Extremal Optimization Approach Applied to Initial Mapping of Distributed Java Programs

Ivanoe De Falco[2], Eryk Laskowski[1], Richard Olejnik[4], Umberto Scafuri[2], Ernesto Tarantino[2], and Marek Tudruj[1,3]

[1] Institute of Computer Science, Polish Academy of Sciences, Warsaw, Poland
[2] Institute of High Performance Computing and Networking, ICAR-CNR, Naples, Italy
[3] Polish-Japanese Institute of Information Technology, Warsaw, Poland
[4] Computer Science Laboratory, University of Science and Technology of Lille, France
{laskowsk,tudruj}@ipipan.waw.pl, richard.olejnik@lifl.fr,
{ivanoe.defalco,umberto.scafuri,ernesto.tarantino}
@na.icar.cnr.it

Abstract. An extremal optimization algorithm for initial Java program placement on clusters of Java Virtual Machines (JVMs) is presented. JVMs are implemented on multicore processors working under the ProActive Java execution framework. Java programs are represented as Directed Acyclic Graphs in which tasks correspond to methods of distributed active Java objects that communicate using a RMI mechanism. The presented probabilistic extremal optimization approach is based on the local fitness function composed of two sub-functions in which elimination of delays of task execution after reception of required data and the imbalance of tasks execution in processors are used as heuristics for improvements of extremal optimization solutions. The evolution of an extremal optimization solution is governed by task clustering supported by identification of the dominant path in the graph. The applied task mapping is based on dynamic measurements of current loads of JVMs and inter-JVM communication link bandwidth. The JVM loads are approximated by observation of the average idle time that threads report to the OS. The current link bandwidth is determined by observation of the performed average number of RMI calls per second.

Keywords: distributed systems, program optimization, evolutionary algorithms.

1 Introduction

Optimization of the execution time of distributed Java programs has been always a challenging task due to specific execution paradigm of object programs and particular architectural features of the JVM. Lots of works have been done in the area of dynamic load balancing and scheduling in clusters and Grids (e.g. [1, 2], a good review of the works on task scheduling on Grids is given in [3]). Usually, in the presented solutions the load balancing is done at Java program runtime based on centralized or distributed load monitoring agents. However, Java program balancing strategies should account for optimization of initial distribution of components of a Java distributed application. This aspect has not yet been sufficiently discussed in the literature,

P. D'Ambra, M. Guarracino, and D. Talia (Eds.): Euro-Par 2010, Part I, LNCS 6271, pp. 180–191, 2010.

and it is the motivation for the research reported in this paper. The problem has been only partially covered in few papers, which propose an initial optimization based on Java objects static object clustering [6, 7] or distributed byte-code clustering [8] in a set of JVMs.

In the paper, we propose an alternative way to solve the initial placement optimization problem for distributed Java programs. Finding the optimal mapping of application tasks onto computing nodes in heterogeneous environments is NP–hard, so, we use an Extremal Optimization (EO) algorithm for mapping of tasks to nodes. Extremal Optimization is a very fast co-evolutionary algorithm proposed by Boettcher and Percus [10, 12]. EO works with a single solution made of a given number of components s_i, each of which is a variable of the problem. There are two kinds of fitness functions used, one for the components and one for a global solution. In an EO algorithm, after an initial random solution is generated, the fitness value is computed for each of the components. The worst variable is randomly updated, so that the solution is transformed into another solution belonging to its acceptable neighbourhood. To avoid stucking in local optima, a probabilistic version of EO /τ-EO/ is used [10].

The paper presents a proposal of a Java program initial optimization algorithm based on the probabilistic EO approach. There are the following general steps in the algorithm:

1. Measuring properties of the executive system: CPU power and network bandwidth,
2. Execution of programs for some representative data to create their graph representation with the use of data dependency analysis methods.
3. Finding the optimal schedule of the program graph in a system of many JVMs.
4. Deployment of the application inside the runtime framework for execution.

