Scalable Computing: Practice and Experience
Volume 13, Number 3, pp. 269–284. http://www.scpe.org
ISSN 1895-1767


# A DISTRIBUTED PROGRAM GLOBAL EXECUTION CONTROL ENVIRONMENT APPLIED TO LOAD BALANCING

JANUSZ BORKOWSKI[*], DAMIAN KOPAŃSKI[*], ERYK LASKOWSKI[†], RICHARD OLEJNIK[‡], AND MAREK TUDRUJ[†*]

**Abstract.** The paper is concerned with a new distributed program design environment based on the global application states monitoring. The environment called PEGASUS (from Program Execution Governed by Asynchronous SUpervision of States) supplies to a programmer a ready to use control primitives to design distributed program execution control in which decisions for synchronous and asynchronous control actions are based on predicates evaluated on global application states. Such strongly consistent global application states are automatically constructed by the run-time system which additionally provides mechanisms for their analysis and organizing the respective program execution control in processes and threads of user programs executed in multicore processors. The PEGASUS control mechanisms are graphically supported in the respective program design framework. The paper first presents main general features of the PEGASUS environment. Next, it presents a method for load balancing inside distributed programs based on a set of parameters which are dynamically measured during program execution. Then, the paper presents how the described load balancing method can be implemented inside the PEGASUS environment taking as an example distributed programs for solving the Traveling Salesman Problem (TSP).

**Key words:** distributed program design tools, global application states monitoring, load balancing;

**1. Introduction.** Distributed programs execution control based on the monitoring of global application states is an important problem in designing many distributed application programs. In recent years, many modern distributed program frameworks were published, which propose new paradigms in the design of distributed programs and systems. As their number is too big to enumerate them all in this paper, we will point out only some of them: active objects as in the ProActive framework [1], message-driven objects as in Charm++ framework [2], new virtualization approach to provide software for chipset services in systems of multicore processors as in vSMP framework of ScaleMP [3], a task based data flow programming model as in StarSS [4], which hides much of internal parallelism from programmers. The proposed solutions put also emphasis on different aspects of the distributed program and system design like FELI support for CMP-based systems [5], which automatically allocate application data to on-chip memories without user programming. All the cited solutions contribute to the development of the design methods for systems based on multicore processors, however, they do not contain any support for the distributed program execution control based on the global application states monitoring.

In the history of programs execution management there were initial tries to include global states monitoring in the execution control but not developed into programming environments meeting common standards. Linda environment [6] was based on a common global tuple space for the exchange of globally available control information. The user processes could write and read in the tuple space. Initial primitives for the design of global control for interactive software components were included in coordination languages. In Manifold and Reo frameworks [7] primitives for inclusion of communicating software components into coordinated structures were embedded. This was done without any notion of a global state introduced in these systems. In the Meta system [8] distributed programs could be designed based on communicating components with the use of a notion of a global application state. In this system, application processes could send messages to a global monitor on their local states. Consistent global states were constructed on these states and some forms of global predicates could be evaluated. In Meta, a complicated formal framework based on guards could construct program control based on global states. Some simplification of the Meta formalism was attempted in the Lomita language [8]. However, it was not successfully efficient due to very costly message broadcasts of ordered states. Global control constructs for the OCCAM language were proposed in [9] with an implementation based on replication of global state variables. The first legacy usable framework for the asynchronous global execution control in distributed programs based on monitoring of global application states was implemented in a graphical parallel program design system PS–GRADE [10] for construction of distributed programs in the C language and standard

[*]POLISH-JAPANESE INSTITUTE OF INFORMATION TECHNOLOGY, UL. KOSZYKOWA 86, 02-008 WARSAW, POLAND ({JANB,DAMIAN.KOPANSKI,TUDRUJ}@PJWSTK.EDU.PL)

[†]INSTITUTE OF COMPUTER SCIENCE POLISH ACADEMY OF SCIENCES, WARSAW, POLAND ({LASKOWSK,TUDRUJ}@IPIPAN.WAW.PL)

[‡]COMPUTER SCIENCE LABORATORY OF LILLE (UMR CNRS 8022), UNIVERSITY OF SCIENCES AND TECHNOLOGIES OF LILLE, FRANCE ({RICHARD.OLEJNIK}@LIFL.FR)

communication libraries: PVM and MPI.

The features of the PS–GRADE and other distributed program design frameworks based on global application states monitoring have been conceptually and practically extended in PEGASUS (from Program Execution Governed by Asynchronous SUpervision of States) which is a basis for this paper. The main extension is the graphically supported distributed program control flow design framework in which the flow of control depends on the predicates computed on distributed application global states.

This paper discusses the use of the facilities of global state monitoring and program execution global control available in the PEGASUS environment to organize load balancing in distributed applications [11]. The features of the PEGASUS framework are original in this sense since, to our knowledge, no operational contemporary environment provides similar ones.

The PEGASUS environment provides global program execution control constructs for distributed programs. They assume a modular structure of parallel programs based on the notions of processes and threads. The PEGASUS global control constructs enable global control of the synchronous and asynchronous character. For synchronous program execution control the constructs logically bind program modules and define the involved global control flow which depends on the application global states. For the asynchronous program execution control, the PEGASUS control constructs enable asynchronous change of process and threads behaviour based on global application states. The monitoring and the management of the global application states in PEGASUS is implemented by the use of special control processes called the synchronizers which automatize the implementation of the program execution control based on global application states. The synchronizers collect local state information from processes and threads, automatically construct global strongly consistent application states, evaluate relevant control predicates on global states and provide a distributed support for sending control Unix-type signals to distributed processes and threads to stimulate the desired control reactions. The repertoire of considered local and global states, the control predicates and the reactions to them are user programmed using a special API provided in the system. It is the run-time system, which provides the necessary implementation of this control model. The design of the global control flow is graphically supported and the provided GUI is decoupled from the API for data processing inside modules. The proposed global control constructs enable better verification and are less error prone.

The PEGASUS framework aims at both hardware and software features to prevent global control overheads. The hosting system for PEGASUS has a triple communication network, built of three separate parallel networks which are serviced by three separate communication libraries whose implementation is not conflicting. One network is used for interprocessor computational data communication (Ethernet). Another network (Infiniband) is used for control data communication including transfers of state information and program execution control signals. The third network (Fast Ethernet) is used for communication involved in processor clock synchronization necessary for detection of strongly consistent application global states.

The contribution of this paper is to show how the necessary control for dynamic load balancing can be implemented under the PEGASUS framework. The PEGASUS framework seems to be especially convenient to design dynamic load balancing algorithms in user programs. For the load balancing under PEGASUS for programs in the C/C++ language, we have adapted the iterative method which has been earlier designed by us for load balancing of object-oriented programs in the Java language [12]. However, the global control infrastructure of PEGASUS enables designing load balancing support for practically any of load balancing methods. A good overview of the known load balancing methods and principal load balancing systems is given in [13].

