From: AAAI Technical Report WS-97-06. Compilation copyright © 1997, AAAI (www.aaai.org). All rights reserved.

A distributed system for genetic programming that dynamically allocates processors

Matthew Evett and Thomas Fernandez
Dept. Computer Science & Engineering, Florida Atlantic University
Boca Raton, Florida 33431
{matt,tfernand}@cse.fau.edu, http://www.cse.fau.edu/~matt

ABSTRACT

AGPS is a portable, distributed genetic programming system, implemented on MPI. AGPS views processors as a bounded resource and optimizes the use of that resource by dynamically varying the number of processors that it uses during execution, adapting to the external demand for those processors. AGPS also attempts to optimize the use of available processors by automatically terminating a genetic programming run when it appears to have stalled in a local minimum so that another run can begin.

1 Introduction

Most distributed systems presuppose a fixed number of processors during execution, though they might provide for this number to vary across distinct invocations. For example [15] and [10] describe systems that determine at start-up the number of available processors and/or their relative speeds and modify their partitioning and load-balancing schemes to optimize the use of these processors. The processors allocated to these systems are either unavailable for other use during execution, or cause other processes on the same processors to suffer very poor performance. If these systems have relatively lengthy execution times, then the user may have to choose not to allocate all available processors to the task so that some processors will be available for other users later during the system's run. The processors may well go unused for the duration of the system's execution, while the system might well have benefited from the use of those extra processors. This is inefficient use of processor resources.

Genetic computation (which we use to describe genetic programming and genetic algorithms) is notoriously computationally demanding. Many of the most interesting applications involving genetic computation can require hundreds of hours of CPU time. Indeed, the results reported in this paper required over 300 hours of CPU time (on Sparcstations) to compile. Luckily, genetic computation is particularly amenable to parallelization (it is "embarrassingly parallel"), which can ameliorate this computational cost somewhat. Thus, a distributed genetic computation system is exactly the type of system where maximizing the use of available processor resources without monopolizing them is particularly important.

We have developed a system, AGPS[1], for genetic programming that runs on distributed and parallel computing platforms through the use of MPICH[7], an implementation of MPI (the Message Passing Interface)[6]. There are other parallel or distributed genetic computation systems (e.g. [1], [9]). But these systems are designed to work within a fixed set of processors. AGPS views processors as a bounded resource and optimizes the use of that resource by dynamically varying the number of processors that it uses during execution, adapting to external demand for those processors.

This paper outlines AGPS and its dynamic processor allocation mechanism and presents a performance analysis of the system. Section 2 briefly describes genetic programming. Section 3 outlines AGPS and Section 4 the processor allocation mechanism. Section 5 presents performance figures for AGPS. Section 6 describes on-going and future development of AGPS.

[1] Adaptive Genetic Programming System

2 Genetic Programming

Genetic programming [11] (GP) is closely related to genetic algorithms [8] (GA). Both are a form of adaptive search. The adaptation mechanism is loosely based on evolution in natural (genetic) systems. The system initially contains a population of randomly generated "prospective solutions" (*population elements* or *individuals*) to the problem to be solved. Each of these individuals is represented by a "gene", which is (usually) a binary string in GAs and a Lisp-like s-expression in GPs. The genetic system evaluates each of the elements via an *evaluation function*, similar to a heuristic function in general search algorithms. The evaluation function returns "high" values for those elements that most nearly solve the given problem, and "low" values for those elements that are far from the mark.

Genetic computation systems differ greatly in how they handle the next few steps in processing populations, but the basic technique goes something like this: there is a *selection* process wherein, typically, those elements with "high" value are more likely to be selected. The end result of the selection process is a *mating pool*. The system uses the elements of the mating pool to create a new population, i.e., to form the next *generation*. Mating pool elements can either be directly copied into the next generation (straight *reproduction*), or can merge in some fashion with another element or elements of the mating pool to form (usually) novel elements (genes). This merging process is usually called *cross-over*. The genetic computation proceeds iteratively for a predefined number generations, and these generations comprise a *run*.

Under the right circumstances (these circumstances are not completely understood, but the evaluation function is particularly important, as is the property that the merging of two high-value individuals has a higher probability of yielding another high-value individual than the merging of two random individuals), because the mating pool always consists of relatively better individuals, the population as a whole tends to maximize its value over successive generations. If all goes well, the genetic computation will yield a solution, or near solution, to the given problem.