Efficient load balancing on Grid platforms requires adequate computational and communication load metrics [4, 5]. In our solution, environment monitoring predicts CPU and network services availability based on current CPU load and network utilization**.** The applied principle is based on the observation that an average idle time that threads report to the OS is directly related to the CPU load. Object behaviour monitoring determines the intensity of communication between active objects. Its principle is based on measuring the number of method calls between active global objects and the volume of serialized data.

In this work, we have selected the ProActive Java-based framework to be an execution management support [11]. This framework for cluster and Grid computing provides a distributed Java API and a set of tools for program management in different environments such as desktop, SMP, LAN, clusters and Grid. The application model is based on Active Objects, Group Communications and Asynchronous Execution with synchronization (Futures mechanism). The framework supports SPMD, task migration, Web Services and Grid. All this is adapted to various protocols such as RMI, SSH, LSF, Globus.

The paper is composed of four parts. First, the principles of the extremal optimization are recalled. Next, the program representation and the executive system features are described. Then, the proposed version of the extremal optimization for Java program scheduling is presented. Finally, results of practical experiments concerning a cluster of JVMs implemented on multicore workstations are shown.

2 Extremal Optimization

Extremal Optimization is based on the Bak–Sneppen model of Self-Organized Criticality (SOC) [9]. SOC is based on the principle that evolution progresses by selecting against the few most poorly adapted species, rather than by expressly breeding those species well adapted to the environment. Each component of an ecosystem corresponds to a species, which is characterized by a fitness value. The least fit species together with its closest dependent species are selected for adaptive changes. As the fitness of one species changes, those of its neighbours are affected. Thus, species coevolve and the resulting dynamics of this extremal process exhibits the characteristics of SOC, such as punctuated equilibrium [9].

Extremal Optimization is a competitive alternative to other nature–inspired paradigms such as Simulated Annealing, Evolutionary Algorithms, Swarm Intelligence and so on, typically used for finding high–quality solutions to NP–hard combinatorial and physical optimization problems. Contrary to the general paradigm of Evolutionary Computation (EC), which assigns a given fitness value to the whole set of the elements of a solution and operates with a population of candidate solutions, EO works with one single solution S made of a given number of components s_i, each of which is a variable of the problem and is thought to be a species of the ecosystem. Once a suitable representation is chosen, a local fitness value φ_i is assigned to each of them. Then, at each time step the overall fitness Φ of S is computed and this latter is evolved, by randomly updating only the worst variable, to a solution S' belonging to its neighbourhood Neigh(S). This last is the set of all the solutions that can be generated by randomly changing only one variable of S by means of a uniform mutation. However, EO is competitive with respect to other EC techniques if it can randomly choose among many S' $\in$ Neigh(S).

Algorithm 1. General EO algorithm

begin
initialize configuration S at will
set $S_{best} := S$
while *maximum number of iterations* N_{iter} *not reached* ***do***
evaluate φ_i *for each variable* s_i *of the current solution S*
rank the variables s_i *based on their local fitness* φ_i
choose the rank k according to $k^{-\tau}$ *so that the variable* s_j *with* $j = \pi(k)$ *is selected*
choose $S' \in Neigh(S)$ *such that* s_j *must change*
accept $S := S'$ *unconditionally*
if $\Phi(S) < \Phi(S_{best})$ ***then***
set $S_{best} := S$
end if
end while
return S_{best} *and* $\Phi(S_{best})$
end

When this is not the case, EO leads to a deterministic process, i.e., gets stuck in a local optimum. To avoid this behaviour, Boettcher and Percus introduced a probabilistic version of EO based on a parameter τ , i.e., τ–EO. According to it, for a minimization problem, the species are firstly ranked in increasing order of local fitness values, i.e.,

a permutation π of the variable labels i is found such that: $\varphi_{\pi(1)} \leq \varphi_{\pi(2)} \leq \ldots \leq \varphi_{\pi(n)}$, where n is the number of species. The worst species s_j has the rank 1, i.e., $j = \pi(1)$, while the best one has the rank n. Then, a distribution probability over the ranks k is considered as follows: $p_k / k^{-\tau}$, $1 \leq k \leq n$ for a given value of the parameter τ. Finally, at each update a rank k is selected according to p_k so that the species s_i with $i = \pi(k)$ randomly changes its state and the solution moves to a neighbouring one $S' \in$ Neigh(S) unconditionally.

