Asynchronous Genetic Algorithms for Heterogeneous Networks Using Coarse-Grained Dataflow

John W. Baugh Jr.[1] and Sujay V. Kumar[2]

[1] North Carolina State University, Raleigh, NC 27695 USA
john.baugh@ncsu.edu
[2] NASA Goddard Space Flight Center, Greenbelt, MD 20771 USA
sujay@hsb.gsfc.nasa.gov

Abstract. Genetic algorithms (GAs) are an attractive class of techniques for solving a variety of complex search and optimization problems. Their implementation on a distributed platform can provide the necessary computing power to address large-scale problems of practical importance. On heterogeneous networks, however, the performance of a global parallel GA can be limited by synchronization points during the computation, particularly those between generations. We present a new approach for implementing asynchronous GAs based on the dataflow model of computation — an approach that retains the functional properties of a global parallel GA. Experiments conducted with an air quality optimization problem and others show that the performance of GAs can be substantially improved through dataflow-based asynchrony.

1 Introduction

Numerous studies have sought to exploit the inherent parallelism in GAs to achieve better performance. A recent report by Cantu-Paz [4] surveys the extensive research in this area and categorizes techniques for parallelization. One of the more straightforward techniques is global parallelization, in which the evaluation of individuals is performed in parallel [3]. Certain variations on global parallel GAs, such as evolving independent subpopulations [8] and hierarchically evolving populations [7], have also been developed. These and other global parallel GAs are *synchronous* in the sense that computations involving subsequent generations may not proceed until those of the current generation are complete. The speedup lost as a result of these synchronization points can be significant, particularly in a heterogeneous, networked environment, since the presence of a single slow processor can impede the overall progress of the GA.

The limitations of global parallel GAs due to end-of-generation synchronization points have been studied by a number of researchers. Most of the reported approaches use localized evolution strategies such as island-based approaches [5, 9] to achieve asynchrony. However, approaches other than global parallelization introduce fundamental changes in the structure of a GA [3]. For example, island-based GAs work with multiple interacting subpopulations whose parameters for

E. Cantú-Paz et al. (Eds.): GECCO 2003, LNCS 2723, pp. 730–741, 2003.

interaction require additional, problem-specific tuning. Poor settings can result in either convergence to an inferior solution or suboptimal parallel performance. Steady-state GAs [10], which work with a single evolving population, are another means of eliminating end-of-generation synchronization points. Instead of placing offspring in subsequent populations, such GAs return them to the original population by an operator that selects individuals to be replaced. In addition to suffering in some cases from problems of premature convergence, steady-state GAs, like island-based approaches, introduce fundamental changes in the GA.

In this paper, we present a new approach for implementing asynchronous GAs that is functionally equivalent to a global parallel GA, and hence to a sequential GA as well. By *functionally equivalent* we mean that the outputs are determined by precisely the same numerical operations and are likewise identical. Equivalence is achieved by "unrolling" the main loop of a global parallel GA, i.e., the loop responsible for advancing from one generation to the next. Inter-generational data dependencies are then captured formally using dataflow graphs, which enable the concurrent processing of multiple generations to the extent allowed by those dependencies. The benefits of functional equivalence between sequential and parallel implementations are substantial. Numerical results obtained from either implementation can be compared one-to-one with assurance that artifacts have not been introduced via parallelization. Further, the additional parameter tuning required when moving from sequential to parallel runs of a GA need not be repeated.

While applicable in other contexts, our approach targets GAs on heterogeneous workstation networks that may need hours, days, or even weeks to complete. In such a scenario participating computers may vary over time in their availability and in the resources that are committed to a given GA run. This type of variability imposes severe performance penalties when extraneous synchronization points are encountered. For all its benefits with compute-intensive runs, though, it is equally appealing that the approach adds very little computational overhead: it is lightweight enough to be imperceptible on runs taking well under a minute to complete.