In general (and this is a *very* broad generalization), GP systems tend to perform better the larger their populations are. To a lesser extent, they perform better the longer their runs are. Like neural networks, GP systems don't necessarily converge, or find a solution; they may get stuck in a local minimum. Runs provide a rather crude mechanism for dealing with these situations. When the system completes a run, it records the best solution found during that run, and starts another run, re-instantiating the population with a completely random set of individuals/genes.

A fuller description of the techniques of genetic programming is far beyond the scope of this paper, but the interested reader should see [11], [12], [2], or [13].

3 AGPS

AGPS provides for the execution of genetic programming on distributed or parallel computational platforms. It is based on a serial GP system described elsewhere [5], and was implemented on MPI so that it would be portable: AGPS runs on heterogenous clusters of Unix workstations, tightly coupled MIMD parallel architectures like the SP-2, etc. AGPS is implemented using G-Framework[3], an object-oriented genetic framework written in C++ for building GA and GP systems. When used to implement GPs the system is similar to Koza's Simple Lisp code [11]. ADF's are not implemented in the current version of G-Framework, but they may be included in future releases. For more details on the analysis and design of G-Framework see [3].

Except for its MPI basis and that it is implemented in C++, AGPS is similar to the system by Niwa and Iba [14] and to several distributed GA systems. The global population is partitioned across the processors. The literature refers to each such partition as an "island" or "meme". Each processor acts mostly as a stand-alone GP process, iterating over generations, and treating its global subpopulation as a normal population. Occasionally (the frequency is set by the user), the processors exchange some of their individuals. This process is called "migration". AGPS provides mechanisms for selecting the migrants, and the communication topology among the processors which specifies the destinations of migrants.

AGPS deals with synchronization issues among the processors in several ways. The system provides for synchronous or asynchronous migration (i.e., migration occurs either after a fixed number of generations by each processor, or migration may involve individuals from different generations.) AGPS can synchronize

the processors at run boundaries. If so, it can terminate runs synchronously (all processors complete a predefined number of generations) or asynchronously (like synchronous, unless a processor discovers a solution, in which case it notifies the other processors, each of which will then terminate its run after finishing their current generation.) Alternatively, the user can not synchronize the processors at run boundaries, in which case migrations may consist of immigrants and emigrants being from different generations and different runs.

As with other GP systems, the user can specify that a run should terminate if "no apparent" progress is being made (the definition of which is specified by a user-supplied parameter). AGPS will then start another run (after the usual process of determining the active set), reinitializing the populations at each processor.

AGPS implements migration through the use MPICH's *send* and *receive* message operators. These are processor-to-processor communication operations. Each message consists of a single string, representing the migrant individuals. Details of this encoding are in [4].

There are many issues relating to the efficacy of the various strategies outlined above, and we are only now beginning to examine them.

4 Processor Allocation

AGPS uses a straightforward mechanism for dynamically allocating processors during execution. Because AGPS is implemented on MPICH, the maximum set of processors must be specified at start-up. This set of processors is referred to as the *eligible set*. AGPS maintains a a simple software agent on each element of the eligible set. These agents use operating system queries to determine the relative load on their host processor. If the load is significantly greater than 1.0 (the default threshold is 1.2), the agent considers its host to be under external (to AGPS) demand.

At regular intervals (the default is during migration) called the *allocation interval*, the agents confer to determine the set of processors that are not in demand. This is called the *active set* (active in the sense that they will be participating in subsequent generations.) The in-demand processors enter a quiescent state through the use of a blocking message receive operation (the `MPI_recv` operator in MPICH). In due course, the AGPS processes on the quiescent processors are swapped to disk, effectively removing the CPU load caused by AGPS on those processors.

At the start of the next allocation interval, the active processors activate the quiescent processors by transmitting a message to them that completes their blocking receive. The agents reevaluate their host's load, form the new active set, and the process continues. Note that at least one processor must be active at all times, else there won't be a processor to activate all the others at the next allocation interval.

After AGPS determines the active set, it must determine the communication topology among the active processors. AGPS uses this topology during migration. Currently, AGPS only supports a ring communication topology. MPICH provides each member of the eligible set with a unique integer identifier from 0 to $n-1$, where n is the size of the eligible set. Processors with lower identifiers (modulus n) are considered to be to the "left", and those with higher numbers are to the "left". During migration, processors communicate with the nearest *active* processor on their left and right. (If only one processor is active, these will be the same.) During migration, each processor sends its migrants to its nearest left and right active neighbors. There are several ways to identify these neighbors that involve iterative communication operations. AGPS instead provides a complete definition of the active set to every processor, and the processors then examine this definition to identify their left and right active neighbors.