Note that only a small number of variables change their fitness, so that only a few connected variables need to be re–evaluated and re–ranked. The only parameters are the maximum number of iterations N_{iter} and the probabilistic selection value τ. For minimization problems τ–EO proceeds as in the Algorithm 1.

3 Executive System Features and Program Representation

The scheduled Java programs are executed in the system under control of the ProActive framework for cluster and Grid computing. ProActive is available as an open source Java library (GPL 2 license) providing an API for parallel, distributed, and multi-threaded computing, also with support for mobility and security. Based on the Active Objects design pattern, ProActive allows for simplified and uniform programming of Java applications distributed on Local Area Networks (LANs), Clusters, Internet Grids and Peer-to-Peer Intranets. Important ProActive features includes: support for active objects mobility, Remote Mobile Objects, Group Communications, OO SPMD, Web Services, Grid support, Various protocols: RMI, SSH, LSF, Globus, PBS.

3.1 ProActive Framework Overview

A distributed ProActive application is composed of a number of active objects. An Active Object is a standard Java object with an attached thread of control. Incoming method calls are stored in a queue of pending requests in each active object, which decides in which order to serve them. Method calls sent to active objects are asynchronous with transparent future objects and the synchronization handled by a *wait-by-necessity* mechanism (Fig. 1).

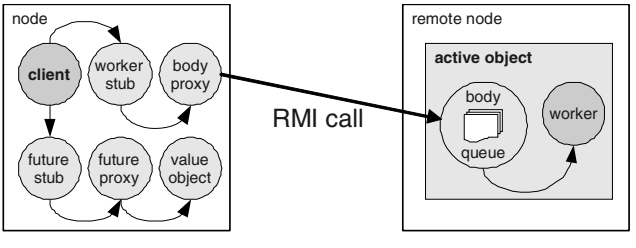

Fig. 1. Components of an active object in ProActive

An active object is composed of the instance of a **Body** with a request queue and the actual instance of some working class. An active object is referred from the outside through working class **stub,** which, in turn, connects to remote active object using **BodyProxy**. Instead of blocking on the execution of one of stub method, a **future** is created and the execution continues. After the asynchronous call to the active object has returned, the caller object holds a reference onto a future object representing the not yet available result of the call. The thread of the **Body** sets the result in the future, which results in the **FutureProxy** having a reference onto a value.

Communication between active objects is implemented by a Remote Method Invocation mechanism (Java RMI). The data to be communicated (Java objects) are serialized and passed as network messages. The communication semantics depends upon the signature of the method, with three possible cases: synchronous invocation, one-way asynchronous invocation, and asynchronous invocation with future result.

Active Objects are created on nodes of a parallel system, the deployment is specified by external XML description and/or API calls. The distribution of active objects is transparent to their clients, invoking methods of remote objects does not require the developer to use explicitly any communication or remote access mechanism.

3.2 System and Program Representation

The executive system consists of N computing resources (nodes), which in general are multicore processors. Each node is identified by an integer value in the range [0, N−1]. The current status of system resources is given by the node power α_i which is the number of instructions computed per time unit on the node i and average load of each node $\ell_i(\Delta t)$ in a particular time span Δt: $\ell_i(\Delta t)$ ranges in [0.0, 1.0], where 0.0 means a node with no load and 1.0 a node loaded at 100%; network bandwidth β_{ij}: the communication bandwidth between any pair of nodes i and j. Hence $(1 - \ell_i(\Delta t))\alpha_i$ represents the power of the node i available for the execution of the tasks scheduled by our algorithm. The current status is supposed to be contained in tables based either on statistical estimations in a particular time span or gathered by tracking periodically and by forecasting dynamically resource conditions [13, 14].

