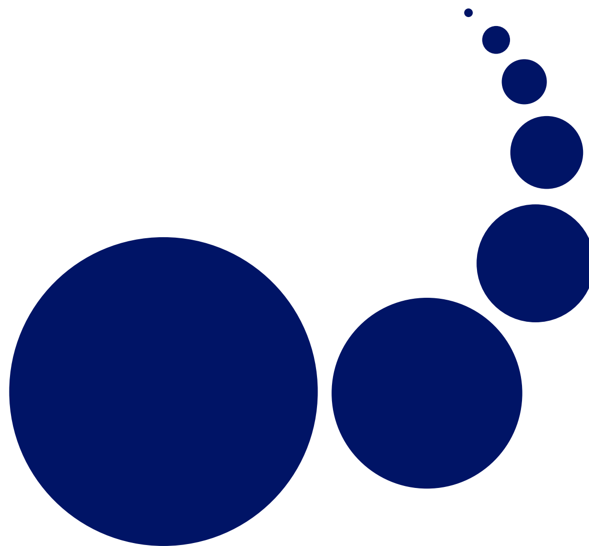
SCALABLE COMPUTING
Practice and Experience

Special Issue: Parallel and Distributed Computing Techniques, Selection of papers from ISPDC 2008

Editor: Marek Tudruj



Volume 10, Number 2, June 2009

ISSN 1895-1767

Editor-in-Chief

Dana Petcu
Computer Science Department
Western University of Timisoara
and Institute e-Austria Timisoara
B-dul Vasile Parvan 4,
2300223 Timisoara, Romania
petcu@info.uvt.ro

Managinig and
TEXnical Editor

Alexander Denisjuk
Elbląg University of Humanities
and Economy
ul. Lotnicza 2
82-300 Elbląg, Poland
denisjuk@euh-e.edu.pl

Book Review Editor

Shahram Rahimi
Department of Computer Science
Southern Illinois University
Mailcode 4511, Carbondale
Illinois 62901-4511
rahimi@cs.siu.edu

Software Review Editor

Hong Shen
School of Computer Science
The University of Adelaide
Adelaide, SA 5005
Australia
hong@cs.adelaide.edu.au

Domenico Talia
DEIS
University of Calabria
Via P. Bucci 41c
87036 Rende, Italy
talia@deis.unical.it

Editorial Board

Peter Arbenz, Swiss Federal Institute of Technology, Zürich, arbenz@inf.ethz.ch
Dorothy Bollman, University of Puerto Rico, bollman@cs.uprm.edu
Luigi Brugnano, Università di Firenze, brugnano@math.unifi.it
Bogdan Czejdo, Fayetteville State University, bczejdo@uncfsu.edu
Frederic Desprez, LIP ENS Lyon, frederic.desprez@inria.fr
David Du, University of Minnesota, du@cs.umn.edu
Yakov Fet, Novosibirsk Computing Center, fet@ssd.sscc.ru
Ian Gladwell, Southern Methodist University, gladwell@seas.smu.edu
Andrzej Goscinski, Deakin University, ang@deakin.edu.au
Emilio Hernádez, Universidad Simón Bolívar, emilio@usb.ve
Jan van Katwijk, Technical University Delft, j.vankatwijk@its.tudelft.nl
Vadim Kotov, Carnegie Mellon University, vkotov@cs.cmu.edu
Janusz S. Kowalik, Gdańsk University, j.kowalik@comcast.net
Thomas Ludwig, Ruprecht-Karls-Universität Heidelberg, t.ludwig@computer.org
Svetozar D. Margenov, IPP BAS, Sofia, margenov@parallel.bas.bg
Marcin Paprzycki, Systems Research Institute, Polish Academy of Science, marcin.paprzycki@ibspan.waw.pl
Lalit Patnaik, Indian Institute of Science, lalit@micro.iisc.ernet.in
Siang Wun Song, University of São Paulo, song@ime.usp.br
Boleslaw Karl Szymanski, Rensselaer Polytechnic Institute, szymansk@cs.rpi.edu
Roman Trobec, Jozef Stefan Institute, roman.trobec@ijs.si
Carl Tropper, McGill University, carl@cs.mcgill.ca
Pavel Tvrdík, Czech Technical University, tvrdik@sun.felk.cvut.cz
Marian Vajtersic, University of Salzburg, marian@cosy.sbg.ac.at
Lonnie R. Welch, Ohio University, welch@ohio.edu
Janusz Zalewski, Florida Gulf Coast University, zalewski@fgcu.edu

SUBSCRIPTION INFORMATION: please visit http://www.scpe.org

Scalable Computing: Practice and Experience

Volume 10, Number 2, June 2009

TABLE OF CONTENTS

Special Issue Papers:

Introduction to the Special Issue i
Marek Tudruj

The Impact of Workload Variability on Load Balancing Algorithms 131
Marta Beltrán and Antonio Guzmán

Model-Driven Engineering and Formal Validation of High-Performance Embedded Systems 147
Abdoulaye Gamatié, Éric Rutten, Huafeng Yu, Pierre Boulet, and Jean-Luc Dekeyser

Relations Between Several Parallel Computational Models 163
Stefan D. Bruda and Yuanqiao Zhang

Experiences with Mesh-like computations using Prediction Binary Trees 173
Gennaro Cordasco, Biagio Cosenza, Rosario De Chiara, Ugo Erra, and Vittorio Scarano

The influence of the IBM pSeries servers virtualization mechanism on dynamic resource allocation in AIX 5L 189
Maciej Mlynski

HeteroPBLAS: A Set of Parallel Basic Linear Algebra Subprograms Optimized for Heterogeneous Computational Clusters 201
Ravi Reddy, Alexey Lastovetsky and Pedro Alonso

Research Paper:

Power-aware Speed-up for Multithreaded Numerical Linear Algebraic Solvers on Chip Multicore Processors 217
Jayanta Mukherjee and Soumyendu Raha

© SCPE, Timişoara 2009

Scalable Computing: Practice and Experience
Volume 10, Number 2, pp. i–ii. http://www.scpe.org

ISSN 1895-1767
© 2009 SCPE

INTRODUCTION TO THE SPECIAL ISSUE: PARALLEL AND DISTRIBUTED COMPUTING TECHNIQUES, SELECTION OF PAPERS FROM ISPDC 2008

Dear SCPE Reader,

We present a selection of papers which are extensions of papers presented at the 7-th International Symposium on Parallel and Distributed Computing, 1–5 July 2008, in Krakow, Poland. The motivation for publishing the selection in the SCPE Journal was, on the one hand, to present the flavour of the research reported at the conference and on the other hand to present some of the most relevant topics currently focused on the research on parallel and distributed computing in general. The selection contains only 6 papers out of about 60 presented at the conference, and thus, is far from covering all relevant topics represented at the ISPDC 2008. This is because not all of the invited authors were patient enough to accept a fairly long paper publishing process. Nevertheless, we hope that the presented papers will bring you closer to the research covered by the ISPDC conferences and will encourage you to participate in future ISPDC editions.

The first paper "The Impact of Workload Variability on Load Balancing Algorithms" is by Marta Beltrán and Antonio Guzmán from King Juan Carlos University in Spain. It concerns an important topic of load balancing in cluster systems, namely adaptativity of the load balancing algorithms to changes of the workload in the system. Adequate accounting for additional load in the hosting system is of great relevance for correct optimization effects. The paper presents a thorough formal analysis of the workload variability metrics and their influence on the quality of load balancing algorithms. Four basic activities appearing in load balancing algorithms are identified, and based on them some algorithmic solutions are proposed to correctly deal with workload variability in system load balancing. The problem of dynamic load balancing algorithms robustness has been discussed. Two different robustness metrics sensitive to the applied type of opimization: local task-oriented or a global one enable selecting task remote execution or migration as load balancing operations. The proposed approach is illustrated with experiments.

The second paper "Model-Driven Engineering and Formal Validation of High-Performance Embedded Systems" is by Abdoulaye Gamatié, Éric Rutten, Huafeng Yu, Pierre Boulet, Jean-Luc Dekeyser, from University of Lille and INRIA in France. The paper is concerned with a very advanced methodology of designing correct parallel embedded systems for intensive data-parallel computing. In their previous research, the authors of the paper designed the GASPARD embedded system design framework. It is based on the hardware/software co-design approach through model-driven engineering. The framework is based on an UML-like model specification language in which hardware and software elements are modelled using a component approach with special mechanisms for repetitive structures. This paper tries to combine the modelling framework of GASPARD with the mechanisms of synchronous languages to achieve design verifiability provided for such languages. The paper shows how GASPARD models can be translated into synchronous models based on data flow equations in order to formally check their correctness. The proposed approach is illustrated with an example of a video processing system.

The third paper "Relations Between Several Parallel Computational Models" is by Stephan Bruda and Yuanqiao Zhang from Bishop's University in Canada. The paper is concerned with theoretical aspects of shared memory systems described by the parallel random access machine PRAM model and aims in studying performance properties of different types of PRAM systems. The attention is focussed on analysing the computational power of two more sophisticated PRAM models (Combining CRCW and Broadcast Selective Reduction), which include data reduction in case of concurrent writes. The paper shows that these two models have equivalent computational power, which is a new result comparing the existing literature. The performance of both models applied to reconfigurable multiple bus machines was studied as a possible architectural solution for current VLSI processor implementations. It was shown that in such systems under reasonable assumptions concurrent-write does not enhance performance comparing the exclusive-write model. Another result important for the VLSI technology is that the Combining CRCW PRAM model (in which data of concurrent writes are arthmetically or logically combined before write) and the exclusive-write on directed reconfigurable busses perform in equivalent way under strong real-time requirements.

The fourth paper "Experiences with Mesh-Like Computations Using Prediction Binary Trees" is by Gennaro Cordasco, Biagio Cosenza, Rosario de Chiara, Ugo Erra and Vittorio Scarano from the University "degli Studi" of Salerno and the University "degli Studi della Biasilicata" of Potenza in Italy. The paper concerns optimization methods for mesh-like computations in clusters of processors. The computations are perfomed assuming a phase-like program execution control using a tiling approach which reduces inter-processor communication.

A temporal coherence is also assumed, which means that task sizes provide similar execution times in consecutive phases. Temporary coherent computations are structured in a Prediction Binary Tree, in which leaves represent computing tiles to be mapped to processors. A phase-by-phase semi-static load balancing is introduced to the scheduling algorithm. The scheduling algorithm is equipped with a predictor, which estimates the computation time of next phase tiles based on previous execution times and modifies the tiles to achieve balanced execution in phases. For this, two heuristics are used to leverage on data locality in processors. The proposed approach is illustrated by the example of interactive rendering with Parallel Ray Tracing algorithm.

The fifth paper "The Influence of the IBM pSeries Servers Virtualization Mechanism on Dynamic Resource Allocation in AIX 5L" is by Maciej Mlynski from ASpartner Limited in Poland. The paper concerns a very up-to-date problem of system virtualization and presents the results of research carried on IBM pSeries servers. IBM is strongly developing the virtualization technique especially on IBM pSeries servers enabling an improved and flexible sharing of system resources between applications. The paper investigates novel facilities for dynamic resource management such as micro-partitioning and partition load manager. They enable dynamic creation of workload logical partitions of system resources and their dynamic mangement. It includes run-time resource re-alocation between logical partitions including setting of sharing specifications as well as run-time adding/removing/setting parameters of resources in the system. It remains an open question how to properly tune parameters of the operating system using the provided virtualization facilities to obtain the best efficiency for a given application program. The paper presents the results of experiments which study the effects of tuning the disk subsystem parameters under the IBM AIX 5L operating system with the use of the provided virtualization facilities on the resulting application execution performance. The results show that even small deterioration in the resource pool status requires an immediate adaptation of the operating system parameters to maintain the required performance.

The sixth paper "HeteroPBLAS: A Set of Parallel Basic Linear Algebra Subprograms Optimized for Heterogeneous Computational Clusters" is by Ravi Reddy, Alexey Lastovetsky and Pedro Alonso from University College Dublin in Ireland and Polytechnic University of Valencia in Spain. The paper concerns the methodology for parallelization of linear algebra computations for execution in heterogeneous cluster environments. The design of the HeteroPBLAS library (Parallel Basic Linear Algebra Subprograms) for heterogeneous computational clusters is presented. The main contribution of the paper is the automation of the parallelization and optimization of the PBLAS, which is done by means of a special user interface and the underlying set of functions. An important element is here a performance model that is based on program code instrumentation, which determines parameters of the application and the executive heterogeneous platform relevant for execution performance of parallel code. The parameter values specified for or returned by execution of the performance model functions are next used for generation and optimal mapping of the parallel code of the library subroutines. The proposed approach is illustrated by experimental results of execution of optimized HeteroPBLAS programs on homogeneous and heterogeneous computing clusters.

Marek Tudruj

Scalable Computing: Practice and Experience
Volume 10, Number 2, pp. 131–146. http://www.scpe.org

ISSN 1895-1767
© 2009 SCPE

THE IMPACT OF WORKLOAD VARIABILITY ON LOAD BALANCING ALGORITHMS*

MARTA BELTRÁN† AND ANTONIO GUZMÁN†

Abstract. The workload on a cluster system can be highly variable, increasing the difficulty of balancing the load across its nodes. The general rule is that high variability leads to wrong load balancing decisions taken with out-of-date information and difficult to correct in real-time during applications execution.

In this work the workload variability is studied from the perspective of the load balancing performance, focusing on the design of algorithms capable of dealing with this variability. In this paper an exhaustive analysis is presented to help users and administrators to understand if their load balancing mechanisms are sensitive to variability and to what degree. Furthermore, solutions to deal with variability in load balancing algorithms are proposed and their utilization is exemplified on a real load balancing algorithm.

Key words: cluster computing, load balancing, workload variability

1. Introduction. Cluster systems are rapidly evolving from laboratories and research centers to business environments due to their potential to take advantage of old and/or idle computers ([9]). In this kind of environment cluster systems are even more dynamic and variable than in the typical controlled research environments.

Many factors contribute to the system state variability ([5]): asynchronous scheduling and communication behavior arising from the autonomy of the individual system nodes (clusters are loosely-coupled systems), performance heterogeneity (processing nodes are rarely homogeneous, differing in many ways), workload variability (the presence of background tasks constitutes an important source of variability), variation in the number of system nodes, independent failure and recovery of nodes and variable communication latencies (network variability due to transmission errors or congestion).

Load balancing is essential to take advantage of all the available resources on cluster systems and therefore, to obtain their optimal performance ([9, 13]), especially on heterogeneous clusters. And the ability to deal with variability is essential for designing efficient and scalable load balancing algorithms. All the decisions related to load balancing are taken considering some kind of system state information. If this state suffers smooth variations, it is easy to maintain updated information about it. But when the system state is highly variable, the information may be out of date by the time the load balancing decisions are taken. Therefore, the probability of taking wrong decisions increases critically ([17]).

As far as we know the impact of workload variability on load balancing performance has not been investigated in detail. Some works have discussed similar problems in other research areas as networking ([1, 10, 11]), multimedia applications ([14]) or performance prediction ([12]), studying the variability of TCP round-trip times or another network aspects, execution times or processes arrivals respectively.

The main contribution of this paper is an exhaustive analysis of the effects of variability on load balancing algorithms performance. Furthermore, solutions for the main problems caused by these effects, related to the state information updating and to the load balancing operations, are presented. The proposed techniques can be used in any variable environment, typically a non-dedicated cluster, to obtain the desired performance with a load balancing algorithm. Two different robustness metrics have been proposed to select the best distribution rule for the algorithm, remote execution or process migration. With the proposed solution only when a high variability causes an intolerable performance degradation the process migration is used to balance the load in the system.

The rest of this paper is organized as follows. Section 2 studies the effects of workload variability on load balancing performance. To do this, load balancing algorithms are divided into four policies or rules. Section 3 proposes solutions to deal with variability in the information rule and discusses three different metrics to quantify the workload variability. Section 4 proposes solutions to deal with variability in the distribution rule and defines the robustness metric. Section 5 presents some experimental results to verify the performed analysis and to select a best variability metric. And finally, Section 6 summarizes conclusions and future work.

*Questions, comments, or corrections to this document may be directed to Marta Beltrán.

†Computer Architecture, Artificial Intelligence and Computer Science Department, Rey Juan Carlos University, Madrid, Spain ({marta.beltran, antonio.guzman}@urjc.es).

2. Effects of workload variability on load balancing. To study the effects of variability on load balancing performance, the structure of a load balancing algorithm has been decomposed in this research with the four rules or stages proposed in [19]. Another structures for the load balancing algorithm could be used to make easier the analysis of the effects of variability, but the results would be the same because it is only an structural issue and finally all the algorithms perform similar stages to balance the system load. These are the four considered rules ([8]):

- **Load Measurement:** Load balancing decisions always rely on some kind of workload or availability information of system nodes. This information is typically quantified by a load index in this first algorithm stage.
- **Information Rule:** The information policy or rule specifies how to exchange the state information between the system nodes. A good information rule should achieve a balance between incurring a low communication overhead for the state information exchange and maintaining an accurate view of the system state.
- **Initiation Rule:** This load balancing stage determines when to initiate a load balancing operation. These operations introduce a non-negligible overhead in the system; the initiation rule must weigh their cost against their expected performance benefit.
- **Load Balancing Operation:** This last rule is decomposed in three more rules: the location, distribution and selection rules.
  The location rule determines the best candidate to participate in the load balancing operation. The distribution rule specifies how to balance the load among the nodes involved in the load balancing operation. This balance can be performed by remote execution (non-preemptive allocation or initial placement of tasks) or by process migration. And finally, the selection rule chooses the load that is going to be distributed or re-distributed with this operation.

Analyzing the effects of high workload variability on these four stages, some important conclusions can be drawn.

The effects of workload variability on the first algorithm stage depend on the selected load index. If it is a good load index, it should be able to show the workload variations summarizing the node state and maintaining its stability. Therefore, this stage performance depends entirely on the selected load index, and with the proper decision, the first algorithm stage will be able to deal with variability.

On the other hand, the impact of variability on the information policy is very important. If there is a low variability this policy has an easy job because it is not difficult to maintain updated system state information in all the cluster nodes. But if the workload variability is high, the job of this policy is more difficult because it has to maintain updated state information in all the cluster nodes without causing a great network overhead with the messages generated to do it.

For the initiation policy, all the effects of variability are related to the information rule. If this rule is capable of dealing with variability and of maintaining updated state information in all the system nodes, the initiation decisions are correct and the variability does not affect this algorithm stage.

Finally, the load balancing operation performance can be affected by the workload variability in the location and distribution rules. The location rule selects the best candidate to participate in the load balancing operation. Supposing that the system state information available to decide about this aspect is updated, there is still one problem: in a highly variable environment the system state may change during this rule. The best candidate for the load balancing operation in a certain moment may be the worst immediately after.

As it has been said before, the distribution rule can be based on remote execution or on process migration. Many authors have discussed about the benefits and drawbacks of process migration for the algorithm distribution rule ([16]). Some authors have concluded that migration cannot provide better performance than other alternatives like remote execution due to its costs, while others have argued that it can significantly improve systems performance, especially when the systems are highly variable, even when its costs are high ([16]). If a task is allocated to a certain system node, and after this, the state of this node suffers a great variation, process migration allows solving this problem reallocating this task. And if the initiation rule takes a wrong decision because the state information is obsolete, the migration is again the mechanism which allows correcting this situation. The performance of algorithms based on remote execution can be affected by variability because there are not mechanisms to migrate processes once they have been assigned to one system node, even if the load balancing decisions were based on wrong state information or if the system state has changed since this assignment.

Considering this discussion, there are three main challenges that need to be faced: the load index selection, the information policy design and the load balancing operation execution. The load index selection is not included in this paper because it is a whole important research area in load balancing, but next sections try to propose solutions for the two other challenges.

3. Information rule capable of dealing with variability. Three different policies have been traditionally proposed for the information rule in load balancing algorithms:

- **Periodic (or batch) policy:** Each system node periodically broadcasts its state information through the system, regardless of whether this information is needed or not. In this kind of policy a critical question is to fix the interval between successive information exchanges, the information period (T_{info}).
- **On-demand policy:** Each system node sends a request message to the rest of the system nodes when it needs the information to make a load balancing decision.
- **Event-driven policy:** The information exchange is asynchronous, taking place when some specific conditions are met (events). These events are typically related to node state changes in a certain degree and in this case, the critical question is to decide what kind of event determine when the information exchange must be performed.

Intuitively, it can be seen that periodic approaches for the information rule provide a simple solution for information exchange, but one important problem is how to fix the interval for this exchange in highly variable environments. A long interval leads to work with obsolete information and therefore, to take load balancing decisions based on old knowledge about nodes state. But a short interval, necessary to manage updated information, implies a heavy communication overhead. The on-demand approach minimizes the number of generated messages, information is sent only when a system node requests it, introducing an operation delay. When some node needs information to evaluate a load balancing decision, it has to wait for the rest of system nodes to send it. This delays should be avoided on cluster systems, specially if the system state is highly variable and the system has a large number of nodes, because the time taken to gather all the state information is enough to make this information old. The event-driven policy might obtain a compromise between the two previous solutions, but it depends on the selected events, i. e., on the conditions which must be met to inform about the nodes state.

The event-driven policy seems to be the best alternative to obtain an information policy capable of dealing with workload variability. With this kind of policy the information exchange is asynchronous, taking place when some specific conditions are met, the events. The critical question is to decide what kind of event determine the information exchange. The proposed solution in this work is based on considering the system workload variability. If the workload of a system node is continuously changing (large variability values), this node must inform very often about its state, but if its workload hardly never changes (low variability values), this information exchange is unnecessary.

Let V denote the workload variability (independently of the metric selected to quantify it, discussed in the next section) and $V_{\max}$ denote its maximum value. The information events are defined in the following way:

$$\frac{V}{V_{\max}} > \chi. \tag{3.1}$$

With this event definition a system node informs the rest of the system nodes about its state when its workload variability is above a certain fraction of its maximum value. The normalization allows to consider system heterogeneity, if one node workload changes a fraction of 0.5 of its maximum value, the meaning is the same for all the system nodes with independence of this maximum value. Events are defined with the 'relative variability' not with absolute values which cannot be compared in different system nodes with different features. The cluster user (or administrator) only has to set the threshold value, χ, used to determine how updated must be the global state information in the system nodes.

3.1. Variability quantification. In this section, different ways of quantifying the system workload variability are proposed to define the information rule events. In this context, the variability metric must quantify how much the workload data differ from individual to individual on a system node. Each node maintains a vector (X) with its last K load index values. The load index quantifies the workload (or the availability, inversely related to this load) of the node at a given instant. Load balancing algorithms use to measure this load index periodically on each node to maintain updated information about system state, therefore the vector X keeps

the index values measured in the last K monitoring intervals. Taking into consideration these values, there are three main alternatives to measure the workload variability ([4]):

- **The range (R):** This metric is obtained with the difference between the largest measurement in a set (the maximum) and the smallest (the minimum).

$$R = \max(X) - \min(X). \tag{3.2}$$

- **The inter-quartile range (IQR):** Another kind of range not sensitive to the extreme values or to the sample size is the inter-quartile range. This metric is the difference between the first and third quartiles of the sample ($Q1$ and $Q3$ respectively).

$$IQR = Q3(X) - Q1(X). \tag{3.3}$$

  Note that $Q1$ is the $25th$ percentile, therefore, the value below which a 25% of observations fall. And $Q3$ is the $75th$ percentile, the value below which a 75% of observations fall.
- **The standard deviation (σ):** This is a well-known variability metric and it is a kind of average of the distances of the observed values from their mean ($\overline{X}$). If many of the observations are far above or below the mean, the deviation value is large.

$$\sigma = \sqrt{\frac{\sum_{i=1}^{K}(X_i - \overline{X})^2}{K-1}}. \tag{3.4}$$

It is important to be able to interpret these two lasts metrics meaning, a little more complex than the range. And to understand that their values cannot be compared due to the differences in their meanings. With the IQR definition the variability can be interpreted as the dispersion of the values in the center of the sample, not taking into account the smallest 25% (smaller than the first quartile, $Q1$) and the largest 25% (larger than the third quartile $Q3$): the IQR is expected to include about half of the data. On the other hand, the standard deviation is a measure of how spread out or close together the sample values are, quantifying the average distance from their mean value.

4. Distribution rule capable of dealing with variability. First, to avoid the negative effects of workload variability on the location rule, negotiated operations should be always used.

Once a system node performs the initiation rule and decides to initiate a load balancing operation, the location rule selects the best system node to be involved in the load balancing operation. Before finishing this rule, the node initiating the operation always asks the the system node selected to perform the load balancing operation to accept this operation. This node can accept or reject the load balancing operation depending on its own state information, always more updated than its state information in the rest of nodes. Therefore, if the selected node has changed its state during the initiation and location rules due to a high variability, and its new workload is higher, it can reject the operation and the location rule has to be performed again.

For the distribution rule, the desirable solution should obtain the costs of remote execution, but having some mechanism to redistribute the allocated tasks in highly variable environments. Thus, the load balancing algorithm should rely on a non-preemptive allocation for the load distribution. This kind of policy is easier to implement and implies a lower cost to the system than a preemptive one ([16]). But, is it enough to maintain the desired system performance in all the possible environments, even in highly variable situations? The answer to this question is not.

In this work a remote execution rule is proposed as the general distribution mechanism, but using a robustness metric to decide if an exceptional migration is needed due to a high variability. The robustness metric is proposed to quantify the degradation of the load balancing algorithm performance when large system state variations take place. This degradation quantification allows the user to decide if process migration is needed to obtain the desired performance with the load balancing algorithm when large state changes occur. These large variations caused by the workload variability will be called perturbations from this point.

A tasks assignment is defined to be robust with respect to a system performance feature against some specific perturbations if the degradation in this feature is limited when the perturbations occurs ([3]). But, what is the degree of this assignment robustness in a dynamic load balancing algorithm? A system user or administrator may need this information to tune properly the algorithm parameters and to be sure of not

needing process migration to correct in real time the tasks allocation to keep the desired system performance when perturbations occur.

Siegel proposed the FePIA procedure ([2, 3]) to quantify a system-task allocation combination robustness. This method, very frequently used for analyzing systems robustness in initial task assignments, constitutes the starting point which will be extended in this paper to quantify the robustness of a dynamic load balancing algorithm. Therefore, when the process allocation performed by the algorithm is robust enough against perturbations, process migration is not needed. On the other hand, if these perturbations caused by variability lead to an unacceptable performance degradation, the process migration must be used to redistribute the tasks among the system nodes.

With the inclusion of this robustness criteria, the distribution rule of the load balancing algorithm can deal with variability but without incurring in unnecessary overheads and costs in low variable environments.

Two different robustness metrics have been defined: the QoS-robustness and the P-robustness. The first metric should be used when the load balancing algorithm objective is related to the individual tasks composing the global application executed on the cluster and the second should be used with global algorithm objectives related to the total application.

4.1. QoS-robustness. This robustness metric should be used when the load balancing algorithm decisions are taken to achieve an objective related to the tasks composing the application executed on the cluster. The initiation mechanism evaluates the tasks response times or their resources utilization or their balance to decide when to begin a new load balancing operation.

In this case the performance feature is a quality-of-service value (QoS) related to the individual tasks: response time, resources utilization, balance, etc. The system is robust when:

$$A \geq \tau_R \cdot \overline{A}, \tag{4.1}$$

where A denotes the real QoS achievement obtained when the application is executed, $\overline{A}$ denotes its expected achievement and τ_R is the robustness tolerance defined by the system designer or user.

Therefore, the load balancing algorithm tries to obtain a certain QoS value related to the application tasks and the assignment made by the algorithm is robust if it obtains a certain degree of the expected QoS, given by τ_R, despite all the possible perturbations in the system.

The achievement value (A) has to be defined to quantify the degree of the QoS achievement, therefore:

$$A = \frac{W_C}{W} \tag{4.2}$$

where W_C is the number of tasks executed on the system accomplishing with the expected QoS and W is the total number of tasks composing the application.

Following the FePIA procedure, the steps proposed to quantify the proposed robustness are:

1. **Performance Features (Ψ):** The selected performance feature must be related to the QoS achievement:

$$\Psi = \{\psi_j\} = \{w_j | 1 \leq j \leq N\},$$

where w_j is the number of tasks allocated to the j_{th} system node that finally achieve the expected QoS and N is the number of system nodes. There is an equivalent magnitude, $\overline{w_j}$, denoting the total number of tasks allocated to this node that are supposed to achieve the expected QoS if there are not system perturbations to avoid it.

