman Problem (TSP). The key to our framework is a communication mechanism that allows multiple computational entities (applets executing on browsers) to communicate effectively to ensure balanced load as well as algorithmic efficiency in terms of excess computation by distributed clients. Compared to existing systems, Oceanus provides superior performance across a wider range of applications, flexibility and ease of adaptation, and scalability to larger numbers of clients.

1 INTRODUCTION

Genetic Algorithms (GAs)(Holland, 1975; Goldberg, 1989) belong to the general class of optimization techniques, which can be applied to many different problems. In spite of its non-deterministic nature, GA has been proven to be a powerful and efficient way of finding good solutions for many problems, and especially for more complex problems that can not be efficiently solved using conventional algorithms.

Due to its population based nature, it is relatively simple to parallelize GA using certain divide-and-conquer

approaches (Cant-Paz, 1995) In this paper, we present a software framework that enables researchers to use the web as the platform for running parallel GA applications. Our experimental evaluation of the distributed algorithms are in the context of Traveling Salesman Problem (TSP). However, our intention in this project is to build a web-based software framework powered by parallel GA to solve a variety of real-world problems, rather than to study TSP alone. Thus far, we have successfully applied our Oceanus framework in Enterprise Security Simulation project, and aircraft design. The results of the those projects are forthcoming.

Although the World Wide Web, in its original design, is not intended for scientific computational purposes, we find it an attractive platform for building a framework to run parallel GA applications for the following of reasons.

1. The web provides a *uniform interface* over heterogeneous and autonomous systems through the use of standardized technologies, such as HTML and Java. We were able to effectively use these and take advantage of the standard interface they provide to expand our framework to include heterogeneous systems.
2. The web provides *cost effective resources*. As we shall explain in Section 2, the millions of computers connected to the web can potentially provide very significant number of CPU cycles and memory storage, provided that the required computation and communication models can be developed.
3. The web is *easy to use*. Most of the web users are comfortable with browsers and related web technologies, such as Java applets. Because of this low entry barrier, we anticipate greater participation in the future.

A number of challenges arise in successfully harnessing the web infrastructure. These include fault tolerance, mechanisms for efficient data exchange and communication, and the many issues parallel genetic algorithms itself raises. We shall explain how the Oceanus framework address these issues in the following sections.

2 RELATED WORK

Prior research has focused on various aspects of parallel GA. There are generally four classes of parallel genetic algorithms, Global, Coarse-grained (also know as Island Model), Fine-grained (also known as Cellular), and various combinations of the three (Cant-Paz, 1995; Whitley, 1993; Gordon, 1993). Oceanus implements coarse-grained GA, which distributes sub-populations, know as *Demes*, to clients for processing. Previous studies has shown that the deme size has a direct impact on the performance of parallel GA (Goldberg, 1995)

Researchers have also previously attempted to use web as a platform for execution of GA. Chong (Chong, 99) has used a combination of Java Servlets and Applets to execute coarse-grained GA in a client and server paradigm. Chong noticed the potential of the web and attempted to exploit the easy to use, and installation-free nature of Java applet. However, as shown by Tanese (Tanese, 1989), a frequent migration among the demes, clients in Chong’s case, is crucial in achieving good quality solution in coarse-grained GA. In Chong’s model, clients communicate with the server using a unidirectional, connection-less mechanism that the Java Servlet provides, which is impossible to support frequent large scale population migration with. Therefore, we believe that Chong’s design is more suited for problems with slowly evolving and relatively independent sub-populations.

In our design, while we still use Java applets to alleviate the need for client-side software installation, we address the communication issue with a persistent, fully duplex communication channel between the clients and the server. As shown in Section 3, such a channel provides the basis of a mechanism that allows frequent information exchange among rapidly evolving sub-populations. While it is true that the server-side overhead associated with these channels increases as the clients are involved, we demonstrate in this paper that the overheads are still small for client numbers of 8-16. It is our hope that in the near future, we can get larger numbers of clients (than those available in our lab) to fully test the scalability of our system. Furthermore, we note that such a channel is necessary to support coarse-grained GA of rapidly evolving subpop-