An application is represented as a weighted directed acyclic graph $G_{dag} = \{P, E\}$, where P is a set of communicating tasks, and E is a set of data transfers between tasks. Each task p_k, k in $\{1..|P|\}$ has a weight γ_k which represents the number of instructions to be executed. These weights are determined either during the sample execution of the program for some representative data or provided by application developer. An edge weight ψ_{km} represents the amount of data to be sent from task k to another m-th task. Similarly, these weights can be sampled or provided explicitly by a developer. A program is executed according to the *macro dataflow* model. Tasks start their execution when all needed data arrived through task incoming edges. When task is finished, it sends produced data to the succeeding tasks. The graphs are static and deterministic.

The program representation corresponds to ProActive distributed application in which a task is a thread of an Active Object and an edge is a method call in another Active Object. To preserve the DAG constraints, all loops are unrolled or encircled inside the body of a single method of an Active Object. The execution of an Active

Object method, which constitutes the task, can start when this object collected all necessary data. During execution, an Active Object communicates only with local objects. At the end of execution, the method sends data to the successors.

4 Task Scheduling by Extremal Optimization

Using the provided system and program representation, the optimization of initial distribution of components of an application translates to the problem of mapping the application divided into P tasks on N nodes. Since tasks address non-dedicated resources, their own local computational and communication loads must be considered to evaluate the computation time of the tasks of the program to be scheduled. There exist several prediction methods to face the challenge of non–dedicated resources [15, 16].

Solution encoding. A scheduling solution S is represented by a vector $\mu = (\mu_1,\ldots, \mu_P)$ of P integers ranging in the interval [0, N−1], where the value $\mu_i = j$ means that the solution S under consideration maps the i–th task of the application onto processor node j. The number of processor cores is not represented inside the solution encoding, however, it is taken into account when estimating the global and local fitness functions while solving the scheduling problem. This will be explained below.

Global fitness function. The global fitness accounts for the time of execution of a scheduled program. The execution time of the scheduled program is provided by a program graph execution simulator. The simulator assigns time annotations to program graph nodes based on the processor computing power availability and communication link throughput available for a given program execution.

Algorithm 2. Program graph execution simulation procedure

```
begin
  Mark entry task of the graph as ready
  While not all tasks are visited do
        T:=The ready task with the earliest starting time
        C:=The core which has the earliest Availability_time
        Place task T on node of core C
        Task_completion_time(T):=Availability_time(C) + Execution_time(T)
        Availability_time(C) := Task_completion_time(T)
        Mark T as visited
        foreach succesor task Sc of task T do
            DRT := Task_completion_time(T) + Communication_time(T, Sc)
            if DRT > Ready_time(Sc) then Ready_time(Sc) :=DRT endif
            if all data of Sc arrived then mark Sc as ready endif
        endforeach
  endwhile
end
```

The program graph is executed in the processor node containing in general multiple cores. Each task is executed on one core following a macro data flow program execution paradigm. The task computation time is evaluated taking into account the concurrent execution of the task i and of all the other tasks assigned to the same node j.

The communication time depends on the bandwidth effectively available on involved node. The total time used by the node j to execute all the tasks assigned to it is determined by the task completion time tc_j of execution of the latest task on any of the cores belonging to the node j. The tc_j value is determined by the simulation of the execution of tasks on cores of the node j. This scheduling strategy assigns tasks (considered as threads) to processor cores using the heuristics presented as Algorithm 2. Algorithm 2 determines also the data ready time DRT for each task.

Knowing the task completion times tc_j for all processor nodes in the system, the **total fitness function Φ** of the solution μ is given as $\Phi(\mu) = \max(tc_j)$ for $0 \leq j \leq N-1$.