2 Dataflow Principles

Dataflow [6] is a term that refers to algorithms or machines whose order of execution is based on the availability and forwarding of data. A dataflow program is a directed graph with nodes that represent operators and directed arcs that represent data dependencies. Nodes are computational tasks, and may be primitive machine-level instructions or arbitrarily complex functions. As a result, the dataflow model is applicable to fine- or coarse-grained parallelism. In addition to supporting varying levels of parallelism, the dataflow model also supports various types of parallelism. For instance, *vectorizing* and *pipelining* are simply special cases of standard flow graphs.

In the dataflow model, data values are carried on tokens, which travel along arcs, which we model as one-place buffers. The status of nodes can be determined

by a simple firing rule: A node is said to be *firable* when the data it needs are available. When a node is fired, its input tokens are absorbed. The computation is performed and the result is sent to its output arcs for other nodes to use. There is no communication between tasks — each task simply receives and outputs data.

The dataflow model has the following properties [1]:

- *parallelism:* nodes may execute in parallel unless there is an explicit data dependence between them;
- *determinacy:* results do not depend on the relative ordering in which nodes execute.

The natural parallelism in the dataflow model occurs because it does not force over-specification of an algorithm. The firing rule only says when a node *can* fire. It does not require that it be executed at any particular time.

3 Using Dataflow for Asynchrony

A synchronous distributed GA (SDGA) based on global parallelism begins with an initial population from which subsequent ones are obtained through a selection process. Here we assume the use of a binary tournament scheme, which selects two individuals at random, evaluates their fitnesses remotely, and produces a single winner. To generate a new population of size P this process is performed P times. Processor loads are dynamically balanced by placing evaluation requests in a task pool. Crossover and mutation operators are then applied and the entire process is repeated until convergence.

The "repeat until convergence" part of the above algorithm forces synchronization at the end of each generation since individuals in subsequent generations cannot be evaluated until all of their individuals are in place. An asynchronous distributed GA (ADGA) is obtained by "unrolling" this loop and building dataflow graphs that capture the algorithm's inter-generational data dependencies. Intuitively, once a sufficient number of individuals have been evaluated in one generation, some of their offspring can be produced and undergo evaluation, even before the prior generation is complete. The extent to which generations are processed concurrently is limited only by the data dependencies derived from the synchronous implementation. Typically a "band" of 2 to 4 generations is active at any one time as the computation unfolds.

Pseudo-code for an ADGA using dataflow is shown in Figure 1. Populations are constructed and named using the *new population* procedure, which initiates enough *dataflow* threads (or "lightweight" processes) to carry out the genetic operations necessary for that generation. As each *dataflow* thread completes its task, the resulting offspring are placed in the subsequent generation. The *succ* function finds and returns the subsequent (or "successor") generation: if it does not exist it is created via *new population*, which has the side effect of forking a new round of *dataflow* threads for the next generation unless a termination condition is met.

```
P_final = empty

main
   new population (P_0)
   while P_final is empty do wait
   return fittest from P_final

procedure new population (P_t)
   if termination condition met
   then P_final = P_t
   else start n/2 threads: dataflow (P_t)

thread dataflow (P_t)
   place 4 random individuals from P_t in graph
      (evaluate remotely, compete, mate)
   write 2 offspring into succ (P_t)

function succ (P_t)
   if P_t+1 is empty
   then new population (P_t+1)
   return P_t+1
```

Fig. 1. Pseudo-code for an Asynchronous GA

An illustration of a running ADGA program is shown in Figure 2. The figure depicts three active generations, each with a population of 10 individuals. Unshaded circles in each population denote empty token positions — a place to put an individual once it is produced. The initial population, *G1*, begins with randomly generated individuals so all of its circles are filled with tokens. The figure shows that some processing has already occurred. Dataflow graphs *D11*, *D12*, *D14*, and *D15* have completed, as indicated by their dashed outlines and the fact that they have produced offspring (shaded circles) in generation *G2*. Dataflow graph *D13*, on the other hand, is still working: it has a solid outline and has yet to produce its offspring in generation *G2*. There is a mix of working and completed dataflow graphs in generation *G2* as well. In generation *G3*, however, no dataflow graphs have completed, and some are still waiting for input. No space will be allocated for generation *G4* until one of the graphs in *G3* is ready to produce its offspring.