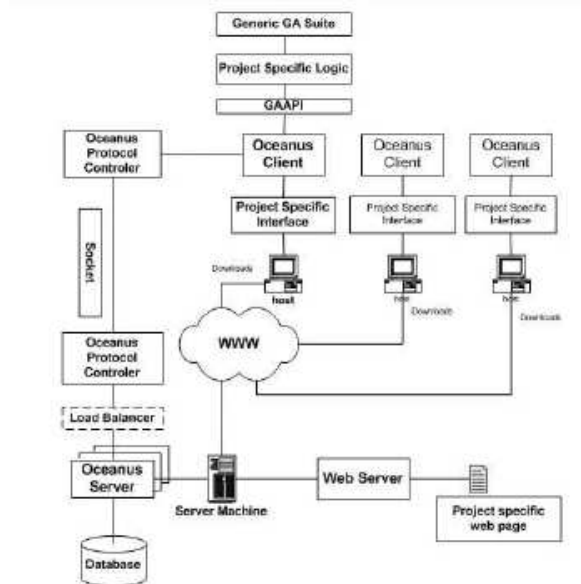


Figure 1: The Oceanus Architecture.

ulations. In addition, we also proposed in our design the use of a load balancer to handle the case of larger scale of client participation.

3 THE WEB-BASED MODEL

The design philosophy in Oceanus is the provision of an underlying genetic programming environment that provides robust, versatile, and efficient mechanisms for problem specific communication and computation. As shown in Figure 1, the web-based model consists of the Oceanus server, Oceanus client applets, Oceanus Protocol Controller, GA-API (GA Application Programming Interface), an optional load balancer, a web server, a database, project specific logic, and project specific user interfaces.

3.1 THE OCEANUS SERVER

The Oceanus server is responsible for initially creating the global population and maintaining it for the rest of the program execution. It also maintains a global migration list. The Oceanus Server listens on a pre-defined TCP port for client requests. Upon receiving a request, the server creates a thread to process the request, and, if necessary, replies to the request in ac-

cordance to the Oceanus Protocol. We describe these tasks in greater detail in the next section.

3.2 THE OCEANUS PROTOCOL CONTROLLER

Due to the ad-hoc nature of the Oceanus' clients participation, there is a need for a protocol to address the communication and cooperation issues between clients and servers. The Oceanus Protocol was designed and developed for this purpose. The Oceanus Protocol Controller implements the Oceanus Protocol. Specifically, it deals with issues of book keeping, error checking, deadlock avoidance, and monitoring progress.

3.3 THE OCEANUS CLIENT APPLLET

An Oceanus client consists of the Oceanus Client Applet (OCA), the *GA-API*, and problem specific logic and user interfaces. The OCA primarily handles communication with the server and with problem specific logic through GAPPI. The GA-API is an API developed for running generic GA. It consists of a set of software objects that correspond to logic entities in GA. GA-API uses Object Oriented Programming (OOP) language concepts to provide well defined software interfaces that can be expanded to encapsulate problem specific logic. The purpose of GA-API is to make the Oceanus model flexible enough to be applied to different problems. In the experimental evaluation of Oceanus, we use TSP as our testbed applications. We also have more complex applications in aircraft design and computer system security running on the Oceanus framework. However, a number of other issues associated with these more complex applications detract from performance evaluation of the underlying framework.

During execution, the OCA is downloaded to the client's computer when an interested web user browses Oceanus web page using a web browser with a Java Virtual Machine. When it first starts execution, OCA is in suspended mode until the user decides to join the project by pressing the start button Figure 2. The OCA then opens a TCP socket to Oceanus Server, which is waiting for a connection at this point. After the connection is successfully established, OCA makes requests to the server by sending an Oceanus Packet (OP), which is an application level packet with a fixed format as shown in Figure 3.3. OPs are sent back and forth between clients and server to exchange information, which includes *system parameters*, including commands such as "GETPOP", "EXEC", and "RETURNPOP", time elapsed, and number of clients; *GA parameters*, such as number of generation, crossover