2. **Perturbation Parameters (Π):** The parameters whose values can impact the proposed performance feature must be identified. System perturbations may modify the system state preventing tasks to accomplish with the expected QoS. The perturbation parameter is then:

$$\Pi = L,$$

where L denotes the total number of tasks that are allocated to system nodes to achieve the expected QoS but finally are not able to do it due to perturbations. Therefore, $\overline{L} = 0$, because the system expects not to fail any prediction about the achievement of the QoS requirements.

3. **Impact of the perturbation parameter on the system performance feature:** With the previous definitions, this relation can be summarized with the following expressions:

$$W_C = \sum_{j=1}^{N} w_j, \qquad \overline{W_C} = \sum_{j=1}^{N} \overline{w_j}, \qquad W_C = \overline{W_C} - L.$$

4. **Robustness metric:** It is the smallest variation in the value of the perturbation parameter that would cause the performance feature to violate its acceptable variation, i. e. a violation of the condition expressed in the equation 4.1. And the load balancing algorithm is not robust if this condition is not fulfilled. The degree of the QoS achievement can be related to the performance feature and to the perturbation parameter by:

$$A = \frac{W_C}{W} = \frac{\sum_{j=1}^{N} w_j}{W} = \frac{\overline{W_C} - L}{W},$$

and, therefore, the robustness radius is:

$$\rho_{QoS}(\Psi, \Pi) = \min_{L:A(L) \geq \tau_R \cdot \overline{A} \wedge \exists L+1:A(L+1) < \tau_R \cdot \overline{A}} (L). \tag{4.3}$$

Therefore, the robustness radius is a discrete magnitude: the largest number of tasks executed on the system that can finally fail in achieving the expected QoS without causing an unacceptable performance degradation in terms of the robustness tolerance given by the system designer or user.

This procedure allows to algorithmically evaluate the discrete robustness radius of the system and to use this value to decide about the need of process migration. The robustness metric warns the designer or user when the system-load balancing algorithm combination is not able to obtain the expected performance in terms of expected QoS achievement (A). This warning can be obtained in two different situations:

- Before executing the application if there is a priori information about it and about the system.
- After executing the application, once this information has been collected, to improve the performance for later executions.

4.2. P-Robustness. When the load balancing algorithm tries to achieve a global objective for the total application (total response time, global resources utilization, system balance, etc) this second robustness metric based on the global system power should be used. In [15], the following system power definition P is proposed:

$$P = \lambda \cdot f\left(QoS, \overline{QoS}\right), \tag{4.4}$$

where λ is the system throughput (finished tasks per time unit) , QoS is the quality of service metric chosen for the evaluated system and $\overline{QoS}$ is the expected or desired quality of service.

The three general QoS metrics proposed in this work are the application response time (T), the system efficiency (ε) and the system balance (B):

- The function f proposed in [15] for the response time metric can be easily used in this work:

$$f\left(T, \overline{T}\right) = \frac{1}{1 + \frac{T}{\overline{T}}}. \tag{4.5}$$

  With this definition, the system power is high when its throughput is high and when the T value is close to its expected value, $\overline{T}$.
- For the efficiency and balance metrics, a new f function has been proposed in this work. Since a high response time value and a high efficiency or balance values have opposite effects on the power figure, this function for the efficiency and balance is:

$$f\left(\varepsilon, \overline{\varepsilon}\right) = \frac{1}{1 + \frac{\overline{\varepsilon}}{\varepsilon}}, \quad f\left(B, \overline{B}\right) = \frac{1}{1 + \frac{\overline{B}}{B}}. \tag{4.6}$$

Therefore, for this robustness metric the performance feature whose degradation is quantified is the whole system power P. In this case, the system is robust when

$$P \geq \tau_R \cdot \overline{P}, \tag{4.7}$$

where P denotes the real system power (equation 4.4) when the application is executed on the cluster, $\overline{P}$ denotes its expected value and τ_R is the robustness tolerance defined by the system designer or user again.

Therefore, the system is robust if it obtains a certain degree of the expected system power, given by τ_R, despite all the possible perturbations in the system.

The steps to obtain the system robustness in this case are the following :

1. **Performance Features (Ψ):** The selected performance feature is the finishing time of each system node:

$$\Psi = \{\psi_j\} = \{M_j | 1 \leq j \leq N\},$$

where M_j is the time at which the j_{th} system node finishes executing all its allocated tasks, $\overline{M_j}$ is its expected value and N is the number of system nodes. The total application response time will be related to this magnitude and to the starting time of the application (T_o):

$$T = \max(M_j) - T_o.$$

Because the last node finishing its work is the one limiting the total application response time.

2. **Perturbation Parameters (Π):** The parameters whose values can impact the proposed performance feature are the processes response times. Therefore the perturbation parameters are continuous in this case:

$$\Pi = \{\pi_i\} = \{t_i | 1 \leq i \leq W\},$$

where t_i denotes the response time of the i_{th} task on the system and again $\overline{t_i}$ is the expected response time for this task.
But with the previous definition, the following steps in the procedure are very difficult to complete, because the current cluster nodes are almost always multitasking systems. Therefore the perturbation parameter is slightly transformed in:

$$\Pi = \{\pi_i\} = \{F_i | 1 \leq i \leq W\},$$

where F_i denotes the time at which the i_{th} task finishes its execution and $\overline{F_i}$ denotes its expected value.

3. **Impact of the perturbation parameter on the system performance feature:**
This impact can be summarized with this expression:

$$M_j = \max(F(q_j(k))) \text{ with } 1 \leq k \leq |q_j|,$$

where F is a vector composed of all the F_i values (for $1 \leq i \leq W$) and q_j is a vector whose indices denote the tasks allocated to the j_{th} system node. For example, if $q_1 = [1\ 5\ 8]$, the cluster node number 1 has been allocated three tasks, whose finishing times are F(1), F(5) and F(8) and $M_1 = \max(F(1), F(5), F(8))$.

4. **Robustness metric:** Again, it is the smallest variation in the value of the perturbation parameter that would cause the performance feature to violate its acceptable variation, i. e. a violation of the condition expressed in the equation 4.7.

$$P \geq \tau_R \cdot \overline{P}. \tag{4.8}$$

Therefore, given the system power definition:

$$\lambda \geq \tau_R \cdot \overline{P} \cdot f^{-1}(QoS, \overline{QoS}) \geq \tau_R \cdot \overline{P} \cdot \overline{f^{-1}(QoS, \overline{QoS})},$$

then

$$\lambda \geq \tau_R \cdot \overline{\lambda}.$$

From this point and taking into account the traditional throughput definition ($\lambda = W/T$):

$$\frac{W}{T} \geq \tau_R \cdot \frac{W}{\overline{\overline{T}}}, \qquad \frac{1}{T} \geq \tau_R \cdot \frac{1}{\overline{\overline{T}}}, \qquad T < \frac{1}{\tau_R} \cdot \overline{T}.$$

where T is the application response time, and if T_o is its starting time:

$$T = \max(M_j) - T_o, \quad \overline{T} = \max(\overline{M_j}) - T_o = \max(\overline{F_i}) - To,$$

and therefore:

$$\max(M_j) - T_o < \frac{1}{\tau_R} \cdot (\max(\overline{F_i}) - T_o), \quad \max(M_j) < \frac{1}{\tau_R} \cdot (\max(\overline{F_i}) - T_o) + T_o.$$

And if the last node in finishing its work must fulfill this condition, all the system nodes must do it:

$$M_j < \frac{\max(\overline{F_i})}{\tau_R} - \left(\frac{1-\tau_R}{\tau_R}\right) \cdot T_o.$$

The robustness radius of M_j against F is then:

$$r_j(\Psi, \Pi) = \min_{T:M_j = \frac{\max(\overline{F_i})}{\tau_R} - \left(\frac{1-\tau_R}{\tau_R}\right) \cdot T_o} ||F - \overline{F}||_2. \tag{4.9}$$

Hence, for the entire system the robustness radius for the P-robustness will be:

$$\rho_P(\Psi, \Pi) = \min(r_j(\Psi, \Pi)). \tag{4.10}$$

Equation 4.9 is the Euclidean distance between any vector composed of the real task finishing times and the vector composed of the expected finishing times. It can be interpreted as the distance from a point to an hyperplane, therefore, using the point to plane expression, this equation can be reduced to:

$$\rho_P(\Psi, \Pi) = \min \left(\frac{\frac{\max(\overline{F_i})}{\tau_R} - \left(\frac{1-\tau_R}{\tau_R}\right) \cdot T_o - M_j(\overline{F})}{\sqrt{|q_j|}} \right). \tag{4.11}$$

With $M_j(\overline{F}) = \max(\overline{F(q_j(k))})$, and $1 \leq k \leq |q_j|$.

5. Experimental Results. To exemplify the application of the proposed solutions, a load balancing algorithm is implemented in this section. With this algorithm and a simple application, very interesting conclusions can be drawn about the utility of the solutions proposed in the previous sections and about the best metric to quantify the wokload variability. Only some examples of the performed experiments are shown in this section, but all the obtained results with different algorithms, applications and cluster systems are considered for conclusions, to make them general and meaningful in most of the typical contexts.

5.1. Example of load balancing algorithm. The following four rules have been implemented to perform the experiments:

- **Load measurement:** In the example algorithm the following new load index has been defined:

$$I = \frac{C \cdot \alpha}{C_{\max}}. \tag{5.1}$$

It can be seen that the proposed index is based on two main parameters, C and α. The first one is a static parameter, the node computing power (C). To compute the index value a normalization is performed on this magnitude, using the computing power of the fastest system node ($C_{\max}$). Therefore the $C/C_{\max}$ ratio gives some kind of 'relative computing power' of the system node. The computing power is a static feature of the system nodes, it can be quantified in the simplest approach using the *bogomips* value given by the Linux operating systems for each system node, or using more sophisticated and complex metrics such as the inverse of the response time of a certain task or process, etc.
The second parameter is a dynamic one, the CPU assignment (α). The node CPU assignment is a dynamic feature containing its availability information and it is defined as the percentage of CPU time that would be available to a new process ([6, 18]). For example, if a 30% of the CPU is available for a a new task, this task should be able to obtain the 30% of the processor time slices, and therefore $\alpha = 0.3$.

With this definition, the proposed index I is inversely related to the node workload: the higher the I value, the less loaded the node. It is an availability index instead of a load index.
Selecting the correct monitoring interval, this index has demonstrated to follow the system state variations (even in highly variable environments) due to the utilization of the dynamic parameter α, which can be computed or predicted with different models. Furthermore, it is a stable index and a good indicator of processes response times on system nodes.

- **Information rule:** Following the rules given in section 3 to deal with variability , the information rule is event-driven with the events defined in equation 3.1. The variability metric and the number of values considered to calculate it (K) can be configured by the user.
- **Initiation mechanism:** To derive the initiation rule for the considered algorithm, the methodology proposed in [7] has been used. This procedure is based on describing qualitatively the requirements for the load balancing algorithm and on identifying an objective function to quantify the achievement of these requirements.
The proposed initiation mechanism is sender initiated, the initiation rule of an overloaded node decides about load balancing operations trying to achieve the selected objective. For our algorithm, two different objectives have been implemented, one related to the individual tasks composing the executed application and another to the complete application.
The two implemented initiation mechanism are based on boundary functions (using this kind of objective function the overhead caused by optimizations is avoided). The implemented objective functions define an upper or lower bound to a certain system parameter and this condition can be directly transferred to the initiation mechanism in the algorithm.
  - **Task response time objective:**
  In this first version of the algorithm implemented to perform the experiments, the aim is to bound the individual processes response time to finish a certain number of tasks per time unit.
  Therefore this first objective is related to the individual tasks response times. The initiation bound is referred to the response time of a new task in its *home node*, that is, the node to which this task is initially assigned. Let $\hat{t_i}$ be the response time of the i_{th} task in its home node, t_i the response time of the i_{th} task regardless it finally runs, and τ be the algorithm tolerance used to define the threshold of the selected objective. This objective function implies that the initiation rule tries to allocate new tasks to nodes where their response times will not be more than τ times greater than in their home nodes:

$$t_i < \tau \cdot \hat{t_i} \;\; \forall i.$$

  In the home node the objective function is achieved if the CPU assignment for the new task is:

$$\alpha = \frac{1}{\tau}.$$

  This CPU assignment is the minimum necessary to accomplish the response time requirements for the new task on this node, but due to system heterogeneity, this may not be the minimum in another node (for example, on a node with more computing power). The objective function is based on the load indexes values, which take into account the nodes computing powers. The minimum index necessary to achieve the objective is:

$$I_{\min} = \alpha_{\min} \cdot \frac{C_{\text{home}}}{C_{\max}} = \frac{1}{\tau} \cdot \frac{C_{\text{home}}}{C_{\max}},$$

  where C_{home} is the computing power of the home node and $C_{\max}$ is the computing power of the most powerful system node. And the initiation mechanism is:

$$I > \frac{1}{\tau} \cdot \frac{C_{\text{home}}}{C_{\max}}. \tag{5.2}$$

  With this initiation mechanism the load balancing algorithm must always look for a node which fulfill this bound to allocate the new task.

– **Total response time objective:**
This second objective is related to the total application response time. In this case the user or administrator determines a bound to the finishing time of the application (T_F):

$$T_F < \tau.$$

Therefore $M_j < \tau \ \forall j$, and: $M_j = \max(t_i^o + t_i)$.
With t_i^o the arriving time for the i_{th} task to the j_{th} node and t_i is the response time of the i_{th} task. The response time predicted for this task is related to the CPU assignment it receives in the node it is allocated:

$$t_i = \frac{t_i^{\text{unloaded}}}{\alpha}.$$

With t_i^{unloaded} the response time of the i_{th} task on the j_{th} node when its processor is unloaded and α the CPU assignment that this node can offer to this new task.
Therefore, the initiation mechanism for this objective is:

$$t_i^o + \frac{t_i^{\text{unloaded}}}{\alpha} < \tau. \tag{5.3}$$

This initiation mechanism is always evaluated on the task *home node*, so this node must have information to evaluate this bound for the rest of system nodes. Each node must know the unloaded CPU response time of the tasks composing the application (t_i^o) and the computing power of the system nodes (C_j). Another important data is the average time consumed to initiate a load balancing operation ($\overline{t_{LB}}$). With all this information, the initiation mechanism of the load balancing algorithm must look for a node which fulfill this bound to allocate the new task:

$$t_i^o(\text{home}) + \overline{t_{LB}} + \frac{t_i^{\text{unloaded}}(\text{home}) \cdot \frac{C_{\text{home}}}{C_j}}{\alpha_j} < \tau. \tag{5.4}$$

It must be pointed that $t_i^o(\text{home}) + \overline{t_{LB}}$ would be the starting time of the new tasks in the receiver node, considering the average time needed to initiate the load balancing operation. The $\overline{t_{LB}}$ is the same for all the possible receiver nodes and depends mainly on the cluster network. The only case in which $\overline{t_{LB}} = 0$ is when a local execution is performed. On the other hand, the rest of values needed to perform the initiation mechanism are well known static parameters except α_j which can be computed from the last index value (I) received from the j_{th} system node.

- **Load balance operation**: Once a load balancing operation is initiated, the load balancing algorithm tries to allocate the new task accomplishing with the algorithm objective using the location rule.
Negotiated operations are always performed, thus, the sender node always asks the receiver one to accept the task and the receiver node can accept or reject the load balancing operation.
The proposed algorithm is based on non-preemptive allocation, and only newly arriving tasks can be selected to perform load balancing operations in the selection rule. Only when the robustness metric recommends it, process migration can be used.

5.2. Application and Experimental Setup. A real heterogeneous cluster called Medusa (Table 5.1) has been used to perform the desired experiments. It is a cluster composed by 32 different nodes interconnected with a Gigabit Ethernet network.

The application executed on this cluster system to evaluate the solutions proposed in this work is composed of different kinds of independent tasks with different CPU requirements.

The CPU-bound task programmed for the experiments ($t100$) performs a simple image processing operation the 100% of its response time, i. e., it can always be running on the processor. The IO-bound tasks compute this operation too, but they have to read from and write to the hard disk (IO operation) a certain fraction of their total response time. For example, $t10$ is a task which is a 10% of its response time processing the image (using the CPU) and the rest of the time performing disk accesses. The task $t20$ is performing the image processing a 20% of its response time, and so on.

Table 5.1
Cluster Medusa (Q=32)

Nodes	Processor	GHz	RAM (MB)	Bogomips
n0-n17	Pentium IV	1.7	256	3394.76
n18-n20	Pentium III	0.864	128	1723.59
n21-n31	Pentium III	0.664	128	1327.10

Table 5.2
Results for the information rule in experiment 1 (low variability)

Information policy	T_{info}/χ	T (s)	S
Periodic	1	345	1.26
Periodic	3	318	1.37
Periodic	10	302	1.44
Periodic	20	275	1.59
Periodic	30	310	1.41
Periodic	60	336	1.30
Periodic	120	549	0.79
Periodic	180	805	0.54
Periodic	300	1098	0.40
Periodic	600	1416	0.31
On-demand	0	232	1.88
Events	0.1	235	1.86
Events	0.3	229	1.90
Events	0.4	218	2.00
Events	0.5	284	1.54
Events	0.7	293	1.49
Events	0.8	327	1.33

In the next sections, the results of three different experiments will be shown, experiment 1 with low variability (dedicated cluster, only executing the example application), experiment 2 with medium variability (non-dedicated cluster with 3 random load peaks during the experiment in system nodes from n0 to n11) and experiment 3 with large variability (non-dedicated cluster with 6 random load peaks during the experiment in all the system nodes). The unpredictable load peaks have been simulated with a set of short tasks arriving to the nodes every 0.3 seconds and consuming CPU and memory resources.

The example application is composed by 320 tasks (W=320), all arriving to the lower half of the system (nodes from n0 to n15) and the elapsed time of this application without load balancing algorithm is $t_{without\ algorithm}$= 436 seconds in the experiment 1, $t_{without\ algorithm}$= 498 seconds in the experiment 2 and $t_{without\ algorithm}$= 545 seconds in the experiment 3.

5.3. Experimental results for the information rule. In the first set of experiments the three traditional information policies have been compared in environments with different variabilities. The second version of the load balancing algorithm has been used, therefore the initiation mechanism tries to allocate tasks to maintain the total response time below a certain value. For the event driven policy, the IQR metric with K=5 has been used to quantify the workload variability.

Tables 5.2, 5.3 and 5.4 show the results obtained for experiments 1, 2 and 3. In all these tables the total application response time (T) and the speedup obtained comparing this time with the response time without load balancing algorithm (S) are shown. For the periodic and event-driven policies different values of the information period (T_{info}) and of the event threshold (χ) have been used.

It can be seen that in the low variability environment (experiment 1) the best speedup results are obtained with the event-driven policy, specifically with a threshold value of χ=0.4 the speedup obtained with the algorithm is 2.00. But the on-demand and periodic policies are able to achieve very similar results, in the case of the periodic approach, with an information interval of T_{info}=20 s, the speedup value is 1.59, and for the on-demand policy the speedup is 1.88.

When the environment variability increases, two phenomenons can be observed. The first, to obtain the best speedup values the information exchange must occur more frequently. Therefore, for the periodic approach the information period must decrease and the same must happen with the event threshold for the event-driven

TABLE 5.3
Results for the information rule in experiment 2 (medium variability)

Information policy	T_{info}/χ	T (s)	S
Periodic	1	377	1.32
Periodic	3	354	1.41
Periodic	10	395	1.26
Periodic	20	416	1.20
Periodic	30	459	1.08
Periodic	60	481	1.04
Periodic	120	699	0.71
Periodic	180	993	0.50
Periodic	300	1344	0.37
Periodic	600	1854	0.27
On-demand	0	342	1.46
Events	0.1	276	1.80
Events	0.3	255	1.95
Events	0.4	288	1.73
Events	0.5	376	1.32
Events	0.7	397	1.25
Events	0.8	412	1.21

TABLE 5.4
Results for the information rule in experiment 3 (large variability)

Information policy	T_{info}/χ	T (s)	S
Periodic	1	423	1.29
Periodic	3	461	1.18
Periodic	10	488	1.12
Periodic	20	501	1.09
Periodic	30	548	0.99
Periodic	60	599	0.91
Periodic	120	625	0.87
Periodic	180	889	0.61
Periodic	300	1245	0.44
Periodic	600	3874	0.14
On-demand	0	417	1.31
Events	0.1	293	1.86
Events	0.3	360	1.51
Events	0.4	408	1.34
Events	0.5	456	1.20
Events	0.7	515	1.06
Events	0.8	539	1.01

policy. These best values can be obtained with χ=0.3 and T_{info}= 3 s for the experiment 2 and with χ=0.1 and T_{info}= 1 s for the experiment 3.

The second, the only information policy capable of achieving similar results in the highly variable system than in the low variability environment is the event-driven policy once the χ value is adequately tuned. The speedup decreases only from 2.00 to 1.86 when the variability changes, although the worsening for the on-demand policy is from 1.88 to 1.31 and for the periodic policy is from 1.59 to 1.29.

Once the event-driven has been selected to deal with variability, the second set of experiments has been performed to determine the best metric to quantify the workload variability (V) for load balancing purposes. Table 5.5 shows the obtained results using different variability metrics in the experiment 3, the most significative due to its large variability value. Different χ values for the events definition and different lengths (K) for the vector with the last load index values have been used. The number of information events (and therefore, the number of information messages, directly related to the network overhead), and the total application response time have been measured to evaluate the load balancing algorithm performance, E and T respectively.

In a low variability environment the results obtained with the three metrics would have been very similar with the K value properly configured. In a stable situation it is not necessary to inform very often about the nodes state and with a few events is enough to have updated information about the global state and to take

Table 5.5
Variability metrics comparison in experiment 3 (large variability)

Event	χ	K	E	T (s)	S
Range	0.1	3	130	341	1.60
Range	0.3	3	104	412	1.32
Range	0.5	3	71	521	1.05
Range	0.7	3	49	596	0.91
Range	0.1	5	141	350	1.56
Range	0.3	5	117	415	1.31
Range	0.5	5	95	533	1.02
Range	0.7	5	63	590	0.92
IQR	0.1	5	92	293	1.86
IQR	0.3	5	69	360	1.51
IQR	0.5	5	31	456	1.20
IQR	0.7	5	17	515	1.06
IQR	0.1	10	101	308	1.77
IQR	0.3	10	73	385	1.42
IQR	0.5	10	45	478	1.14
IQR	0.7	10	28	543	1.00
σ	0.1	5	84	306	1.78
σ	0.3	5	58	374	1.46
σ	0.5	5	25	470	1.16
σ	0.7	5	8	526	1.04
σ	0.1	10	81	328	1.66
σ	0.3	10	52	399	1.37
σ	0.5	10	19	502	1.09
σ	0.7	10	7	588	0.93

correct load balancing decisions. The range metric can obtain good results in this kind of situation, because there are not important differences between successive load index values. Anyway, this metric may have better performance with low K values, because the higher this value is the more probability of having measurement errors affecting the variability value (the range increases with the size of the data set as it has been said in section 3.1). On the other hand, the IQR metric can obtain very good results in these environments, even better than the range, because it ignores these extreme values. And finally, the standard deviation is able to obtain similar results than the other two metrics with low variability values.

But the results showed in Table 5.5 are quite different because in this case the system is suffering a high variability. The χ value which leads to the best response times is very low in all the cases because there are a lot of state changes in the system nodes and therefore, more events and information messages are needed to maintain updated state information and to make correct load balancing decisions.

As it was expected, the range metric performance is very poor in this kind of environment. Even with a low K value, (K=3) the effects of the extreme values on the metric value turns it into a bad election.

However, the IQR metric avoids these problems because it ignores the extreme values. And, furthermore, it can react to unforeseen load changes in a short interval of time, enough to maintain up-to-date state information in almost all the environments. Table 5.5 shows that this metric can obtain the best response time value with an acceptable number of events.

On the other hand, the standard deviation metric suffers the consequences of smoothing the state changes. Its slow variation with the load index changes increases the response time and decreases this metric performance in environments with high variability, because the load balancing decisions are very often based on obsolete state information. The load index changes take longer to have effects on the variability values with this metric than with the range or with the IQR, due to the use of the data arithmetic mean in its calculation and this leads to a low number of events and to a poor information updating.

To conclude, it can be seen that the best results have been obtained with K=5. With the application used in these example experiments, 5 load index values are enough to measure the variability, while larger values suppose a too long-term approximation. But it has been observed that with other applications, the best value for K could be another one. Therefore, the optimal value for this parameter is application-dependent and must be tuned properly depending on the application executed on the cluster.

5.4. Experimental results for the load balancing operation. In these last experiments, the convenience of process migration in experiments 1, 2 and 3 has been evaluated using the robustness metrics derived in section 4. In these three experiments the two versions of the load balancing algorithm with the two proposed objectives have been used.

With all the equations proposed in section 4 to perform the robustness analysis, the robustness radius for different system configurations can be theoretically computed.

First, the robustness radius for the load balancing based on the task response time objective can be computed with the QoS-robustness, because it is an objective related to the individual tasks composing the application. The robustness tolerance has been fixed in τ_R=0.85 and the expected achievement is $\overline{A}$=1, therefore:

$$A \geq 0.85 \cdot \overline{A} = 0.85.$$

The application executed in these experiments is composed of 320 tasks, therefore:

$$A = \frac{W_C}{W} = \frac{W_C}{320} \geq 0.85, \quad W_C \geq 320 \cdot 0.85 = 272,$$

and:

$$W_C = \overline{W_C} - L \geq 272, \quad L \leq 48.$$

The robustness radius for this system-application combination is 48 using the QoS-robustness. On the other hand, if the total response time objective is considered, the P-robustness radius must be computed, because it is a global objective related to the total application. With the same robustness tolerance, the following condition must be fulfilled:

$$P \geq 0.85 \cdot \overline{P}.$$

In this case, to compute the expected system power:

$$P = \lambda \cdot \frac{1}{1 + \frac{T}{\overline{T}}}, \quad \overline{P} = \overline{\lambda}\frac{1}{1 + \frac{T}{\overline{T}}} = \frac{320}{\overline{T}} \cdot 0.5.$$

Because in the ideal situation $T = \overline{T}$. In the example application the expected response time is $\overline{T}$=300 s. Therefore:

$$\overline{P} = 0.53.$$

And the P-robustness condition is in this case:

$$P \geq 0.85 \cdot 0.53 = 0.45.$$

Knowing the desired finishing time for all the tasks composing the application, the vector $\overline{F}$ can be built. And with this vector and the rest of available information once the application have been executed on the system one time (supposing that there is not a priori information), the robustness radius for all the system nodes can be computed. In this case the minimum robustness radius is the one corresponding to the node $n2$. For this node the vector q is:

$$q_2 = [2\ 7\ 21\ 34\ 58\ 63\ 94\ 103\ 120\ 193\ 206\ 315].$$

This vector stores the identifiers of all the application tasks allocated to this node, therefore, this node has executed 12 tasks ($|q_2| = 12$) . Using the expression 4.11 for this node and supposing the starting time $T_o = 0$:

$$\rho_2 = \frac{\frac{300}{0.85} - 281}{\sqrt{12}} = 20.76\ s$$

with

$$M_2(\overline{F}) = \max(\overline{F(q_2(k))}) = F(206) = 281\ s.$$

Table 5.6
QoS-robustness for experiments 1, 2 and 3

Experiment	L	Kind of system
1	14	QoS-robust
2	20	QoS-robust
3	33	QoS-robust

Table 5.7
P-robustness for experiments 1, 2 and 3

Experiment	$\max(\|F(k) - \overline{F(k)}\|)$	Kind of system
1	6.45	P-robust
2	15.67	P-robust
3	22.45	No P-robust

The meaning of this results is very simple to interpret. In the case of the QoS-robustness, only 48 tasks or less can fail in achieving the objective of having a response time below $\bar{t}$ to comply with the robustness condition given in equation 4.1. It has to be pointed that it is a discrete radius because it is a number of tasks.

On the other hand, for the P-robustness the robustness radius is continuous because it is a time measurement. The meaning of this radius value is the maximum deviation a task can have from its expected finishing time without causing a violation of the condition given in equation 4.7. Therefore, the maximum difference $|F(k) - \overline{F(k)}|$ should be 20.76 s to have a robust system-application combination.