Under PEGASUS, all load balancing control (except for sending load states reports and receiving work control signals by worker threads) is done by synchronizers in the background of the primary computations (in parallel with them). The synchronizers (organized in a hierarchical distributed structure) receive load reports, judge the load imbalance, take decisions upon balancing operations and instruct the worker processes and threads how to continue computations, including the migration of the computational load. The direct contacts of worker threads with the infrastructure of synchronizers is by shared memory inside processors, so it is fast and scarcely disturbing the thread's computational work since the load balancing control is done asynchronously not causing any busy waiting. Only transfers of control data at higher levels of the synchronizer tree is done by message passing. But this is done after strong reduction of control data at lower levels of the synchronizer tree and it is done only when the load balancing decisions can not be taken at these lower levels. In fact, such transfers for global decisions are unavoidable in any load balancing distributed infrastructure, so, in this respect, the performance of load balancing under PEGASUS is not worse than in other systems. However,

in PEGASUS, a programmer has a ready to use convenient global control infrastructure based on automatic construction of global consistent load state to design and master load balancing for distributed computations. The presented approach partially leverages also our earlier works, reported in [14, 15].

The paper consists of three parts. In the first part the features of the assumed PEGASUS program design and execution environment are described. In the second part, the principles of the proposed load balancing strategy, implemented using global application states monitoring are presented. The third part describes the design of an exemplary application which is the algorithm for load balancing in solving the Traveling Salesman Problem by the Branch & Bound method.

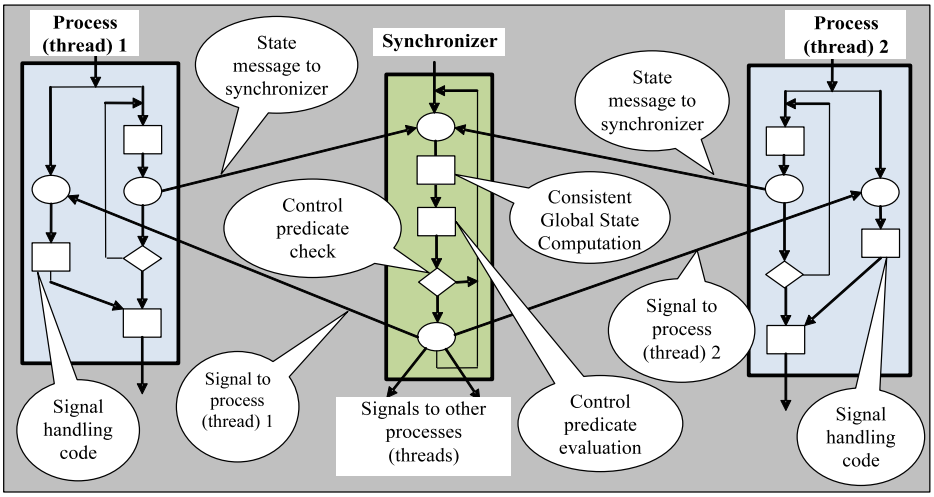



Fig. 2.1: Processes (threads) co-operating with a synchronizer.

**2. Distributed program execution control based on global application states .** The PEGASUS environment enables designing distributed programs written in the C/C++ languages built of processes and threads in which program execution control is governed by the global states of the entire distributed application. The environment works under control of the Linux operating system and provides the run-time framework for the designed programs. The PEGASUS environment includes a built-in programming infrastructure which supports designing program global execution control based on predicates computed on global application states. Program execution control under PEGASUS framework has features of the asynchronous and synchronous control. The asynchronous control enables constructing internal process and threads behavior which depends on the global application states. It corresponds to a local process/thread control by global states. It is organized by process/thread reporting reporting of local states to special monitors of global states which reconstruct global application states and organize reactions to control signals issued after an analysis of global states.

The synchronous control enables constructing program execution control in which the flow of control between program modules depends on the global application states, discovered by the mentioned above global state monitors. The synchronous control paradigm includes gathering local states responsible for decisions on global control flow inside programs, computing control predicates on global states and issuing signals to the flow of control switching blocks in programs. in response to the signals the switches direct the flow of control in the program in desired way.

The design of the program execution control in PEGASUS is graphically supported. The system enables Graphical User Interface for designing distributed program control flow graphs using global high level control flow primitives which are global application states sensitive. The graphical phase of design is further supported by the Application Program Interface to define the attributes of the global control flow. The system provides also the API to design asynchronous control of design the process and thread programs whose behavior is depending in the asynchronous way on the global application states.

The executive distributed systems are assumed to be built of multicore processors interconnected by a message passing network. The network is used for interprocessor computational data communication at the

process/thread levels with the use of standard message passing MPI library. For computational data communication between threads inside multicore processors communication by shared memory is used, supported by the OpenMP and Pthreads libraries. The implementation of global program execution control requires control data communication between processors and processor cores. This communication is decoupled from the computational data communication and is implemented using separate software and hardware means. The programmer specifies the variables which are to be used as attributes of this control using special API. It is the system which assures the necessary software/hardware communication environment for respective control data transfers and automatically generates the necessary fragments of program code. His code uses additional dedicated networks for control data transfers necessary for implementation of global control mechanisms. A programmer is able to control assignments of processes to processor nodes and threads to processor cores.

**2.1. Asynchronous model of program execution control based on global states..** The asynchronous program execution control is based on special control processes called the synchronizers introduced to the API and GUI of distributed programs. The general scheme for organizing the asynchronous program execution control is shown in Fig. 2.1.

Application program processes (threads) send messages on their *states* to special globally accessible processes (threads) called *synchronizers*. A synchronizer collects local state messages, determines the application's strongly consistent global state, evaluates control predicates and stimulates desired reactions to the signals in application components.

A strongly consistent global state (SCGS) means a set of fully concurrent local states detected unambiguously by a synchronizer [16]. Processor node clocks are synchronized with a known accuracy to enable the construction of strongly consistent global states by projecting the local states of all processes or threads on a common time axis and finding time intervals which are covered by recognized local states in all participating processes or threads [17].

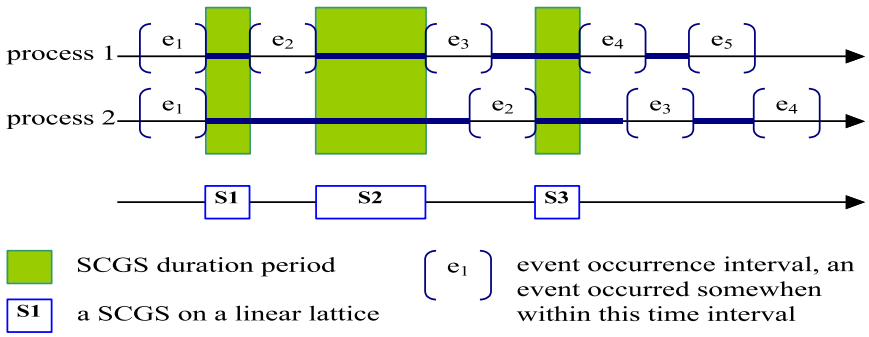



Fig. 2.2: Detection of strongly consistent global states.

The way of reconstructing strongly consistent global states from messages on local process/thread states is shown in Fig. 2.2. The synchronizers receive messages on local states by means of messages on starting and ending events of the states accompanied by time stamps which are given with the accuracy of the local clocks synchronization in processors. The time stamps are treated by the SCGS detection algorithms as time intervals with the event appearance time points expressed with the accuracy equal to the clock synchronization skew. For multicore processors, the threads issue state messages with the time stamps based on the processor clock shared by all existing threads. The strongly consistent states are those which are not covered by the uncertainty interval of events. A strongly consistent state for a group of processes/threads appears between time points not covered by any uncertainty intervals of the appearance of constituent local state events.

