Multi-agent System for Irregular Parallel Genetic Computations

J. Momot, K. Kosacki, M. Grochowski, P. Uhruski, and R. Schaefer

Institute of Computer Science, Jagiellonian University, Kraków, Poland
{momot,kosacki,grochows,uhruski,schaefer}@ii.uj.edu.pl

Abstract. The paper presents the multi-agent, parallel computing system (MAS) composed of a platform of software servers and a set of computing agents. The generic actions of agents and the system government are so designed that it can perform irregular concurrent genetic computations in heterogeneous computer network with a number of computation nodes and connection topology varying in time. The effectiveness of MAS solution is discussed in terms of average migration and communication overheads. Additionally, the MAS system with autonomous, diffusion-based scheduling is compared with low-level distributed implementation, which utilizes the centralized greedy scheduling algorithm.

1 Introduction

The multi-agent system (MAS) seems to be an attractive way to overcome transparency and scalability problems in distributed computing systems. Moreover MAS's are well suited to maintain loosely-coupled, locally synchronized parallel tasks due to the autonomous activity of agents (there is no overhead for global synchronization) and low cost of local communication among the agents located close together. Distributed computing systems composed of mobile tasks have recently been intensively developed (see e.g. PYRAMID in NASA [5]). The advantages of scheduling by agent migration have also been proved in [4,6]. A particular case of such loosely-coupled parallel computations is multi-deme, parallel genetic algorithms. They are an advantageous tool for solving difficult global optimization problems, especially in case of problems with many local extrema (see e.g. [1]). The Hierarchical Genetic Strategy (HGS) introduced by Kołodziej and Schaefer [2] constitutes one of their instances. The main idea of the HGS is to run a set of dependent evolutionary processes, called *branches*, in parallel. The dependency relation has a tree structure with the restricted number of levels m. The processes of lower order (close to the root of the structure) represent a chaotic search with low accuracy. They detect the promising regions on the optimization landscape, in which more accurate process of higher order are activated. Populations evolving in different processes can contain individuals which represent the solution (the phenotype) with different precision. This precision can be achieved by binary genotypes of different length or by different phenotype scaling. The strategy starts with the process of the lowest order 1, called

M. Bubak et al. (Eds.): ICCS 2004, LNCS 3038, pp. 623–630, 2004.

the root. After the fixed number of evolution epochs the best adapted individual is selected. We call this procedure a *metaepoch* of the fixed period. After every metaepoch a new process of order 2 can be activated. This procedure is called a *sprouting operation* (SO). Sprouting can be generalized in some way to HGS branches of higher order up to $m-1$. Sprouting is performed conditionally according to the outcome of the *branch comparison operation* (BCO). BCO can be also used to reduce branches of the same degree that checks the same region of the optimization landscape. Details of both the SO and BCO depend strongly upon the implementation of the HGS.

The first HGS implementation [2] utilizes the Simple Genetic Algorithm (SGA) as the basic mechanism of evolution in every process. The next implementation of the HGS_RN was obtained by using real-number encoding, normal mutation, and the simple arithmetic crossover instead of SGA (see [3]).

2 Autonomous Agent's Platform

The paper depicts an agent approach to the HGS application, which utilizes the MAS platform (see [4] and references inside). The MAS platform is composed of software servers statically allocated on computer nodes that perform information, migration, and hibernation policies for mobile computing units called agents. The MAS computing application is composed of intelligent, mobile agents that wrap the computational tasks. They can be dynamically created and destroyed. Each of them, with partial autonomy, can decide on its current location. The computing application is highly transparent with respect to the hardware platform, the number of computers and the configuration and addressing of the computer network. In order to facilitate agent building, which should combine the computational and scheduling purposes, *Smart Solid* architecture was introduced. Each Smart Solid Agent is represented by the pair $A = (S, T)$ where T is the computational task executed by the agent, including all data required for computation, S is shell responsible for the agent logic required to maintain the computational task, including the communication capabilities and scheduling mechanism. To perform scheduling the shell S enforces on computational task T the following functionalities:

- T has to be able to denominate the current requirement for computational power and RAM,
- T must allow pausing and continuing of its computation (pausing is needed for the hibernating task in case of agent migration or partitioning, and continuation is needed to restore the paused job),
- T must allow partitioned into two subtasks $T \rightarrow \{T_1, T_2\}$.