The inputs to each dataflow graph are the randomly selected individuals that will be used in the genetic operations. For instance, dataflow graph *D13* takes individuals 7, 2, 5, and 0 from generation *G1* and produces its offspring in positions 4 and 5 of generation *G2*. This behavior is more clearly seen in Figure 3, which provides a detailed view of dataflow graph *D13*. As shown in the figure, individuals 7 and 2 compete for position 4, and individuals 5 and 0 compete for position 5. This processing is performed by nodes in the graph, each

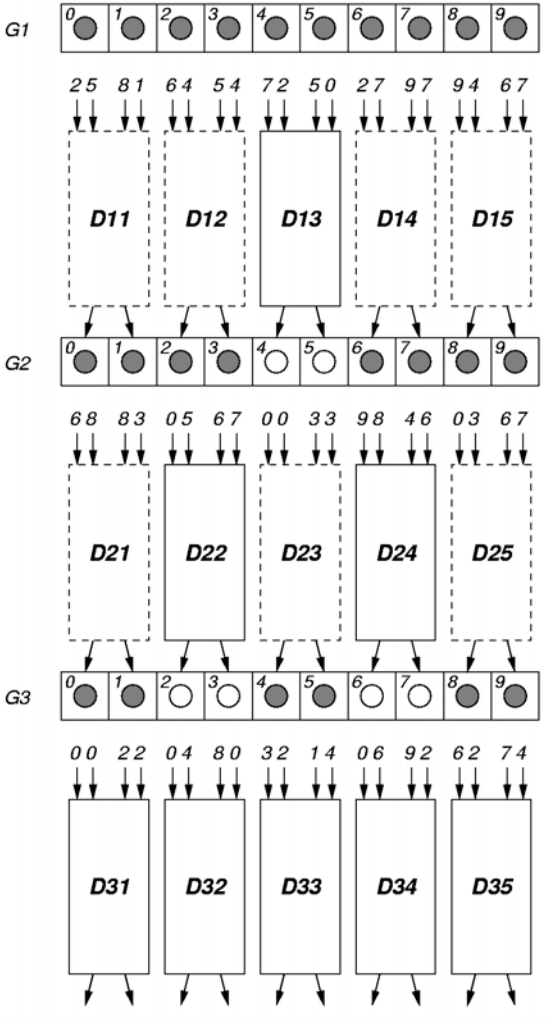

Fig. 2. Dataflow Graphs Dynamically Unfolding

being implemented by concurrent threads that block until their requisite inputs are available.

The *Copy* nodes ensure that an individual can be selected and processed simultaneously by other dataflow graphs. The need to copy is a result of having data flow through the model via tokens instead of being referenced as variables — a fundamental requirement of the dataflow model. Pointer copying is sufficient here, ensuring implementation efficiency. *Compare* nodes are used to keep track of an incumbent organism — the fittest seen during the GA run. Other nodes in the graph —*Evaluate*, *Compete*, and *Mate*— perform the usual genetic

operations. True parallelism is obtained in the implementation of the *Evaluate* nodes, which place in a task pool a request to evaluate the individual's fitness on a remote processor; each blocks until the result becomes available.