To validate these conclusions, an experimental analysis has been performed. The obtained results are shown in Tables 5.6 and 5.7 for the QoS-robustness and for the P-robustness respectively. Real executions of the studied application have been performed on the Medusa cluster to measure the average L values (number of tasks that are executed without achieving the desired response time $\bar{t}$), the objective achievement value (A), the maximum deviations of the tasks finishing times from their ideal finishing times($\max(|F(k) - \overline{F(k)}|)$), the application response time (T) and the system power (P) in experiments 1, 2 and 3. And the conditions expressed in equations 4.1 and 4.7 have been evaluated to verify if the predictions made with the robustness radius about the system robustness are correct.

The results shown in Tables 5.6 and 5.7 verify the proposed robustness metrics and the expressions proposed to compute their associated robustness radius. The L value is always below the QoS robustness radius of 48 and therefore the system is always able to fulfill the robustness condition expressed in 4.1. The same thing occurs with the P-robustness in experiments 1 and 2. But on the other hand, the condition expressed in equation 4.7 cannot be fulfilled for the experiment 3. Therefore, the $\max(|F(k) - \overline{F(k)}|)$ value is above the P-robustness for this experiment and process migration would be needed to achieve the desired system robustness.

6. Conclusions and Future Work. A deep analysis of the effects of variability on load balancing performance on cluster systems has been presented. This analysis has demonstrated that these effects are very important for the information rule and the load balancing operation (assuming a good load index selection).

The main contribution of this paper is the proposal of specific solutions for considering the workload variability in this algorithm stage without causing unnecessary overheads on the system nodes or network. These solutions allow to improve the load balancing algorithms performance in highly variable environments such as non dedicated clusters.

For the information rule, an event-driven solution is proposed. The events must be determined with the load index variability on the system nodes. Experimental results have demonstrated that the best variability metric for load balancing purposes is the inter-quartile range (IQR) in all the possible environments. The number of load index values considered for its calculation (K) and the threshold for the information events (χ) must be configured to optimize the load balancing performance depending on the degree of variability and on the executed application.

For the location rule negotiated load balancing operations should be used to give the selected node the chance to reject the operation if its state has changed during the load balancing decisions.

And finally, for the distribution rule, a non preemptive allocation is proposed. But a robustness metric must be used to determine if process migration is needed in certain variable situations to maintain the desired system

performance. Two new robustness metrics, the QoS-robustness and the P-robustness have been proposed for individual tasks algorithm objectives and for global objectives, respectively.

Both robustness metrics have demonstrated their accuracy in warning the user or administrator when process migration should be used to manage the workload variability because it is causing intolerable performance degradations. A very interesting line for future research is incorporating the robustness computation to the load balancing algorithm to decide in real-time which distribution rule should be used depending on the workload variability: remote execution or process migration.

REFERENCES

[1] J. Aikat, J. Kaur, F. D. Smith, and K. Jeffay, *Variability in TCP round-trip times*, in Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement, ACM Press, 2003, pp. 279–284.

[2] S. Ali, A. Maciejewski, H. Siegel, and J.-K. Ki, *Definition of a robustness metric for resource allocation*, in Proceedings of the International Symposium on Parallel and Distributed Processing, 2003, pp. 42–50.

[3] S. Ali, H. Siegel, and A. Maciejewski, *The robustness of resource allocation in parallel and distributed computing systems*, in Proceedings of the Third International Symposium on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks, 2004, pp. 2–10.

[4] T. Anderson and J. D. Finn, *The new statistical analysis of data*, Springer, 1997.

[5] R. J. Anthony, *Self-configuration in parallel processing*, in Proceedings of the 16th International Workshop on Database and Expert Systems Applications, 2005.

[6] M. Beltran and J. L. Bosque, *Estimating a workstation CPU assignment with the DYPAP monitor*, in Proceedings of the Third International Symposium on Parallel and Distributed Computing, 2004, pp. 64–70.

[7] M. Beltrán, J. L. Bosque, and A. Guzmán, *Initiating load balancing operations*, in Proceedings of the EuroPar2005, 2005, pp. 292–301.

[8] M. Beltrán, A. Guzmán, and J. L. Bosque, *Dealing with heterogeneity in load balancing algorithms*, in Proceedings of the Fifth International Symposium on Parallel and Distributed Computing, 2006, pp. 123–132.

[9] R. Buyya, *High Performance Cluster Computing: Architecture and Systems, Volume I*, Prentice-Hall, 1999.

[10] K.-T. Chen, P. Huang, C.-Y. Huang, and C.-L. Lei, *The impact of network variabilities on TCP clocking schemes*, in Proceedings of the 4th Annual Joint Conference of the IEEE Computer and Communications Societies, vol. 4, 2005, pp. 2770–2775.

[11] J. J. Evans and C. S. Hood, *Network performance variabilty in NOW clusters*, in Proceedings of the IEEE International Symposium on Cluster Computing and the Grid, 2005, pp. 1047–1054.

[12] R. Gusella, *Characterizing the variability of arrival processes with indexes of dispersion*, IEEE Journal on Selected Areas in Communications, 9 (1991), pp. 203–211.

[13] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 2003.

[14] C. Hughes, P. Kaul, S. Adve, R. Jain, C. Park, and J. Srinivasan, *Variability in the execution of multimedia applications and implications for architecture*, in Proceedings of the 28th Annual International Symposium on Computer Architecture, vol. 4, 2001, pp. 245–265.

[15] P. Jogalekar and M. Woodside, *Evaluating the scalability of distributed systems*, IEEE Transactions on Parallel and Distributed Systems, 11 (2000), pp. 589–603.

[16] D. S. Milojicic, F. Douglis, D. Paindaveine, M. Wheeler, and Z. Zhou, *Process migration*, ACM Computing Surveys, 32 (2000), pp. 241–299.

[17] R. Mirchandaney, D. Towsley, and J. A. Stankovic, *Analysis of the effects od delays on load sharing*, IEEE Transactions on Computers, 38 (1989), pp. 1513–1525.

[18] R. Wolski, N. Spring, and J. Hayes, *Predicting the CPU availability of time-shared unix systems on the computational grid*, in Proceedings of the 8th International Symposium on High Performance Distributed Computing, IEEE, 1999, pp. 105–112.

[19] C. Xu and F. C. M. Lau, *Load Balancing in Parallel Computers: Theory and Practice*, Kluwer Academic Publishers, 1997.

Edited by: Marek Tudruj, Dana Petcu
Received: January 22th, 2009
Accepted: June 28th, 2009

Scalable Computing: Practice and Experience
Volume 10, Number 2, pp. 147–161. http://www.scpe.org
ISSN 1895-1767
© 2009 SCPE

MODEL-DRIVEN ENGINEERING AND FORMAL VALIDATION OF HIGH-PERFORMANCE EMBEDDED SYSTEMS

ABDOULAYE GAMATIÉ[*], ÉRIC RUTTEN[†], HUAFENG YU[‡], PIERRE BOULET[‡], AND JEAN-LUC DEKEYSER[‡]

Abstract. The study presented in this paper concerns the safe design of high-performance embedded systems, specifically dedicated to intensive data-parallel processing as found, for instance, in modern multimedia applications or radar/sonar signal processing. Among the important requirements of such systems are the efficient execution, reliability and quality of service. Unfortunately, the complexity of modern embedded systems makes it difficult to meet all these requirements.

As an answer to this issue, this paper proposes a combination of two models of computation (MoCs) within a framework, called Gaspard, in order to deal with the design and validation of high-performance embedded systems. On the one hand, the repetitive MoC offers a powerful expression of the parallelism available in both system functionality and architecture. On the other hand, the synchronous MoC serves as a formal model on which a trustworthy verification can be applied. Here, the high-level models specified with the repetitive MoC are translated into an equational model in the synchronous MoC so as to be able to formally analyze different properties of the modeled systems. As an example, a clock synchronizability analysis is illustrated on a multimedia system in order to guarantee a correct interaction between its components. For the implementation and validation of our proposition, a Model-Driven Engineering (MDE) approach is adopted. MDE is well-suited to deal with design complexity and productivity issues. In our case, the OMG standard MARTE profile is used to model embedded systems. Then, automatic transformations are applied to these models to produce the corresponding synchronous programs for analysis.

Key words: design, high-performance embedded systems, repetitive MoC, synchronous languages, formal validation

1. Introduction.

High-performance computing is increasingly becoming important in embedded systems. Very typical application domains are medical imaging or state-of-the-art multimedia devices. Nowadays, multimedia mobile devices such as Personal Digital Assistant (PDA), multimedia cell phones and mobile multimedia players are ubiquitous. These devices provide many functionalities, such as mp3 playback, camera, video and mobile TV. These features, together with their small size and long power supply make them very attractive in the market. However, they make the design of systems more challenging due to their ever increasing complexity and the need of high quality products in terms of reliability and usability.

In high-performance computing, parallelism plays a central role in order to get efficiency as much as possible. The multimedia application examples mentioned above are characterized by intensive data-parallel processing. Unlike general parallel applications, which are often concerned by code parallelization, intensive data-parallel applications concentrate on the regular data partitioning and access. The data manipulated in these applications are generally represented as multidimensional data structures, e.g. arrays. The highly desirable design approaches for such applications are those providing designers with concepts to suitably represent data manipulations, and techniques that trustworthily guarantee important implementation requirements.

The current study addresses this issue by combining technologies for an efficient design of high-performance embedded systems and formal validation of designs.

1.1. Motivation example: video processing.

Let us consider a video processing system as illustrated in Fig. 1.1: images are captured via a CMOS sensor, then downscaled and displayed on a Thin Film Transistor (TFT) screen. TFT refers to active matrix screens on laptop computers, which offers sharper screen displays and broader viewing angles than does passive matrix. The best known application of TFT is in Liquid Crystal Display (LCD) technology.

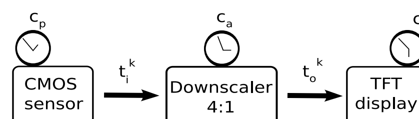


Fig. 1.1. *A video processing system.*

The downscaler transforms a VGA signal (640×480 pixels per frame) into a QVGA signal (320×240 pixels per frame). The required downscaling ratio is therefore 4:1. This operation is interesting when visualizing high

[*]CNRS-LIFL, INRIA, Parc de la Haute Borne, Bât A, 40 av. Halley, 59650 Vil. d'Ascq, France
[†]INRIA Rhône-Alpes, 655 av. de l'Europe, Montbonnot, 38334 Saint-Ismier cedex, France
[‡]USTL-LIFL, INRIA, Parc de la Haute Borne, Bât A, 40 av. Halley, 59650 Vil. d'Ascq, France

quality live video on TFT screen while using low power and real-time previews in recent generation cellular phones. Such phones include several multimedia functionalities, provided by dedicated modules and processors integrated in chips. A more detailed operational view of the whole video processing system is as follows: the CMOS sensor gathers data and sends to the downscaler a flow of pixels, denoted by t_i^k. After the receipt of a sufficient number of pixels, the downscaler transforms these pixels. The resulting pixels denoted by the flow t_o^k are sent to the TFT screen, which waits for receiving enough data before displaying images. Each component (sensor, downscaler and display) is assumed to hold a logical clock that defines its activation instants. These clocks are related by affine relations, characterizing the frequencies at which images are received, transformed and displayed. *We want to model such a system and analyse how its components' clocks must be synchronized so that the video can be displayed w.r.t. quality of service requirements.*

1.2. Rationale and contribution. In order to answer the above demand, we consider the GASPARD framework (`https://gforge.inria.fr/projects/gaspard2`), which is dedicated to the design of high-performance embedded systems [11]. GASPARD promotes a software/hardware co-design based on *model-driven engineering* (MDE) [21]. Models are described by using the OMG MARTE (*Modeling and Analysis of Real-Time and Embedded systems*) standard profile [19], combined with a few native UML concepts. The Hardware Resource Model (HRM) concepts of MARTE enable to describe the hardware part of a system. The Repetitive Structure Modeling (RSM) concepts allow one to describe repetitive structures. Finally, the Generic Component Modeling concepts are used as the base for component modeling.

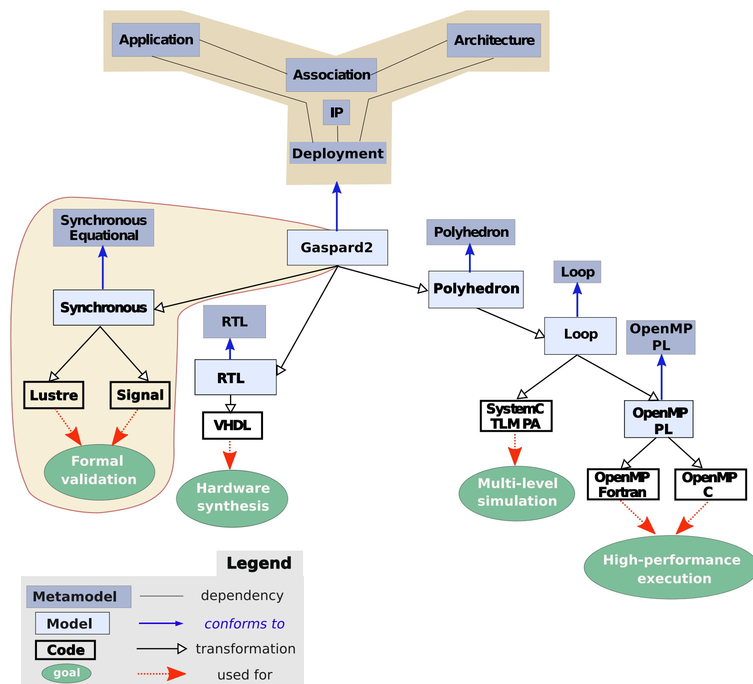


FIG. 1.2. *Embedded system design within the* GASPARD *framework.*

In the application functionality part of Fig. 1.2, the focus is put on the expression of data dependencies between components in order to describe an algorithm. The manipulated data are mainly multidimensional arrays. Furthermore, a form of reactive control can be modeled via the notion of execution modes. In the hardware architecture part, similar mechanisms are used to describe regular architectures in a compact way. Regular parallel execution units are more and more present in embedded systems, especially in System-on-Chip (SoCs). HRM is fully used to model these concepts. The association part concerns the allocation of the application functionality onto available execution resources, and the scheduling. The allocation model also takes advantage of the repetitive and hierarchical representation offered by MARTE to express mappings at different granularity levels and in a factorized way.

There are several models and languages for the programming of high-performance systems such as MPI [18] or the DARPA high productivity languages: Chapel [5], Fortress [1] and X10 [6]. These languages consider

specific architectures models. This does not facilitate SoC design in which one often needs to program various specialized architectures. In GASPARD, the adopted MARTE profile is rich enough to enable the description of several types of SoC architectures.

In addition to the previous design aspects, GASPARD proposes a notion of *deployment* specification in order to generate compilable code from a SoC model. The corresponding concepts enable *i)* to describe a relation between a MARTE representation of each elementary component and a text-based code, e.g. an *intellectual property* (IP); and *ii)* to enrich the transformed high-level models with non functional information such as the average execution time or the power consumption for design space exploration. The deployed models are refined towards different technologies for various purposes: formal validation with synchronous languages [2] as shown in this paper, hardware synthesis at the register transfer level (RTL) with VHDL, simulation at transaction level modeling (TLM) in SystemC, high-performance execution in OpenMP Fortran and C. The refinement distinguishes several abstract levels at which the manipulated concepts are characterized by a dedicated metamodel. The transitions from one level to another are defined by automatic model transformations.

Among the similar existing co-design frameworks, we can mention those relying on *platform-based design* (PBD) [20]. Their main idea is to facilitate the design by enabling successive refinements of high level specifications of system functionality and architecture with reusable components so as to rapidly meet the implementation requirements of the system. However, the design flexibility may be significantly reduced since the deployment choice is only limited to the available pre-defined components.

In this paper, we define a bridge between the modeling formalism of GASPARD and the synchronous languages so as to get access to formal validation tools. More precisely, we show how models defined with the repetitive model of computation (MoC) of GASPARD are transformed into a set of dataflow equations in synchronous languages. This transformation is implemented according to an MDE technique. As an example of analysis that can be applied to the resulting synchronous code, we address the synchronizability between the components of the video processing system shown in Fig. 1.1. For that, we use the affine clock notion of the SIGNAL language [22]. More generally, this paper gives a summary of [14, 13] and extends them with the translation of inter-repetition dependencies during the transformation of GASPARD models into synchronous programs. This extension is important for the modeling of control-oriented computations in the repetitive MoC. The implementation of all these aspects is exposed using MDE. The metamodel corresponding to the generated synchronous equational models is detailed as well as the transformation rules. These aspects, which are more related to the implementation are not addressed in [14, 13].

1.3. Outline. In the following, Section 2 describes the basic notions of the repetitive MoC which is used in GASPARD to express the parallelism inherent to systems. Section 3 deals with the translation of the models described with the repetitive MoC into synchronous programs. The implementation of this translation using MDE is addressed in Section 4. Section 5 presents how a clock synchronizability analysis can be carried out on the video processing system example, introduced previously by using the synchronous approach. Finally, Section 6 gives concluding remarks.

2. A repetitive model of computation. The repetitive MoC relies on the domain-specific language ARRAY-OL [3, 8], which is a mixed graphical-textual functional language enabling to specify both *task parallelism* and *data parallelism* in data intensive applications. In ARRAY-OL, manipulated data are *infinite multidimensional arrays*. These arrays are also *toroidal* because one may sometimes need to represent spatial or frequential dimensions representing physical tori (e.g., hydrophones around a submarine) or some toroidal frequency domains (e.g., obtained with Fast Fourier Transforms). The repetitive MoC is implemented by the RSM package of MARTE.

2.1. Basic concepts. To illustrate the basic concepts of the repetitive MoC, let us consider the downscaler component in Fig. 1.1. It is composed of two parts: a horizontal filter that reduces the number of pixels from a 640-line to a 320-line by interpolating packets of 8 pixels; and a vertical filter that reduces the number of pixels from a 480-line to a 240-line by interpolating packets of 8 pixels as well. The model of the downscaler in the GASPARD environment is illustrated in Fig. 2.1. It is represented by an aggregate component consisting of different levels: at the top-level, a *repetitive task* referred to as `Downscaler`; at the intermediate level, a *hierarchical task* represented by a directed acyclic graph where the nodes are the repetitive tasks `Horizontal filter` and `Vertical filter`; and at the low level, *elementary tasks* `Hfilter` and `Vfilter` that are repeated within the associated repetitive tasks. The whole downscaler model receives an infinity of frames, denoted by the input 3-D array $(640, 480, \infty)$, and produces an infinity of transformed frames, $(320, 240, \infty)$. In the model,

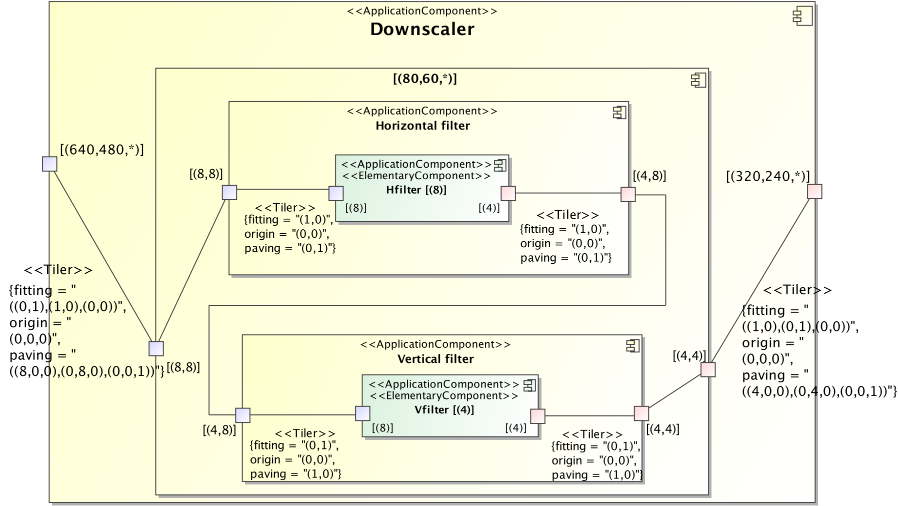


FIG. 2.1. *A model of downscaler in* GASPARD.

∞ is represented by the symbol $\star$. The way pixels are extracted from (*resp.* inserted in) these infinite arrays is described by *tilers*. The next paragraphs present each specific concept illustrated in the downscaler model.

Overview. The basic specification concepts of GASPARD are summarized by the grammar given below. The notation $x : X$ in this grammar means that X is the type of x, and $\{X\}$ denotes a set of objects of type of X.

$$
\begin{array}{lcll}
Task & ::= & Interface; Body & (r1)\\
Interface & ::= & i, o : \{Port\} & (r2)\\
Port & ::= & id; type; shape & (r3)\\
Body & ::= & Body^h \mid Body^r \mid Body^e & (r4)\\
Body^e & ::= & some\ function\ f & (r5)\\
Body^r & ::= & t_i, t_o : \{Tiler\}; (\mathbf{r}; Task) \mid & \\
 & & t_i, t_o : \{Tiler\}; (\mathbf{s_r}; Task); \{Connexion; \overrightarrow{d}; c_p\} & (r6)\\
Connexion & ::= & p_i, p_o : Port & (r7)\\
Tiler & ::= & Connexion; (F; \mathbf{o}; P) & (r8)\\
Body^h & ::= & \{Task\}; \{Connexion\} & (r9)
\end{array}
$$

Atomic computation: elementary tasks. An *elementary task* (rule ($r5$)) consists of functions that are executed atomically, e.g. the `Hfilter` task in Fig. 2.1.

Data-parallelism: repetitive tasks. A repetitive task (rule ($r6$)) expresses the data-parallelism by replicating a task into several *instances* that consume the elements of its input arrays and produce the elements of output arrays. Each instance executes independently of the others. The sub-arrays consumed and produced by repeated task instances have the same shape. They are referred to as *patterns* or *tiles*. For example, in Fig. 2.1 the execution of the repetitive task `Horizontal filter` will lead to the replication of 8 instances of the elementary task `Hfilter`.

The way the tiles are constructed is defined via *tilers* (rule ($r8$)), which are associated with each array (i.e. each edge in the graphical representation). A tiler extracts (resp. stores) tiles from (resp. in) an array based on some information: F is a *fitting* matrix (how array elements fill the tiles), $\mathbf{o}$ is the *origin* of the *reference tile* (for the *reference repetition*), and P is a *paving* matrix (how the tiles cover arrays).

The *repetition space* indicating the number of task instances is itself defined as a multidimensional array with a shape. Each dimension of the repetition space can be seen as a parallel loop and its shape space gives the bounds of the nested parallel loops. In Fig. 2.1, the shape of repetition space in `Horizontal filter` is (8).

Given a tile, let its *reference element* denote its origin point from which all its other elements can be extracted. The *fitting* matrix is used to determine these elements. Their coordinates, represented by $\mathbf{e_i}$, are built as the sum of the coordinates of the reference element and a linear combination of the fitting vectors, the

whole modulo the size of the array (since arrays are toroidal) as follows:

$$\forall \mathbf{i}, \mathbf{0} \leq \mathbf{i} < \mathbf{s}_{\text{pattern}}, \mathbf{e_i} = \text{ref} + F \times \mathbf{i} \bmod \mathbf{s}_{\text{array}} \tag{2.1}$$

where $\mathbf{s}_{\text{pattern}}$ is the shape of the pattern, $\mathbf{s}_{\text{array}}$ is the shape of the array and F is the fitting matrix. Fig. 2.2 illustrates the fitting result for a $(2,3)$-pattern with the tiling information indicated on the same figure. The fitting index-vector $\mathbf{i}$, indicated in each point-wise element of the pattern, varies between $\binom{0}{0}$ and $\binom{1}{2}$. The reference element is characterized by index-vector $\binom{0}{0}$.

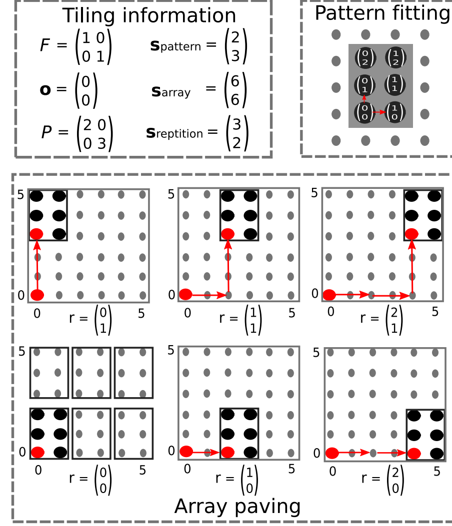


Fig. 2.2. *Example of paving and fitting scenarios.*

Now, for each repetition instance, one needs to specify the reference elements of the input and output tiles. The reference elements of the reference repetition are given by the *origin* vector, $\mathbf{o}$, of each tiler. The reference elements of the other repetitions are built relatively to this one. As above, their coordinates are built as a linear combination of the vectors of the *paving* matrix as follows:

$$\forall \mathbf{r}, \mathbf{0} \leq \mathbf{r} < \mathbf{s}_{\text{repetition}}, \text{ref}_{\mathbf{r}} = \mathbf{o} + P \times \mathbf{r} \bmod \mathbf{s}_{\text{array}} \tag{2.2}$$

where $\mathbf{s}_{\text{repetition}}$ is the shape of the repetition space, P the paving matrix and $\mathbf{s}_{\text{array}}$ the shape of the array. The paving illustrated by Fig. 2.2 shows how a $(2,3)$-patterns tile a $(6,6)$-array. Here, the paving index-vector $\mathbf{r}$, varies between $\binom{0}{0}$ and $\binom{2}{1}$.

Task instances may sometimes depend on other task instances in the same repetition space (second alternative of rule $(r6)$). For example, this happens when computing the sum of the elements of an array by considering the partial sum previously calculated at each step. Such a constraint therefore induces a total execution order on a repetitive task.

In the corresponding rule $(r6)$ of the summary grammar, *Connexion* represents the pair of ports connected by the inter-repetition dependency link: an input of $Task$ and an output of $Task$. The vector $\overrightarrow{d}$ specifies the coordinates of the link in the repetition space. For each repetition instance, c_p denotes a pattern to be used as input by the next repetition instance. Initially, c_p holds a default value, let us call it def. There could be at the same time several inter-repetition dependencies within a task since an instance may require values from more than one instances to compute its outputs. This is why rule $(r6)$ specifies a set of dependency link vectors $\{Ird\}$.

Fig. 2.3 illustrates a simplified notation for a repetitive task with an *inter-repetition dependency*. This task encodes an automaton. TFC is a transition function that computes, given some input events p_i^k from its environment (used in the transition conditions) and a current state c_p, the value of the next state p_o^k (resulting from the transition). Here, the dependency vector (-1) indicates that the result p_o^k depends on the state c_p computed in the previous step. The initial state s_i is given as the default value def. This example is a very classical encoding of an automaton as a sequential circuit.

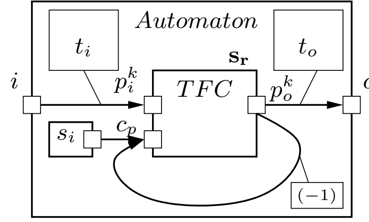


FIG. 2.3. *An automaton as a repetitive task with inter-repetition dependency.*

Task parallelism: hierarchical tasks. They are represented by hierarchical acyclic graphs in which each node consists of a task, and edges are labelled by the arrays exchanged between these tasks (e.g. `Horizontal filter` and `Vertical filter` in Fig. 2.1. This naturally leads to hierarchical description of tasks (rule ($r9$)).

3. Projection of GASPARD on the synchronous model.

3.1. The synchronous language SIGNAL. The synchronous language SIGNAL [16] belongs to the family of dataflow languages. SIGNAL handles unbounded series of typed values $(x_\tau)_{\tau \in \mathbb{N}}$, called *signals*, denoted as `x` and implicitly indexed by discrete time. At a given instant, a signal may be present, at which point it holds a value; or absent and denoted $\perp$. The set of instants where a signal `x` is present represents its *clock*, noted `^x`. Two signals `x` and `y`, which have the same clock are said to be *synchronous*, noted `x ^= y`. A *process* (or *node*) is a system of equations over signals that specifies relations between values and clocks of the signals. A *program* is a process. SIGNAL relies on the following six primitive constructs:

Relations. `y:= f(x1,...,xn)` $\stackrel{def}{\equiv}$ $\forall \tau \geq 0 : y_\tau \neq \perp \Leftrightarrow x1_\tau \neq \perp \Leftrightarrow ... \Leftrightarrow xn_\tau \neq \perp$, and $y_\tau = f(x1_\tau, ..., xn_\tau)$. Here, `y, x1,...,xn` are signals and `f` is a point-wise n-ary relation extended canonically to signals. This construct imposes `y, x1,...,xn` *i)* to be simultaneously present, and *ii)* to hold values satisfying the equation `y:= f(x1,...,xn)` whenever they occur.