The shell S can perform the following actions: (a-1) execute T; (a-2) pause T; (a-3) continue T; (a-4) denominate own load requirement; (a-5) compute gradient, and check the migration condition; (a-6) partition $T \rightarrow \{T_1, T_2\}$ and create $A_i = (S_i, T_i), i = 1, 2$; (a-7) migrate; (a-8) disappear. These actions allow A to accomplish two goals: (G-1) - perform computation of carried task, (G-2) - find a

better execution environment. The easiest possibility to get (G-1) is to execute (a-1). A more extended decision algorithm fitted to the HGS implementation will be presented in the next section. If a new agent is created and also when the shell recognizes a significant change of load on the local servers, the agent then checks if it should continue to realize (G-1). If no, then it passes to (G-2) by using the scheduling mechanism based on the diffusion phenomenon. Briefly depicting that mechanism we can say that the shell queries task requirements for resources (action (a-4)) and computes load concentration on the local server and nearest surrounding servers (action (a-5)). To achieve those values the shell communicates with both the computational task and the local server on which the agent is located. That information is necessary to compute agent *binding energy* on the server and local load concentration gradient. The gradient specifies a possible direction for migration (destination server). If the binding energy on the destination server exceeds the binding energy on the current server more than the predefined threshold, then the agent migrates (action (a-7)). The agent can keep migrating until it finds sufficient resources. If agent A is too large to migrate, then the actions (a-6) and (a-8) are performed, and both produced agents A_1 and A_2 will start scheduling independently.

3 HGS Agents

We concentrated on the agent-oriented project of the HGS that searches global extrema of the objective function. The HGS produces a tree with nodes being demes, and the edges being created when the sprouting operation occurs. A tree is created by the stochastic process, thus its size (number of demes) might be different in various runs of the algorithm. The HGS is concurrent in its nature, because computations performed in each branch do not have much influence on each other. The only point where two demes meet is when a deme sprouts. The information passed to produce a new deme is very small - only one individual. If the HGS is synchronized regularly (as it was done in [2]), all demes stop after each metaepoch, which enables us to relatively easy employ operations of *branch reduction* (BR) and *conditional sprouting* (CS) in those points of global synchronization. CS searches all the demes of one level and checks, using BCO, whether the deme that is to be sprouted will not be too close to any of them. BR requires a comparison of each pair of the demes on the same level and, if they are searching the same area, reducing them to a single deme. These operations are complex and very time consuming. Furthermore, in the case of implementation in distributed environment, they require more communication.

The architectural idea that naturally springs to one's mind is to put one deme into an agent and let it do the calculations. However, the time needed to process one deme is small in relation to the time needed to create and maintain an agent. Thus we have decided to construct an agent in such a way that it can contain a limited number of demes. When demes sprout, children (new demes) are stored in the same agent until there is no room for them. If this occurs, a new agent with all demes that could not be stored is created.

Instead of CS we introduce *local conditional sprouting* (LCS), which behaves in exactly the same way as CS but only within one agent. This means that before creating another deme it is checked whether there are any other demes of the same level that are very close to this new deme. If there are no such cases, then we create a new deme, otherwise we don't. Another mechanism is the 'killing agent'. It walks around the platform asking other agents for data about demes computed by them and finally reduce all demes that are scanning the same region to a single one. So this agent would actually perform an operation similar to BR and also reduce redundant demes that were sprouted due to a lack of global CS.

We use a very restrictive stop condition for the demes of higher levels. A deme finishes when either it has computed a maximal fixed number of metaepochs or it is stopped when it did not make enough progress in the best fitness.

Following the idea given above we have implemented the HGS as a set of Smart Solid Agents. In T (computational task space of an agent) we store a chart of active demes ("populations") and a chart of sprouted demes ("sprouted"). The agent can denominate the requirements for computational power by estimating the upper bound of a number of metaepochs to be computed by all the active populations. T can be paused after the metaepoch for each contained deme is finished. Its activity, when trying to achieve (G-2), can be described by the code below:

```
if (thereIsBetterEnvironment()) {
    computeGradient(); // (a-5)
    if (thereIsSufficientEnvironment()) {
        pause(); migrate(); continue(); // (a-2), (a-7), (a-3)
    } else {
        // create two agents, both with half of the populations of old agent
        partition T → {T1, T2} and create Aj = (Sj, Tj), j = 1, 2; // (a-6)
        disappear(); // (a-8)
    }
}
```

The following code is activated when the agent tries to achieve (G-1).

```
execute T; // (a-1)
if (mustDoPartitioning) {
    // create two new agents first with "parents" second with "children"
    partition T → {T1, T2} and create Aj = (Sj, Tj), j = 1, 2; // (a-6)
}
disappear(); // (a-8)
```