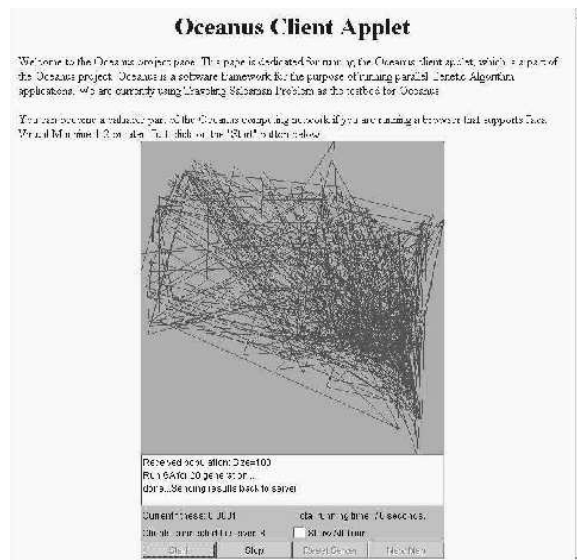


Figure 2: The Oceanus web page

ratio, and mutation ratio; *GA subpopulation*; *problem specific parameters*, such as map dimension, number of cities.

4 THE DISTRIBUTED ALGORITHM

In Oceanus, we implement coarse-grained parallel GA in a client-server topology. In such a topology, special consideration needs to be taken to efficiently handle population and client management, migration, and scalability issues. The rest of this section focuses on these issues.

4.1 MANAGING POPULATION AND CLIENTS

The Oceanus server creates and maintains a global population. The server divides the global population into a pre-determined number of demes. The server also maintains a list of clients that are currently connected to it, along with the index of the demes each of the client is currently working on. Since the level of participation is not known before hand, it is often possible to have fewer clients than there are demes. This means that a static binding between a client and

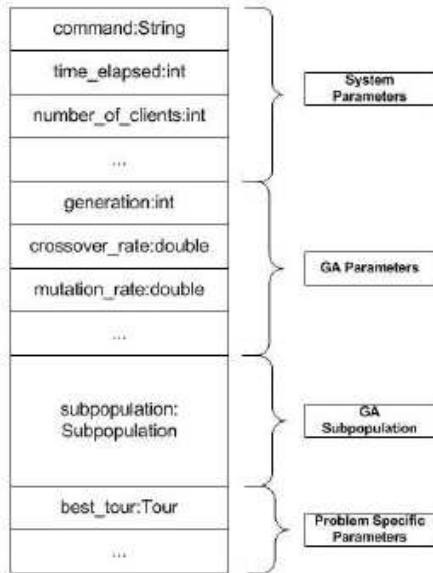


Figure 3: The Oceanus Packet

a deme will lead to poor performance. To illustrate this point, consider in an extreme case in which there are n demes and there is only 1 client. If we statically assign a deme to this client, then effectively only $1/n$ th of the total population is used in the GA process. To avoid this situation, we assign demes to clients in a circular manner. The following procedure shows the logic used to do this.

```

Procedure get_next_deme
  (IPaddr, clientList, demeList):

  Input:  IPaddr, client IP address
         clientList, a list of clients that are
           connected
         demeList, the list of demes
  Output:

begin:
  cur_deme = -1;
  for i = 0 to n, do
    if IPaddr=clientList[i].addr, then
      cur_deme = clientList[i].deme;

  /* free the old deme */
  signal(demeList[cur\_deme]);

  if ++cur_deme >= demeList.length, then
    cur_deme=0; /* circle back */
  end if

```

```

break;
end if
end for

/* client not connected before */
if cur_deme == -1, then
  cur_deme = find_free_deme();

/* no more free demes */
/* refuse connection */

if cur_deme == -1, then
  return false;
end if
end if

/* try to lock the new deme */
wait(demeList[cur_deme]);

record_time(IPaddr); /*record the time*/

return cur_deme;
end

```

Notice that semaphores are used to ensure synchronization among clients. One potential problem in using semaphores is the possibility of client starvation. If one client, for some reason, holds a semaphore for a long time, this will cause other clients to wait indefinitely, and therefore causes starvation. To solve this, the Oceanus server enforces bounded waiting using a timer. When the server assigns a deme to a client, it locks the deme using semaphore for this client and records the client's IP and the time at which this occurs. A separate timer thread periodically checks the time records of the locked demes. If the difference between the current time and the recorded time is greater than a predefined time-out value, the timer thread cancels the semaphore the client had previously, and therefore allows other clients to make progress. If the client does eventually come back, the server discards its results and re-assign a deme to this client. In the current version of Oceanus, we use a pre-defined value for the timer. More sophisticated timing schemes are possible, such as using the Jacobson algorithm to estimate the client response time. However, these are not the focus of this project.