The way a synchronizer operates is shown in Fig. 2.3. The SCGS detecting program is present in every synchronizer. The algorithm works permanently and on each received local state message automatically it tries to detect a next SCGS. For each detected SCGS, one or more control conditions (control predicates) are computed. Predicates are specified as blocks of code in C. If a predicate value is true, then a number of control signals are sent by the synchronizer to selected application processes (threads). Inside programs allocated to the same processor signals are sent as Unix signals, which work asynchronously in the way which resembles

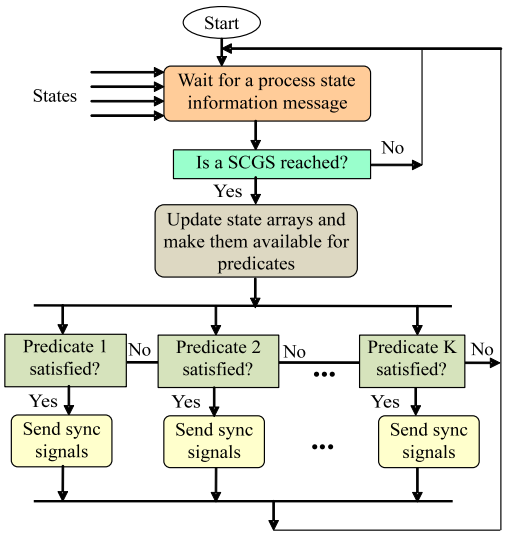



Fig. 2.3: Synchronizer control flow diagram.

interrupts. For signals sent to a remote processor, Unix signals are converted into messages sent by message passing with the use of the MPI communication library. When a message reaches a distant processor it is again converted — this time into a Unix signal, which is asynchronously received by the code of a target process or thread. Together with a signal some data can be transferred.

Two types of reaction to control signals in target processes or threads are possible. The first type of the reaction is a signal-driven activation (interrupt), which breaks current computation and activates a reaction code associated with the region. After completion of the reaction code the broken computing resumes. The second type of reaction to a signal is a signal-driven cancellation. It stops current computation and activates a cancellation handling procedure associated with the region. Program execution resumes just after the abandoned region. The described mechanism of distant Unix signals resembles the paradigm of distributed interrupts.

Since the messages representing Unix signals have to be transmitted over an external message passing network. The transfer time can be here very long. An important problem is the control of the validity of reactions for the signals which travel with a long delay. Sometimes the signaled program, which is not suspended but continues execution after sending the local state message, is advancing execution too far in respect to the acceptable time distance between the local signal and the reaction cased by it. In the code of a process (thread), regions sensitive to incoming control signals can be marked by special delimiters. If the process (thread) control is inside a region sensitive to a signal and the signal arrives, then a reaction is triggered. Otherwise, the reaction is neglected.

**2.2. Synchronous model of program execution control based on global states.** The second distributed program execution control mechanism in the PEGASUS environment involving the global state monitoring concerns defining the flow of control in distributed programs based on the global application state monitoring. PEGASUS framework provides global parallel control structures which are on PARALLEL DO (PAR) and JOIN constructs, embedded (if needed) into standard control statements of high level languages (IF, WHILE...DO, DO...UNTIL, CASE) but governed by predicates on application global states, for some of them see Fig. 2.4. Such global control constructs are graphically supported in PEGASUS. The programmer design its flow of control in the distributed program by drawing a flow of control graph using specially prepared GUI.

The predicate *GP3* in the synchronizer *S* assigned to the *N0* logical processor is evaluated based on a global state generated from local state messages received from the program blocks *P1, ..., Pn*, assigned to logical processors *N1,...,Nn* (for the simplicity of the graph representation we do not draw the respective state message transfer edges). Logical processors are further assigned to physical processors by the program mapping facility provided in the PEGASUS run-time. Based on the value of *GP3* a binary control signal is sent to the switch *SW*, which governs the flow of execution control in the PARALLEL DO–UNTIL construct. Usually the program blocks which participate in global control constructs are additionally asynchronously controlled based

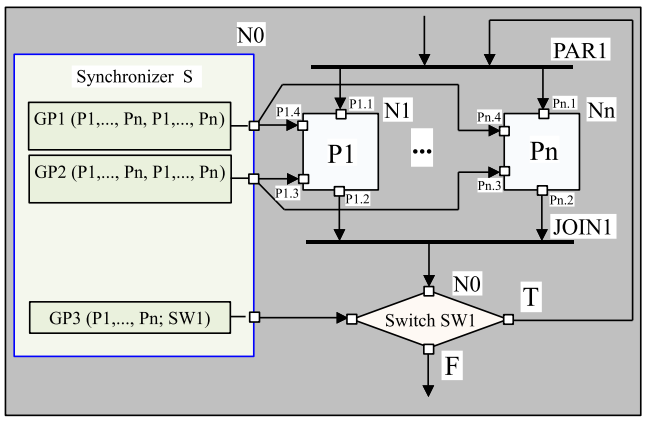



Fig. 2.4: PARALLEL DO-UNTIL construct with an asynchronous control predicates GP1, GP2 and a control flow predicate GP2.

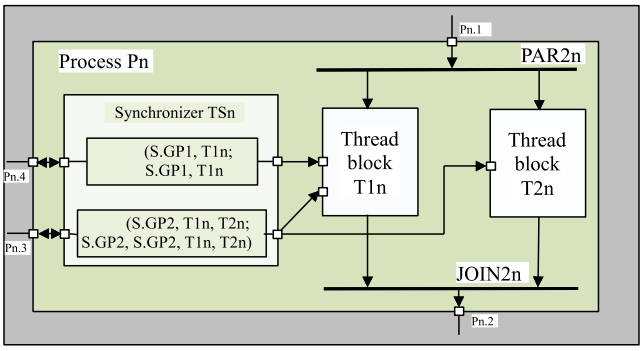



Fig. 2.5: Structure of a terminal program block

on global application states. It is done by the predicates *GP1* and *GP2*. They receive local state messages from *P1, ..., Pn*, program blocks evaluate a predicate condition and send control signals to *P1, ..., Pn*.

