Distributed Java Programs Initial Mapping Based on Extremal Optimization

E. Laskowski\(^{(1)}\), M. Tudruj\(^{(1,3)}\), I. De Falco\(^{(2)}\), U. Scafuri\(^{(2)}\), E. Tarantino\(^{(2)}\), R. Olejnik\(^{(4)}\),

\(^{(1)}\) 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

1 Introduction

The motivation for the research reported in the paper is optimized execution of distributed Java programs in clusters of Java Virtual Machines (JVM) implemented on workstations and desktop Grids. Optimization of the execution time of distributed object programs has been always a challenging task due to specific execution paradigm of object programs and particular architectural features of the JVM. Several research papers have been written on dynamic load balancing and scheduling in clusters and Grids [1, 2]. In the cited solutions load balancing is done at Java program runtime based on centralized or distributed load monitoring agents. A good review of the works on task scheduling on Grids is given in [3]. Efficient load balancing on Grid platforms requires adequate computational and communication load metrics [4, 5]. 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, however it has been 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.

This paper proposes a new way to solve the initial placement optimization problem for distributed Java programs. Finding the optimal mapping of application tasks onto the computing nodes in heterogeneous environment 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 [9]. EO works with a single solution made of 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 sticking in local optima, a probabilistic version of EO /\(\tau\)-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 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 a Method Call Graph (MCG) with the use of method dependency graph and measured data.
3. Finding the optimal schedule of the MCG graph in a system of many JVMs.
4. Deployment of the application inside the ProActive framework for execution.

Environment monitoring (system observation) 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. It also accounts for the maximal number of RMI calls per second. Object behaviour monitoring determines the intensity of communication between active objects. Its principle is based on measuring the number of method calls between ProActive active (global) objects and the volume of serialized data.

Execution of distributed Java programs in clusters and Grids is usually done using an execution management support. In this work, we have selected the ProActive Java-based framework for cluster and Grid computing [11]. This framework 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 Remote Mobile Objects, Group Communications and Asynchronous Execution with synchronization (Futures mechanism), OO SPMD, task migration, Web Services and Grid support. All this is adapted to various protocols such as rmi, ssh, LSF, Globus.

The paper will be 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 Applied to Initial Deployment of Java Programs

In nature, highly specialized, complex structures often emerge when their most inefficient elements are selectively driven to extinction. Such a view 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. This idea has been applied successfully in the Bak–Sneppen model [9] which shows the emergence of Self–Organized Criticality (SOC) in ecosystems. According to that model, each component of an ecosystem corresponds to a species, which is characterized by a fitness value. The evolution is driven by a process where 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 was proposed by Boettcher and Percus and draws upon the Bak–Sneppen mechanism, yielding a dynamic optimization procedure free of selection parameters. It represents a successful method for the study of NP–hard combinatorial and physical optimization problems [10, 12] and 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 such NP–hard problems. Differently from the well–known paradigm of Evolutionary Computation (EC), which assigns a given fitness value to the whole set of the components of a solution based upon their collective evaluation against a cost function and operates with a population of candidate solutions, EO works with one single solution $S$ made of a cost function and operates with a population of candidate solutions based upon their collective evaluation against a problem. Differently from the well–known paradigm of finding high–quality solutions to such NP–hard problems, EO is competitive with respect to other EC techniques if it can randomly choose among many $S’ \in \text{Neigh}(S)$. 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 $\tau$, i.e., $\tau$–EO. According to it, for a minimization problem, the species are firstly ranked in increasing order of fitness values, i.e., a permutation $\pi$ of the variable labels $i$ is found such that: $\phi_i(1) \leq \phi_i(2) \leq \ldots \leq \phi_i(n)$, where $n$ is the number of species. The worst species $s_j$ is of rank 1, i.e., $j = \pi(1)$, while the best one is of 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 $\tau$. Finally, at each update a generic 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 $S’ \in \text{Neigh}(S)$ unconditionally. The only parameters are the maximum number of iterations $N_{\text{eo}}$ and the probabilistic selection value $\tau$. For minimizing problems $\tau$–EO proceeds as in the Algorithm 1.