Delay. `y:= x $ 1 init c` $\stackrel{def}{\equiv}$ $\forall \tau \; x_\tau \neq \perp \Leftrightarrow y_\tau \neq \perp$ *and* $\forall \tau > 0$: $y_\tau = x_k$, where $k = \max\{\tau' \mid \tau' < \tau$ and $x_{\tau'} \neq \perp\}$, $y_0 = c$. Here, `y, x` are signals and `c` is an initialization constant. This construct imposes *i)* `x` and `y` to be simultaneously present while *ii)* `y` must hold the value carried by `x` on its previous occurrence.

Undersampling. `y:= x when b` where `b` is Boolean $\stackrel{def}{\equiv}$ $\forall \tau \geq 0 : y_\tau = x_\tau$ if $b_\tau = true$, else $y_\tau = \perp$. Here, `y, x, b` are signals and `b` is of Boolean type. This construct imposes *i)* `y` to be present only when `x` is present and `b` holds the value *true*, while *ii)* `y` must hold the value carried by `x` at those logical instants. The expression `y:= when b` is equivalent to `y:= b when b`.

Deterministic merging. `z:= x default y` $\stackrel{def}{\equiv}$ $\forall \tau \geq 0 : z_\tau = x_\tau$ if $x_\tau \neq \perp$, else $z_\tau = y_\tau$. Here, `z, y, x` are signals. This construct imposes *i)* `z` to be present when either `x` or `y` are present while *ii)* `z` must always hold the value of `x` uppermost, otherwise it takes the value of `y`.

Composition. `(| P | Q |)` $\stackrel{def}{\equiv}$ union of the equations of `P` and `Q`, leading to the conjunction of the constraints associated with processes `P` and `Q`. It is commutative and associative.

Hiding (or restriction). `P where x` $\stackrel{def}{\equiv}$ `x` is local to the process `P`.

SIGNAL offers a process frame that enables the definition of sub-processes. Sub-processes that are only specified by an interface without internal behavior are considered as external, and may be separately compiled processes.

A useful notion of SIGNAL is the *oversampling* mechanism. It consists of a temporal refinement of a given clock c_1, which yields another clock c_2, which is faster than c_1, meaning that c_2 contains more instants than c_1.

In the SIGNAL process given in Fig. 3.1, called `k_Overspl`, `k` is a constant integer parameter (line `1`). The clock signals `c1` and `c2` respectively denote input represented by "`?`" and output represented by "`!`" (line `2`). Here, `c2` is a 4-oversampling of `c1`. The `event` type is associated with clocks. It is equivalent to a Boolean type where the only taken value is *true*. The local signals `cnt` and `pre_cnt` serve as counter to define 4 instants in `c2` per instant in `c1` (lines `3` and `4`).

```
1: process k_Overspl = {integer k;}
2:   (? event c1; ! event c2; )
3:   (| cnt := (k-1 when c1) default (pre_cnt-1)
4:    | pre_cnt := cnt $ 1 init 0
5:    | c1 ^= when (pre_cnt <= 0)
6:    | c2 := when (^cnt) |)
7:  where integer cnt, pre_cnt;
8: end; %process k_Overspl%
```

c1 :	tt	⊥	⊥	⊥	tt	⊥	⊥	...
cnt :	3	2	1	0	3	2	1	...
pre_cnt :	0	3	2	1	0	3	2	...
c2 :	tt	tt	tt	tt	tt	tt	tt	...

Fig. 3.1. *Specification of clock oversampling and an associated execution trace.*

There is a graphical syntax of Signal that is very similar to block diagrams. In such a syntax, a box represents a process and a connection between boxes represents the communication of signal values between processes (e.g. see Fig. 3.3 and 3.5).

3.2. Translation of Gaspard in Signal. We present two interpretations of Gaspard models in Signal according to the way the parallelism is considered.

3.2.1. Parallel interpretation. The translation of Gaspard models into synchronous models is structural. It is greatly facilitated by the similarity between Gaspard and Signal since both have a recursive block-diagram structure.

Structural translation. The recursive algorithm following the grammatical structure is as follows, starting with rule (r1) in the previous grammar:

- **(r1)** Each Gaspard task is represented by a Signal process/node, with an interface according to (r2), and a body translating the task body according to (r4).
- **(r2)** Each input and output array of an interface is translated according to (r3).
- **(r3)** Each port is translated as a Signal flow. The infinite dimension of an array can be suitably represented by an infinite flow of arrays of the remaining dimensions. Therefore, we concretely enforce this representation by modeling Gaspard arrays into flows of sub-arrays (Fig. 3.2).
- **(r4)** The body is translated according to the appropriate rule, being either an elementary task (r5), a repetitive task (r6), or a hierarchical task (r9).
- **(r5)** An elementary task E is represented by one equation, a Signal *Relation* construct, defining the outputs by applying a function to the inputs.
- **(r6)** Repetitive tasks are modeled by a compound Signal node, where: *i)* input tilers are transformed according to (r8); *ii)* the repeated computation is represented by a node with a composition of $|\mathbf{r}|$ instances of the node representing the repeated body, obtained by (r1); and *iii)* output tilers are transformed according to (r8). Fig. 3.3 graphically illustrates the Signal model corresponding to a repetitive task R with one input A_1 and one output A_2.
  When a repetitive task contains a dependency between repetitions, its associated model includes data dependencies between the repetition model instances (Fig. 3.4). The initial value is an input of the first task instance.
- **(r7)** Connexions are translated as assignments between the ports p_i and p_o.
- **(r8)** Each tiler is represented by a node: for input tilers, this node takes as input the array given by the connexion in (r7), and where each pattern is produced as output, by an equation extracting it from the input array. The indexes for each pattern are obtained by applying the paving and fitting equations (Section 2). Output tilers are represented by a node, where each pattern, given by the connexion in (r7), is inserted in the output array by an equation.
- **(r9)** Hierarchical tasks are modeled by the synchronous composition of nodes representing each of the sub-tasks, obtained by (r1), with the appropriate data dependencies defined by the connections in (r7).

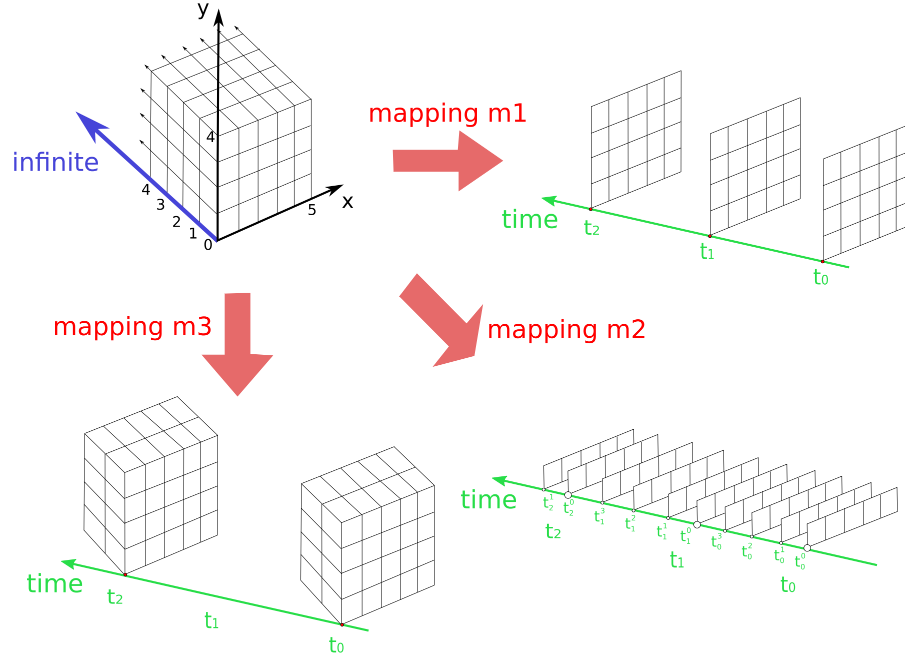


FIG. 3.2. *Space-time mapping of a* $[5,4,\infty]$*-array w.r.t. different granularity levels.*

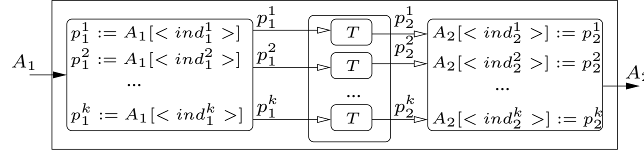


FIG. 3.3. *Parallel model of a repetitive task.*

The above parallel model is meant to be intuitive and simple, and may certainly be optimized. However, the considered optimizations can be quite different according to the intended target operations (e.g. efficient code generation or verification).

3.2.2. Serialized interpretation. While the parallel model fully preserves the data-parallelism of repetitive tasks, the *serialized* model rather sequentializes the repetitions. It is more compact than the parallel model because it only defines one instance of the repeated task in a repetitive task. It features a mono-processor execution of repetitions. More generally, a system will be modeled by combining both parallel and serialized models.

The main difference between the rules in the parallel and serialized interpretations concerns those describing repetitive tasks (r6) and tilers (r8), as follows:

(r6') Repetitive tasks are modeled by a compound SIGNAL node (see Fig. 3.5), where: *i)* the input tilers are encoded by *Array to Flow* according to (r8'); *ii)* the repeated task is represented by a single node instance T obtained by (r1); and *iii)* the output tilers are encoded by *Flow to Array* according to (r8').
The synchronous model of a repetitive task containing a dependency between repetitions is similar to the above model. We just need to handle the value transmission between two dependent repetition instances. Repetitions are performed sequentially, one at each logical instant. The value transmission is realized via by using simply a delay or register (Fig. 3.6).

(r8') Each tiler is represented by a node defined by the *Array to Flow* and *Flow to Array* components, which play a central role in the serialized model. Their definition partially relies on the oversampling mechanism introduced before.
Let us consider that the incoming array A_1 is received at the instants $\{\tau_{i,i\in\mathbb{N}}\}$ of a given clock c_1. Then, for each τ_i, the tile production algorithm is applied to the received array: the task instance T

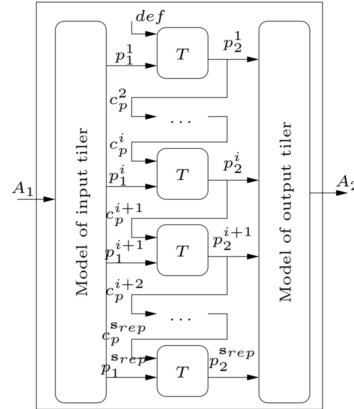


FIG. 3.4. *Parallel model of a repetitive task with an inter-repetition dependency.*

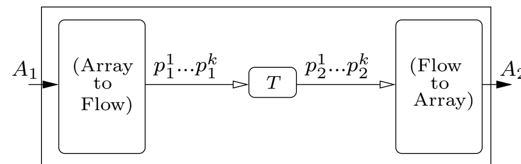


FIG. 3.5. *Serialized model of a repetitive task.*

is provided with the flow of tiles t_1^j and produces the tile flow t_2^j; the produced tile flow is therefore associated with a clock $c_2 = k \uparrow c_1$, where $0 \leq j \leq k$. The constant integer k is the number of paving iterations deduced from the repetition space of the task.

For more details about the correctness of the above parallel and serialized interpretations of GASPARD models, the reader can refer to [14].

4. Implementation of the translation using MDE. Our translation is implemented by a prototype transformation tool relying on MDE. This tool enables to automatically generate synchronous programs from GASPARD models [23].

4.1. Metamodeling. We present the main concepts defined in the metamodels considered during the implementation of our translation.

4.1.1. Repetitive structure modeling in MARTE. A subset of the MARTE profile, called RSM package, is based on our repetitive MoC and is dedicated to the modeling of regular structures (either in application or architecture). All the concepts we focus on are clearly identified in this package. Fig. 4.1 depicts the basic UML stereotypes associated with RSM. The `Tiler` stereotype, on the bottom right-hand side, comprises the `origin`, `paving` and `fitting` attributes for tiling operations.

The package indicates further stereotypes such as the `InterRepetition` dependency link presented previously. The `Reshape` stereotype enables us to express complex link topologies in which the elements of a multidimensional array are redistributed in another multidimensional array. It is a combination of two `Tilers`, identified with the `srcTiler` and the `targetTiler` relations in Fig. 4.1. A complete description of all these concepts is provided in [19].

4.1.2. A metamodel for synchronous equational systems. The proposed metamodel aims at the different target synchronous data-flow languages (LUSTRE, LUCID SYNCHRONE and SIGNAL) at the same time. These languages have considerable common aspects, which enable their code generation with the help of only one metamodel. A metamodel for synchronous equational systems is therefore proposed. Note that contrarily to the SIGNALMETA metamodel [4] dedicated to SIGNAL, our metamodel is not intended to have exactly the same expressivity as the target languages. The next paragraphs give details about the relevant sub-parts of this metamodel for synchronous equational systems. The defined generic concepts correspond to the notions introduced in Section 3.1. The metamodel is presented in several parts: `Signals`, `Equation`, `Node` and `Module`.

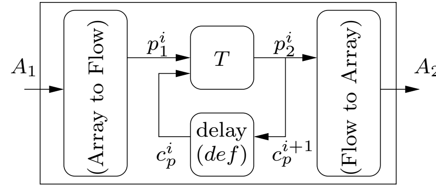


FIG. 3.6. *Serialized model of a repetitive task with an inter-repetition dependency.*

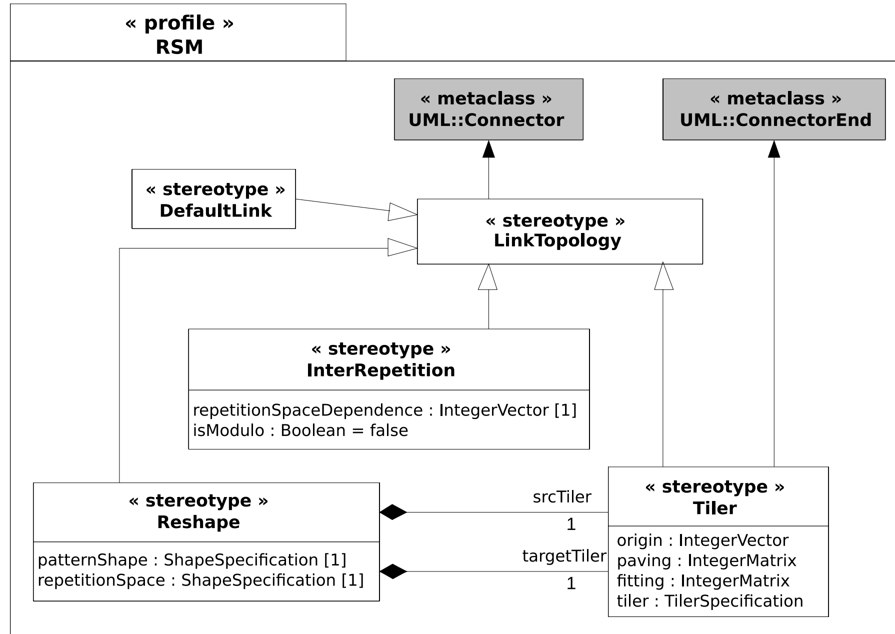


FIG. 4.1. *The RSM package of MARTE.*

Signal. The concept of `Signals` (Fig. 4.2) refers to variables. `SignalDeclaration` declares the name and type of a `Signal`. A `Signal` can be a local or an interface variable, respectively captured by `SignalLocalDec` and `SignalInterDec`. A declared `Signal` can be referenced. The `SignalUsage` represents an operation on a `Signal`. If a `Signal` is an array, a `SignalUsage` can be an operation on its sub-parts. When such an array is divided into several sub-parts (tiles), it will be associated with the same number of `SignalUsage`. Every `SignalUsage` has an `IndexValueSet`. `Signals` are used in both sides of equations. Each side is part of the equation arguments, and a `SignalUsage` is associated with at least one equation `Argument`.

Equation. `Equations` (Fig. 4.3) define relations between signals, considered as its `Arguments`. But `Signals` and `Arguments` do not have a direct relation; `SignalUsages` play this intermediate role. An `Equation` has an `EquationRightPart` and at most one `EquationLeftPart`. The latter has `Arguments` as `Equation` outputs.

`EquationRightPart` is either an `ArrayAssignment` or a node `Invocation`. `ArrayAssignment` has `Arguments`, and indicates that the `Equation` is an array assignment. `Invocation` is a call to another `Node`, where `FunctionIdentifier` indicates the called function. An `Invocation` may have an ordered list of `Arguments`, which are used as the inputs of the function. `Equations` can be assembled in an `EquationSystem`.

Node. Functionalities are modeled as `Nodes` (see Fig. 4.4). Such a `Node` has an `Interface`, a `LocalDeclaration`, some `NodeVariables` and an `EquationSystem`. `NodeVariable` is the container of `Signals` and `SignalUsages`.

Module. All `Nodes` are grouped in a `Module` (Fig. 4.5), which represents the whole system model. A module contains one `Node` as the *main* function of the system. Each `Node` is either completely defined or linked to an external function through IP deployment (see Section 1.2). These IP concepts, such as `CodeFile` and `Implementation` are also grouped in the `Module`.

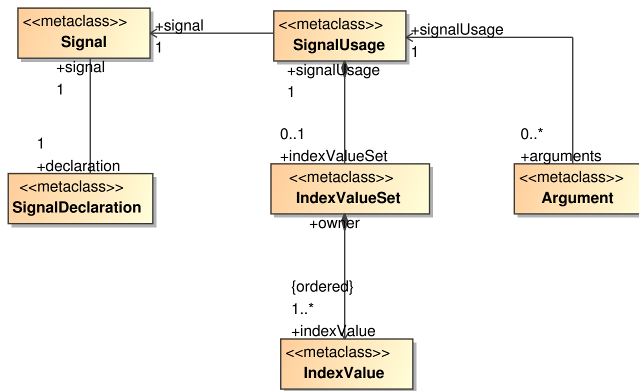


Fig. 4.2. *`Signal` part of our synchronous metamodel.*

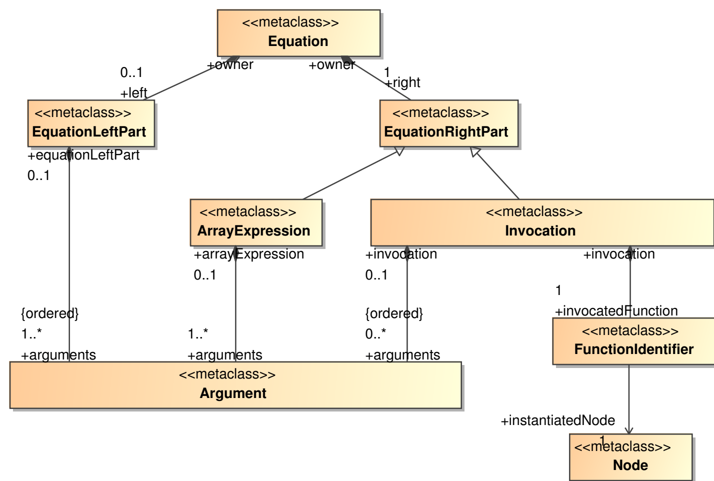


Fig. 4.3. *`Equation` part of our synchronous metamodel.*

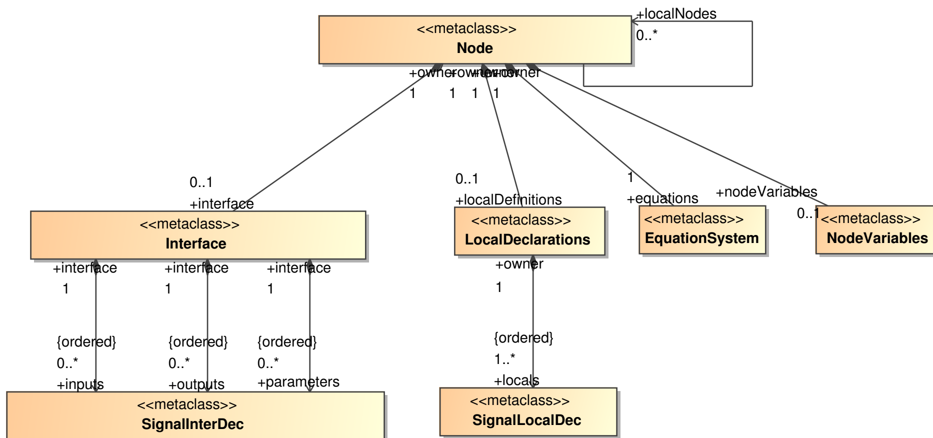


Fig. 4.4. *`Node` part of our synchronous metamodel.*

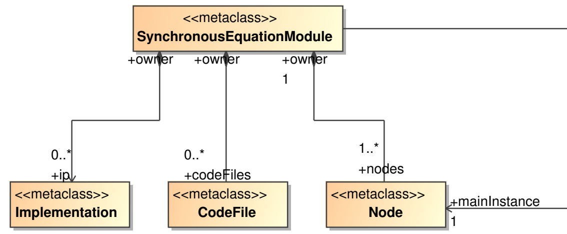


Fig. 4.5. *`Module` part of our synchronous metamodel.*

4.2. Transformations towards synchronous programs. The transformation of Gaspard models into synchronous programs consists of two steps (Fig. 4.6):

1. transformation of models specified with MARTE into synchronous models;
2. code generation from synchronous models obtained from the first step.

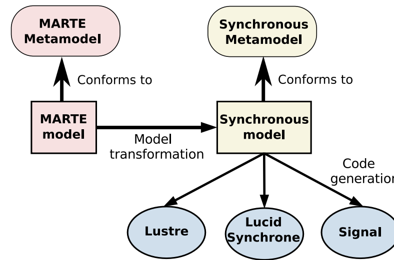


Fig. 4.6. *From MARTE models to synchronous dataflow programs.*

Our transformation rules are represented through a tree structure (Fig. 4.7). There are mainly twenty transformation rules. The root rule is *GModel2SModel*. It transforms a Gaspard model into a synchronous program by recursively calling its sub-rules: *GTiler2SNode* devoted to the transformation of `tiler` connectors, *GApplication2SNode* devoted to the transformation of the different Task notions of Gaspard, etc. A more complete presentation of all rules is given in [23].

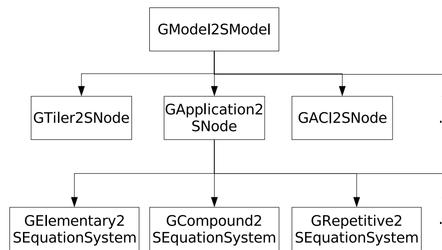


Fig. 4.7. *Hierarchy of the transformation rules*

The above transformation rules as well as the code generation are achieved within the Eclipse Modeling Framework (EMF) [9]. Gaspard models are exported as EMF models, which are then transformed into EMF synchronous equational models, used finally to generate synchronous programs. This transformation chain has been implemented by using some specific technologies.

The MoMoTE tool (MOdel to MOdel Transformation Engine), based on EMF, is a Java framework that enables to perform model to model transformations. It is composed of an API and an engine. It takes input models that conform to some metamodels and produces output models that conform to other metamodels. A transformation by MoMoTE is composed of hierarchical rules. These rules are integrated into an Eclipse plugin that is automatically invoked during model transformations.

The implemented code generation is based on templates. EMF JET [10] and MoCodE (MOdels to CODe Engine) are used to build code generators for each of the target synchronous languages. JET is a template based

code generation tool. These templates are used to generate JAVA implementation classes. MoCodE consists of an API with an engine that performs model to text transformation. It takes a set of models as inputs. Then, its engine recursively takes out elements from input models and executes a corresponding JAVA implementation class on them. These JAVA classes finally generate the target code. The size of our implemented transformation chain is about 5.10^3 lines of Java code.

5. Validation of design properties. We present a typical use of the analysis techniques and tools associated with synchronous languages to analyze the video processing system introduced in Section 1, via a corresponding generated code. The addressed issue specifically concerns clock synchronizability between the system components in order to satisfy some non functional requirements.

5.1. Clock synchronizability analysis with the synchronous technology. Let us denote by c_p, c_a and c_i the respective logical clocks of the sensor, the downscaler and the display. They respectively represent the pixel production rate in the sensor, the bloc computation frequency within the downscaler, and the image production rate on the display. The whole model works as follows: the sensor produces its output data pixel by pixel; the downscaler periodically performs an operation whenever it receives from the sensor a fixed number of pixels; and the TFT screen periodically displays an image whenever it receives from the downscaler a fixed number of blocs of transformed pixels. A step in c_p, c_a and c_i corresponds to the production of respectively a single pixel by the sensor, a transformed block of pixels by the downscaler, and an image by the TFT display. From the point of view of GASPARD, the clock steps associated with a component correspond to its paving iterations. We therefore derive the following constraints between above logical clocks:

- C_1: c_a is an affine undersampling of c_p, i.e., $c_p \overset{(1,\phi_1,d_1)}{\rightarrow} c_a$;
- C_2: c_i is an affine undersampling of c_a, i.e., $c_a \overset{(1,\phi_2,d_2)}{\rightarrow} c_i$;

Now, let us consider a specification requirement of the video display functionality, consisting of a constraint on the actual production rate, noted c_i', of displayed images in the cell phone. This constraint, denoted by C_3, states a relation between the pixel production rate c_p and c_i' as follows: $c_p \overset{(1,\phi_3,d_3)}{\rightarrow} c_i'$. Then, we need to guarantee the compatibility of this new constraint with the previous set of constraints $\{C_1, C_2\}$. I.e., we want to establish a synchronizability relation between clocks c_i' and c_i w.r.t. $\{C_1, C_2, C_3\}$. This will ensure that the expected rate of the TFT display c_i, which depends on the rate of the downscaling process c_a, satisfies the requirement.

This issue cannot be addressed by only using the usual definition of clock synchronization in synchronous languages. Instead, we consider affine clock systems in order to define under which condition c_i' and c_i are synchronizable. So, from C_1, C_2 and C_3, this synchronizability property is checked by using the following property, which has been proved and implemented in the SIGNAL compiler:

$$c_i' \text{ and } c_i \text{ are synchronizable} \Leftrightarrow \begin{cases} \phi_1 + d_1\phi_2 = \phi_3 \\ d_1 d_2 = d_3 \end{cases} \tag{5.1}$$

This issue is solved quite easily with synchronous models while it is not possible with GASPARD only. The result of this analysis can be used to adjust the paving iteration parameters of the downscaler model so as to satisfy the non functional requirements imposed on the whole system.

In [22], Smarandache *et al.* combine the ALPHA language and the synchronous language SIGNAL to design and validate embedded systems by defining affine clocks relations. Our approach differs from this work in that we propose a synchronous model of a whole data-parallel application instead of describing only its clock information as it is the case with [22]. As a result, our model allows us to address both functional and non functional properties of the application using the synchronous technology. Another similar work concerns the design of *n-synchronous Kahn networks* [7] in which authors consider the LUCID SYNCHRONE language in order to address the correct-by-construction development of high performance stream-processing applications. This work also defines clock synchronizability properties that can be applicable to GASPARD models specified in LUCID SYNCHRONE. Note that in both [22] and [7], the analysis relies on clocks, which give a qualitative view of time. This is not the case of [15], which is another interesting work where authors use linear relations to analyze synchronous programs so as to verify quantitative time properties. Such an approach would be very helpful when dealing with time durations.

5.2. Other analyses. In addition to the above clock relation analysis, the synchronous technology also enables to address further model design properties imposed by the GASPARD underlying specification formalism, ARRAY-OL, as shown in [14]: single assignment, functional determinism, absence of cyclic data-dependencies. The compilers of synchronous languages provide us with very efficient data-dependency analysis techniques to address such properties.