The control construct shown in Fig. 2.4 is a replicated control construct which means that the program blocks *P1, ..., Pn* have the same internal configuration and the same program code. The program blocks *P1, ..., Pn* can contain nested other high level control constructs on program blocks. The nesting can be repeated by special clicking on the block with the highest index. If the nesting is not to be introduced since the block does not have any high level conditional control construct nested inside, another special click on the block opens a terminal block design window. The internal structure of a program terminal block is shown in Fig. 2.5. Since the PARALLEL-DO-UNTIL construct from Fig. 2.4 was replicated, Fig. 2.5 represents the control flow of the block *Pn* with the highest index in the replication. The terminal block structure is designed of a number of thread blocks (*T1n, T2n*) placed under a PAR construct and a thread level synchronizer (*TSn*). The thread blocks composed of a number of threads with the same code but working on separate data. The thread level synchronizer implements an asynchronous control model in respect to the thread blocks *T1n, T2n*. Threads in the thread blocks can send local state messages to *TSn*. *TSn* reconstructs the strongly consistent global states of threads belonging to the process *Pn*. Upon reaching an SCGS, *TSn* activates control predicates evaluation on the global states. As a result *TSn* can send control signals to threads in *T1n, T2n* thread blocks to modify their behaviour asychronously. The exit from the thread blocks is synchronized by the JOIN2 block which works as a barrier for all threads in the blocks *T1n, T2n*. *TSn* can also send some discovered global states together with some data to the higher level synchronizer *S* beyond the *Pn* process. The synchronizer *S* can collect such partial states of the synchronizers *TSi* to construct some more global program states to evaluate some global predicate on the reconstructed more global states. Depending on the values of the predicates the synchronizer *S*

can send control signals back to the thread level synchronizers *TSi*. *TSi* can send control signals to the threads working inside their processes *Pi*.

Fig. 2.6 shows a graphical representation of the global PARALLEL IF control construct. In this construct the Switch *SW* control block supervises parallel execution (PAR construct) of a set of replicated program blocks *P1, ..., Pn*. The blocks are assigned to *P1, ..., Pn* parallel processors. The Switch is controlled by the control signal generated by the global predicate *GP1* embedded in the synchronizer. The predicate is evaluated based on the global state composed of local states provided by program blocks *B1, ..., Bn* existing somewhere in the program. For the simplicity of the graph representation we do not draw the respective state message transfer edges. Program blocks *P1, ..., Pn* are additionally asynchronously controlled by the predicates *GP2* and *GP3*. The predicates are evaluated based on the local states provided by the blocks *P1, ..., Pn*. The exit from the PAR construct is synchronized by the JOIN lock working as a barrier. More details on the PEGASUS framework can be found in [11] and [18].

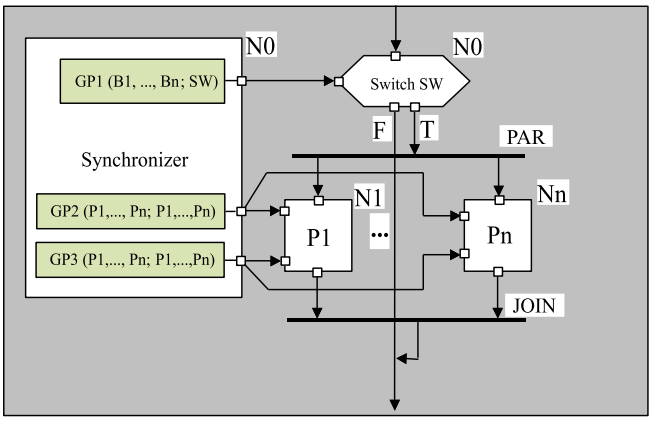



Fig. 2.6: PARALLEL IF construct with a control flow predicate GP1 and an asynchronous control predicate GP2, GP3.

**3. Load balancing algorithm.** The efficient execution of parallel programs requires balancing of computational loads among available nodes of parallel system. The problem of load balancing is NP–hard [19], thus it is necessary to apply different kinds of heuristics from various fields, such as prefix sum, recursive bisection, space filling curves, work stealing and graph partitioning.

In the static case of the load balancing problem, i.e. when the computational tasks co-exist for the entire duration of the parallel program and their workload is known, it is possible to solve it using the graph partitioning algorithm. That's why METIS [20], a well known graph partitioning framework, is widely used for mapping and static load balancing of parallel applications.

When we encounter changes of a workload and the varying availability of resources, the problem of load balancing becomes a much more hard to solve. In such a case, the only feasible approach is to apply dynamic, on–line load balancing. Also for irregular problems, i.e. when the workload depends on the processed data, the only solution is the dynamic approach.

There exist several dynamic load balancing strategies [21]. The simplest ones are based on the greedy heuristics, where the largest workloads are moved to the least loaded processors until the load of all processors is close to the average load. The more sophisticated algorithms use some refinement strategies, where the number of objects migrated is reduced or the communication between different objects is also considered.

Depending on the execution model of parallel application, dynamic load balancing is implemented by a migration of the application components (processes or threads) or by a data redistribution among computing nodes. In both cases, the goal is to achieve such a distribution of workloads that guarantees the highest possible efficiency of the overall application execution. In the strategy presented in the paper we focus at data migration as a basic load balancing mechanism.

**3.1. Load balancing based on global states.** In the approach presented in the paper, we use the global state monitoring infrastructure provided by the PEGASUS environment, as a tool to achieve dynamic

load balancing of parallel applications. The motivation for this approach is as follows: global state monitoring provides very efficient, low-latency way to detect and distribute load balancing informations, while asynchronous global control enables for instant response and load balancing accomplishment.

In the proposed implementation of the load balancing control, the complete load balancing algorithm is placed in sychronizers which are fully programmable. This way we are able to use many existing load balancing methods like those based on graph partitioning (METIS [20], Zoltan [22], etc.). However, it would require an extensive load redistribution by many time-consuming distant thread-to-thread load transfers, to follow the global optimal work partition. Thus, we propose a new strategy to avoid such flaws and to allow for real load migration only if unavoidable by other methods.

The overall scheme of load balancing, presented here, is an extended version of our former algorithm for Java-based distributed applications, see [12] for more details. Besides improvements inside balancing heuristics, the main difference, compared to the algorithm from [12], consits in the adoption of general parallel application model, applicable for wider range of computational problems.

The load balancing approach, proposed in the paper, consists of two main steps: **detection** of imbalance and its **correction**. The first step uses some measurement infrastructure to detect the functional state of the computing system and executed application. In the second step, we migrate some load from overloaded computing nodes to underloaded computing nodes to balance the workloads.

The goal of the algorithms is to balance the load of computing nodes of the parallel system. As it will be explained later, during detection of imbalance and its correction, the algorithm takes into account both computational demands of the application and its inter-process communication pattern.

**3.2. System and application observations.** The on–line detection of the state of the system is necessary since the computing nodes (workstations) can be heterogeneous, moreover they can have different and variable computing capabilities over time. A load imbalance occurs when the differences of loads between the computing nodes become too big.

Applications' observation is necessary since they can be irregular in general, thus their workloads can change in an unpredictable way. The application observation mechanism provides knowledge of the application behavior during its execution. This knowledge is necessary to undertake adequate and optimal load balancing decisions.

There are two types of measurements in the proposed load balancing method for the PEGASUS environment:

**System level observations** – they provide general functional indicators, e.g. CPU load, which are universal for all kinds of applications. The parameters are measured using software sensors, for that the load balancing mechanism of the PEGASUS environment installs observation kernels on computing nodes.

**Application specific observations** – they include measurements which have to be done in each application since they furnish data on application–dependent behavior. An example of this kind of data is the workload of a process (or thread). It must be bound to the application since such data can depend for example on the amount of data to be processed in the future which is known only to the application logic.