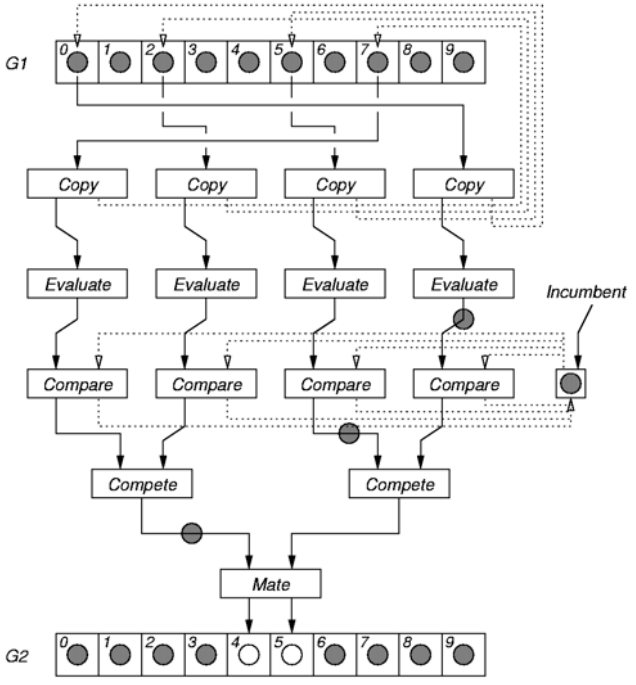

Fig. 3. Details of Dataflow Graph *D13*

4 Analysis and Results

Realizations of the SDGA and ADGA approaches, as described above, have been conveniently implemented in the Java programming language using its multithreading capabilities and socket libraries for network communication. The implementations have been shown to be both efficient and portable across multiple platforms and operating systems — even within a single GA run. Experiments have been conducted with homogeneous as well as heterogeneous systems of processors, and simple empirical models have been developed to predict execution times. We begin by describing these models and then comparing predicted results with those obtained on a simple 0/1 knapsack problem and on a more complex air quality management problem.

4.1 Homogeneous System of Processors

Consider a homogeneous network of computers consisting of N identical processors. For a single generation of a GA to complete, P organisms must be evaluated. It is assumed that all of the N processors start simultaneously, and that each takes time t_{comp} to execute a fitness evaluation and time t_{comm} for communication with the client. The tasks associated with the GA can then be laid out in blocks, with each block representing the tasks performed by N processors in time $t_{comp} + t_{comm}$, as shown in Figure 4.

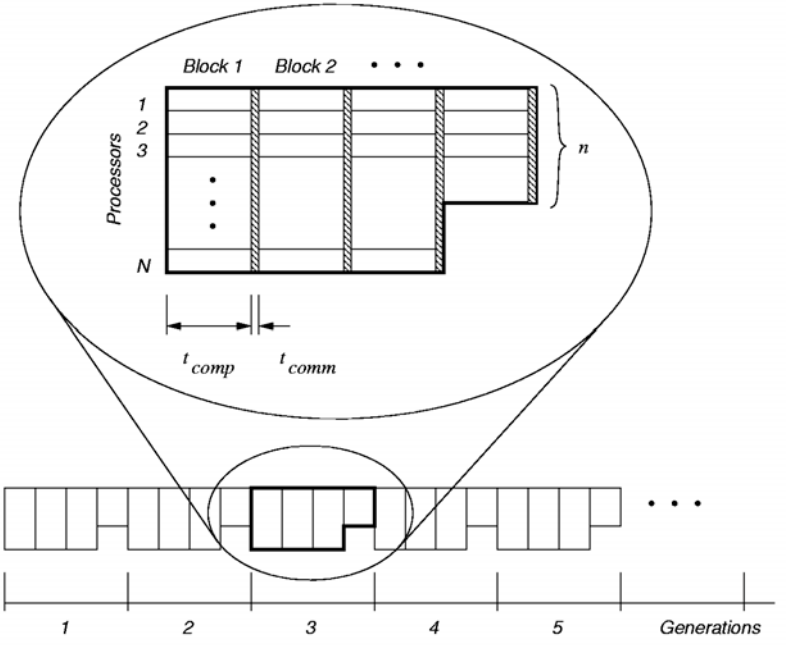

Fig. 4. GA Tasks Executing on N Homogeneous Processors

The pattern of blocks repeats itself until the end of a generation, at which point some number of evaluations n remain to be performed. Since N individuals are evaluated in each block, the total number of blocks in a generation is equal to $\lceil \frac{P}{N} \rceil$. From the figure, the time taken for a single generation (T_g) and the total time taken by an SDGA (T_{sync}) can be estimated as:

$$T_g = \lceil \frac{P}{N} \rceil (t_{comp} + t_{comm}) \tag{1}$$

$$\begin{aligned} T_{sync} &= T_g\, G \\ &= \lceil \frac{P}{N} \rceil (t_{comp} + t_{comm})\, G \end{aligned} \tag{2}$$