The functioning of task T is described in the code below:

```
do {
    for (int i = 0; i < populationsCount; i++) {
        if (populations[i].endOfComputing()) {
            kill(populations[i]);
        } else {
```

```
            popualtions[i].metaepoch();
            if (populations[i].canSprout()) {
                sprouted.insert(populations[i].sprout());
            }
        }
    }
    if (isPlaceForSprouted()) { storeSprouted(); }
    else {
        mustDoPartition = true;
        return;
    }
} while (thereAreAnyLivingPopulations())
storeResults(); // returns results to the Requester unit
```

A partition is done polymorphously; it acts differently depending on the state of the "mustDoPartition" flag (see code below).

```
if (mustDoPartition) {
    T1 ← T; T2 ← sprouted;
} else {
    T1 ← first half of T; T2 ← second half of T;
}
```

Aside from Smart Solid Agents dynamically created and destroyed, the HGS application contains a single requester unit that sends the first computing agent on the platform. All the results computed by the agents are sent to the requester which stores them and then chooses the best.

4 Experiments

The MAS-HGS implementation was tested with sample inputs to check its runtime properties. The experiments were conducted within a network of 45 PC machines connected by a TCP/IP protocol based network. Machines ranged from PIII, 256Mb RAM up to dual PIV 1Gb RAM machines and worked under Linux and MS Windows operating systems.

Table 1. The average results of HGS computations with diffusive scheduling

Objective	Agents amount		Execution times [sec]			Parallel	Over-
function	Total	Average	Migration	Communication	Computation	time [sec]	head %
1	2	3	4	5	6	7	8
Rastrigin	193,0	42,7	358,9	141,7	7923,0	187,1	5,94
Griewangk	183,7	22,0	168,55	110,95	6421,3	288,2	4,17
Schwefel	163,0	40,4	288,6	138,0	22259,4	558,1	1,88

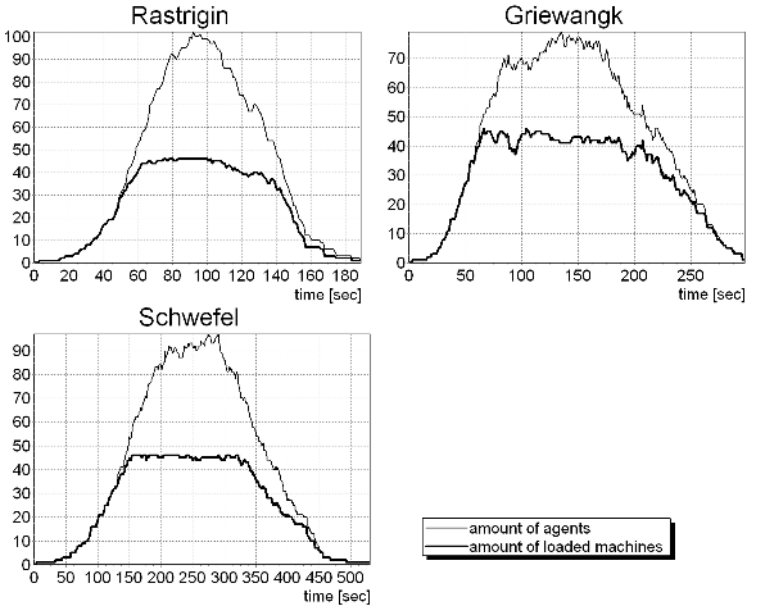

Fig. 1. The dynamics of HGS computing agents.

We performed computations for three well known global optimization benchmarks: 20-dimensional Rastrigin, 10-dimensional Griewangk, and 10-dimensional Schwefel. Each row of Table 1 contains values averaged over 10 runs performed for the particular objective. The Total Agents amount (column 2) is the number of agents produced in the whole run, while Average (column 3) is the mean number of active agents during each run for each objective. Columns 4-6 contain sums of all agents' migration, communication, and computation times. Column 7 shows the average parallel time, which includes migration and communication, measured by the requester. It is a good approximation of the mean wall clock time period of the total experiment. The computation time, stored in column 6, can be greater then the serial time for the whole computation, because significant negative synergy may occur among agents allocated on the same computer. The last column, Overhead %, shows the fraction of execution times that agents spend on migration and communication.

Figure 1 presents the amount of actively computing agents varying during the experiment runtime. We chose runs whose parameters were closest to the mean ones presented for each objective in the Table 1. A high agent dynamic is caused by the HGS strategy that sprouts many demes in the first phase of the search and then demes are reduced to ones that search close to the local extrema. Sprouting is stochastic so it is impossible to predict when the particular agent will be partitioned and, in consequence, to plan the optimal a'priori allocation for each new agent.

Table 2. Speedup of HGS computations for the Griewangk objective with Round-Robin and Diffusive Scheduling. Selected experiments with largest (a) and smallest (b) amounts of demes are shown. Section (c) presents mean values for all conducted experiments (10).