In extremal optimization for task scheduling, a local fitness function should possibly detect and measure a quality of assignment of a task on a processor, so that the relevant element(s) of the solution can be selected for improvement. We propose an approach based on a local fitness function composed of three local fitness sub-functions, which well characterize task assignments to processors. Tasks with large fitness function have big probability of selection for improvement.

Local fitness sub-function 1. For a system of heterogeneous processors interconnected by a heterogeneous network in which our program is executed with sharing resources with other system load, the local fitness function 1 (LFF1) of a task is the delay of the execution start of a task comparing the data ready time DRT of a task. We call this delay the initial execution delay.

Local fitness sub-function 2. The second local fitness function (LFF2) supports the LFF1 by moving the tasks, which are placed on the overloaded node, to other nodes. We estimate the execution completion time of all processors and select the processor node which finished program execution as the latest one. We assign to all the tasks assigned to this node, the maximal delay value found during the calculation of fitness function 1 (LFF1) or a special constant value *Const*, thus increasing the probability they would get selected during the ranking process. Thus LFF2(i) = max (LFF1(i)) if there exists i for which LFF1(i) $\neq 0$ otherwise LFF2(i) = *Const*.

Local fitness sub-function 3. The third local fitness function (LFF3) also supports the LFF1 by moving the tasks, which belong to dynamic critical path of the graph and that are improperly placed on nodes, to other nodes. The dynamic critical path is the longest path in the scheduled graph. We determine the dynamic critical path by traversing the graph from the sink task to the entry task. Then, we look for the tasks on critical path, whose parent task from the critical path is placed on a different node than the task's node. We assign to all those tasks whose parents are assigned to different nodes, the maximal delay value found during the calculation of fitness function 1 (LFF1) or a special constant value *Const*, thus increasing the probability they would get selected during the ranking process. Thus LFF3(i) = max (LFF1(i)) if there exists i for which LFF1(i) $\neq 0$ otherwise LFF3(i) = *Const*.

The **local fitness function:** the local fitness function (LFF) of a task i is evaluated using the LFF1 and one of LFF2 or LFF3 sub-functions. We have designed and used three variants of local fitness function. It is done in the following way:

Variant 1 – the local fitness function uses only the LFF1 sub-function:

LFF(i) = LFF1(i)

Variant 2 – the local fitness function uses the LFF1 and LFF2 sub-functions:

LFF(i) = LFF1(i) for tasks for which LFF1(i) $\neq 0$ and LFF2 = 0,

LFF2(i) for tasks i which are assigned to the processor node which has finished program execution as the latest one.

Variant 3 – the local fitness function uses the LFF1 and LFF3 sub-functions:

LFF(i) = LFF1(i) for tasks for which LFF1(i) ≠0 and LFF3 = 0,

LFF3(i) for tasks i which belong to the dynamic critical path and that are assigned to the different processor node than the node of its parent.

5 Experimental Results

During experiments we have used two sets of synthetic graphs and the graph of a medical application – ART algorithm (reconstruction of tomographic scans [19]).

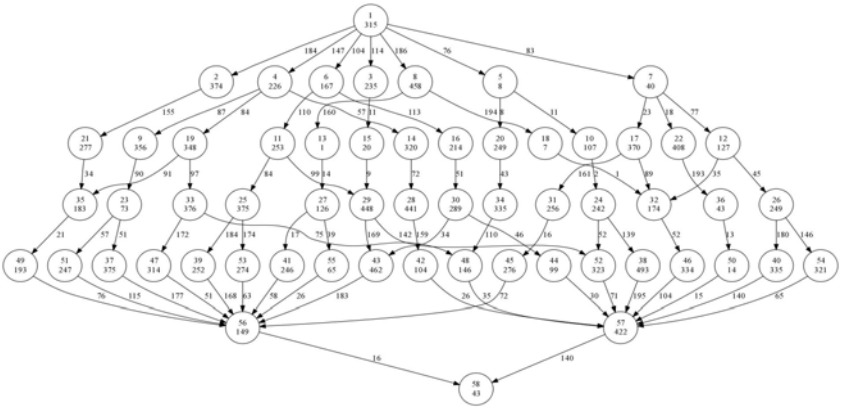

Fig. 2. The structure of a synthetic exemplary application graph (gen-3a)