In the case of an ADGA, the processors are not constrained by the lack of available tasks at the end of a generation since, in practice, a sufficient number are available from subsequent generations to avoid idling. The total number of tasks in an ADGA evaluation is PG. Since there are N processors the total time taken by an ADGA (T_{async}) can be estimated as:

$$T_{async} = \frac{P\,G}{N}(t_{comp} + t_{comm}) \tag{3}$$

4.2 Heterogeneous System of Processors

To model a heterogeneous system, n_s identically slow processors are introduced into the system of N processors. Each of these slow processors is assumed to require a factor of f more processing time to evaluate an individual. The quantities t and t_{slow} are defined to be the sum of t_{comp} and t_{comm} for fast and slow processors, respectively. As with the homogeneous case, the tasks on a heterogeneous system can be laid out in blocks, where in this case each block is of width t_{slow}. Figure 5 shows GA tasks on a heterogeneous system with a single slow processor and f equal to 4. As depicted in the figure, for an SDGA, the presence of a slow processor clearly leaves idle a large number of faster processors.

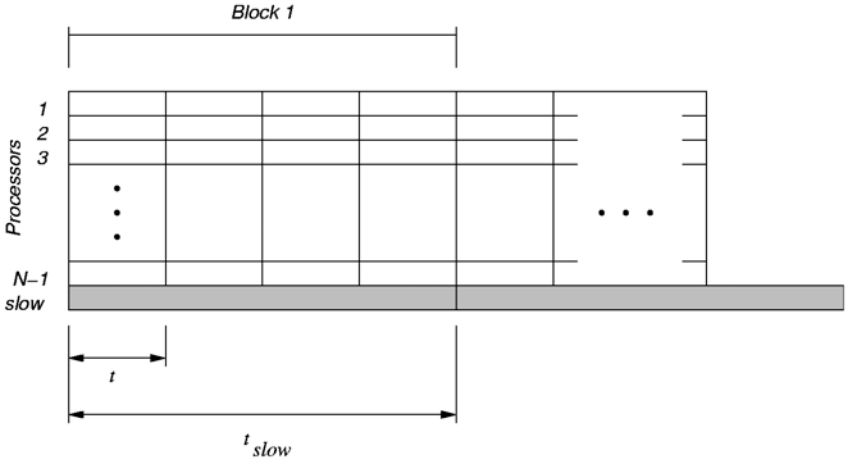

Fig. 5. GA Tasks Executing on Heterogeneous Processors

The number of blocks in a generation can be estimated as:

$$n_b = \lceil \frac{P}{f(N - n_s) + n_s} \rceil \tag{4}$$

Depending on the ordering of tasks, the number of tasks that remain at the end of generation becomes important. The number present in the final block of a generation (δ_1) can be estimated as:

$$\delta_1 = P - (n_b - 1)\,(f(N - n) + n) \tag{5}$$

If there are more tasks in the last block than fast processors, the slow processors will receive tasks to evaluate. Taking these factors into account, the total time taken by an SDGA can be estimated as:

$$T_{sync} = \begin{cases} (n_b - 1) f\, t\, G + t\, G & \text{if } \delta_1 \leq (N - n_s) \\ n_b\, t\, f\, G & \text{otherwise} \end{cases} \tag{6}$$

Since end-of-generation synchronizations are eliminated in an ADGA, the overall GA execution can be thought of as an ordering of PG tasks among processors. The number of blocks is estimated as:

$$n_b = \lceil \frac{PG}{f(N - n_s) + n_s} \rceil \tag{7}$$

At the end of the GA execution, if the last block contains more tasks than the number of fast processors, the slow processors will be involved in the final computations. The number of tasks present in the final block of GA execution (δ_2) can be estimated as:

$$\delta_2 = PG - (n_b - 1)\,(f(N - n_s) + n_s) \tag{8}$$

The estimated time taken by an ADGA is:

$$T_{async} = \begin{cases} (n_b - 1) f t + t & \text{if } \delta_2 \leq (N - n_s) \\ n_b\, t\, f & \text{otherwise} \end{cases} \tag{9}$$

4.3 0/1 Knapsack Problem

The 0/1 knapsack problem is representative of the large class of problems known as combinatorial optimization problems. Informally stated, the objective of the knapsack problem is to select items that maximize profit without exceeding capacity. As such, the problem is fine grained since fitness evaluation is typically inexpensive.