4.2 POPULATION MIGRATION

In Oceanus, we implement a migration strategy based on the *blackboard* model (Barr, 1989) The Oceanus server maintains a centralized migration list, which serves as the blackboard for client to exchange information, in this case, populations of solutions.

As shown in Figure 4, when a client, C_i , returns after finishing the assigned number of GA generations

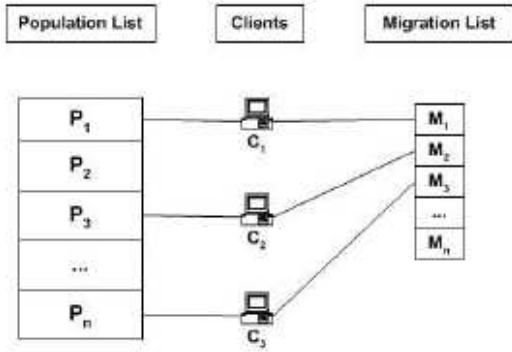


Figure 4: The Blackboard Migration Model.

on a deme of size n , it copies a fraction of its population to its reserved slot, M_i , in the migration list. The client then replaces a portion of the population in the deme that it is currently holding with the population in the rest of the migration list. The exact number of entities that C_i replaces can be expressed as $S = (n - 1)M_s$, where n is the number of clients and M_s is the size of each of their contributions. This is done to replace the worst S individuals in the current deme. Overall, this process takes $O(n \log n)$ to sort deme, where n is the size of the deme, and $O(m)$ time to copy and replace the population, where m is the size of the migration list. Our observation from experimental results indicates that for client sizes ranging from 8-16, this overhead is small compared to the time a client takes to perform the GA computation. However, we note that this migration process will become the bottleneck for system performance as the number of clients grows. To address this potential problem, a load balancer is introduced in the Oceanus design. The load balancer is a central server that accepts requests from the client. Instead of further processing these requests as an Oceanus server would do, the load balancer forwards these requests to a list of Oceanus servers in a round robin fashion. This round robin distribution of requests among the servers is advantageous as it probabilistically ensures that each server has populations of similar quality at about the same time of the genetically evolving process.

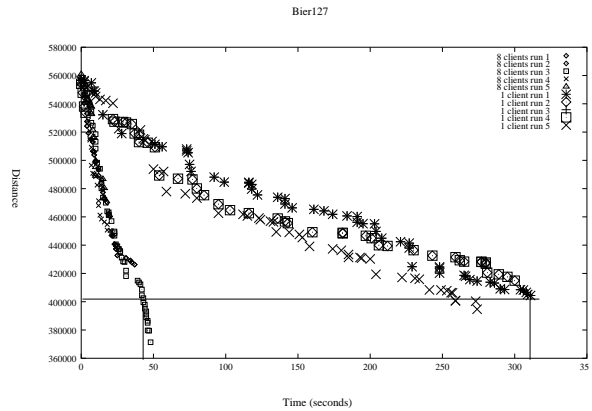


Figure 5: The results of running 1 client and 8 clients on the bier127 map. Each case is repeated for 5 runs.

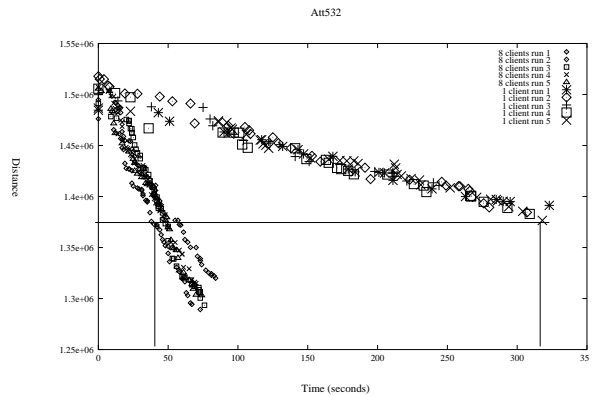


Figure 6: The results of running 1 client and 8 clients on the att532 map. Each case is repeated for 5 runs.