	Serial	Diffusive Scheduling			Round-Robin	
	time [sec]	Total agents amount	Parallel time [sec]	Speedup	Parallel time [sec]	Speedup
(a)	6891	244	371	18,57412	318	21,66981
	6766	299	299	22,62876	334	20,25749
(b)	3387	122	374	9,05615	163	20,77914
	2687	94	228	11,78509	131	20,51145
(c)	**4221,9**	**183,7**	**288,2**	**14,44104**	**204,9**	**20,66099**

The mean number of active agents (see column 3 in Table 1) is lower while the maximum is about two times greater than the number of processors (see Figure 1), so the upper limit of speedup was temporarily activated.

The irregularity of parallelism in the agent-oriented application may be measured as the ratio between the maximum and the mean number of agents active during the whole computation. This ratio reaches the maximum value for the Griewangk benchmark. In this case we performed a detailed speedup comparison with a low level message passing (using fast RMI communication) distributed application with a predefined number of fixed PC nodes, which utilizes centralized Round-Robin (RR) scheduling performed by the master unit. The RR is one of the well known greedy policies that can handle cases with randomly appearing tasks. It is currently implemented as follows: each new sprouted deme is registered by the master unit using the message sent by the process holding the parent deme. Next, the master unit introduces the message to the process on a selected machine, pressing it to start computation for the new deme. Both messages contain only deme parameters (deme branch degree, standard deviation of mutation, etc.) and the seed individual (see [3]). Each deme sends the report with the computation result to the master after the stop condition is satisfied.

The appearance of the tasks is totally random, therefore a low-level distributed application with fast explicit communication and RR scheduling is close to the optimal solution in the case of clusters and LANs dedicated to parallel computation. This solution may be treated as a reference point for the fastest solution towards multi-agent solution.

5 Conclusions

– Multi-agent systems (MAS) are well suited to irregular parallel computations because the number of processing units (agents) may be dynamically

adapted to the degree of concurrency of the algorithm. Moreover they allow utulizing of a multi purpose, heterogeneous computer network with a number of computation nodes and connection topology varying in time.
- The flexibility of the presented MAS solution was obtained by both architectural and implementation properties. We used Java language and CORBA framework services due to their transparency and support for object migration (serialization). The MAS is composed of a network of software servers and a set of Smart Solid Agents. Each one of the agents is a pair $A = (S, T)$, where T stands for task space and S is a shell that provides all services related to scheduling and tasks' communication. Diffusion scheduling ensures proper agent location in the dynamic network environment.
- Preparing a parallel computing application of this kind we have to specify only the task space T of agent classes. However, we are burdened by specifying all details associated with explicit low-level distributed programming.
- Smart Solid Architecture used to implement the HGS imposes coarse-grained parallelism. Grain size may be easily configured by changing agent capacity (maximum number of demes).
- Total overhead caused by the use of Agent paradigm is low in comparison with the computation time (about 5%, see Table 1). Comparison of this solution with a fine-grained, fast low-level message passing application on dedicated workstation cluster shows moderate losses of the average speedup (see Table 2 section (c)). Presented solution becomes faster for larger problems, even winning with Round Robin (see Table 2 section (a)).
- Speedup computed in all the tests is far from linear. It is caused not only by migration and communication overheads, but mainly by HGS irregularity. Though a mean number of agents is lower than the number of PCs, the maximum number is much higher.

References

1. Cantu-Paz E.: *Efficient and Accurate Parallel Genetic Algorithms.* Kluwer 2000.
2. Schaefer R., Kołodziej J.: Genetic search reinforced by the population hierarchy. in *De Jong K. A., Poli R., Rowe J. E. eds. Foundations of Genetic Algorithms 7* Morgan Kaufman Publisher 2003, pp. 383-399.
3. Wierzba B., Semczuk A., Kołodziej J., Schaefer R.: Hierarchical Genetic Strategy with real number encoding. *Proc. of the 6th Conf. on Evolutionary Algorithms and Global Optimization* Łagów Lubuski 2003, Wydawnictwa Politechniki Warszawskiej 2003, pp. 231-237.
4. Grochowski M., Schaefer R., Uhruski P.: Diffusion Based Scheduling in the Agent-Oriented Computing Systems. Accepted to *LNCS*, Springer 2003.
5. Norton Ch.: "PYRAMID: An Object-Oriented Library for Parallel Unstructured Adaptive Mesh Refinement" accepted to *LNCS*, Springer 2001.
6. Luque E., Ripoll A., Cortés A., Margalef T.: A distributed diffusion method for dynamic load balancing on parallel computers. *Proc. of EUROMICRO Workshop on Parallel and Distributed Processing*, San Remo, Italy, January 1995. IEEE CS Press.