Both SDGA and ADGA implementations are applied to the 0/1 knapsack problem with anywhere from 3 to 30 processors. To assess their scalability with increased problem size, fitness evaluation times are artificially varied to achieve four different levels of granularity based on the ratio of t_{comp} to t_{comm}. Since

t_{comm} is approximately 250 milliseconds in our set up, t_{comp} times are artificially set to 250, 500, 750 and 1000 milliseconds, resulting in granularity factors of 1 through 4. To simulate a heterogeneous system, a slow processor is introduced with f set to 5. GA runs conducted with a population size of 100 for 200 generations yield the results shown in Figure 6. Although t_{comp} and t_{comm} are underpredicted in the model, the trends are as expected, with execution times increasing with problem granularity, and the ADGA scaling better than the SDGA.

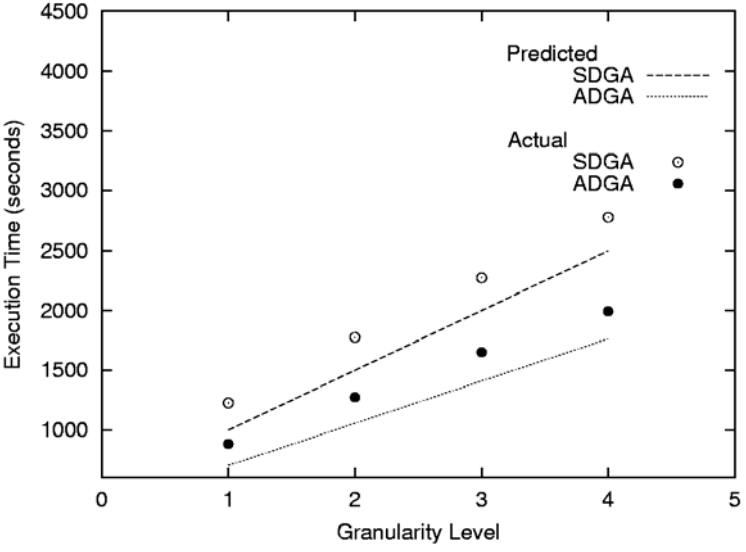

Fig. 6. Execution Time vs. Granularity using 15 Processors: 0/1 Knapsack Problem

4.4 Air Quality Optimization

Tropospheric ozone formed from the emissions of vehicles and industrial sources is considered a major pollutant. As a result, air quality management strategies may be necessary for geographic regions containing hundreds of sources, with each in turn having thousands of processes. Formal search strategies using GAs can be applied to find cost-effective ways of reducing ozone formation. For instance, an ambient least cost (ALC) model [2] is an optimization approach that incorporates source marginal control costs and emission dispersion characteristics to compute the source emissions at the least cost. A number of modeling techniques can be used to determine dispersion characteristics, such as the Empirical Kinetic Modeling Approach (EKMA), a Lagrangian box model that is used in this study. Because of the execution times typically required for EKMA, this GA formulation is somewhat coarse grained.

Experiments for an air quality management study around Charlotte, NC, were conducted on a network of workstations with as many as 19 processors. To simulate a heterogeneous system, a slow processor with an f factor of 5 is used. In each case, the GA was run for 50 generations using a population size of 50. The execution times are found to be in close agreement with the values predicted by the empirical model, as shown in Figure 7. Better agreement here than in the knapsack problem is likely due to increased problem granularity. Similar to earlier trends, the SDGA is outperformed by the ADGA; the execution times of the SDGA follow a step function pattern implying that, in between each step, there is no marginal benefit in using additional processors.