Furthermore, in [24], we dealt with both functional and non functional properties of an application case study, consisting of the multimedia processing module of a cellular phone. We applied the SIGALI model-checker (from the synchronous technology) [17] to synchronous programs generated from GASPARD models extended with controlled computations specification [12].

6. Concluding remarks. In this paper, we presented an approach to deal with the reliable design of high-performance embedded systems by combining the repetitive and synchronous models of computation. While the former model suits for the adequate expression of data-parallelism and task parallelism in such systems, the latter model allows one to reason about critical design properties of the systems. These models of computation are respectively implemented in the GASPARD and synchronous design frameworks. We showed how GASPARD models can be translated into synchronous models in order to formally check the correctness of the systems initially designed with the MARTE standard profile in GASPARD. Here, we illustrated a synchronizability analysis on a simple system example by using the synchronous technology. Further design correctness properties can be also straightforwardly checked with the same technology as it has been exposed in [14, 24].

REFERENCES

[1] E. ALLEN, D. CHASE, J. HALLETT, V. LUCHANGCO, J.-W. MAESSN, S. RYU, G. S. JR., AND S. TOBIN-HOCHSTADT, *The Fortress Language Specification Version 1.0 Beta*, tech. report, Sun Microsystems, Inc., Mar. 2007.

[2] A. BENVENISTE, P. CASPI, S. EDWARDS, N. HALBWACHS, P. LE GUERNIC, AND R. DE SIMONE, *The synchronous languages twelve years later*, Proceedings of the IEEE, 91 (2003), pp. 64–83.

[3] P. BOULET, *Formal Semantics of* ARRAY-OL*, a Domain Specific Language for Intensive Multidimensional Signal Processing*, tech. report, INRIA, France, March 2008. available online at `http://hal.inria.fr/inria-00261178/fr`.

[4] C. BRUNETTE, J.-P. TALPIN, A. GAMATIÉ, AND T. GAUTIER, *A metamodel for the design of polychronous systems*, Journal of Logic and Algebraic Programming, (2009).

[5] D. CALLAHAN, B. CHAMBERLAIN, AND H. ZIMA, *The Cascade High Productivity Language*, in 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments, IEEE Computer Society, Apr. 2004, pp. 52–60.

[6] P. CHARLES, C. GROTHOFF, V. SARASWAT, C. DONAWA, A. KIELSTRA, K. EBCIOGLU, C. VON PRAUN, AND V. SARKAR, *X10: an object-oriented approach to non-uniform cluster computing*, in 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications, New York, NY, USA, 2005, ACM Press, pp. 519–538.

[7] A. COHEN, M. DURANTON, C. EISENBEIS, C. PAGETTI, F. PLATEAU, AND M. POUZET, *N-sychronous Kahn networks*, in ACM Symp. on Principles of Programming Languages (PoPL'06), Charleston, South Carolina, USA, January 2006.

[8] A. DEMEURE AND Y. DEL GALLO, *An array approach for signal processing design*, in Sophia-Antipolis conference on Micro-Electronics (SAME'98), SoC Session, France, Oct. 1998.

[9] ECLIPSE, *Eclipse Modeling Framework*. http://www.eclipse.org/emf.

[10] ———, *EMFT JET*. http://www.eclipse.org/emft/projects/jet.

[11] A. GAMATIÉ, S. LE BEUX, E. PIEL, A. ETIEN, R. BEN ATITALLAH, P. MARQUET, AND J.-L. DEKEYSER, *A model driven design framework for high performance embedded systems*, Research Report 6614, INRIA, France, August 2008. `http://hal.inria.fr/inria-00311115/en`.

[12] A. GAMATIÉ, E. RUTTEN, AND H. YU, *A Model for the Mixed-Design of Data-Intensive and Control-Oriented Embedded Systems*, Tech. Report 6589, INRIA, France, July 2008. `http://hal.inria.fr/inria-00293909/en`.

[13] A. GAMATIÉ, É. RUTTEN, H. YU, P. BOULET, AND J.-L. DEKEYSER, *Modeling and formal validation of high-performance embedded systems*, in 7th International Symposium on Parallel and Distributed Computing (ISPDC'2008), Krakow, Poland, IEEE Computer Society, july 2008, pp. 215–222.

[14] A. GAMATIÉ, E. RUTTEN, H. YU, P. BOULET, AND J.-L. DEKEYSER, *Synchronous Modeling and Analysis of Data Intensive Applications*, Eurasip Journal on ES, 2008 (2008).

[15] N. HALBWACHS, Y. PROY, AND P. ROUMANOFF, *Verification of real-time systems using linear relation analysis*, Formal Methods in System Design, 11 (1997), pp. 157–185.

[16] P. LE GUERNIC, J.-P. TALPIN, AND J.-C. LE LANN, *Polychrony for System Design*, Journal for Circuits, Systems and Computers, 12 (2003), pp. 261–304.

[17] H. MARCHAND, P. BOURNAI, M. L. BORGNE, AND P. L. GUERNIC, *Synthesis of discrete-event controllers based on the Signal environment*, Discrete Event Dynamic System: Theory and Applications, 10 (2000), pp. 325–346.

[18] MESSAGE PASSING INTERFACE FORUM, *MPI Documents*. `www.mpi-forum.org/docs/docs.html`.

[19] OBJECT MANAGEMENT GROUP, *A UML profile for MARTE*, 2007. `http://www.omgmarte.org`.

[20] A. SANGIOVANNI-VINCENTELLI, *Quo Vadis, SLD? Reasoning About the Trends and Challenges of System Level Design*, Proceedings of the IEEE, 95 (2007), pp. 467–506.

[21] D. SCHMIDT, *Guest editor's intro.: Model-driven engineering*, Computer, 39 (2006), pp. 25–31.

[22] I. Smarandache, T. Gautier, and P. Le Guernic, *Validation of mixed* Signal-Alpha *real-time systems through affine calculus on clock synchronisation constraints*, in World Congress on Formal Methods (2), 1999, pp. 1364–1383.
[23] H. Yu, A. Gamatié, E. Rutten, and J.-L. Dekeyser, *Embedded Systems Specification and Design Languages, Selected Contributions from FDL'07 Series: Lecture Notes Electrical Engineering, Vol. 10, ISBN: 978-1-4020-8296-2, Villar Eugenio (Ed.)*, Springer Verlag, 2008, ch. 13: Model Transformations from a Data Parallel Formalism towards Synchronous Languages.
[24] H. Yu, A. Gamatié, E. Rutten, and J.-L. Dekeyser, *Safe design of high-performance embedded systems in an mde framework*, Innovations in Systems and Software Engineering, 4 (2008), pp. 215–222.

Edited by: Marek Tudruj, Dana Petcu
Received: February 6th, 2009
Accepted: June 28th, 2009

Scalable Computing: Practice and Experience
Volume 10, Number 2, pp. 163–172. http://www.scpe.org

ISSN 1895-1767
© 2009 SCPE

RELATIONS BETWEEN SEVERAL PARALLEL COMPUTATIONAL MODELS*

STEFAN D. BRUDA† AND YUANQIAO ZHANG†

Abstract. We investigate the relative computational power of parallel models with shared memory. Based on feasibility considerations present in the literature, we split these models into "lightweight" and "heavyweight," and then find that the heavyweight class is strictly more powerful than the lightweight class, as expected. On the other hand, we contradict the long held belief that the heavyweight models (namely, the Combining CRCW PRAM and the BSR) form a hierarchy, showing that they are identical in computational power with each other. We thus introduce the BSR into the family of practically meaningful massively parallel models. We also investigate the power of concurrent-write on models with reconfigurable buses, finding that it does not add computational power over exclusive-write under certain reasonable assumptions. Overall, the Combining CRCW PRAM and the CREW models with directed reconfigurable buses are found to be the simplest of the heavyweight models, which now also include the BSR and all the models with directed reconfigurable buses. These results also have significant implications in the area of real-time computations.

Key words: parallel computation, shared memory parallel models, reconfigurable buses, parallel random access machine, broadcast with selective reduction, reconfigurable multiple bus machine, reconfigurable network, concurrent-read concurrent-write conflict resolution rules, real-time

1. Introduction.

The concurrent-read concurrent-write parallel random access machine (CRCW PRAM) is the most convenient model of parallel computation and so it is used extensively in analyzing parallel solutions to various problems. Lower bounds on the PRAM are in particular very strong. The Priority CRCW PRAM is sometimes considered [19] to be at the upper level of feasible parallel models. The broadcast with selective reduction (BSR) on the other hand is at present the most powerful model of parallel computation, with the Combining CRCW PRAM falling somewhere in between. By logical extension of the Priority CRCW PRAM being at the upper end of the feasibility chain [19], the Combining CRCW PRAM and the BSR should not be considered feasible; however, efficient implementations have been proposed for both [3].

It is widely believed (though to our knowledge not proven) that Priority CRCW PRAM is strictly less powerful that the Combining CRCW PRAM, which is in turn strictly less powerful than the BSR. In all, one can identify two "categories" of models of parallel computation: we thus call the Priority CRCW PRAM and the models below it in terms of computational power *lightweight* models, with the *heavyweight* models represented by the Combining CRCW PRAM and the BSR.

The first purpose of this paper is then to analyze the relation between these two categories in terms of computational power, as well as the relations between the models within the heavyweight class. Although not necessarily explicit, we always have in mind real-time computations, so we are using a strong notion of relationship: we say that model A is (not necessarily strictly) more powerful than model B only if $t(n)$ computational steps of model B using polynomial resources can be simulated in $O(t(n))$ steps of model A using polynomial resources. We often accomplish this by showing how one model is capable of simulating one computational step of the second model in constant time (and the other way around whenever we want to prove an equality).

As expected, we find that the class of heavyweight models is strictly more powerful than the class of lightweight models. Surprisingly, we also find that all the heavyweight models are however equivalent with each other, despite the perceived high computational power (and low feasibility) of the BSR.

We also enhance the usability of models with reconfigurable buses. Such models (namely the reconfigurable multiple bus machine or RMBM for short, and the reconfigurable network or RN) have been studied in the past [5, 21], being recognized as realistic models for VLSI design and other parallel machines [6, 15]. While the more complex variants of the PRAM embrace the combining of values written into memory, such a combination is justly disregarded (in as much as little to no work exists on the matter) on such distributed resources as buses. That such complex combinations are not necessary (and that Collision is sufficient for any computation) is known [7]. We further find that under a reasonable assumption not even Collision is necessary, that concurrent-write does not add anything over exclusive-write.

*This research was supported by the Natural Sciences and Engineering Research Council of Canada. Part of this research was also supported by Bishop's University.

†Department of Computer Science, Bishop's University, 2600 College St, Sherbrooke, Quebec J1M 1Z7, Canada, (stefan@bruda.ca, yqzhang@cs.ubishops.ca)

Our results are rather significant, as we essentially free the analysis of parallel algorithms and problems from a number of restrictions. On shared memory models, whether using the powerful Broadcast instruction of the BSR diminishes the practicality of the analysis (specifically, we are not aware of any real shared memory machine that implements Broadcast, yet as a consequence of our result such an instruction can be used with no consequences in the design of algorithms for such machines) becomes immaterial. On reconfigurable buses, we allow the use of any form of concurrent-write without penalty, given that any concurrent-write can be simulated in constant time by an exclusive-write machine. At the same time, we also offer a strict delimitation between the two classes of heavyweight and lightweight models of parallel computation.

Furthermore, these results become particularly significant in conjunction with previous work on real-time parallel computations [8] and the characterization of models with reconfigurable buses [9]. We show in effect that the Combining CRCW PRAM is as powerful as reconfigurable buses (thus being useful for the design and analysis of VLSI circuits). We also show that the Combining CRCW PRAM and the exclusive-write on directed reconfigurable buses are the simplest models that can solve exactly all the problems solvable under no matter how tight real-time constraints. In effect, we "simplify" the domain of real time, thus strengthening previous results on real-time computations.

We proceed as follows: The next section presents the necessary preliminaries. The shared memory hierarchy is the subject of Section 3, followed by the discussion on exclusive-write being universal on reconfigurable buses, that can be found in Section 4. The global hierarchy of lightweight and heavyweight models (with shared memory or reconfigurable buses) is summarized in Section 5. Real-time considerations are addressed in Section 6. We conclude in Section 7. Results proved elsewhere are henceforth introduced as Propositions, whereas results proved in this work are introduced as Theorems. Intermediate results are all Lemmata.

2. Preliminaries. PARITY_n denote the following problem: given n integer data $x_1, x_2, \ldots, x_n$, compute the function $\text{PARITY}_n(x_1, x_2, \ldots, x_n) = (\sum_{i=1}^{n} x_i) \mod 2$.

2.1. The PRAM and the BSR. Shared memory models of parallel computation are represented by the *parallel random access machine* (or PRAM for short) and by its extension, the *broadcast with selective reduction* (or BSR).

A PRAM [2, 19] consists in a number of processors that share a common random-access memory. The processors execute the instructions of a parallel algorithm synchronously. The shared memory stores intermediate data and results, and also serves as communication medium for the processors. The model is further specified by defining the memory access mode; we thus obtain exclusive-read exclusive-write (EREW), concurrent-read exclusive-write (CREW) and concurrent-read concurrent-write (CRCW) PRAM. While reading concurrently from the shared memory is defined straightforwardly, writing concurrently into the shared memory requires the introduction of a conflict resolution rule (for the case in which two or more processors write into the same memory location). Four such conflict resolution rules are in use: Common (the processors writing simultaneously in the same memory location must write the same value or else the algorithm fails), Collision (multiple processors writing into the same memory location garble the result so that a special "collision" marker ends up written at that location instead of any processor-provided data), Priority (processors are statically numbered and the memory location receives the value written by the lowest numbered processor), and Combining (where a binary, associative reduction operation is performed on all the values sent by all the processors to the same memory location and the result is stored in that memory location).

Given the obviously increased computational power as well as the straightforward implementation of concurrent-read machines, we will not consider exclusive-read variants. For similar reasons, exclusive-write machines will receive a spotty consideration if any.

The BSR model [2, 4] is an extension of the Combining CRCW PRAM. All the read and write operations of the Combining CRCW PRAM can also be performed by the BSR. In addition, all the BSR processor can write simultaneously into all the memory locations (the Broadcast instruction). Every Broadcast instruction consists in three steps: In the *broadcasting step*, all the n participating processors produce a datum d_i and a tag g_i, $1 \leq i \leq n$, destined to all the m memory locations. In the *selection step* each of the m memory locations uses a limit l_j, $1 \leq j \leq m$ and a selection rule $\sim \in \{<, \leq, =, \geq, >, \neq\}$ to test the received data; the datum d_i is selected for the next step if and only if $g_i \sim l_j$. Finally, the *reduction step* combines all the data d_i destined for memory location j, $1 \leq j \leq m$ and selected in the previous step using a binary, associative operator $\mathcal{R}$, and then writes the result into memory location j. The Broadcast instruction is performed simultaneously for all the processors and all the memory locations.

Typically, the reduction operator $\mathcal{R}$ of the BSR as well as the Combining operator of the Combining CRCW PRAM can be any of the following operations: Σ (sum), Π (product), $\bigwedge$ (logical conjunction) $\bigvee$ (logical disjunction), $\bigoplus$ (logical exclusive disjunction), max (maximum), and min (minimum).

We denote by X CRCW PRAM$(p(n), t(n))$ the class of problems of size n that are solvable in time $t(n)$ on a CRCW PRAM that uses $p(n)$ processors and X as collision resolution rule, $X \in \{\text{Common}, \text{Collision}, \text{Priority}, \text{Combining}\}$. Similarly, we denote by BSR$(p(n), t(n))$ the class of problems of size n that are solvable in time $t(n)$ on a BSR that uses $p(n)$ processors.

The notion of uniform families of PRAM or BSR machines exists [18]. However, we do not need such a notion in this paper. Still, we assume as customary that one location in the shared memory has $O(\log n)$ size for an input of size n, and that the number of memory locations are upper bounded by a polynomial in the number of processors used. Again as usual, we assume that every BSR or PRAM processor has a constant number of internal registers, each of size $O(\log n)$. Finally, we assume that every PRAM processor knows its number as well as the total number of processors in the system. The following result regarding the PRAM will be used later:

Proposition 2.1. [12] $\text{PARITY}_n \notin$ Priority CRCW PRAM$(\text{poly}(n), O(1))$.

2.2. The RMBM and the RN. An RMBM [21, 22] consists of a set of p *processors* and b *buses*. For each processor i and bus j there exists a *switch* controlled by processor i. Using these switches, a processor has access to the buses by being able to read or write from/to any bus. A processor may be able to *segment* a bus, obtaining thus two independent, shorter buses, and it is allowed to *fuse* any number of buses together by using a *fuse line* perpendicular to and intersecting all the buses. DRMBM, the *directed* variant of RMBM, is identical to the undirected model, except for the definition of fuse lines: Each processor features two fuse lines (*down* and *up*). Each of these fuse lines can be electrically connected to any bus. Assume that buses i_1, i_2, …, i_k are all connected to the down [up] fuse line of some processor. Then, a signal placed on bus i_j is transmitted in one time unit to all the buses i_l such that $i_l \geq i_j$ [$i_l \leq i_j$]. If some RMBM [DRMBM] is not allowed to segment buses, then this restricted variant is denoted by F-RMBM [F-DRMBM] (for "fusing" RMBM).

For CRCW RMBM, the most realistic conflict resolution rule is Collision, where two values simultaneously written on a bus result in the placement of a special, "collision" value on that bus. We do not need to consider other conflict resolution rules such as Common, Priority, and even Combining, since all of these rules are in fact equivalent to the seemingly less powerful Collision rule (or even exclusive-write, see Theorem 4.2).

An RMBM (DRMBM, F-DRMBM, etc.) *family* $R = (R_n)_{n \geq 1}$ is a set containing one RMBM (DRMBM, etc.) construction with $p(n)$ processors and $b(n)$ buses for each $n > 0$. A family R solves a problem π if R_n solves all inputs of π of size n. We say that some RMBM family R is *uniform* if there exists an NL Turing machine M that, given n, produces the description of R_n using $O(\log(p(n)b(n)))$ work tape cells. We henceforth drop the "uniform" qualifier, with the understanding that any RMBM family described here is uniform. If some family $R = (R_n)$ solves a problem π, and each R_n, $n > 0$, uses $p(n)$ processors, $b(n)$ buses, X as write conflict resolution rule ($X \in \{\text{CREW}, \text{Common CRCW}, \text{Collision CRCW}, \text{Priority CRCW}, \text{Combining CRCW}\}$), and runs in $t(n)$ time, then we say that $\pi \in X$ RMBM$(p(n), b(n), t(n))$ (or $\pi \in X$ F-DRMBM$(p(n), b(n), t(n))$, etc.), and that R has *size complexity* $p(n)b(n)$ and *time complexity* $t(n)$. Whenever we state a property that holds for any conflict resolution rule we drop X (thus writing $\pi \in$ RMBM$(p(n), b(n), t(n))$, etc.).

It should be noted that a directed RMBM can simulate an undirected RMBM by simply keeping all the up and down fuse lines synchronized with each other.

An RN [5, 22] is a network of processors forming a connected graph whose vertices are the processors and whose edges represent fixed connections. Each edge incident to a processor corresponds to a (bidirectional) port of the processor. A processor can partition its ports such that all the ports in the same block of the partition are electrically connected (fused) together. The edges that are thus connected together form a bus. CREW, Common CRCW, Collision CRCW, etc. are defined as for the the RMBM model. The edges are directed in a *directed* RN (DRN). The concept of (uniform) RN family is identical to the RMBM concept. For some write conflict resolution rule X ($X \in \{\text{CREW}, \text{Common CRCW}, \text{Collision CRCW}, \text{Priority CRCW}, \text{Combining CRCW}\}$), X RN$(p(n), t(n))$ [X DRN$(p(n), t(n))$] is the set of problems solvable by RN [DRN] uniform families with $p(n)$ processors ($p(n)$ is also called the *size complexity*) and $t(n)$ running time using the conflict resolution rule X. We drop X when the property refers to any conflict resolution rule.

As in the RMBM case, we note that for any undirected RN there exists a directed RN with the same size complexity and the same running time that simulates it (the directed RN provides a couple of edges of opposite directionality for every edge in the undirected RN).

3. The Shared Memory Hierarchy. Recall that we called Priority CRCW PRAM and all the models of less computational power lightweight, while the Combining CRCW PRAM and the BSR were called heavyweight. We show now that this terminology is introduced for good reason. Specifically, we show that all the heavyweight models have the same computational power, and that they are strictly more powerful than the lightweight models. The first result of this paper is thus stated as follows:

THEOREM 3.1. *For* $X \in \{\text{Collision}, \text{Common}\}$ *and for any* $n \in \mathbb{N}$,

$$\begin{aligned} X \text{ CRCW PRAM}(\text{poly}(n), O(t(n))) &\subseteq \\ \text{Priority CRCW PRAM}(\text{poly}(n), O(t(n))) &\subsetneq \\ \text{Combining CRCW PRAM}(\text{poly}(n), O(t(n))) &= \\ \text{BSR}(\text{poly}(n), O(t(n))). \end{aligned}$$

Proof. Theorem 3.1 is a direct consequence of Lemmata 3.2 to 3.5 below. Specifically, the inclusions shown in the theorem are proven in the mentioned lemmata one by one. ∎

We complete all the proofs below by showing how the model on the right hand side of the inclusion simulates in constant time one computational step of the model on the left hand side. Once this is shown, the inclusion that needs to be proved becomes immediate.

We note that some of the hierarchy analyzed here has also been investigated earlier [14]. In particular, Lemma 3.2 has been established in a stronger version [14] (namely, that the two models are equivalent). For completeness however we include here our proof of this result.

LEMMA 3.2.

$$\forall n \in \mathbb{N} : \text{Collision CRCW PRAM}(\text{poly}(n), O(t(n))) \subseteq \text{Priority CRCWPRAM}(\text{poly}(n), O(t(n)))$$

Proof. A Collision CRCW PRAM with k processors p_i, $1 \le i \le k$ and m memory locations u_j, $1 \le j \le m$ is readily simulated by a Priority CRCW PRAM with $2k+m$ processors denoted by $p_i^{\downarrow}$ $(1 \le i \le k)$, $p_i^{\uparrow}$ $(1 \le i \le k)$, and p_j^m $(1 \le j \le m)$. (Note however that the processor group p_j^m and the processor group $p_i^{\downarrow}$ plus $p_i^{\uparrow}$ take turns in the simulation, so the actual number of processors required is $\max(2k, m)$; however, the explicit differentiation eases the presentation.)

In addition to the original memory locations u_j, we use two more "banks" of the same size $u_j^{\downarrow}$ and $u_j^{\uparrow}$, $1 \le j \le m$. A Collision CRCW PRAM step (read, compute, write) is then simulated as follows:

1. For every $1 \le i \le k$, both the processors $p_i^{\downarrow}$ and $p_{k+1-i}^{\uparrow}$ perform the same read, compute, and write cycle as the original p_i, with the following addition: Whenever processor $p_i^{\downarrow}$ writes into memory location u_j, it also writes its number j into memory location $u_j^{\downarrow}$; similarly, whenever processor $p_{k+1-i}^{\uparrow}$ writes into memory location u_j, it also writes its number $k+1-i$ into memory location $u_j^{\uparrow}$.
2. Every processor p_i^m writes the collision value into memory location u_j if and only if $u_j^{\downarrow} \ne u_j^{\uparrow}$.

That the above simulation takes constant time is immediate. Note further that after Step 1 of the simulation the location $u_j^{\downarrow}$ $[u_j^{\uparrow}]$ contains the index of the lowest [highest] ranked processor that modified the memory location u_j (indeed, we operate on a Priority CRCW model, so only the lowest numbered processor succeeds in writing into a given memory location; we then chose the processors numbers in appropriate manner for the desired property to happen). Then, whenever $u_j^{\downarrow} \ne u_j^{\uparrow}$, more than one processor wrote into the given memory location. A collision occurred, so Step 2 places a collision marker accordingly. The simulation is complete. ∎

LEMMA 3.3.

$$\forall n \in \mathbb{N} : \text{Common CRCW PRAM}(\text{poly}(n), O(t(n))) \subseteq \text{Priority CRCWPRAM}(\text{poly}(n), O(t(n)))$$

Proof. We simulate now a computational step of a Common CRCW PRAM with k processors and m memory locations in constant time using a Priority CRCW PRAM. The simulation will use the same number k of processors (we denote them by p_i, $1 \le i \le k$) and the same memory space (denoted by u_j, $\le j \le m$). The simulation proceeds as follows:

1. All the processors p_i carry on the computational step prescribed by the Common CRCW PRAM algorithm, including the operation of writing into the shared memory (recall however that we are now using the Priority conflict resolution rule).
2. Each processor p_i that wrote a value v_i into memory location u_j in Step 1 also remembers v_i by storing it into an otherwise unused internal register ρ.
3. Every p_i that wrote a value into location u_j in Step 1 read the content of u_j and compares it with the content of its register ρ.
   (a) If the contents of u_j and ρ are the same then either (a) p_i is the sole processor which wrote into u_j, (b) p_i is the highest priority processor which wrote into u_j, or (c) p_i and the highest priority processor agree on the value written into u_j. None of these cases violate the Common resolution rule, so p_i does not do anything.
   (b) If on the other hand u_j and ρ contain different values, then the value written into u_j by p_i disagrees with the value written in the same location by some other processor, which in turn violates the Common resolution rule. So p_i aborts the algorithm and reports the failure.

Note that in effect we chose *one* of the processors writing concurrently into a memory location as representative for all the others (given that we have a Priority machine at our disposal, that representative turned out to be the processor with the highest priority; however the way we chose a representative is immaterial). Every processor which wants to write a value in some memory location compares now its value with the value already written by its representative; if the value is different, then the Common conflict resolution rule is violated; otherwise all is good and the overall algorithm continues with the next step.

The above proof uses the usual definition of Common, as presented in Section 2. Still, we note that sometimes this definition is termed "Fail-safe Common," case in which the "Fail Common" or "Tolerant" variant is also defined [2, 13]. In such a variant, any computational step that violates the Common resolution rule is discarded completely (that is, for all the processors in the system) and the algorithm continues with the next step (instead of aborting the computation). A proof for this modified variant of Common is readily possible. Indeed, all we need is "backup" copies of the memory locations used by the algorithm, plus one memory location used to signal any violation to everybody. The processors now use the backup memory to perform all the simulation described above, except that they set the violation flag instead of aborting the algorithm whenever a violation of the Common resolution rule occurs (since is is just a flag, using Priority as conflict resolution rule will do just as well as almost any other rule). At the end of the computational step, all the processors inspect the flag and write again the values they wanted to write in the first place (this time in the main memory, not the backup) only if the flag is not set; otherwise they all proceed to the next step immediately, thus leaving the main working memory unchanged. □

Lemma 3.4.

$$\forall n \in \mathbb{N} : \text{Priority CRCW PRAM}(\text{poly}(n), O(t(n))) \subsetneq \text{CombiningCRCW PRAM}(\text{poly}(n), O(t(n)))$$

Proof. A Priority CRCW PRAM with k processors p_i, $1 \le i \le k$ and m memory locations u_j, $1 \le j \le m$ is readily simulated by a Combining CRCW PRAM with the same number of processors (denoted by p'_i) and $2m$ memory locations (denoted by u_i and u'_i):

1. Each processor p'_i performs the same read and compute operations as p_i. Instead of writing (to memory location u_j), p'_i however performs a "dry run" by writing its number into memory location u'_j using a Combining CRCW write with min as combining operation.
2. Each processor p'_i performs now the real write operation: It writes into the memory location u_j in which it wanted to write to begin with, but does so only if its number matches the value stored in u'_j. The min as combining operation performed over the locations u'_j in the previous step ensures that a matching occurs only for the lowest numbered processor, as desired.

That Combining CRCW PRAM$(\text{poly}(n), O(t(n))) \neq$ Priority CRCW PRAM$(\text{poly}(n), O(t(n)))$ is an immediate extension of Proposition 2.1. Indeed, by Proposition 2.1 PARITY$_n$ is not solvable in constant time on a Priority CRCW PRAM with polynomial number of processors. On the other hand, PARITY$_n$ is trivially solvable in constant time using a Combining CRCW PRAM with n processors: Every processor p_i of such a machine, $1 \le i \le n$ holds one input datum x_i and writes it into some designated memory location μ by performing a Combining CRCW using Σ as combining operation; then p_1 performs a modulo operation on μ and thus μ contains the output of PARITY$_n$ for the given instance, as desired. □

LEMMA 3.5. $\forall n \in \mathbb{N}$: Combining CRCW PRAM$(\text{poly}(n), t(n)) = \text{BSR}(\text{poly}(n), O(t(n)))$.