The PEGASUS global states monitoring infrastructure is a convenient tool to organize both the system and the application level observations. In our implementation, application program processes and system observation agents send messages on local state changes to *load balancing synchronizer*, where they are processed and appropriate reactions are computed using the method described in next sections. Similarly, reactions are organized as asynchronous program execution control. Load balancing logic is implemented as control predicates inside *a load balancing synchronizer*. Figure Alg. 3.1 presents a general scheme of the proposed algorithm. The rest of this section describes functions and symbols used in the flowchart in Alg. 3.1.

**3.3. Detection of load imbalance.** Computing nodes composing the parallel system can be heterogeneous, therefore to detect the load imbalance, we need to known both the processors availability and their computing power. The heterogeneity of nodes disallows us to directly compare measurements taken on computing nodes whose computing powers are different. Thus, to compare the computing nodes' load, we propose to use the availability index of a CPU computing power on the node $n$:

$$\mathrm{Ind}_{\mathrm{avail}}(n) = \mathrm{Ind}_{\mathrm{power}}(n) * \mathrm{Time}^{\%}_{\mathrm{CPU}}(n)$$

where:

$\mathrm{Ind}_{\mathrm{power}}(n)$ — computing power of a node $n$, which is the sum of computing powers of all cores on the node,

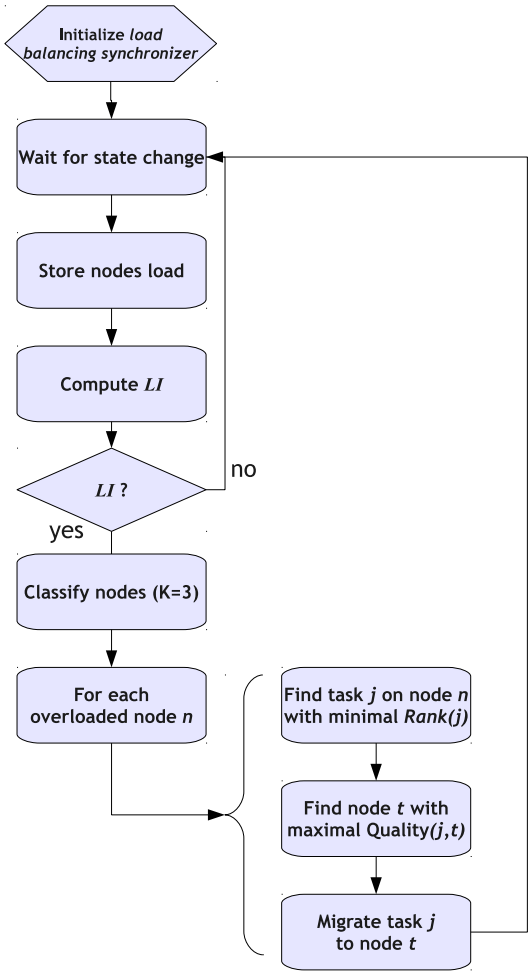



Fig. 3.1: General scheme of load balancing algorithm

$\text{Time}^{\%}_{\text{CPU}}(n)$ — the percentage of the CPU power available for computing threads on the node $n$, periodically estimated by observation agents on computing nodes.

The way the availability index of a CPU computing power is computed is not obvious, so some explanation follows. The computing power of the node is determined by the *calibration process* [23]. For each node, the calibration should be performed in a consistent way to enable comparisons of calibration results (they can be expressed in MIPS, MFLOPS or similar). The calibration needs to be done only once when the nodes join the system. The percentage of the CPU power available for a single computing thread is computed as a quotient of the time during which the CPU was allocated to the probe thread against the time span of the measurement (see [23] for more details and the description of the implementation technique). $\text{Time}^{\%}_{\text{CPU}}(n)$ value is the sum of the percentage of CPU power available for the number of probe threads equal to the number of CPU cores on the node.

A load imbalance $LI$ is defined as the difference of the availability indexes between the most heavily and the least heavily loaded computing nodes composing the cluster, which can be determined as:

$$LI = \max_{n \in N}(\text{Ind}_{\text{avail}}(n)) \leq \alpha * \min_{n \in N}(\text{Ind}_{\text{avail}}(n))$$

where:
$N$ — the set of all computing nodes, $\alpha$ — a positive constant number.

Power indications $\text{Ind}_{\text{power}}$ and CPU time use rate $\text{Time}_{\text{CPU}}$ are collected and sent to *load balance synchronizer* by local system agents as state messages.

The proper value of the $\alpha$ coefficient can be determined using both statistical and experimental approaches. Following our previous research [12] on load balancing algorithms for Java-based distributed computing environment, we can restrict the value to the interval $[1.5 \ldots 2.5]$. These experimental values give good results because they are neither too restrictive nor too tolerant for the load imbalance.

**3.4. Correction of load imbalance.** At the beginning of this step, we know the state of computing nodes (i.e. their availability indexes), which is an outcome of the detection of load imbalance. Now, we detect overloaded computing nodes and then we'll re-balance them (i.e. transform into the normally loaded by the migration of the load).

**3.4.1. Classification of computing nodes.** We classify $n$ computing nodes into three categories, based on the computed availability indexes: overloaded, normally loaded and underloaded. To build categories, we use the K-Means algorithm [24] with $K = 3$. The three centers that we choose are the minimum, average and maximum availability indexes, where the average index is simply the average of indexes measured during the last series of measures over the whole cluster.

**3.4.2. Selection of candidates for migration.** To correct load imbalance, we migrate the load from overloaded computing nodes to underloaded ones. The load for migration has to be selected carefully, so that the migration does not cause the deterioration of overall computing conditions.

The loads are represented by the data processing activities of the threads which are running on computing nodes. Two parameters are used to characterize the load that we want to migrate:

a) the *attraction* of a load to a computing node,
b) the *weight* of the load.

The attraction of a load to a computing node is expressed in terms of communication, i.e. it indicates how much a particular thread communicates with others allocated to the same node. A strong attraction means frequent communication, so, the less the load is attracted by the current computing node, the more interesting it is to be selected as a migration candidate. The computational weight of the load gives the quantity of load which could be removed from the current node and placed on another.

Both the *attraction* and *weight* are application-specific metrics, which should be provided by an application programmer in the form of state messages sent to *load balance synchronizer*:

1. $\mathrm{COM}(t_s, t_d)$ is the communication metrics between threads $t_s$ and $t_d$,
2. $\mathrm{WP}(t)$ is the load weight metrics of a thread $t$.

In our load balancing algorithm we prefer to move an entity whose quantity of work is neither too big, nor too small. Thus, the smaller the distance is to the average thread loads, the more the load is interesting for migration.

The attraction of the load $j$ to the actual computing node:

$$\mathrm{attr}(j) = \sum_{o \in L^*(j)} (\mathrm{COM}(j, o) + \mathrm{COM}(o, j))$$

where:
$L^*(j)$ — the set of threads, placed on the same node as a thread $j$ (excluding $j$).

The load deviation compared to the average quantity of work of the node $j$:

$$\mathrm{ldev}(j) = |\mathrm{WP}(j) - m_{WP}| \tag{3.1}$$

where:
$m_{WP} = \frac{\sum_{o \in L(j)} \mathrm{WP}(o)}{|L(j)|}$,
$L(j)$ — the set of threads, placed on the same node as the thread $j$ (including $j$).