We have designed and used three variants of local fitness function. All parameters necessary for computing the value of any variant of local fitness functions are obtained during the execution of program graph simulation procedure (see Algorithm 2).

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
    N := the node of task T
    C := the core of N which has the earliest Availability_time
    Place task T on core C of node N
    TaskCompletion_time(T) := Availability_time(C) + Execution_time(T)
    Availability_time(C) := TaskCompletion_time(T)
    Mark T as visited
end while
for each successor task Sc of task T do
    DRT := TaskCompletion_time(T) + Communication_time(T, Sc)
    if DRT > Ready_time(Sc) then
        Ready_time(Sc) := DRT
    endif
    if TaskCompletion_time(T) > LastParent_time(Sc) then
        LastParent_time(Sc) := TaskCompletion_time(T)
    endif
end for
end

Local fitness function a. For the system of heterogeneous processors inter-connected by a heterogeneous network in which our program is executed with sharing resources with other system load, the local fitness function (LFFa) 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.

\[ LFFa(t) = Availability\_time(t) – Ready\_time(t) \]

Local fitness function b. The second local fitness function (LFFb) is the extension of the LFFa function. The LFFb moves the tasks belonging 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 different node than the task's node. We assign to all those tasks whose parents are assigned to different node, the maximal delay value found during the calculation of fitness function a (LFFa) or a arbitrary constant value, thus increasing the probability they would get selected during the ranking process.

\[ LFFb(t) = LFFa(t) \quad \text{when } t \text{ does not belong to DCP} \]

\[ Const \quad \text{when } t \text{ belong to DCP and its parent} \]

\[ \text{on DCP is not on the same node} \]

where \( Const = \max (LFFa(t)) \) if there exists \( t \) for which \( LFFa(t) \neq 0 \) otherwise \( Const = \text{arbitrary value} > 0 \)

Local fitness function c. The local fitness function (LFFc) of a task is the complete delay of task execution comparing the execution under optimal conditions, i.e. there is no communication overhead nor resource contention between tasks and the task is executed on the fastest processor. We call this delay the total execution delay.

\[ LFFc(t) = TaskCompletion\_time(t) – LastParent\_time(t) – FastestExecution(t) \]

where \( FastestExecution(t) \) – the execution time of task \( t \) on the fastest processor

3 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 [14]). The first set of synthetic graphs consists of seven randomly generated graphs (gen-1...3, gen-3a...d), with layered structure. 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, then 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 following extremal optimization algorithm variants have been used during the experiments: eo-a, eo-b, eo-c, based on described in the paper local fitness functions (local fitness function LFFa, LFFb and LFFc respectively). For the comparative experiments we used a list scheduling algorithm with the ETF (Earliest Task First) heuristics. The ETF implementation is based on the description from [13]. We used a cluster of 7 homogeneous dual core processor nodes for program graph scheduling and program execution under ProActive.

Comparison of real execution times of an exemplary application, scheduled by different methods is presented in Fig. 1 and Fig. 2. The different variants of EO method obtained similar quality of initial mappings of applications. For synthetic graphs, the best results are obtained by eo-b algorithm, however the eo-c method is only marginally worse. For ART application graph, the eo-c method is the best one among the different EO variants. The typical execution time increase, comparing

Figure 1: The real execution times of the scheduled synthetic program graphs for different scheduling algorithms.

Figure 2: The real execution times of ART program graphs for different scheduling algorithms.
the EO and ETF algorithm, is below 10% (the only exception is gen-m1 graph, for which ETF was able to find the optimal solution). 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.

In another experiment we empirically extrapolated 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 very much longer than the running times of ETF algorithm. We consider this as the main drawback of EO method. Among investigated EO variants, the eo-a and eo-c are the fastest, since the eo-b method introduces additional run-time overhead due to the dominant sequence tracking.

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

4 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