The first set of synthetic graphs consists of seven randomly generated graphs (*gen-1...3, gen-3a...d*), with layered structure. The graph structure of synthetic graphs is shown in Fig. 2 (on an example of *gen-3a* graph). Nodes are denoted by the node number (upper level) and the node weight (lower level). Each task (node) of this graph represents a mixed float- and integer-based generic computation (random generation of matrix of doubles, then floating-point matrix-by-vector multiplication, than rounding to integers and integer sort) with execution time defined by node weight (the weight controls the number of iterations of the generic computation). The second set of synthetic graphs consists of two hand-made graphs with known optimal mappings (*gen-m1, gen-m2*), with a general structure similar to the structure of randomly generated graphs. The structure of ART algorithm graph is shown in Fig. 3 (*art-ct* graph, only the upper and the lower parts of the graph are shown, due to its large height). The sizes of all graphs used during research are the following: 35, 36, 58, 58, 56, 62, 49, 44, 86, 211.

To be able to compare several variants of extremal optimization algorithms we used a cluster of 7 homogeneous dual core processor nodes for program graph scheduling and program execution under ProActive.

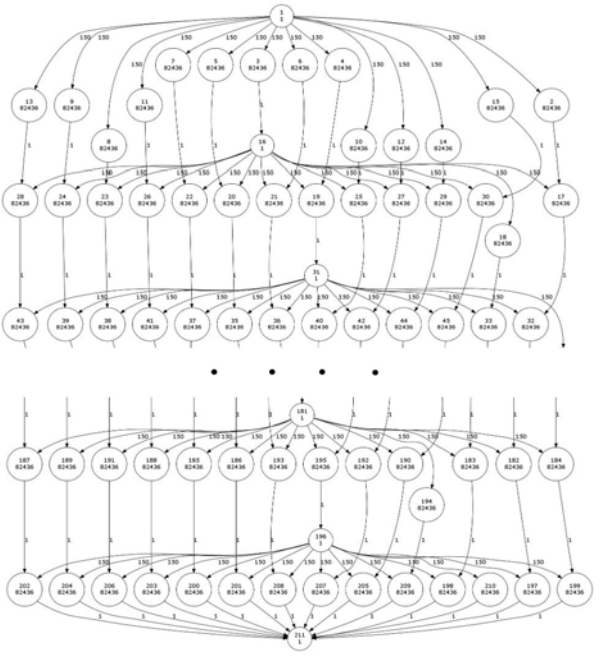

Fig. 3. The structure of the graph of ART algorithm

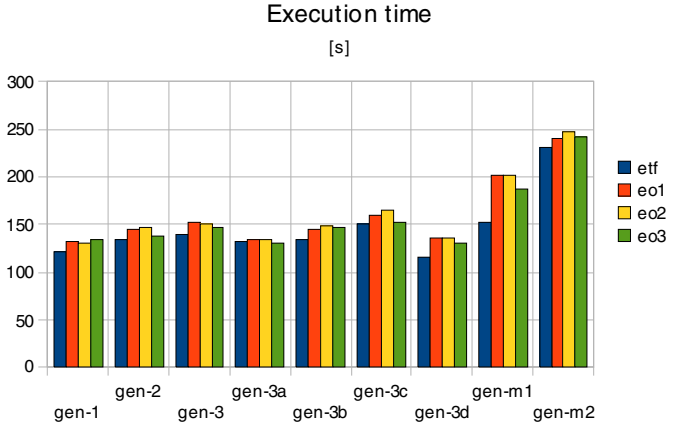

Fig. 4. The real execution times of the scheduled synthetic program graphs for different scheduling algorithms

The following extremal optimization algorithm variants have been used during the experiments:

eo1 – the algorithm utilizing variant 1 of the local fitness function (using only the task start delay as a optimization criterion (LFF1 only),

eo2 – the algorithm utilizing variant 2 of the local fitness function (based on a combination of LFF1 and LFF2),

eo3 – the algorithm utilizing variant 2 of the local fitness function (based on a combination of LFF1 and LFF3).