The formula above allow to compute the attraction of an entity to the local node in order to compare it with the attractions of other loads of this node. The comparison formula are:
(1) global attraction measure:

$$\mathrm{attr}^{\%}(j) = \frac{\mathrm{attr}(j)}{\max_{o \in L(j)}(\mathrm{attr}(o))}$$

(2) load deviation compared to the average quantity of work:

$$\mathrm{ldev}^{\%}(j) = \frac{\mathrm{ldev}(j)}{\max_{o \in L(j)}(\mathrm{ldev}(o))}$$

The element to migrate is the one for which a weighted sum of last two parameters has the minimal value:

$$\mathrm{Rank}(j) = \beta * \mathrm{attr}^{\%}(j) + (1 - \beta) * \mathrm{ldev}^{\%}(j) \tag{3.2}$$

where:
$\beta$ — a real between 0 and 1. Its choice remains experimental. Let us notice however that the bigger $\beta$ is, the bigger is the weight of the object attraction.

**3.4.3. Selection of the target computing node for migration.** The target computing node for migration should be both underloaded and strongly attracted by migrated load. In order to select the target, we use the formula which is based on weighted sum of two selection criteria.

The first criterion to qualify a computing node as a migration target is the computing node power availability indexes. We prefer the one whose availability index is the highest, because it is actually the least loaded. We also take into account the number of *waiting* threads in the potential targets ($T_{\mathrm{wait}}(n)$ – the set of waiting threads on a node $n$). We consider them, however, only as potential load, which must be taken under consideration together with the related load currently processed on the computing node.

The second criterion is based on the attraction of a selected load entity to this node. A relationship of attraction of the load $j$ to node $n$ is defined as follows:

$$\mathrm{attrext}(j, n) = \sum_{e \in T(n)} (\mathrm{COM}(e, j) + \mathrm{COM}(j, e))$$

where:
$T(n)$ — the set of threads, placed on a node $n$.

The formula to select the target for migration is as follows (we normalize all the values related in the interval $[0 \ldots 1]$):

$$\mathrm{Quality}(j, n) = \gamma * \mathrm{attrext}^{\%}(j, n) + (1 - \gamma) * \mathrm{Ind}^{\%}_{\mathrm{avail}}(n) \tag{3.3}$$

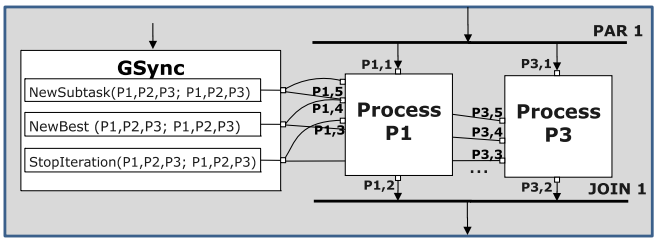
with $\gamma \in [0 \ldots 1]$ and

$$\mathrm{attrext}^{\%}(j, n) = \frac{\mathrm{attrext}(j, n)}{\max_{e \in N}(\mathrm{attrext}(j, e))}$$

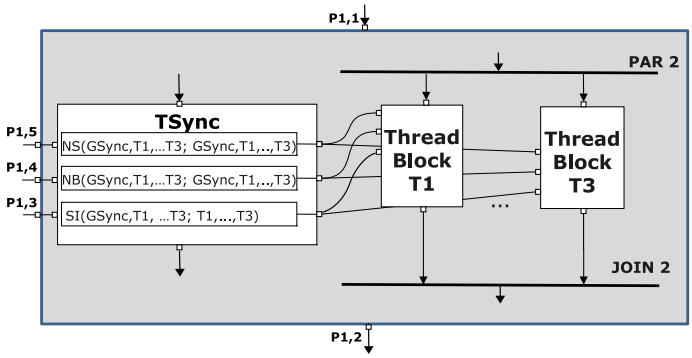
$$\mathrm{Ind}^{\%}_{\mathrm{avail}}(n) = \frac{\mathrm{Ind}^{*}_{\mathrm{avail}}(n)}{\max_{e \in N}(\mathrm{Ind}^{*}_{\mathrm{avail}}(e))}$$

$$\mathrm{Ind}^{*}_{\mathrm{avail}}(n) = \mathrm{Ind}_{\mathrm{avail}}(n) - \mathrm{Ind}_{\mathrm{avail}}(n) * \frac{|T_{\mathrm{wait}}(n)|}{|T(n)|} \tag{3.4}$$

We evaluate the above equations for all potential computing node targets for a load which is a candidate for migration. As the new location for the load we choose the computing node which maximizes equation 3.3. The value of the coefficient $\gamma$ has to be determined using experimental verification.




(a)




(b)

Fig. 3.2: (a) Control flow graph for the TSP program. (b) Control flow graph for worker process of TSP.

## 4. Example of load balancing based on global state monitoring.

**4.1. General assumptions on the solution search.** We will present now how the described load balancing method can be applied to a Traveling Salesman Problem (TSP) solved by a parallel Branch and Bound (B&B) method. The global control in the algorithm supports both organizing a reduced search of the TSP problem solution space due to the B&B method and load balancing of the search computations in distributed processes. A parallel solution space search is done following a "master-slave" method implemented using a number of worker search processes assigned to multicore processors. Each worker process is composed of a number of worker threads and a thread level synchronizer. The solution space is organized as a tree of salesman trajectories which are gradually being developed and searched by worker threads under control of synchronizers. The solution space is divided into search subproblems (subtasks) which correspond to subtrees of the total solution tree. To reduce the search in the solution space, the global state monitoring with a bounding function is organized in all executive processes and threads.

Synchronizers are organized in a hierarchical structure and play the role of masters in the solution search algorithm. A central global synchronizer at the highest hierarchy level divides the set of all search subproblems into subtask pools meant for execution by worker processes. The subtasks are directed for execution to worker threads by the thread level synchronizer placed in each worker process. The thread synchronizer evaluates an estimate of the search work to be done (the current process load), based on state messages sent to it by worker threads. Then, the thread synchronizer sends the estimate to a global load balancing synchronizer. Based on it, the global synchronizer computes the deviation of the load of worker processes. This deviation is used to work out the load balancing decisions for the processes (i.e. the rate and amount of subtask delivery).

The global synchronizer maintains also the value of the best known solution found so far in worker processes. The global synchronizer updates this value if a thread synchronizer in a worker process informs it about a still better solution it has found. The global synchronizer distributes the updated globally best *min_dist* solution to all thread synchronizers of all worker processes. The thread synchronizers distribute this value to all their

threads. The threads use this *min_dist* value to bound further development of every partial trajectory solution (containing a partial set of towns) they consider. They also compare to *min_dist* every full trajectory solution (containing the full set of towns) found in the search process and inform the thread synchronizer on each better new solution (the trajectory and its distance value). The thread synchronizers compare the new received *min_dist* values to those they know and send them to the global synchronizer and back to worker threads if it is smaller than the currently known *min_dist*.