5 EXPERIMENTAL RESULTS.

We report here on the experimental results from execution of various instances of TSP using Oceanus. To encode the problem using GA, we assign a unique integer to each city for a given map of cities. A tour can be represented by a list of integers, each representing a city. A city also comes with its two dimensional coordinates, and therefore the cost of a tour can be calculated as the Euclidean distance of the tour. The fitness of a tour is calculated as the inverse of its cost. The crossover is done using the Partially Mapped Crossover(PMX)(Davis, 1991) operator. GA parameters, such as population size, deme size, crossover rate, mutation rate, and migration intervals, are stored in a initialization file. As the Oceanus server starts, it reads the parameters from the file and randomly generates the initial population of tours.

We demonstrate the results of running 1 client and 8 clients respectively for 5 runs (Figures 5 and 6). Two maps from the the TSPLIB were used, *bier127* and *att532*.

5.1 PARALLEL EFFICIENCY

We can see that in both graphs (Figures 5 and 6), as the distance of the tour approaches a benchmark (marked by the horizontal line), the computation performed by 8 clients are approximately 7-8 times faster than the that performed by 1 client. We believe that this demonstrates that the population management and migration strategy adopted by Oceanus server is largely efficient and the server does not become the bottleneck for performance for a client pool of this 8-16 clients. We observe similar results for other problems of this magnitude and beyond.

5.2 ALGORITHMIC EFFICIENCY

A perhaps more subtle point demonstrated in these results is the fact that the algorithmic efficiency of GA is preserved in distributed execution. This is a very important aspect that is often relaxed in many implementations of parallel GA. Our algorithmic efficiency is attributed to the fact that the demes in the global population are assigned to client(s) in a circular fashion, as explained in Section 4.1. Therefore, in the single client cases, the entire population is processed, deme by deme, in a sequential manner with population migration among these demes.

6 CONCLUSIONS

Oceanus provides a flexible, efficient, and easy to use framework for executing GA over the web. In contrast to prior work, Oceanus provides a tight coupling between clients to ensure high algorithmic efficiency while incurring low communication costs. In doing so, it provides a basis for diverse applications in which the fitness measures across populations may be rapidly varying. Oceanus provides a well defined application interface, which has been utilized to support a variety of applications. Finally, in configurations of 8-16 clients, Oceanus exhibits excellent parallel and algorithmic efficiency. We expect to present more detailed scalability results at the conference as well as execution profiles of other applications in aircraft design and computer system security.

References

- [1] BARR, A., AND FEIGENBAUM, E. A. *The Handbook of Artificial Intelligence IV*. Addison Wesley, 1989.
- [2] CANT-PAZ, E. A survey of parallel genetic algorithms. Tech. rep., IlliGAL, University of Illinois at Urbana-Champaign, 1995.
- [3] CHONG, F. S. Java based distributed genetic programming on the internet. Tech. rep., School of Computer Science, The Univ. of Birmingham, U.K., 1999.
- [4] DAVIS, L., Ed. *Handbook of genetic algorithms*. New York : Van Nostrand Reinhold, 1991.
- [5] GOLDBERG, D. E. *Genetic algorithms in search, optimization, and machine learning*. New York: Addison-Wesley, 1989.
- [6] GOLDBERG, D. E., KARGUPTA, H., HORN, J., AND CANT-PAZ, E. Critical deme size for serial and parallel genetic algorithms. Tech. rep., IlliGAL, University of Illinois at Urbana-Champaign, 1995.
- [7] GORDON, V. S., AND WHITLEY, D. Serial and parallel genetic algorithms as function optimizers. In *The 5th International Conference on Genetic Algorithms* (Urbana-Champaign, 1993), pp. 177-183.
- [8] HOLLAND, J. H. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MC, 1975.

- [9] REINELT, G. A traveling salesman problem library, 1990. web page, <http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/>.
- [10] TANESE, R. *Distributed Genetic Algorithm for Function Optimization*. PhD thesis, University of Michigan, 1989.
- [11] WHITLEY, D. A genetic algorithm tutorial. Tech. rep., Colorado State University, 1993.