For the comparative experiments we used a ETF list scheduling algorithm (Earliest Task First). The ETF implementation is based on the description in [18].

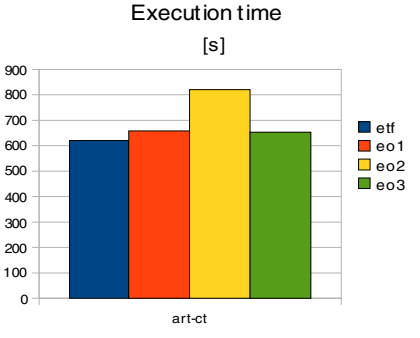

Fig. 5. The real execution times of ART program graphs for different scheduling algorithms

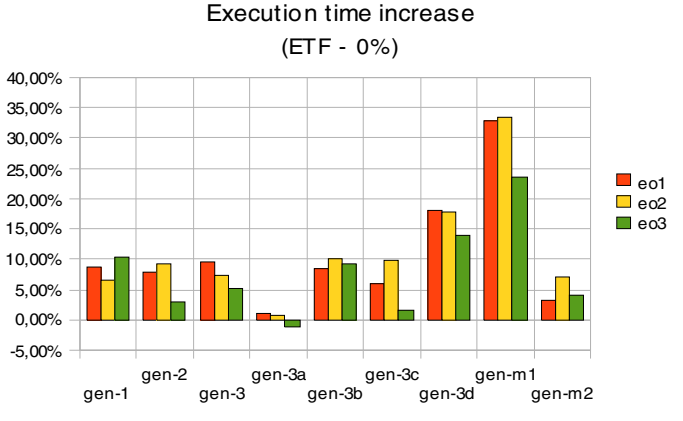

Fig. 6. Real execution time increase different EO methods, comparing the ETF algorithm

Comparison of real execution times of an exemplary application scheduled by different methods is presented in Fig. 4 and Fig. 5. The different variants of EO method have obtained similar quality of initial mappings of applications. The best results are obtained by **eo3** algorithm. The detailed difference in quality of mappings between

EO methods and ETF algorithm is shown in Fig. 6. The typical execution time increase, comparing the EO and ETF algorithm, is below 10% (the only exception are gen-3d and gen-m1 graphs). This result confirms that the **eo3** algorithm gives the best results among different variants of EO methods.

The experimental results show that EO technique is able, in general, to draw the same level of the quality of results as classical scheduling and mapping approaches like ETF algorithms. In our opinion, it is a quite good result, taking into account the simplicity of the basic principle of extremal optimization method.

The second part of experimental research was an empirical extrapolation of actual time complexity of presented algorithms. For this purpose we used a set of large, randomly generated graphs (the number of nodes from 350 to 3500), which were scheduled by extremal optimization and ETF algorithms. The actual running times confirmed the theoretical complexity of EO and ETF methods, which is approximately $C(n^2)$ for EO and $C(n^3)$ for ETF (where n is the size of the graph). Although the time complexity of the EO methods is lower than that of ETF, the actual running times of different kinds of the EO algorithms were much longer than the running times of ETF algorithm. We consider this as the main drawback of EO method.

Experimental results indicate that extremal optimization technique can be useful for large mapping and scheduling problems when we pay special attention to runtime optimization of an EO algorithm. For small sizes of application graphs, it is suggested to use classic scheduling methods as ETF list scheduling.

6 Conclusions

The paper has shown how to optimize a distributed program schedule by using the extremal optimization technique. For homogeneous systems, the extremal optimization algorithm has delivered results comparable to the ETF algorithms. The execution times of the scheduled programs determined by simulation were close to the real execution time of the synthetic programs corresponding to the scheduled graphs.