Proof. That Combining CRCW PRAM$(\text{poly}(n), t(n)) \subseteq \text{BSR}(\text{poly}(n), O(t(n)))$ is immediate from the definition of the BSR. Surprisingly enough, the reverse inclusion is also true. We show now this reverse inclusion. Specifically, we show how one BSR computational step is simulated by a Combining CRCW PRAM in constant time.

Consider a BSR with k processors and m memory locations. Every BSR processor p_i is simulated by a set of m PRAM processors r_{ij}, $1 \le j \le m$. The PRAM memory is doubled, every BSR memory location u_j will be simulated by two PRAM memory locations u_j^d and u_j^l, $1 \le j \le m$. Finally, the PRAM uses extra processors p_j^u, $1 \le j \le m$ (once more processors p_j^u and r_{ij} actually take turns in the simulation so we differentiate them for convenience only; the actual number of processors used is in fact smaller as these two groups can overlap with each other). The PRAM simulation of a BSR Broadcast step (read, compute, Broadcast) then proceeds as follows:

- *Read and Compute:* For $1 \le i \le n$, all the processors r_{ij}, $1 \le j \le m$ perform the reading and the computation prescribed for p_i. Every time some processor wants to read the value of u_j it will read the value of u_j^d instead. Processors r_{ij} will then *all* hold the values of the datum d_i and the tag g_i originally computed by p_i.
  Note that there are n groups of processors; all the m processors in the same group perform the same computation. This may appear wasteful but is intentional and will be used to simulate the Broadcast instruction.
- *Selection limits:* Every processor p_j^u computes the limit l_j associated with u_j in the selection phase of the BSR step, and stores it in the memory location u_j^l.
- *Broadcast instruction:* r_{ij} will be responsible for the data written by the BSR processor p_i into memory location r_j:
  1. r_{ij} reads l_j from memory location u_j^l so that it holds d_i, g_i, and l_j;
  2. r_{ij} then computes the selection criterion $g_i \sim l_j$ as prescribed by the BSR algorithm;
  3. r_{ij} writes d_j into memory location u_j^d if and only if $g_i \sim l_j =$ True, using a Combining CRCW write with the combining operator prescribed by the BSR algorithm.

  Note that for some fixed i all the processors r_{ij}, $1 \le j \le m$ contain identical data, so p_i's replacement covers all the memory locations, thus realizing the desired broadcast.

In effect, we use one PRAM processor for every pair processor–memory location in the BSR algorithm. This allows for an immediate simulation of the broadcast phase of a Broadcast instruction: Instead of broadcasting, every PRAM processor is now responsible for writing to one memory location only; but then since we have as many processors as memory locations, we nonetheless write to all the memory locations at once, as desired. The correctness of the rest of the simulation is immediate, and so is the overall constant running time. ∎

4. On the Power of Concurrent-Write on Reconfigurable Buses. It has been shown [7] that Collision is universal on reconfigurable buses, meaning that all the other conflict resolution rules can be simulated by Collision with constant overhead. Under certain assumptions, this result can be strengthened. Indeed, one can conceivably identify two variants of concurrent-write on reconfigurable buses:

DEFINITION 4.1. *Concurrent-write (or CW for short) on reconfigurable buses is defined as follows:*

Strong CW *Any two signals arriving simultaneously at the same bus are considered concurrent-write.*

Weak CW *Two signals from two different processors arriving simultaneously at the same bus are considered concurrent-write. However, a signal that is split and arrives two times at some bus is not considered concurrent-write.*

Strong CW is implied in the earlier work that established the universality of Collision [7]. Weak CW, however appears just as realistic, for indeed a bunch of fused buses form an electrical, longer bus; then it makes no sense to consider a signal that travels on two different paths, for the signal is simply placed on the bus and propagates along it according to the physical laws of electricity.

As it turns out, the definition of CW makes a significant difference in terms of universality of Collision: under weak CW, Collision is unnecessary on reconfigurable buses; instead, exclusive-write is all we need. This is all put together as follows:

THEOREM 4.2.

1. *Under strong CW,* Collision *is universal on reconfigurable buses (directed or undirected), meaning that* Collision *can simulate all the other conflict resolution rules with constant time overhead and with polynomial resources.*

2. *Under weak CW, exclusive-write is universal on reconfigurable buses (directed or undirected), meaning that exclusive-write can simulate all the forms of concurrent-write with constant time overhead and with polynomial resources.*

Proof. The strong CW case has been established for the directed case earlier [7, 8]. The undirected case is easily derived from the proofs supporting the directed case (sketched below): one only need to replace the graph accessibility problem with its undirected variant and nondeterministic logarithmic space with deterministic logarithmic space as done (in different contexts) earlier [5, 6].

The weak CW case has been established for undirected buses elsewhere [6]. For the directed case, we can easily modify the strong CW proof [7] as follows:

The simulation of all the conflict resolution rules is accomplished via a nondeterministic logarithmic space-bounded Turing machine. This machine computes first the content of all the buses, unfused. Successive computations of the graph accessibility problem (which is complete for nondeterministic logarithmic space [20]) then determine which buses are fused. Once the fused buses are determined, the final content of the fused buses is computed. This Turing machine is in turn simulated by a DRMBM which constructs the graph of possible configurations of the machine and then solves the graph accessibility problem to determine the outcome of the computation of the machine. The construction of the graph does not involve any concurrent-write. Overall, the only computation involving concurrent-write is the computation of the graph accessibility problem; all the other computations do not involve concurrent-write at all.

The graph accessibility problem is solved by a DRMBM that works as follows [8]: Finding whether vertex t is accessible from vertex s in a directed graph with n vertices given as an incidence matrix I can be computed in constant time by a DRMBM with $(n^2-n)/2$ processors and n buses. Denote each processor by p_{ij}, $1 \le i < j \le n$, and let p_{ij} know the value of both I_{ij} and I_{ji}. Then, each p_{ij}, $1 \le i < j \le n$ directionally fuses buses i and j if and only if $I_{ij} = True$; simultaneously, p_{ij} fuses buses j and i if and only if $I_{ji} = True$. It is easily proven that, for any s, t, $1 \le s, t \le n$, a signal placed on bus s reaches bus t if and only if vertex t is accessible from vertex s; the content of the emitted signal is immaterial. Indeed, note that the electrical connections between buses actually form the original graph. Bus t will then receive the signal placed on bus s if and only if there exists a path that connects the two vertices. If there are multiple such paths, then the signal will arrive on bus t multiple times; however, there is only one signal in the whole system, so under weak CW there is simply no possibility for a collision to happen! Under the weak CW model the graph accessibility problem is thus computed by an exclusive-write machine. As already mentioned, none of the other computations that simulate all the conflict resolution rules on a DRMBM involve any concurrent-write, so we conclude that exclusive-write is universal and so we complete the proof for DRMBM.

The simulation of a DRN by a Turing machine is similar to the simulation of a DRMBM, except for some supplementary issues related to initial data distribution. These issues however do not change the fact that concurrent-write occurs only in the computation of the graph accessibility problem, which proceeds in the same manner as in the DRMBM case [7]. ☐

5. Relations Between Several Parallel Computational Models. The results established in Section 3 can be combined with earlier results that establish the BSR as being as powerful as the models with directed reconfigurable buses [9]. They can be further combined with the universality of Collision or exclusive-write, as discussed in Section 4. We can then put everything together as follows:

COROLLARY 5.1. *For any $n \in \mathbb{N}$ and for any $X \in \{\text{Collision}, \text{Common}\}$,*

$$\begin{gathered} X \text{ CRCW PRAM}(\text{poly}(n), O(t(n))) \subseteq \text{Priority CRCW PRAM}(\text{poly}(n), O(t(n))) \subsetneq \\ \text{Combining CRCW PRAM}(\text{poly}(n), O(t(n))) = \text{BSR}(\text{poly}(n), O(t(n))) = \\ \text{DRMBM}(\text{poly}(n), \text{poly}(n), O(t(n))) = \text{DRN}(\text{poly}(n), O(t(n))) = \\ Y \text{ DRMBM}(\text{poly}(n), \text{poly}(n), O(t(n))) = Y \text{ DRN}(\text{poly}(n), O(t(n))) \end{gathered}$$

where $Y =$ CREW under the weak CW assumption and $Y =$ Combining CRCW under the strong CW assumption.

Proof. The first three relations are established in Theorem 3.1. The next two relations are established elsewhere [9]. The last two relations are given by Theorem 4.2. ☐

6. Real-Time Implications. The class of problems in NL (problems solvable in nondeterministic logarithmic space) with the addition of (any kind of) real-time constraints is denoted by NL/rt [8]. Let rt-PROC$^M(f)$

denote the class of those problems solvable in real time by the parallel model of computation M that uses $f(n)$ processors (and $f(n)$ buses if applicable) for any input of size n. The following strongly supported conjecture is then established.

PROPOSITION 6.1. [8] rt-PROC$^{\text{DRMBM}}(\text{poly}(n)) = \mathsf{NL}/rt$.

We can further define [9] the following classes of models of parallel computation:

$\mathbb{M}_{<GAP}$ contains exactly all the models that cannot compute the graph accessibility problem (GAP) in constant time, and cannot compute in constant time any problem outside NL.

$\mathbb{M}_{\equiv GAP}$ contains exactly all the models that can compute GAP in constant time, but cannot compute in constant time any problem outside NL.

$\mathbb{M}_{>GAP}$ contains exactly all the models that can compute GAP in constant time and can compute in constant time at least one problem not in NL; to our knowledge, no model has been shown (or is even believed) to belong to this class.

Proposition 6.1 is then extended as follows:

PROPOSITION 6.2. [9] *For any models of computation M_1, M_2, and M_3 such that $M_1 \in \mathbb{M}_{<GAP}$, $M_2 \in \mathbb{M}_{\equiv GAP}$, and $M_3 \in \mathbb{M}_{<GAP}$, it holds that*

$$\begin{aligned}\text{rt-PROC}^{M_1}(\text{poly}(n)) &\subseteq \mathsf{NL}/rt\\ \text{rt-PROC}^{M_2}(\text{poly}(n)) &= \mathsf{NL}/rt\\ \text{rt-PROC}^{M_3}(\text{poly}(n)) &\supset \mathsf{NL}/rt\end{aligned}$$

The class $\mathbb{M}_{\equiv GAP}$ is established by Proposition 6.2 as the class of complete models for real time. Indeed, exactly all the models in this class can solve all the real-time problems in the presence of no matter how tight timing constraints. This class has been previously populated with the BSR, the DRMBM, and the DRN.

Now Theorem 3.1 adds to $\mathbb{M}_{\equiv GAP}$ the Combining CRCW PRAM. In effect, we are simplifying the real-time domain, showing that the power of reconfiguration as well as the power of the Broadcast instructions are not only not needed to solve real-time problems, but they are also unnecessary. The PRAM thus becomes the most convenient tool for characterizing computations from a real-time perspective, being the simplest of the complete models for real time.

The real-time domain is also simplified with respect to reconfigurable buses, at least under the weak CW model. Indeed, Theorem 4.2 adds the CREW variant of directed reconfigurable buses to $\mathbb{M}_{\equiv GAP}$. Therefore, for directed reconfigurable buses running in real time concurrent-write is not necessary, as exclusive-write is universal.

Our previous characterization of real-time computations (Proposition 6.1) is strengthened by Proposition 6.2 only to the degree that the class $\mathbb{M}_{\equiv GAP}$ is populated; in this paper we therefore further strengthened this result (by adding simpler models to this class).

7. Conclusions. It has been widely believed that the all-powerful Combining conflict resolution rule does add computational power to the PRAM model, and that the BSR's Broadcast instruction adds further power. We are to our knowledge the first to establish formally a hierarchy of the PRAM variants that both confirms and contradicts the mentioned belief. Indeed, we showed that Combining does add computational power over "lesser" rules. However, we also showed that surprisingly enough the Broadcast instruction does not add computational power over Combining. In fact we established an intriguing collapse of the hierarchy of parallel models at the top of the food chain, where the Combining CRCW PRAM and the BSR turn out to have identical computational power.

This result offers substantial and unexpected aid to the analysis of real-life shared memory massively parallel machines. Indeed, the power of BSR's Broadcast instruction has attracted attention in various areas of parallel algorithms; algorithms with running time as fast as constant have been developed for various problems, most notably in the areas of geometric and graph computations [16, 17]. Of course, constant time algorithms for such practically meaningful problems are very attractive, and so is the power of the BSR. Nevertheless, the model tends to be frowned upon given its apparent implementation complexity, even if very efficient (optimal) implementations have been proposed [3]. We now showed that whether the BSR is feasible or not is irrelevant, as a Combining PRAM with the same performance and with all the attractive properties of the BSR is automatically available—in essence, one can freely choose between these two models, depending on no matter what

issues (ranging from practical feasibility to convenience to mere taste) with the formally supported knowledge that the results are portable to the other model.

We are not aware of any massively parallel machine that implements the BSR's Broadcast instruction, but many such machines implement the PRAM model [1, 10, 11, 23]. Our result shows that such an instruction—and thus the full power of the BSR model—can be used in algorithm design and analysis while keeping the analysis or the design practically pertinent.

We also note that most of our proofs are constructive, so we also set the basis for automatic conversion back and forth between models. True, we did not have efficiency in mind so our constructions are likely not to be optimal; historically however suboptimal algorithms have always been optimized sooner or later, so we believe that our—however sub-optimal—implicit conversion algorithms are nonetheless an additional significant contribution (beside establishing the actual equivalence).

This all being said, we note that simulating the Broadcast instruction on a Combining CRCW PRAM implies a (polynomial) blow-up in the number of processors used. This does not change complexity-theoretical results, but does have practical implications. Unfortunately, we believe that such a tight simulation (with constant time slowdown) of the Broadcast instruction is not possible without a blow-up in the number of processors.

As a second significant result we found that exclusive-write can simulate all the forms of concurrent-write on reconfigurable buses under the reasonable weak CW assumption. Concurrent-write has always been regarded as problematic on distributed resources such as buses. We also believe that weak CW is a realistic assumption, given the physical (electrical) realization of reconfigurable buses. VLSI design uses reconfigurable buses extensively. The universality of Collision or exclusive-write (depending on whether we choose the strong or weak CW rule) is significant for VLSI design, as Collision and especially exclusive-write are easily implemented in silicon. More work is necessary for the refinement of the process (of converting a general machine into an exclusive-write machine) before they become useful in practice, but the most important step (or showing that they are possible) is done here.

Beside the significance of our results by themselves (of bringing the BSR into the area of feasible models of parallel computation for massively parallel systems and of showing that exclusive-write is universal on reconfigurable buses), these results become particularly significant in conjunction with previous work on real-time parallel computations [8] and the rest of the characterization of models with reconfigurable buses [9].

Indeed, we showed previously [9] that models with reconfigurable buses have the same computational power as the BSR. Theorem 3.1 then extends this equivalency to the Combining CRCW PRAM. One consequence is that the Combining CRCW PRAM turns out to be just as powerful as reconfiguration. Reconfiguration is however one of the realistic models for VLSI design [15, 22]. We then expect that bringing the Combining CRCW PRAM to the table has the potential of inspiring new design and analysis techniques for VLSI algorithms. That is, our results are not significant only to the area of massively parallel systems, but also to the area of VLSI circuits.

As a consequence of Theorem 3.1, the Combining CRCW PRAM also joins the family of models that can solve all the problems known to be solvable under no matter how tight real-time constraints, and so does the exclusive-write models with directed reconfigurable buses. This strengthens out previous results on real-time computations by establishing the PRAM and the exclusive-write DRMBM or DRN as the simplest complete models of computation for real time.

REFERENCES

[1] F. Abolhassan, R. Drefenstedt, J. Keller, W. J. Paul, and D. Scbeerer, *On the physical design of PRAMs*, Computer, 36 (1993), pp. 756–762.

[2] S. G. Akl, *Parallel Computation: Models and Methods*, Prentice-Hall, Upper Saddle River, NJ, 1997.

[3] S. G. Akl and L. Fava Lindon, *An optimal implementation of broadcasting with selective reduction*, IEEE Transactions on Parallel and Distributed Systems, 4 (1993), pp. 256–269.

[4] S. G. Akl and G. R. Guenther, *Broadcasting with selective reduction*, in Proceedings of the IFIP 11th World Congress, G. X. Ritter, ed., San Francisco, CA, 1989, North-Holland, Amsterdam, pp. 515–520.

[5] Y. Ben-Asher, K.-J. Lange, D. Peleg, and A. Schuster, *The complexity of reconfiguring network models*, Information and Computation, 121 (1995), pp. 41–58.

[6] Y. Ben-Asher, D. Peleg, and A. Schuster, *The power of reconfiguration*, Journal of Parallel and Distributed Computing, 13 (1991), pp. 139–153.

[7] S. D. Bruda, *The graph accessibility problem and the universality of the Collision CRCW conflict resolution rule*, WSEAS Transactions on Computers, 10 (2006), pp. 2380–2387.

[8] S. D. Bruda and S. G. Akl, *Size matters: Logarithmic space is real time*, International Journal of Computers and Applications, 29 (2007), pp. 327–336.

[9] S. D. Bruda and Y. Zhang, *Why shared memory matters to VLSI design: The BSR is as powerful as reconfiguration*, in Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium, Miami, FL, Apr. 2008.

[10] M. Forsell, *A scalable high-performance computing solution for network on chips*, IEEE Micro, 22 (2002), pp. 46–55.

[11] ———, *On the performance and cost of some PRAM models on CMP hardware*, in Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium, Miami, FL, Apr. 2008.

[12] M. Furst, J. B. Saxe, and M. Sipser, *Parity, circuits, and the polynomial-time hierarchy*, Mathematical Systems Theory, 17 (1984), pp. 13–27.

[13] T. Hagerup and T. Radzik, *Every robust CRCW PRAM can efficiently simulate a PRIORITY PRAM*, in Proceedings of the Second Annual ACM Symposium on Parallel Algorithms and Architectures, ACM, 1990, pp. 117–124.

[14] J. JaJa, *An Introduction to Parallel Algorithms*, Addison Wesley, 1992.

[15] R. Miller, V. L. Prasanna-Kumar, D. I. Reisis, and Q. F. Stout, *Parallel computations on reconfigurable meshes*, IEEE Transactions on Computers, 42 (1993), pp. 678–692.

[16] J.-F. Myoupo and D. Seme, *Work-efficient BSR-based parallel algorithms for some fundamental problems in graph theory*, The Journal of Supercomputing, 38 (2006), pp. 83–107.

[17] J.-F. Myoupo, D. Seme, and I. Stojmenovic, *Optimal BSR solutions to several convex polygon problems*, The Journal of Supercomputing, 21 (2002), pp. 77–90.

[18] I. Parberry, *Parallel Complexity Theory*, John Wiley & Sons, New York, NY, 1987.

[19] L. Stockmeyer and U. Vishkin, *Simulation of parallel access machines by circuits*, SIAM Journal on Computing, 13 (1984), pp. 409–422.

[20] A. Szepietowski, *Turing Machines with Sublogarithmic Space*, Springer Lecture Notes in Computer Science 843, 1994.

[21] J. L. Trahan, R. Vaidyanathan, and R. K. Thiruchelvan, *On the power of segmenting and fusing buses*, Journal of Parallel and Distributed Computing, 34 (1996), pp. 82–94.

[22] R. Vaidyanathan and J. L. Trahan, *Dynamic Reconfiguration: Architectures and Algorithms*, Springer, 2004.

[23] X. Wen and U. Vishkin, *PRAM-on-chip: first commitment to silicon*, in Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures, ACM, 2007, pp. 301–302.

Edited by: Marek Tudruj, Dana Petcu
Received: February 7th, 2009
Accepted: June 28th, 2009

Scalable Computing: Practice and Experience
Volume 10, Number 2, pp. 173–187. http://www.scpe.org
ISSN 1895-1767
© 2009 SCPE

EXPERIENCES WITH MESH-LIKE COMPUTATIONS USING PREDICTION BINARY TREES*

GENNARO CORDASCO[†], BIAGIO COSENZA[†], ROSARIO DE CHIARA[†], UGO ERRA[‡] AND VITTORIO SCARANO[†]

Abstract. In this paper we aim at exploiting the temporal coherence among successive phases of a computation, in order to implement a load-balancing technique in mesh-like computations to be mapped on a cluster of processors. A key concept, on which the load balancing schema is built on, is the use of a Predictor component that is in charge of providing an estimation of the unbalancing between successive phases. By using this information, our method partitions the computation in balanced tasks through the Prediction Binary Tree (PBT). At each new phase, current PBT is updated by using previous phase computing time for each task as next-phase's cost estimate. The PBT is designed so that it balances the load across the tasks as well as reduces *dependency* among processors for higher performances. Reducing dependency is obtained by using rectangular tiles of the mesh, of almost-square shape (i. e. one dimension is at most twice the other). By reducing dependency, one can reduce inter-processors communication or exploit local dependencies among tasks (such as data locality). Furthermore, we also provide two heuristics which take advantage of data-locality. Our strategy has been assessed on a significant problem, Parallel Ray Tracing. Our implementation shows a good scalability, and improves performance in both cheaper commodity cluster and high performance clusters with low latency networks. We report different measurements showing that tasks granularity is a key point for the performances of our decomposition/mapping strategy.

Key words: scheduling, load balancing, performance prediction, mesh-like computation, performance evaluation.

1. Introduction.

1.1. Parallel Computing. The evolution of computer science in the last two decades has been characterized by the architectural shift that has brought centralized computation paradigm toward distributed architectures where data processing and data storing are cooperatively performed on several nodes, interconnected by a network.

The problem of scheduling a parallel program to a set of homogeneous processors for minimizing the completion time (that is the time when the last processor completes its job) of the program has been extensively studied (see [11] for a comprehensive presentation). Indeed, dividing a computation (henceforth, *decomposition*) into smaller computations (*tasks*) and assigning them to different processors for parallel executions (named *mapping*), represent two key steps in the design of parallel algorithms [12]. The whole computation is usually represented via a *directed acyclic graph* (DAG) $\mathcal{G} = \{\mathcal{N}, \mathcal{E}\}$ which consists of a set of nodes $\mathcal{N}$ representing the tasks and a set of edges $\mathcal{E}$ representing interactions and/or dependencies among tasks.

The number and, consequently, the size of tasks into which a given computation is decomposed determines the *granularity* of the decomposition. It may appear that the time required to solve a problem can be easily reduced, by simply increasing the granularity of decomposition, in order to perform more and more tasks in parallel, but this is not always true. Typically, interactions between tasks, and/or other important factors, limit our choice to coarse-grained granularity. Indeed, when the tasks being computed are mutually independent, the granularity of the decomposition does not affect the performances. However, dependencies among tasks incur inevitable communication overhead when tasks are assigned to different processors. Moreover, the finer is the adopted granularity by the system, the more is the generated inter-tasks communication. The interaction between tasks is a direct consequence of the fact that exchanging information (e.g. input, output, or intermediate data) is usually needed.

A good mapping strategy should strive to achieve two conflicting goals: (1) balance the overall load distribution, and (2) minimize tasks inter-processors *dependency*; by mapping tasks with a high degree of mutual dependency onto the same processor. As an example of dependency, many mapping strategies exploits tasks' locality to reduce *inter-processors communications* [16] but it should be emphasized that dependency can also refer to other issues such as locality of access to memory (effective usage of caching).

The mapping problem becomes quite intricate if one has to consider that: (1) task sizes are not uniform, that is, the amount of time required by each task may vary significantly; (2) task sizes are not known a

*A portion of this paper was presented at ISPDC 2008 [8].

[†]ISISLab, Dipartimento di Informatica ed Applicazioni "R.M. Capocelli", Università degli Studi di Salerno, Salerno (Italy), {cosenza, cordasco, dechiara, vitsca}@dia.unisa.it.
[‡]Dipartimento di Matematica e Informatica, Università degli Studi della Basilicata, Potenza (Italy), ugo.erra@unibas.it.

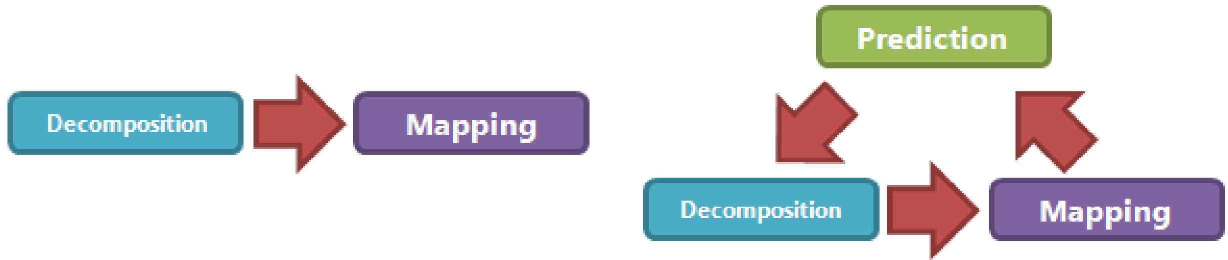


Fig. 1.1. *Interaction between components of a Parallel Scheduler; arrows indicate how components influence each others: (left) Traditional approach; (right) Our system with the Predictor.*

priori; (3) different mapping strategies may provide different overheads (such as scheduling and data-movement overhead). Indeed, even when task sizes are known, in general, the problem of obtaining an optimal mapping is an NP-complete problem for non-uniform tasks (to wit, it can be reduced to the 0-1 Knapsack problem [7]).

Mesh-like computations. In this paper, we focus our study on mesh-like computations, where a set of t independent tasks are represented as items in a bidimensional mesh. Edges among items in this mesh represent tasks dependencies. In particular, we are interested in *tiled* mapping strategies where the whole mesh is partitioned into m *tiles* (i. e., contiguous 2-dimensional blocks of items). Tiles have almost-square shape, that is, one dimension is at most twice the other: in this way, assuming the load in processors is balanced (in terms of nodes), the dependencies inter-processors are minimized because of isoperimetric inequality in the Manhattan grid.

Tiled mappings are particularly suitable to exploit the local dependencies among tasks, be it the *locality of interaction*, i. e., when computation of an item requires other nearby items in the mesh or when there is a *spatial coherence*, i. e., when computation of neighbors item access to some common data. Hence, tiled mapping, in the former case, reduces the interaction overhead, and, in the latter case, improves the reuse of recently data access (cache).

We are interested in decomposition/mapping strategy for step-wise mesh-like computations, i. e. data is computed in successive phases. We assume that each task size is roughly similar among consecutive phases, that is, the amount of time required by item p in phase f is comparable to the amount of time required by p in phase $f + 1$ (*temporal coherence*).

1.2. Our result. In this paper we present a decomposition/mapping strategy for parallel mesh-like computations that exploits the temporal coherence, among computation phases, to perform load balancing on tasks. Our goal is to use temporal coherence to estimate the computing time of a new computation phase using previous phase computing time.

We introduce an iterative novel approach to implement decomposition/mapping scheduling. Our solution (see Figure 1.1) introduces a new component in the system design, dubbed *Predictor* that is in charge of providing an estimation of the computation time needed by a given tile, at each "phase".

The key idea is that, by using the Predictor, it is possible to obtain a balanced decomposition without using a fine-grained granularity that may increase the inter-tasks communication of the systems, due to the interaction between clients, and, therefore, may harm the performances of the whole computation.

Our strategy performs a semi-static load balancing (decisions are made before each computing phase). Temporal coherence is exploited using a *Prediction Binary Tree* where each leaf represents a tile which will be assigned to a worker as a task. At the beginning of every new phase, the mapping strategy, taking into account the previous phase times as estimates, evaluates the chance of updating the binary tree. Due to the temporal coherence property it provides a roughly balanced mapping. We also provide two heuristics which exploit the PBT in order to leverage on data locality.

We validate our strategy by using interactive rendering with Parallel Ray Tracing [24] algorithm, as a significant example of such a kind of computations. In this example our technique is applied rather naturally. Indeed, interactive Ray Tracing can be seen as a step-wise computation, where each frame to be rendered represents a phase. Moreover, each frame can be described as a mesh of items (pixels) and successive computations are typically characterized by temporal coherence.