AGPS uses the `MPI_allReduce` operator to compute the active set. This operator is somewhat similar to parallel prefix operators on some parallel architectures. `MPI_allReduce` applies a given operator, o, across an input value on each processor. Thus, if o is the addition operator, the result will be the sum all the input values. This result is provided to each processor.

In using `MPI_allReduce`, each processor's input value is a binary string, and the reduction operator is a logical bitwise OR (the addition operator would suffice, too). The input value for the processor of rank $r \in \{0, 1, \ldots, n-1\}$, is the binary string consisting of 0's everywhere except for bit r, which is 1 if the processor wishes to participate in the active set, and 0 otherwise. Equivalently, the input value equals 2^r. The result of the reduction operator is a boolean string,

s, that defines the active set: if the rth bit is set, processor r is in the active set. Each processor examines the s to determine which processors are its nearest left and right active neighbors.

5 Results

We tested AGPS on a problem domain similar to Koza's Simple Symbolic Regression problem described in Section 7.3 of [11]. The AGPS attempts to discover a function (of two variables) that passes through 11 pairs of x and y coordinates. The target function is $y = x^3 + 0.3x^2 + 0.4x + 0.6$. The fitness function is the sum of the absolute differences in the dependent variable (y) at 11 data points: the integers zero through ten. The terminal set includes $\{x, \Re\}$ and the function set includes the basic arithmetic functions, $\{+, -, *, \%\}$. ($\Re$ is the *ephemeral constant generator.*) The termination condition is 11 hits. Each of the 11 data points is considered a hit if the value of the individual evaluated at that data pointit is less than 0.1 away from the value of the target function at that data point. All terminals, inputs and results of the functions are double precision real numbers. We used an *elitist graduated over-selection strategy* to select individuals from the population for reproduction and crossover (detailed in [5].)

AGPS was run for 5 sets of 100 runs on configurations of 1, 2, 3, 4 and 5 processors. Figure 1 shows the number of runs out of 100 that reached the termination criteria for each of the five sets. The results suggest that the probability of a run being successful increases with the number of processors used (and thus the global population size). Furthermore the increase may be super-linear. (A related form of super-linearity was observed in [1].)

Figure 2 shows the average of the maximum number of hits achieved by an element of the global population (i.e. the populations of the processors combined). The averages are taken across the 100 runs of each data set. The results suggest that increasing the number of processors leads to better performance; a larger global population tends to lead toward a higher average of hits per individual.

The conclusion that larger populations lead to better GP performance is not, in itself, surprising. This effect has been noted elsewhere (e.g. [11]). The interesting feature here is that the effect is noticeable even

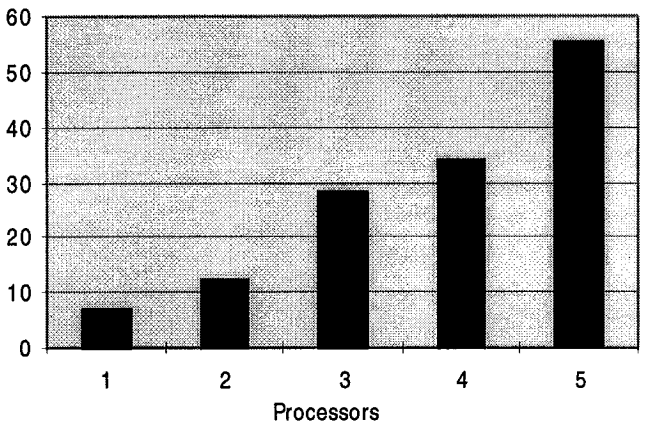

Figure 1: Percent (of 100) runs during which a solution (11 hits) was found.

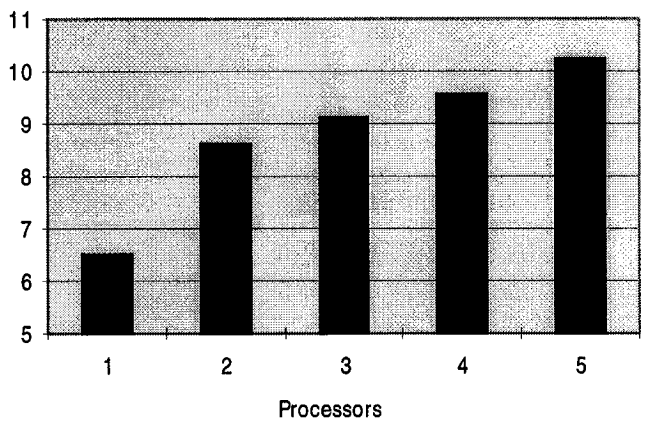

Figure 2: Average of the maximum number of hits achieved by an individual during a run.

when the population is distributed with only limited interaction between the memes.