**4.2. General load balancing strategy and its global implementation.** The control flow graph of the TSP by the B&B method in PEGASUS is shown in Fig. 3.2(a). We have a replicated PAR construct in this graph with 3 parallel worker processes $P_i$, controlled by the predicates of the global synchronizer *GSync*. Worker processes are composed of worker thread blocks and the thread synchronizers *TSynch*, Fig. 3.2(b). Each worker thread block contains replicated worker threads which are assigned to the same processor core. *TSyncs* periodically report their current loads to the global synchronizer *GSync*. Based on these reports *GSync* works out load balancing decisions which direct pools of search subtasks to *TSyncs* in the way to balance loads in worker processes. *TSyncs* also report to *GSync* the best solutions with the trajectory lengths smaller than the *min_dist* values known to their threads. *TSyncs* send to *GSync* asynchronous requests for new subtasks if the pool of the subtasks they have for worker threads is close to exhaustion.

There are 3 predicates evaluated in the *GSync* synchronizer. The *NewSubtask* predicate computes the global mean load in the system and compares it to the loads reported by *TSyncs*. Inside this predicate the $\mathrm{ldev}(j)$ parameter (see equation 3.1 in the previous section) is computed for each process. Based on $\mathrm{ldev}(j)$ load balancing decisions are taken. The decisions are different for different search stages for the entire TSP problem. In the non-terminal search stages in which not all search subtasks have yet been distributed by *GSync* to search worker processes (more exactly the *TSync* synchronizers), the decision determines the number of new search subtasks which are to be sent to the processes which are soon to be idle. In the terminal stage of the solution search when all subtasks have already been distributed, the decision determines the subtask migration from the most loaded worker process to the least loaded one. In this case, the *NewSubtask* predicate of *GSync* selects the process which is too heavily loaded and which should migrate part of its subtasks to another less loaded process. We avoid direct communication of subtasks between threads of different worker processes located on different processors, so the migration is executed at the level of the TSync synchronizers. The selection for migration decision is done using the equation 3.2 given in the previous section. However, in our TSP problem, communication between worker threads does not appear since worker threads are independent, so the migration decision is taken only based on the $\mathrm{ldev}(j)$ parameter. The decision upon the target process to receive load migration is taken in general based on the $\mathrm{Quality}(j, n)$ parameter (see equation 3.3 in the previous section). In our example, due to the independent tasks, this parameter is reduced to $\mathrm{Ind}^*_{\mathrm{avail}}(n)$ (equation 3.4). The selection of the target for migration is done by *GSync* based on the information on partial processor power available in particular computing nodes ($\mathrm{Ind}_{\mathrm{avail}}(n)$ parameter), supplied periodically by *TSyncs*.

The *NewBest* predicate in *GSync* maintains the best known solution *min_dist* based on the local best solutions found so far. This value is determined based on the *min_dist* state messages sent from the *TSync* synchronizers in worker processes. The globally best solution is distributed back to all *TSyncs* by sending to them the appropriate control signal with the new best solution sent in the MPI message. *TSyncs* distribute the signals to all the threads they cooperate with, together with the placement of the new best solution in the memory shared by *TSyncs* and their threads, available on the processor node. The *StopIteration* predicate in *GSync* takes care of the quality of the TSP solution found so far. The TSP solution search can be stopped if a sufficiently good solution has been found in worker processes even if not entire search tree has been scanned. In this case, the *StopIteration* predicate of *GSync* distributes the *StopIteration* signal to all *TSyncs*.

**4.3. Load balancing implementation at the level of threads.** The TSP program execution control inside a worker thread is shown in Fig. 4.1. When a thread is activated and all its programs are initiated the thread sends a "thread started" report to the *TSync* synchronizer. Then the tread waits for the first subtask pool to be received from TSync. When the new subtask pool arrives, the thread starts working on a subtask from it. A subtask consists of a vector of towns with a length smaller than the total number of towns which forms a trajectory with a known length. The thread gradually develops the trajectory by adding new towns and computing the length of each formed trajectory. The length of each new trajectory is compared to the current *min_dist* known to the thread. If the length of a new partial trajectory is bigger than or equal to *min_dist*, the trajectory is considered non perspective and it is not further developed (the B&B bounding). A new full

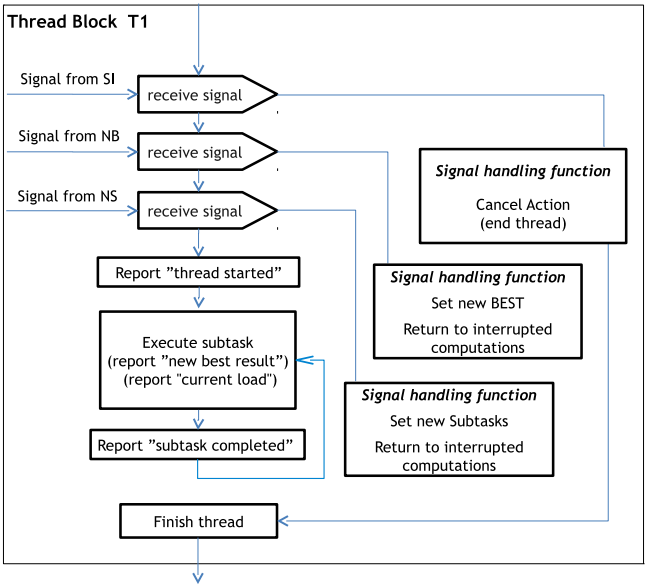



Fig. 4.1: TSP worker thread internal control.

trajectory with the length not smaller than *min_dist* is rejected. Only partial or full trajectories which have the lengths smaller than *min_dist* are maintained and registered. A full trajectory shorter than *min_dist* is reported to the *TSync* synchronizer as a "new best result". Its length is set as a new *min_dist* for a thread. Then the search is continued for the subtask by backtracking in its search tree to the highest partial trajectory which can be further developed. If the entire solution tree of the current subtask for a thread has been inspected, the thread sends a state message "subtask completed" to *TSync* and starts executing a next subtask from its pool. When a new subtask arrives to a thread, the number of search steps to be done $S = (N - t)!$ is computed as the current load of the subtask for the thread, where $N$ is the total number of towns and $t$ is the number of towns in the trajectory given in the subtask specification. During a subtask execution $S$ is decreased by 1 for each step which creates a partial perspective trajectory or a full trajectory. $S$ is decreased by $p!$ after each step which creates a partial non perspective trajectory where $p$ is the difference between $N$ and the number of towns in the created trajectory. The sums of $S$ values for all subtasks present in a thread are periodically sent by threads to *TSyncs* as "current load" reports. On detection of each load SCGS a *TSync* computes the current load *PS* of its worker process as a sum of load values concurrently sent by all threads it controls. *TSyncs* periodically send *PS* to *GSync* which, on detection of respective SCGS computes the mean load of the system and the load deviations ldev($j$) for particular processes. Based on the load deviations, load balancing decisions are taken followed by control signals sent to *TSyncs*.

In Fig. 4.1 we can see three control signals which can come to the thread from the *TSync* synchronizer. The first signal (Stop Iteration) is generated by *SI* predicate of *TSync*. It corresponds to the situation when the search activity in a thread is to be canceled. Next signal (New Best) is sent by the *NB* predicate of *TSync*. It informs the thread that the new best solution has been found and makes it available to the thread code. The last signal (New Subtask) comes from the *NS* predicate of the *TSync*. It brings new subtasks to be executed by the thread. The arrival of each of these signals breaks the basic search loop of the thread and activates a special signal handling function. For signal from *NS* and *NB* the handling function sets a new subtask or the new best solution in the thread. Then, the execution control returns to the interrupted activity. In the case of the signal from *NS* which comes while the thread is idle, the control is passed to the execution of a new subtask. The signals from *SI* cancel the interrupted basic search activity and the execution of the thread is finished.