For Parallel Ray Tracing, our technique experimentally exhibits good performances improvements, with different granularity (size of tiles), with respect to the static assignment of tiles (tasks) to processors. Furthermore we also provided an extensive set of experiments in order to evaluate: (1) the optimal granularity with different number of processors; (2) the scalability of our proposed system; (3) the correctness of the predictions exploiting temporal coherence; (4) the effectiveness of the locality coherence heuristics exploiting spatial coherence; (5) the impact of resolution; (6) the overhead induced by the PBT.

It should be said that, besides other graphical applications (e.g. image dithering), there are further examples of mesh-like computations where our techniques can be fruitfully used, covering simple cases, such as matrix multiplication, but also more complex computations, such as *Distributed Adaptive Grid Hierarchies* [17].

1.3. Previous Works. Decomposition/Mapping scheduling algorithms can be divided into two main approaches: list scheduling and cluster-based scheduling. In list scheduling [18], each task is first assigned a priority by considering the position of the task within the computation DAG $\mathcal{G}$. Then tasks are sorted on priority and scheduled following this order on a set of available processors. Although this algorithm has a low complexity, the quality of scheduling is generally worse than that of algorithms in other classes. In cluster-based scheduling, processors are treated as clusters and the completion time is minimized by moving tasks among clusters [4, 12, 26]. At the end of clustering, heavily communicating tasks are assigned to the same processor. While this reduces inter-processor communication, it may lead to load imbalances or idle time slots [12].

In [15], a greedy strategy is proposed for the dynamic remapping of step-wise data parallel applications, such as fluid dynamics problems, on a homogeneous architecture. In these types of problems, multiple processors work independently on different regions of the data domain during each step. Between iterations, remapping is used for balancing the workload across the processors and thus, reducing the execution time. Unfortunately, this approach does not take care of locality of interaction and/or spatial coherence.

Several online approaches have also been proposed. An example is the work stealing model [3]. In this model when a processor completes its task it attempts to steal tasks assigned to other processors. We notice that, although online strategies are shown to be effective [3] and stable [1], they introduce communication overhead anyway. Furthermore, it is worth noting that online strategies, like work stealing, can be integrated with our assignment policy. In that case, being our load balancing efficient, online strategies introduce smaller overheads.

Many researchers have explored the use of time-balancing models to predict execution time in heterogeneous and dynamic environments. In these environments, performance processors are both irregular and time-varying because of uneven underlying load on the various resources. In [25] authors use a conservative load prediction in order to predict the resource capacity over the future time interval. They use expected mean and variance of future resource capabilities in order to define an appropriate data mappings for dynamic resources.

2. Our Strategy. Our strategy is based on a traditional *data parallel model*. In this model, tasks are mapped onto processors and each task performs similar operations on different pieces of data (*Principal Data Items* (PDIs)). Auxiliary global information (*Additional Data Items* (ADIs)) is replicated on all the workers. This parallelization approach is particularly suited to the *Master-workers paradigm*.

In this paradigm, the master divides the whole job (the whole mesh) into a set of tasks, usually represented by *tiles*. Then, each task is sent to a worker which elaborates the tile and sends back the output. If other tiles are not yet computed, the master sends another task to the worker. Finally, the master obtains the results of the whole computation reassembling partial outputs.

Crucial point in this paradigm is the granularity of the mesh decomposition: in fact, the relationship between m, number of tiles, and n, number of workers, strongly influences the performances.

There are two opposite driving forces that act upon this design choice. The first one is concerned about the *load balancing* and requires m to be larger than n. In fact, if a tile corresponds to a zone of the mesh which requires a large amount of computation, then, it requires much more time with respect to a simpler tile. Then, a simple strategy to obtain a fair load balancing is to increase the number of tiles, so that the complexity of a zone of the mesh is shared among different items.

On the opposite side, two following considerations suggest a smaller m. An algorithm that has large m requires larger *communication costs* than an algorithm with smaller m, considering both the latency (more messages are required; therefore, messages may be queued up) and the bandwidth (communication overhead for each message). Other considerations that would suggest to use small m are (a) the *locality of interaction* and (b) the *spatial coherence* that are motivated because (i) computation of a task relies usually on nearby tasks

and (ii) two close tasks usually access some common data. Then, in order to make an effective usage of the local cache for each node, it is important that the tiles are large enough, so that each worker can exploit spatial coherence of tiles, having a good degree of (local) cache hits.

Our strategy takes into account all the considerations above, by addressing the uneven spread of the load by using a Predictor component (the PBT), with a negligible overhead. The goal we aim to is to keep the load balanced without resorting to increase the number of tiles. Thereby, our solution does not increase significantly the data-movement overhead and it reduces tasks interactions. Therefore, we are able to address simultaneously and positively all the issues above, by providing a technique that uses a moderate amount of tiles.

2.1. The Prediction Binary Tree. In this section we present how we use the Prediction Binary Tree (PBT) to help balancing the load among the computing items. The PBT is in charge of directing the tiling-based load balancing strategy as follows: each computing phase is split into a set of m tiles (we assume, here, for sake of simplicity that $m = n$ but the arguments apply to general cases) and tiles size is adjusted accordingly to (an estimated) tile computing time that is set as the computational time as measured during the preceding phase. The hypothesis is that the computing time required by a tile on two consecutive phases are quite similar because of temporal coherence.

Now we define the Prediction Binary Trees and then describe an on-line algorithm which, before each computing phase, resizes unbalanced tiles in such a way to minimize the mesh computing time. A PBT T stores the current tiling being defined as a rooted binary tree with exactly m leaves, in which each (internal) node has 2 children. The root of T, called r, represents the complete mesh. The two children of an internal node v store the two halves (more details follow on how the mesh is split) of the mesh represented by v. Consequently, each level of T represents a partition of the mesh. Moreover, each internal node v represents a tile which is the sum of the tile assigned to the leaves of the tree rooted in v and consequently, the leaves of T (henceforth $L(T)$) represents a partition of the mesh. In order to maintain a good spatial coherence and minimize tasks interaction, the children of an internal node v which belongs to an odd (resp. even) level of T are obtained halving the tile in t along two horizontal (resp. vertical) axes. This assures that tiles have an almost-square shape (i. e. one dimension is at most twice the other). Each leaf $ell \in L(T)$ also stores two variables: $e(\ell)$ that is the estimate of the time for computing tile in ℓ and $t(\ell)$ that is time used by a worker to compute (in the last phase) the tile in ℓ. Figure 2.1 gives an example of a PBT, with the corresponding mesh partition on the left.

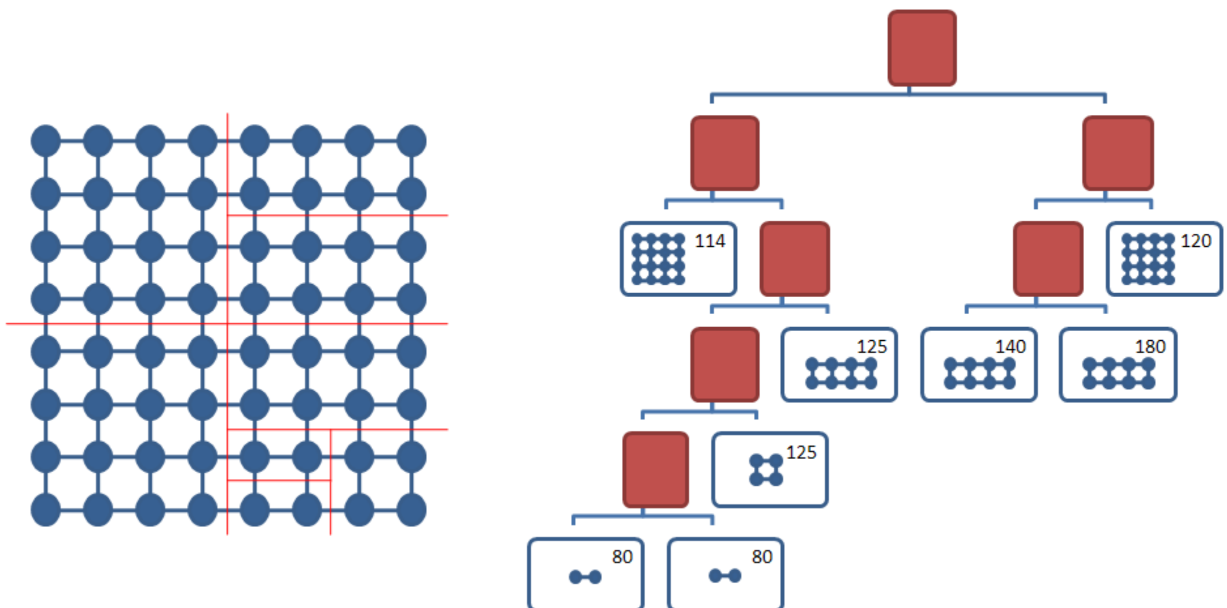


FIG. 2.1. *An example of a PBT tree: the mesh on the left has been computed with the computation times (in ms) for each tile shown on the leaves.*

The PBT stores the subdivision of tiles and each leaf of T is a task to be assigned to a worker. At the end of each phase, the PBT receives (with the tile output) also the information about the time that each worker has spent on the tile. This time is received as $t(\ell)$ for each leaf, and is used as estimate by copying it into $e(\ell)$. By using the previous phase times as estimates, the PBT is efficiently updated for the next phase. Here we describe an effective and efficient way of changing the PBT structure so that the next phase can be executed (given the temporal coherence) more efficiently, i. e., equally balancing the load among the processors.

First we define the variance as a metric to measure the (estimated) computational unbalance that is expected given the tiling provided by the PBT T:

$$\sigma_T^2 = \frac{1}{m} \sum_{\ell \in L(T)} (e(\ell) - \mu_T)^2,$$

where $e(\ell)$ represents the estimated time to compute the corresponding tile to the leaf ℓ of T and μ_T is the estimated average computational time, that is, $\mu_T = \frac{1}{m}\sum_{\ell \in L(T)} e(\ell)$. Clearly, the smaller the variance σ_T^2 is, the better is T's balancing of the load to the processors.

Given a PBT T at the end of a phase, the estimated computation time associated to each leaf, $e(\ell)$, is taken by the computation time $t(\ell)$ at the phase just executed; then, we use a greedy algorithm that finds the new PBT T^*. The idea of the algorithm PBT-Update (shown as Algorithm 1) is to perform a sequence of simultaneous *split-merge* operations, that consists in splitting a tile whose estimated load is "high", and merge two tiles (stored at sibling nodes) whose (combined) estimated load is "small".

We now prove, by means of the following theorem, that the PBT-Update algorithm terminates.

THEOREM 2.1. *Algorithm PBT-Update terminates after a finite number of iterations.*

Proof. We will show that PBT-Update goes from a PBT $T = T^{(0)}$ to a PBT $T^{(s)} = T^*$ through a set of PBTs $T^{(1)}, T^{(2)}, \ldots, T^{(s-1)}$ in such a way that $\sigma^2_{T^{(i)}} > \sigma^2_{T^{(i+1)}}$ for each $i = 0, \ldots, s-1$.

Let T be a PBT tree and T' be obtained from T by halving a leaf ℓ_a into two leaves ℓ_{a_1} and ℓ_{a_2} and merging two sibling leaves ℓ_{b_1} and ℓ_{b_2} into ℓ_b.

We prove first that, if $e(\ell_a)^2 > 4e(\ell_{b_1})e(\ell_{b_2})$ then $\sigma_T^2 > \sigma_{T'}^2$. So, it is not possible to improve the variance of the (estimation of the) computational time by means of a simultaneous split-merge operation if $e(\ell_a)^2 \leq 4e(\ell_{b_1})e(\ell_{b_2})$ which is the test in line 1 of the algorithm.

Algorithm 1 PBT-Update

1: $T \leftarrow CurrentPBT$
2: **for all** $\ell \in L(T)$ **do**
3: copy computational time $t(\ell)$ in estimated time $e(\ell)$
4: **end for**
5: **while** true **do**
6: let ℓ_a be the leaf in T with max $e(\ell)$, $\forall \ell \in L(T)$
7: let ℓ_{b_1}, ℓ_{b_2} be the two siblings such that $e(\ell_{b_1}) \cdot e(\ell_{b_2})$ is minimized over all the pairs of siblings in $L(T)$
8: **if** $e(\ell_a)^2 \leq 4\, e(\ell_{b_1}) \cdot e(\ell_{b_2})$ **then**
9: **return** T
10: **else**
11: Split ℓ_a in ℓ_{a_1} and ℓ_{a_2} // Now ℓ_a is internal
12: $e(\ell_{a_1}) \leftarrow e(\ell_a)/2$
13: $e(\ell_{a_2}) \leftarrow e(\ell_a)/2$
14: Merge ℓ_{b_1} and ℓ_{b_2} into ℓ_b // Now ℓ_b is a leaf
15: $e(\ell_b) \leftarrow e(\ell_{b_1}) + e(\ell_{b_2})$
16: **end if**
17: **end while**

Let us evaluate the difference between the variance on T and the variance on T'.

$$\sigma_T^2 = \frac{1}{m} \sum_{\ell \in L(T)} (e(\ell) - \mu_T)^2 = \frac{1}{m} \left(\sum_{\ell \in L(T)} e(\ell)^2 - \frac{1}{m} \left(\sum_{\ell \in L(T)} e(\ell) \right)^2 \right).$$

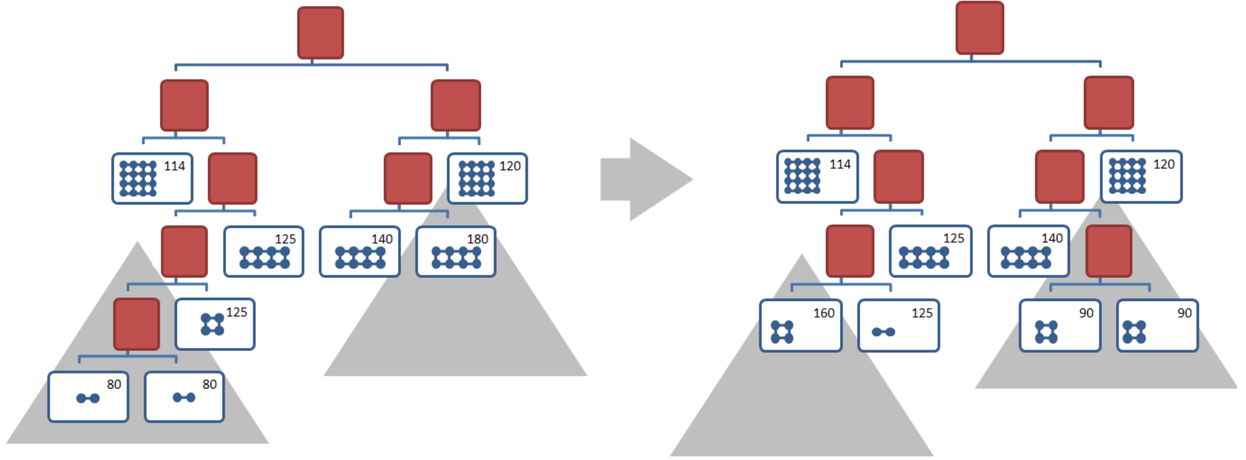


FIG. 2.2. *A merge and split operation on the PBT tree of Figure 2.1 where the estimation times $e(\ell)$ drive the updates.*

Hence, we have

$$\sigma_T^2 - \sigma_{T'}^2 = \frac{1}{m}\left(\sum_{\ell \in L(T)} e(\ell)^2 - \frac{1}{m}\left(\sum_{\ell \in L(T)} e(\ell)\right)^2\right) - \frac{1}{m}\left(\sum_{\ell \in L(T')} e(\ell)^2 - \frac{1}{m}\left(\sum_{\ell \in L(T')} e(\ell)\right)^2\right).$$

Since, by the operations executed in lines 1-1 and 1 of the algorithm, it holds that $\sum_{\ell \in L(T)} e(\ell) = \sum_{\ell \in L(T')} e(\ell)$, then, we have that:

$$\sigma_T^2 - \sigma_{T'}^2 = \frac{1}{m}\left(\frac{e(\ell_a)^2}{2} - 2e(\ell_{b_1})e(\ell_{b_2})\right).$$

Then, if $e(\ell_a)^2 > 4e(\ell_{b_1})e(\ell_{b_2})$, a split-merge operation can improve the variance of the times on the tree. The result follows by the observation that the variance is positive, by definition. ☐

Finally, it should be noticed that the improvement on the variance is proportional to $e(\ell_a)^2 - 4e(\ell_{b_1})e(\ell_{b_2})$. Then at each step, the greedy algorithm PBT-Update chooses ℓ_a and the siblings pair ℓ_{b_1} and ℓ_{b_2} (in lines 1-1) in order to have the higher (local) improvement in variance.

An example of a PBT is shown in Figure 2.1, and one of the updates of the PBT-Update algorithm is shown in Figure 2.2.

Exploiting Local Coherence. In this paragraph we investigate how to leverage the PBT in order to better exploit data locality: the rationale behind this investigation is that jobs carried out by two siblings workers in the PBT, or by the same worker in consecutive computing phases, will probably follow similar memory access patterns.

In order to exploit locality we define the concept of *affine tiles*; in particular we will consider two kinds of affinity: processor-tile affinity and tile-tile affinity. A tile is affine to a processor if it has been assigned to that processor in the previous phase; two tiles are affine if they are neighbors in the mesh. When a worker asks for a tile the master node tries to assign it an affine tile. Then, the intent is to use affine tiles in order to exploit data-locality.

We implemented two heuristics in order to determinate affine tiles. The first heuristic is based on a greedy strategy, dubbed *PBT-Greedy*. For each computing phase, if a tile is not involved in a merge/split operation then it maintains his affinity with the processor it has been assigned to during previous phase. In case of tiles

involved in a merge/split operation, let's consider a tile a split in tiles a_1 and a_2 and tiles b_1 and b_2 merged into tile b: a_1 is assigned to processor that handled a before; b is assigned to processor that handled b_2 before; a_2 is assigned to processor that handled b_1. Just this last assignment, on a total of 3 assignments, will, probably, not exploit cache and for this reason we say that this heuristic is 2/3 effective in leveraging locality.

In the second heuristic, named *PBT-Visit* the affinity is defined by visiting the PBT: tiles are assigned following the in order visit. One can easily check that a subset of affine tiles, tiles "near" in the mesh, is assigned to the same processor phase-by-phase.

3. Case study: Parallel Ray Tracing. Ray Tracing algorithms [19, 20] are a widely used class of techniques for rendering images with the intent of achieving a high grade of realism. Ray Tracing is at the core of many global illumination algorithms. The input for Ray Tracing is a scene description that specifies the geometry of objects together with the definition of every object materials, position/orientation of the lights. The output is an image of the scene as seen through a virtual camera.

For sake of clarity we will shortly summarize the simplest form of Ray Tracing algorithm. For each pixel (x, y) in the final image a ray is casted from the virtual camera through the scene: this is called the *primary ray*. If an intersection between the primary ray and a surface is found then different parameters are considered in order to compute the light intensity at the point: the material of the surface, the position and the color of lights, whether or not the point is shadowed by other surfaces. In the Whitted-style Ray Tracing [24] a ray can be reflected and/or refracted according to surface properties and the process is repeated recursively with these new rays. At the end, the process adds the light intensities at all intersection points in order to get the final color of the pixel. Ray Tracing is considered a computationally intensive algorithm because it depends on the amount of rays shot throughout the scene and this amount can be easily increased by modifying lights properties, objects positions and materials.

Since its introduction several techniques have been explored to accelerate Ray Tracing. In animated scenes we report an interesting observation about the fact that a new frame can be very similar to the previous frame if the viewpoint did not change drastically. This similarity is an instance of the concept of *temporal coherence* (cft. Section 1) and can be exploited to reduce the amount of calculations needed for every new frame [6].

Let p be the pixel of generic coordinates (x, y) in frame f_i and let p' be the pixel with the same coordinates (x, y) (i. e. the same pixel) in frame f_{i+1}. Let r be the ray through p and r' the ray through pixel p'. The idea of the temporal coherence is based upon a simple consideration: the ray r and the ray r' will follow similar paths across the scene.

Parallel Ray Tracing. The Ray Tracing algorithm has been defined "embarrassingly parallel" [10] because no particular effort is needed to segment an instance of the problem in tasks considering that there is no strict dependency between parallel tasks. Each task can be computed independently from every other task in order to achieve a speed up by executing them in parallel. There are two different approaches in designing a parallel ray tracer: object-based and screen-based [5]. In objects-based approach the scene is distributed among clients. For each ray casted the clients forward rays between clients. In the screen-based approach the scene is replicated on each client and the rendering of pixels is assigned to different clients. The second approach is the one investigated in this paper by a frame to frame load partitioning schema.

Speeding up parallel Ray Tracing for interactive use on multi-processor machine has received a big impulse during last years, thanks to an efficient implementation designed to fit the capabilities of modern CPUs [2] and the use of commodity PC clusters [22]. In particular, several techniques are employed to amortize communication costs and manage load balancing. In [22] is suggested a task prefetching and work stealing, whereas [9] is presented a distributed load balancer.

3.1. Exploiting PBT for Parallel Ray Tracing. In order to exploit the PBT to accelerate Parallel Ray Tracing (PRT) algorithm we provide here a mapping between the concepts of the general case, introduced in previous sections, and the concepts strictly bounded to the Ray Tracing. The computation carried out in the PRT is the rendering of a sequence of frames. Every frame is rendered pixel by pixel; in terms of PBT acceleration each of these pixels is an item of the mesh. The memory buffer where each frame of the sequence is rendered can be considered a bidimensional mesh that represents an image: the PDIs managed by nodes are portions of this frame. The information that is available on every worker (ADI) and used to perform the assigned task is the scene description. The number of primitives, usually triangles, the dimension of the textures

to be mapped on the geometry and the number of light sources are elements that increase the computational complexity of a scene to be rendered.

The master divides the frame buffer in tiles, which are rectangular areas of pixels. These tiles are assigned to workers to be rendered. Since two rays will follow similar path if they are close, in order to make an effective usage of the local cache for each node, it is important that the tiles are contiguous and large enough, so that each worker can exploit spatial coherence of tiles, having a good degree of (local) cache hits. Another task performed by the master is to handle the frame buffer for both visualization or to save it into a file.

The granularity of our decomposition strategy is chosen defining $m = k \cdot n$ where m is the number of tiles, n is the number of workers and k is multiplicative constant. The greater is k, the smaller is tiles sizes. We experimentally tested several values of k and found that no large k is needed since, after small values of k, performances degrades due to the higher communication cost (cf. Section 4.2).

4. Experiments and Results. Our serial implementation of Ray Tracing algorithm exploits some, but not all, optimizations techniques used by last cutting-edge ray tracers. Actually, the kind of serial implementation that is used is not relevant for our purposes. Special attention is paid to the acceleration structure. We use a Kd-tree built to minimize the number of traversal and intersection steps, done using the well-know Surface Area Heuristic [23]. Our Kd-tree implementation also provides a fast Kd-tree traversal by using a cache friendly data-layout [21].

We implemented a synchronous render system, with a synchronization barrier at the end of each frame for visualization and camera update purpose. Furthermore, we adopt a demand driven task management strategy where a task manager maintains a pool of already constituted tasks. On receipt of a request from a worker, the task manager dispatches the next available task from the pool (for $k > 1$). We also added a threshold to the number of single merge/split updates into the PBT-Update algorithm in order to avoid to perform many small changes to the tree that would not affect much the overall performances.

We coded our system in C++, compiling it with Intel C++ Compiler 10.1 for Linux. We used MPI [13] for node communications, having care to disable Nagle algorithm [14] in order to decrease latency.

4.1. Setting of the experiments. Because the aim of this work is to exploit temporal coherence in load balancing, we decided to use a distributed memory system, as a cluster of workstations, and test scenes with remarkable unbalance between tiles. We ran several tests on three hardware platforms:

Hydra: an IBM BladeCenter Cluster of 33 nodes (1 master node, 32 worker nodes). Each node has an Intel Pentium IV processor running at 3.20 GHz, with 1 GB of main memory and CentOS 5 Linux as operating system with OpenMPI version 1.1.1 for message passing. All the nodes are interconnected with a Gigabit Ethernet network.

Cacau: a NEC Xeon EM64T Cluster available at HLRS High Performance Computing Center at the Universität Stuttgart. We used up to 64 nodes, each equipped with 2 Intel Xeon EM64T processors and 1 GB of main memory, interconnected with an Infiniband network.

ENEA: an IBM HS21 Cluster with 256 nodes available at CRESCO Project computing platform, Portici ENEA Center. Each node is equipped with 2 Xeon Quad-Core Clovertown E5345 at 2.33 GHz and 16 GByte RAM. The nodes are interconnected with an Infiniband network.

We tested our scheme on two scenes, each of them with different shading aspects. Since the focus is on manage unbalancing, we used a modified standard ERW6 test scene (see Figure 4.1) (about one thousand primitives in total). Unbalancing is due to the surface shading properties used in the scene. We developed two versions of this test scene: ERW6, has one point light source; ERW6-4 has four light sources. The more light sources are present in the scene the bigger is unbalancing because of the increased number of rays to be shot. In both test scenes, we have a predefined walk-through of the camera around the scene, with movements in all directions and rotations too. The image resolution is 512×512 pixels, unless differently stated.

4.2. Results. We performed several test in order to evaluate and estimate: the effectiveness of the PBT, its scalability, the optimal tiles granularity for different number of processors, the impact of temporal and spatial coherence, how the resolution affects the scheduling technique, the overhead incurred by the calculations for the PBT on the total computing time.

In some tests, we compared the technique that uses PBT with a *regular subdivision* technique of equal-sized tiles.

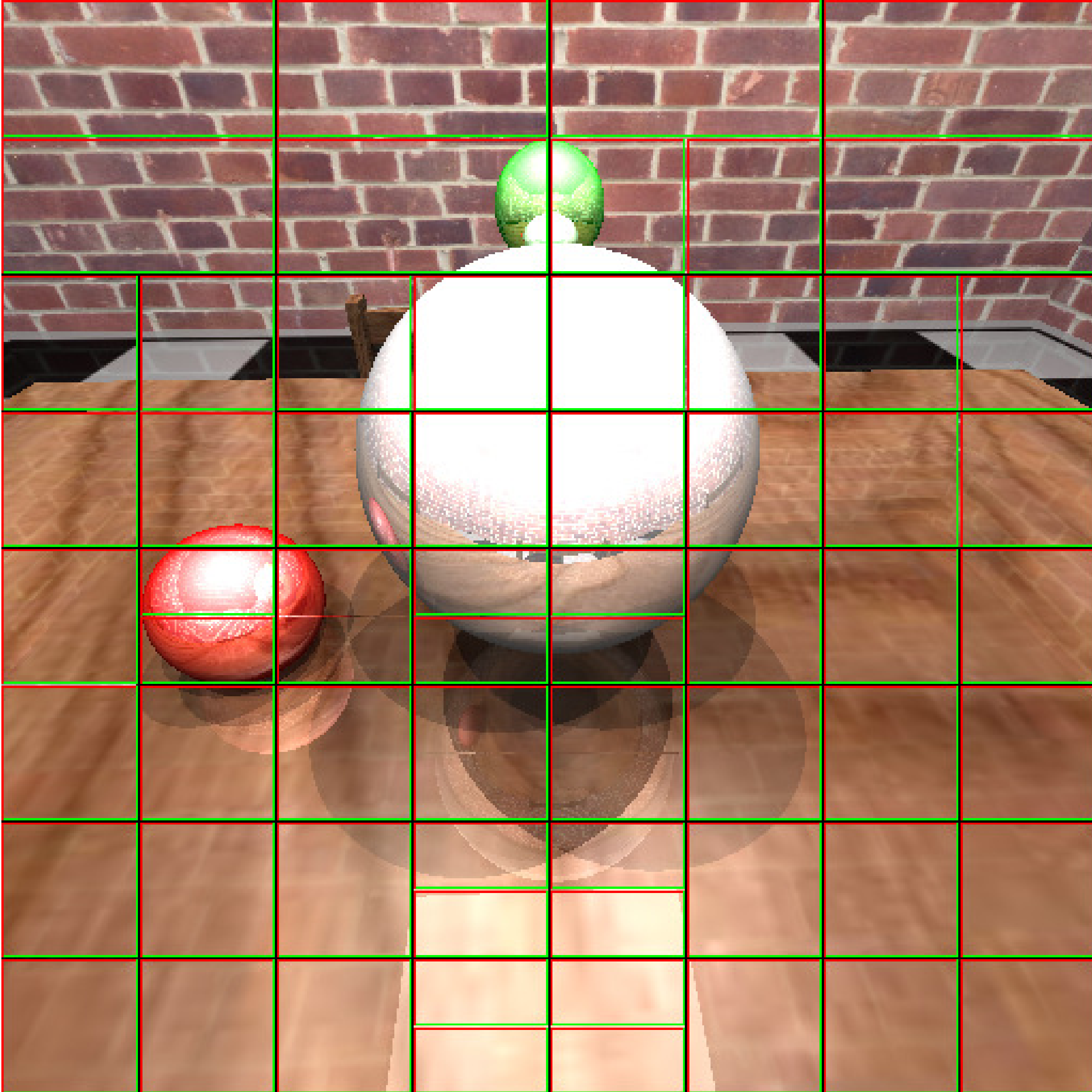


FIG. 4.1. *A frame in the walk-trough for scene ERW6-4, with the tiling shown.*

Effectiveness of Prediction Binary Tree. In these tests, ran on Hydra, we evaluate the improvement provided by using the PBT instead of a regular subdivision strategy.