References

1. Jimenez, J.B., Hood, R.: An Active Objects Load Balancing Mechanism for Intranet. In: Workshop on Sistemas Distribuidos y Paralelismo, WSDP 2003, Chile (2003)
2. Yamaguchi, S., Maruyama, K.: Autonomous Load Balance System for Distributed Servers using Active Objects. In: 12th International Workshop on Database and Expert Systems Applications, Munich, Germany, pp. 167–171 (September 2001)
3. Don, F.: A taxonomy of task scheduling algorithms in the Grid. Parallel Processing Letters 17(4), 439–454 (2007)
4. Toursel, B., Olejnik, R., Bouchi, A.: An object observation for a Java adaptative distributed application platform. In: International Conference on Parallel Computing in Electrical Engineering (PARELEC 2002), pp. 171–176 (September 2002)
5. Olejnik, R., et al.: Load balancing in the SOAJA Web Service Platform. In: 4th Workshop on Large Scale Computations on Grids (LaSCoG 2008), pp. 459–465. IEEE CS, Los Alamitos (October 2008)

6. Fiolet, V., et al.: Optimizing distributed data mining applications based on object clustering methods. In: Parallel Computing in Electrical Engineering 2006 (PARELEC 2006), pp. 257–262. IEEE CS, Los Alamitos (2006)
7. Laskowski, E., et al.: Java Programs Optimization Based on the Most–Often–Used–Paths Approach. In: Wyrzykowski, R., Dongarra, J., Meyer, N., Waśniewski, J. (eds.) PPAM 2005. LNCS, vol. 3911, pp. 944–951. Springer, Heidelberg (2006)
8. Laskowski, E., et al.: Byte-code scheduling of Java programs with branches for Desktop Grid. Future Generation Computer Systems 23(8), 977–982 (2007)
9. Sneppen, K., et al.: Evolution as a self–organized critical phenomenon. Proc. Natl. Acad. Sci. 92, 5209–52136 (1995)
10. Boettcher, S., Percus, A.G.: Extremal optimization: an evolutionary local–search algorithm. In: Bhargava, H.M., Kluver, N.Y. (eds.) Computational Modeling and Problem Solving in the NetworkedWorld, Boston (2003)
11. Baude, F., et al.: Programming, Composing, Deploying for the Grid. In: Cunha, J.C., Rana, O.F. (eds.) GRID COMPUTING: Software Environments and Tools. Springer, Heidelberg (January 2006)
12. Boettcher, S., Percus, A.G.: Extremal optimization: methods derived from coevolution. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 1999), pp. 825–832. Morgan Kaufmann, San Francisco (1999)
13. Fitzgerald, S., et al.: A directory service for configuring high–performance distributed computations. In: Sixth Symp. on High Performance Distributed Computing, Portland, OR, USA, pp. 365–375. IEEE Computer Society, Los Alamitos (1997)
14. Czajkowski, K., et al.: Grid information services for distributed resource sharing. In: Tenth Symp. on High Performance Distributed Computing, pp. 181–194. IEEE Computer Society, San Francisco (2001)
15. Wolski, R., et al.: The network weather service: a distributed resource performance forecasting service for metacomputing. Future Generation Computer Systems 15(5-6), 757–768 (1999)
16. Gong, L., et al.: Performance modeling and prediction of non–dedicated network computing. IEEE Trans. on Computers 51(9), 1041–1055 (2002)
17. Hwang, J.-J., et al.: Scheduling Precedence Graphs in Systems with Interprocessor Communication Times. SIAM. J. Comput. 18(2), 244–257 (1989)
18. Karypis, G., Kumar, V.: Multilevel k-way Partitioning Scheme for Irregular Graphs. J. Parallel Distrib. Comput. 48(1), 96–129 (1998)
19. Kak, A.C., Slaney, M.: Principles of Computerized Tomographic Imaging. IEEE Press, New York (1988)