**5. Conclusions.** Monitoring of global application states creates an efficient and flexible basis for distributed program execution control including the load balancing support. Global control constructs for the control flow design and the design of asynchronous control based on global application states monitoring with control signal dispatching provide flexible infrastructure for distributed applications implementation and system-

level optimizations. This was the motivation for the research on a new distributed program design framework PEGASUS which has been used in this paper as a basic program design and execution infrastructure. In the PEGASUS framework the semantics of control constructs at the process and thread levels takes into account global application states involved in synchronous and asynchronous program execution control. The proposed methods include new program execution paradigms and the corresponding software architectural solutions. The infrastructure enables an easy and convenient design of the load balancing logic in applications.

Dynamic load balancing based on distributed application global states monitoring in the PEGASUS environment has been illustrated by an example of a distributed program for solving the Traveling Salesman Problem by a Branch and Bound method. For this problem, we have designed a load balancing method in which the troublesome load migration between distributed threads mapped to different processors (by transfers of tasks between distributed threads) can be avoided in the initial program execution stages and replaced by controlling the initial computational tasks distribution by synchronizers. Only when all task pools have been distributed, the real task migration is used when it is unavoidable at this stage.

The described PEGASUS framework is in final implementation stage based on a cluster of contemporary multicore processors. The processors in the cluster are interconnected by three dedicated different networks used for control and computational data communication. Interprocess control communication including Unix signal propagation between processors is organized by the use of message passing over Infiniband network. Computational data communication and data exchange for processor clock synchronizatin purposes is performed by dedicated Ethernet networks. C/C++ language with the MPI2, OpenMP and Pthreads libraries are used for writing application programs and the framework control code.

This paper has been partially sponsored by the MNiSW research grant No. NN 516 367 536.

## REFERENCES

[1] Baude et al., Programming, Composing, Deploying for the Grid, In GRID COMPUTING: Software Environments and Tools, J. C. Cunha, O. F. Rana (Eds), Springer, January 2006.

[2] L.V. Kalé, S. Krishnan, CHARM++: A Portable Concurrent Object Oriented System Based on C++, Proc. of OOPSLA'93, ACM Press, Sept. 1993, pp. 91–108.

[3] `http://www.scalemp.com/architecture`

[4] Y. Etsion, A. Ramirez, R. Badia, E. Ayguade, J. Labarta, M. Valero, Task Superscalar: Using Processors as Functional Units, Usenix Workshop on Hot Topics In Parallelism (HotPar), January 2010.

[5] C. Villavieja, Y. Etsion, A. Ramirez, N. Navarro, FELI: HW/SW Support for On-Chip Distributed Shared Memory in Multicores, Euro-Par 2011, LNCS 6852, Part I, pp. 280-292, Springer 2011.

[6] N. Carriero, D. Gelernter, *Linda in Context*, in Communications of the ACM 32 (4), April 1989, pp. 444–459

[7] `http://www.cs.ucy.ac.cy/courses/EPL441/manifold/tut.pdf`, `http://reo.project.cwi.nl/`

[8] K. Marzullo, D, Wood, *Tools for constructing distributed reactive systems*, Technical Report 14853, Cornell University, Department of Computer Science, Feb. 1991.

[9] M. Tudruj, *Fine–Grained Global Control Constructs for Parallel Programming Environments*, in Parallel Programming and Java: WoTUG–20, IOS, 1997, pp. 229–243.

[10] M. Tudruj, J. Borkowski, D. Kopański, *Graphical Design of Parallel Programs with Control Based on Global Application States Using an Extended P–GRADE System*, in Distributed and Parallel Systems, Cluster and GRID Comp., Kluver, Vol. 777, 2004.

[11] M. Tudruj, J. Borkowski, Ł. Maśko, A. Smyk, D. Kopański, E. Laskowski, Program Design Environment for Multicore Processor Systems with Program Execution Controlled by Global States Monitoring. 10th International Symposium on Parallel and Distributed Computing, Cluj-Napoca, July, 2011, ISPDC 2011, IEEE CS, pp. 102-109.

[12] R. Olejnik, I. Alshabani, B. Toursel, E. Laskowski, M. Tudruj, Load Balancing Metrics for the SOAJA Framework, Scalable Computing: Practice and Experience, 2009, Vol. 10, No. 4.

[13] K. Barker, N. Chrisochoides, An Evaluation of a Framework for the Dynamic Load Balancing of Highly Adaptive and Irregular Parallel Applications, Supercomputing 2003, Phoenix, ACM, 2003.

[14] J. Borkowski, M. Tudruj, D. Kopański, Global predicate monitoring applied for control of parallel irregular computations, 15th Euromicro Conference on Parallel, Distributed and Network-Based Processing PDP'07, Naples, Italy, IEEE CS, 2007, pp. 105-111.

[15] J. Borkowski, M. Tudruj, Tuning the Efficiency of Parallel Adaptive Integration with Synchronizers, 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008), 2008, pp. 351-357.

[16] O. Babaoglu, K. Marzullo, Consistent global states of distributed systems: fundamental concepts and mechanisms, Distributed Systems, Addison-Wesley, 1995.

[17] S. D. Stoller, Detecting Global Predicates in Distributed Systems with Clocks, Distributed Computing, Vol. 13, N. 2, 2000, pp. 85-98.

[18] J. Borkowski, M. Tudruj, Dynamic Distributed Programs Control Based on Global Program States Monitoring, Scalable Computing: Practice and Experience, Journal, Vol. 13, N. 2, June 2012, pp. 173–186

[19] H. El-Rewini, T. G. Lewis, H. H. Ali, Task Scheduling in Parallel and Distributed Systems, Prentice Hall 1994.

[20] G. Karypis, V. Kumar, Multilevel graph partitioning schemes, Proc. 24th Intern. Conf. Par. Proc., III. CRC Press, 1995, pp. 113–122.
[21] A. Bhatele, S. Fourestier, H. Menon, L. V. Kale, F. Pellegrini, Applying graph partitioning methods in measurement-based dynamic load balancing, PPL Technical Report 2012, University of Illinois, Dept. of Computer Science, URL: papers/201107_ScotchCharm.
[22] K.D. Devine, E.G. Boman, L.A. Riesen, U.V. Catalyurek, C. Chevalier, Getting Started with Zoltan: A Short Tutorial, Sandia National Labs Tech Report SAND2009-0578C, 2009.
[23] G. Paroux, B. Toursel, R. Olejnik, V. Felea, A Java CPU calibration tool for load balancing in distributed applications, ISPDC/HeteroPar'04, IEEE CS 2004, pp. 155–159.
[24] J.A. Hartigan, M.A. Wong, A K-Means clustering algorithm, Applied statistics, Vol. 28, pp. 100-108, 1979.

*Edited by:* José Luis Vásquez-Poletti, Dana Petcu and Francesco Lelli
*Received:* Sep 20, 2012
*Accepted:* Oct. 20, 2012