To measure the performance of AGPS's dynamic processor allocation we watched the load and process queues on the processors in the eligible set during execution. To test that processors became quiescent during times of heavy demand, we ran CPU intensive tasks on eligible processors during AGPS runs. Investigation of the process queues on those processors showed the AGPS process was swapped to disk during heavy load. We are in the process of conducting a more formal performance analysis of AGPS's allocation strategy. We are making use of MPICH's MPE library (provides MPI profiling) toward that end.

6 Future Work

At the time of submission of this paper, we've been experimenting with AGPS for only a few months. Much remains to be implemented and formally analyzed. We outline just a few of those items here.

We are particularly interested in implementing a more complicated mechanism for entering and exiting an on-going AGPS process. Currently, the maximal set of processors that can participate in a run of AGPS is fixed at start-up. It is impossible (without restarting the whole system) later to add to the eligible set a processor that was unavailable at start-up. We are examining the feasibility of modifying AGPS to permit the insertion of new processors into the eligible set, and, conversely, to provide a mechanism for processors to remove themselves from the eligible set. Such a soluation would allow AGPS to provide flexible processor allocation to those systems that might not provide pre-emptive multitasking on all processors. This may require some modification to MPICH, so we are proceeding gingerly.

We are investigating more sophisticated techniques of resource management. These techniques include finer control of when processors can withdraw or enter the active set, and load-balancing by dynamically manipulating meme size: processors that wait on others for their immigrants slowly increase their meme size to make better use of that wasted time in the future.

7 Conclusion

AGPS provides for efficient use of the bounded resource consisting of a set of available processors. Though our performance analysis is only just begun, AGPS seems to provide better performance as it makes use of more processors. AGPS's dynamic processor allocation scheme provides a mechanism for making maximal use of available processors without monopolizing those processors for the long periods necessitated by genetic computing systems.

References

[1] D. Andre and J. Koza. Parallel genetic programming: a scalable implementation using the transputer network architecture. *Advances in Genetic Programming, Vol II*, 1996.

[2] P. Angeline and K. Kinnear, editors. *Advances in Genetic Programming, Vol. II.* MIT Press, Cambridge, MA, 1996.

[3] T. Fernandez. Analysis and design of the evolution of strategy objects. Technical report, Florida Atlantic University, 1997.

[4] T. Fernandez and M. Evett. Agps: a portable, distributed system for genetic programming. Technical report, Dept. Computer Science and Engineering, Florida Atlantic University, 1997. To be submitted as a late paper to GP-97.

[5] T. Fernandez and M. Evett. The impact of training period size on the evolution of financial trading systems. In J. Koza, editor, *GP-97, Proceedings of the Second Annual Conference*, 1997. to appear.

[6] Message Passing Interface Forum. Mpi: A message-passing interface standard. Technical report, Computer Science Dept., University of Tennessee, Knoxville, TN, 1994.

[7] W. Gropp and E. Lusk. User's guide for mpich, a portable implementation of mpi. Technical Report ANL/MCS-TM-ANL-96/6, Argonne National Laboratory, Mathematics and Computer Science Division, 1996.

[8] J. Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, MI, 1975.

[9] H. Juillé and J. Pollack. Massively parallel genetic programming. *Advances in Genetic Programming, Vol II*, 1996.

[10] M. Evett K. Stoffel and J. Hendler. A polynomial-time inheritance algorithm based on inferential distance. Technical report, Dept. Computer Science and Engineering, Florida Atlantic University, 1997. Submitted to IJCAI97.

[11] J. Koza. *Genetic programming: on the programming of computers by means of natural selection.* MIT Press, 1992.

[12] J. Koza. *Genetic programming II: Automatic Discovery of Reusable Subprograms.* MIT Press, Cambridge, MA, 1994.

[13] J. Koza, editor. *Genetic Programming 1996, Proceedings of the First Annual Conference*, Cambridge, MA, 1996. MIT Press.

[14] T. Niwa and H. Iba. Distributed genetic programming: Empirical study and analysis. In J. Koza, editor, *Genetic Programming 1996, Proceedings of the First Annual Conference*, Cambridge, MA, 1996. MIT Press.

[15] Y. Xu and M. Evett. Parallelization of the canal subsystem of the everglades landscape model. Technical report, Dept. Computer Science, Florida Atlantic Univ., February 1997.