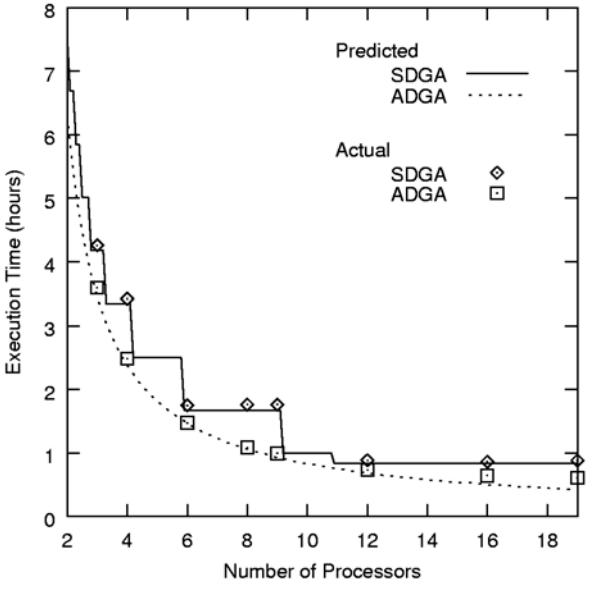

Fig. 7. Execution Time vs. Processors: Air Quality Optimization

5 Final Remarks

The growing acceptance of GAs has led to widespread use and attempts at solving larger and more challenging problems. A practical approach for doing so may rest on the ability to use available computer resources efficiently. Motivating the algorithmic developments in this paper is the expectation that a heterogeneous collection of personal computers, workstations, and laptops should be able to contribute their cycles to the solution of substantial problems without inadvertently detracting from overall performance. Removing the end-of-generation synchronization points from global parallel GAs is necessary to meet this expectation. The application of loop unrolling and dataflow modeling described herein

has been shown to be effective in keeping available processors from idling even when substantial variations exist in the processors' capabilities.

Although other asynchronous approaches might be used, one that is functionally equivalent to a simple, sequential GA offers real benefits with respect to parameter tuning. In a significant study on air quality management [references temporarily withheld for blind review process], our research team was able to move with little effort between atmospheric models that varied widely in their computational demands — from simple ones that can be solved using sequential GAs, to ones that require 20 minutes to evaluate a single individual on a high-end workstation: the same basic algorithm and parameters could be (and were) used in either case.

The GA implementations described in this paper are part of *Vitri*, an object-oriented framework implemented in Java for high-performance distributed computing [references temporarily withheld for blind review process]. Among its features are basic support for distributed computing and communication, as well as visual tools for evaluating run-time performance, and modules for heuristic optimization. It balances loads dynamically using a client-side task pool, allows the addition or removal of servers during a run, and provides fault tolerance transparently for servers and networks.

References

1. Arvind and D. E. Culler. Dataflow architectures. *Annual Reviews in Computer Science*, 1:225–253, 1986.
2. S. E. Atkinson and D. H. Lewis. A cost-effective analysis of alternative air quality control strategies. *Journal of Environmental Economics*, pages 237–250, 1974.
3. E. Cantu-Paz. Designing efficient master-slave parallel genetic algorithms. Technical report, University of Illinois at Urbana-Champaign, Urbana, IL, 1997.
4. E. Cantu-Paz. A survey of parallel genetic algorithms. Technical Report 97003, University of Illinois at Urbana Champaign, May 1997.
5. V. Coleman. The DEME mode: An asynchronous genetic algorithm. Technical Report UM-CS-1989-033, University of Massachusetts, May 1989.
6. Computer. Special issue on data flow systems. 15(2), 1982.
7. J. Kim and P. Zeigler. A framework for multiresolution optimization in a parallel/distributed environment: Simulation of hierarchical GAs. *Journal of Parallel and Distributed Computing*, 32:90–102, 1996.
8. Yu-Kwong Kwok and Ahmad Ishfaq. Efficient scheduling of arbitrary task graphs to multiprocessors using a parallel genetic algorithm. *Journal of Parallel and Distributed Computing*, 47:58–77, 1997.
9. M. G. Schleuter. Asparagas: An asynchronous parallel genetic optimization strategy. *Proceedings of the Third International Conference on Genetic Algorithms*, pages 422–427, 1989.
10. J. E. Smith and T. C. Fogarty. Self adaptation of mutation rates in a steady state genetic algorithm. In *Proceedings of IEEE International Conference on Evolutionary Computing*, volume 72, pages 318–323, 1999.