The results are shown in Figure 4.2 for both scenes. Results are obtained using different granularity and $n = 32$ workers. Our technique offers a speedup, ranging from 5 to 15 percent, for all the values of k tested. When k is large, the performances degrade due to two factors: the number of updates on the PBT increases. Our test shows that there are few updates for smaller values of k, but they grow quickly as k increases. The second is related to the heuristic that we have chosen. Indeed, measuring the rendering time for tiny tile has some approximation problems due to discretization. Our algorithm gives good performances for small values of m. For big values of m, the decomposition algorithm may be a bottleneck.

Optimal Granularity. We tried to estimate the optimal granularity of the schema, i. e., the optimal choice of the number of tiles m, with 8, 16, 32 and 64 processors (see Figure 4.3). The rationale behind this test is to determinate how our schema affects the tuning of the granularity in the parallel implementation, compared with the regular subdivision schema.

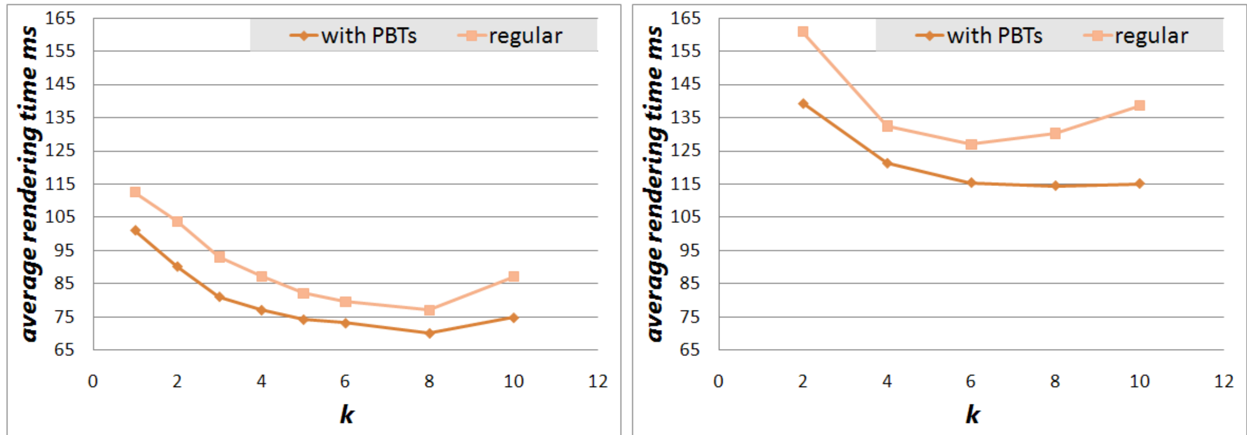


FIG. 4.2. *Average per frame rendering time on increasing k, comparing regular and PBT-based job assignments. (Left: ERW6. Right: ERW6-4)*

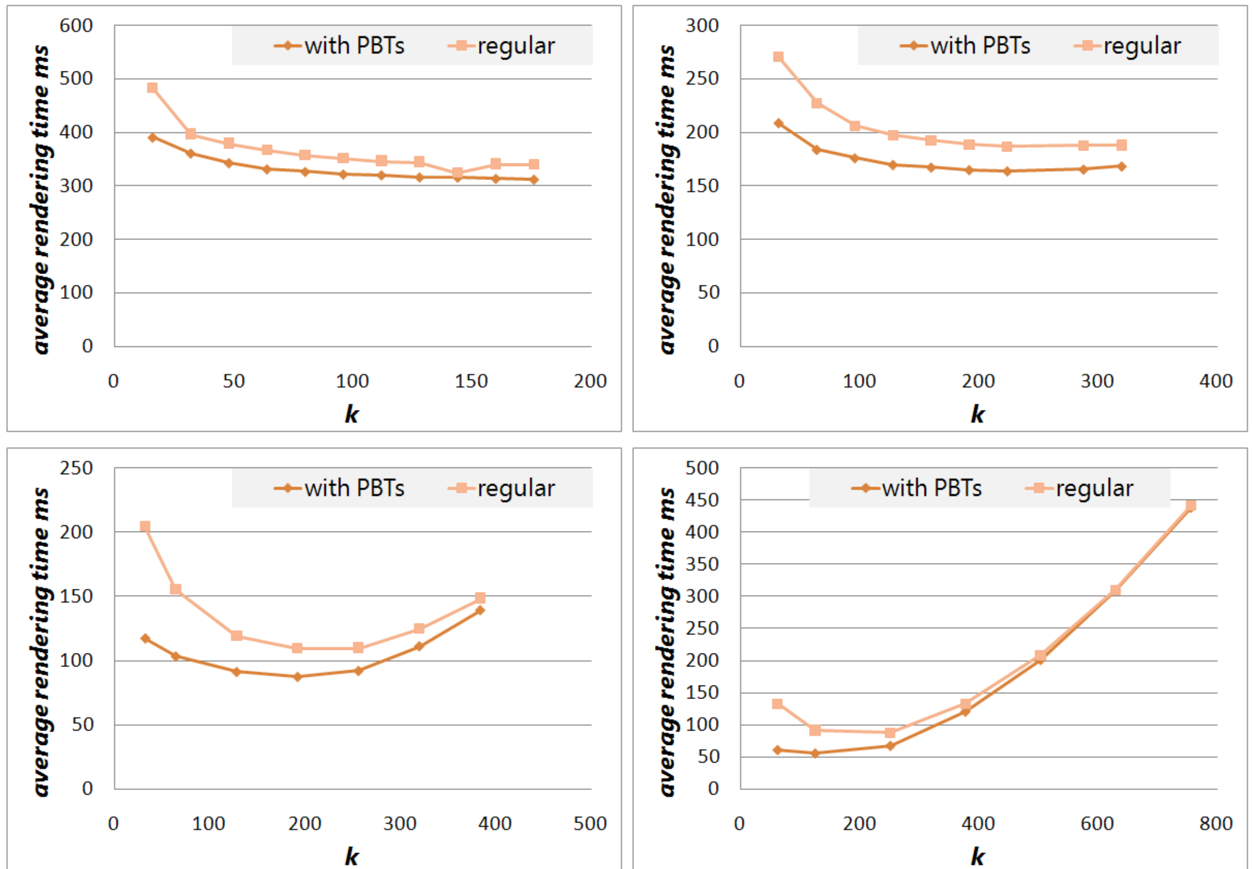


FIG. 4.3. *Optimal subdivision granularity with both regular and PBT subdivision. Test done with 8 (top-left), 16 (top-right), 32 (bottom-left) and 64 (bottom-right) processors. The horizontal axis represents the number of tiles, whereas vertical axis represents rendering time. ERW6-4 test scene.*

Test results, performed on Cacau, raise several interesting considerations. The optimal granularity for the PBT comes with a lower number of tiles, in respect of the normal regular approach. In particular the PBT works better with coarser tiles, introducing beneficial effects because of the lower overhead of communication.

Scalability. We compared our schema based on the PBT against the regular subdivision schema with 2, 4, 8, 16, 32 and 64 processors, in order to evaluate the efficiency of our strategy (see Figure 4.4). The tests, ran on ENEA, shows that our schema works always better than the regular one, and presents almost linear scalability. In all the tests the granularity coefficient k is fixed to 4. However, the test with 64 processors also shows that when the number of tiles increases the use of an adaptive subdivision is still not enough in order to assure a good scalability. We conjecture that, in order to improve scalability, the value of k should be tuned in such a way that tile's sizes do not become extremely small.

Temporal coherence tests. In order to explore the performances of the PBT in exploiting temporal coherence, we checked it under different conditions. We ran a set of tests on Hydra in order to evaluate the correctness of the prediction made using the PBT. First we tested two different task granularity (we recall that we consider $m = k \cdot n$, where m is the number of tiles and n is the number of worker): we have chosen $k = 1$ (one tiles for each worker) and $k = 4$ (four tiles, on average, for each workers). Moreover, we considered two different camera speeds (1x and 2x). In all the tests the number of workers n is 32. To make a comparison, we measured the total amount of tiles which has been estimated correctly using 85th, 90th and 95th percentile (see Table 4.1). As an example row 2 (Perc. 90th) represents the percentile of estimations having an error up to 10%. In other words, when $k = 1$ and the camera speed is 1x the 93.2% of estimations have an error smaller than 10%, while when $k = 4$ and the camera speed is 2x the 79.8% of estimations have an error smaller than 10%.

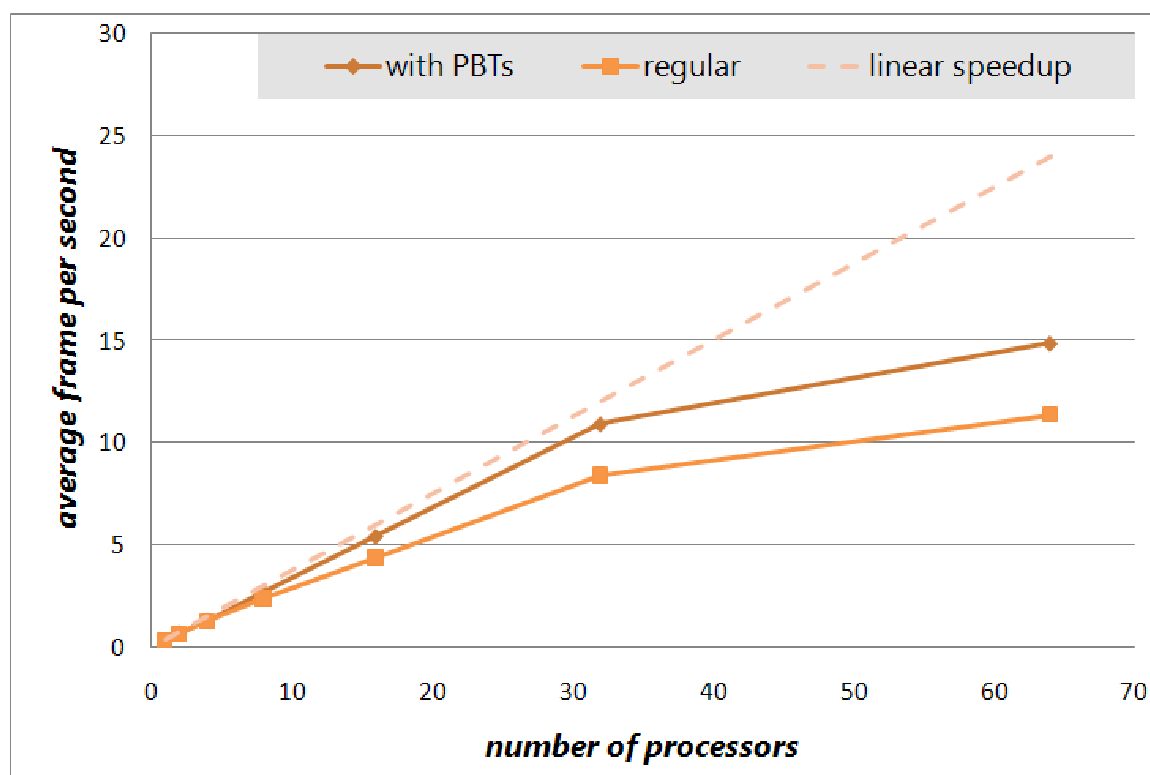


Fig. 4.4. *Scalability. Frame rates on increasing number of processors comparing the regular subdivision, our PBT-based subdivision and the optimal linear speedup. ERW6-4 test scene.*

Spatial coherence tests. In this paragraph we investigate how the PBT exploits data locality using the two locality-aware heuristic described in Section 2.1. The idea is to let workers to better use their CPU cache.

The test considers 4 different scenes with an increasing number of triangles, ranging from less than 30000 to about 950000. In Table 4.2 are reported the number of triangles and the number of nodes in the Kd-tree. The difference between scenes is an increasing number of amphitheaters, all of them with a simple diffusive shader (see Figure 4.5). Scenes are animated by a predefined walk-through of the camera.

The rationale behind the test is to verify that, whenever the size of the Kd-tree is bigger than the cache size, a drop of the performance can be measured, due to the number of cache misses.

Table 4.1
Results of the predictions in 85th, 90th and 95th percentile.

Corr. Perc.	$k = 1$ **1x**	$k = 1$ **2x**	$k = 4$ **1x**	$k = 4$ **2x**
85	96.2%	95.3%	92.1%	89.7%
90	93.2%	92%	86.2%	79.8%
95	92.6%	84%	68%	55%

Table 4.2
A simple description of the test scenes used for spatial coherence tests. For each scene, we report the corresponding number of primitives and the Kd-tree's size.

Test scene	Triangles	Nodes in Kd-tree
1 Amphitheater	29759	251017
4 Amphitheaters	119026	999237
16 Amphitheaters	476098	3970097
32 Amphitheaters	952130	7970097

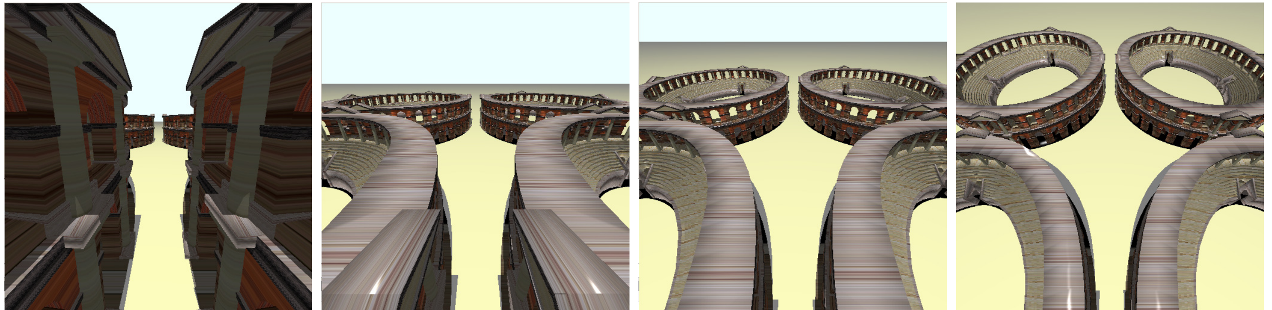


Fig. 4.5. *Amphitheater test scene, used in spatial coherence test set.*

All tests shows that improvements obtained by using the PBT (with both the *locality-aware* heuristics and a random assignment named *PBT-Random*) with respect to the regular assignment (cf. *Exploiting Local Coherence*, Subsection 2.1). With more details, on small scenes, the whole scene fits into the cache and then the assignment strategy does not matter. When the scene becomes larger, so that it does not fit into the cache, the data locality also provides a modest improvement of the performances of the system (around $1 - 5$ ms) using both the *locality-aware* heuristics. This test has been performed on Hydra.

Impact of Resolution. In Parallel Ray Tracing the resolution represents the total amount of work. We ran a set of tests on ENEA, with a fixed task granularity ($k = 4$), using different resolutions in order to show that our implementation of PRT scales linearly with the number of the primary rays. In order to have a measure of the effectiveness of our technique with higher workloads, we tested the PBT with three well-known resolutions. In particular we compared both techniques (PBT and regular subdivision) with PAL (720×576), HD720 (1280×720) and HD1080 (1920×1080) resolutions on the same test scene (ERW6-4).

The tests show that the PBT is effective with all the workloads and its ability in balancing the work between nodes is beneficial with heavy loads (see Figure 4.6).

Total time analysis. The last test is focused on how the whole computing time (i. e., the parallel rendering time) is spent.

The time spent by the master node in computing out PBT-based subdivision schema from a given prediction is serial code and pure overhead introduced by our approach. This overhead corresponds to the time spent in subdividing the image in tiles and updating the PBT.

Our purpose is to determinate: the ratio between time spent by the workers in local rendering and communications; how unbalancing affects performance; how much time is spent by the master node in serial code.

Figure 4.7 shows the test results for 16 and 32 processors at different granularity. The time spent in updating the PBT is proportional to m, hence to the granularity and the number of processors, but in our test is always

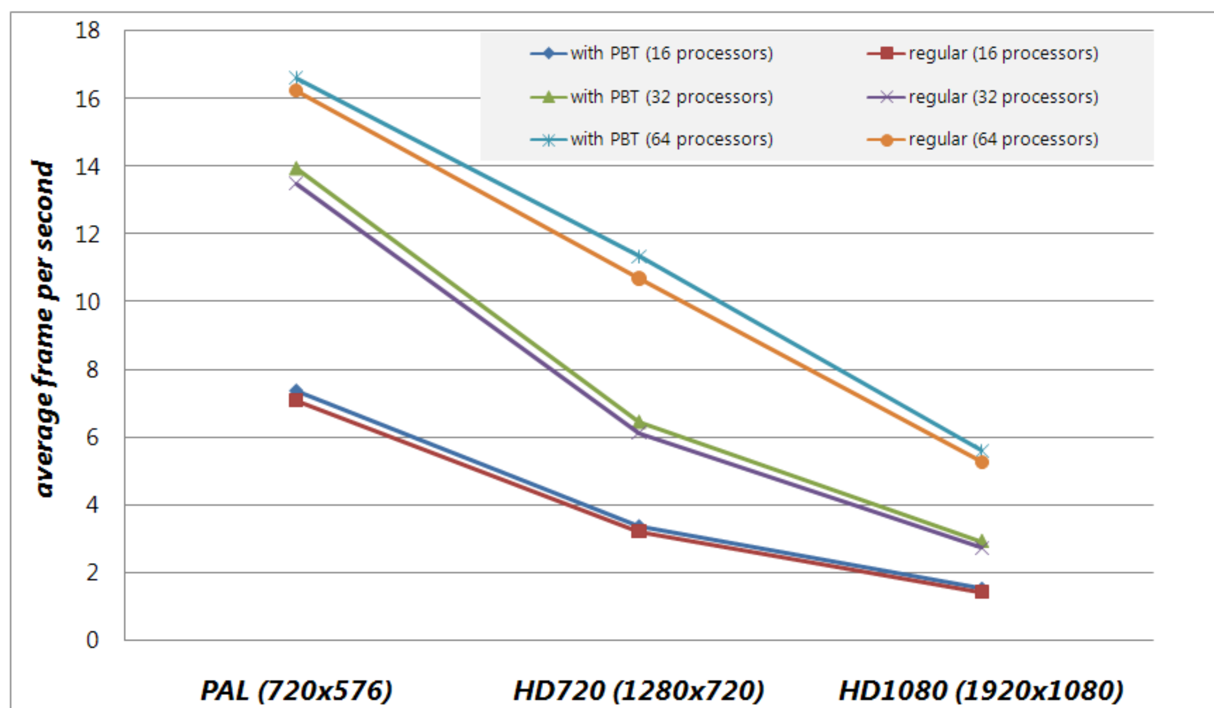


FIG. 4.6. *Average frame rate using different resolutions: PAL(*720×576*), HD720(*1280×720*), HD1080(*1920×1080*), comparing regular and PBT-based job assignments. ERW6-4 test scene.*

lower than 5 ms. Thus the overhead by the adaptive subdivision is small compared with the gain in balancing. This test has been performed on Cacau.

5. Conclusion. In this paper we present a scheduling strategy based on a data structure called PBT. To assess the effectiveness of the proposed scheduling strategy we carried out some experiments, that provided a large amount of results. Our scheduling strategy: (i) improves load balancing; (ii) allows to exploit temporal coherence among successive computation phases; (iii) minimizes the inter-processors dependency. By some assumptions on temporal coherence, we showed that an estimate of next phase workload can be used to quickly divide the mesh in almost-squared tiles assigned to each worker. PBT is effectively used to evaluate the load balance of each phase and, eventually, to update tasks assignment in order to reduce their completion time.

We tested our strategy on a significant problem: Parallel Ray Tracing. We carried out an extensive set of experiments where our PBT-based strategy is compared against the regular subdivision schema. We showed that by using our technique, the optimal granularity comes with a lower number of tiles. Moreover, the PBT approach assures a better scalability.

The predictions used in our approach are based on temporal coherence. We proved that for simple scenes (e.g. a predefined walk-through of the camera around the scene), the proposed estimation heuristic is affordable. Spatial coherence for data locality has been exploited by using two locality-aware heuristic during assignment. However we showed that such heuristic provides an improvement on performance only with larger scene (at least 1 million of triangles).

Tests with higher resolutions show the ability of the PBT in balancing the work with heavy loads.

Finally we provided a total time analysis of the computing time and the overhead introduced by the PBT. The time spent in updating the PBT is proportional to the number of tiles, but in our test is always lower than 5 ms. Indeed, the overhead of the adaptive subdivision is small compared with the gain in balancing.

The variety of architectures used on our tests suggests that the technique improves performances in both cheaper commodity cluster and high performance clusters with low latency networks.

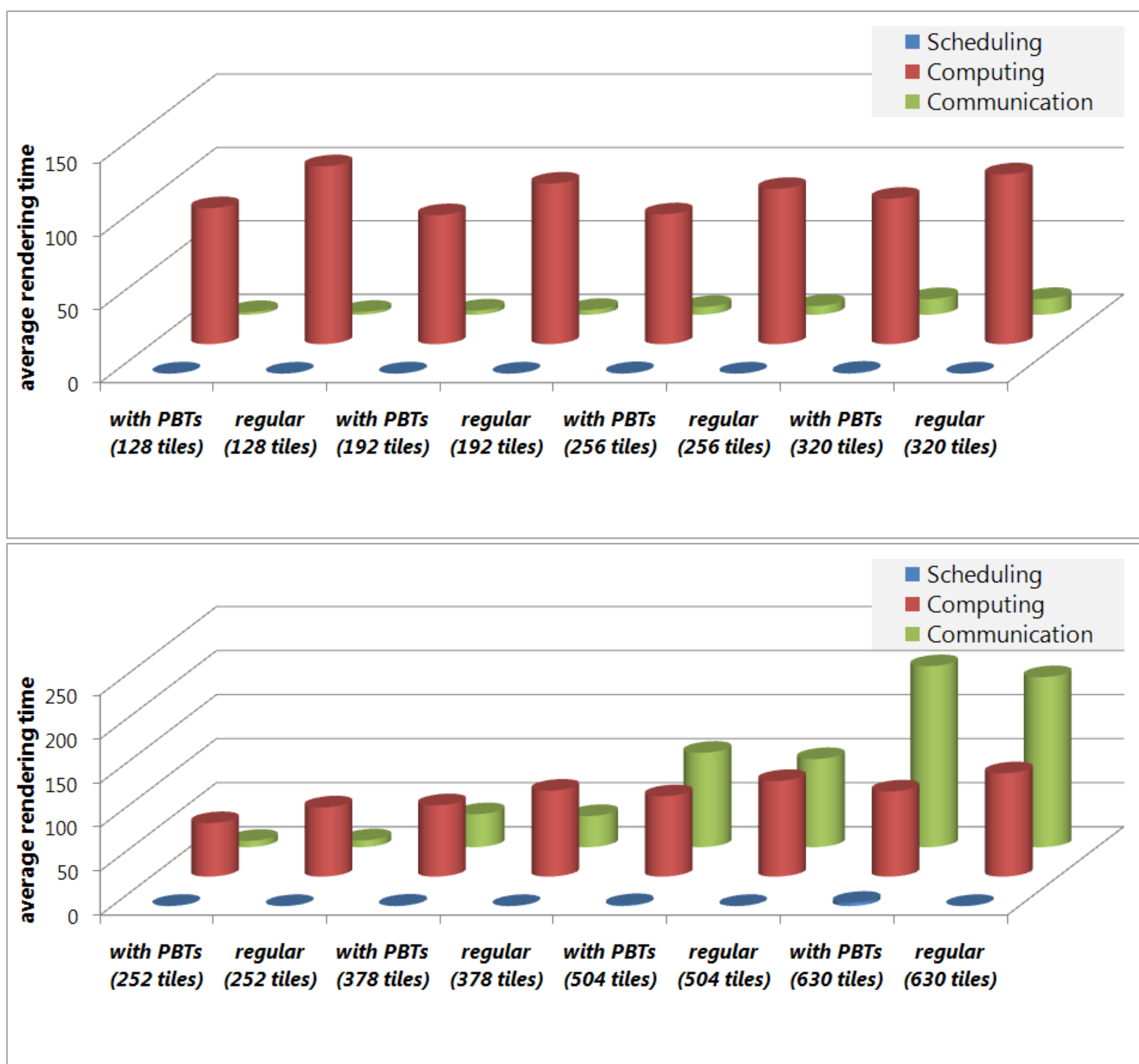


Fig. 4.7. *Contributions in rendering time with 32 (up) and 64 (down) processors, at different granularities, with both adaptive and regular subdivision techniques. The total rendering time is split in: the time spent in subdivision (i. e. update the PBT for adaptive subdivision); the maximum per-node compute time, such as an estimate of the load balancing; the remaining time, mostly due by communications. Rendering times shown are the average in a test scene (ERW6-4) of 600 frames.*

Acknowledgments. A portion of this work was carried out under the HPC-EUROPA++ project (project number: 211437), with the support of the European Community—Research Infrastructure Action of the FP7.

The authors gratefully thank for their collaboration in providing some of the computational resources the ENEA (Ente per le Nuove Tecnologie, l'Energia e l'Ambiente)—Research Center in Portici (Napoli, Italy) and the HLRS Supercomputing Center at Universität Stuttgart (Germany).

REFERENCES

[1] P. Berenbrink, T. Friedetzky, and L. A. Goldberg, *The natural work-stealing algorithm is stable*, in SIAM J. Comput., 32(5):1260–1279, 2003.
[2] J. Bigler, A. Stephens, and S. G. Parker, *Design for parallel interactive ray tracing systems*, in IEEE Symposium on Interactive Ray Tracing, 0:187–196, 2006.
[3] R. D. Blumofe and C. E. Leiserson, *Scheduling multithreaded computations by work stealing*, in Journal of ACM, 46(5):720–748, 1999.

[4] C. Boeres and V. Rebello, *Cluster-based static scheduling: Theory and practice*, in Proceedings of the 14th Symposium on Computer Architecture and High Performance Computing (SCAB-PAD'02), page 133, Washington, DC, USA, 2002. IEEE Computer Society.
[5] A. Chalmers and E. Reinhard, editors. *Practical Parallel Rendering*. A. K. Peters, Ltd., Natick, MA, USA, 2002.
[6] J.Chapman, T. W. Calvert, and J.Dill, *Exploiting temporal coherence in ray tracing*, in Proceedings on Graphics interface '90, pages 196–204, Toronto, Canada, 1990.
[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.
[8] B. Cosenza, G. Cordasco, R. De Chiara, U. Erra, and V. Scarano, *Load Balancing in Mesh-like Computations using Prediction Binary Trees*, in International Symposium on Parallel and Distributed Computing (ISPDC '08), pages 139–146, ISBN: 978-0-7695-3472-5, 2008. IEEE Computer Society.
[9] D. E. DeMarle, C. P. Gribble, S. Boulos, and S. G. Parker, *Memory sharing for interactive ray tracing on clusters*, in Parallel Computing, 31(2):221–242, 2005.
[10] G. C. Fox, R. D. Williams, and P. C. Messina, *Parallel Computing Works!*, Morgan Kaufmann, May 1994.
[11] K. Hwang and Z. Xu., *Scalable Parallel Computing: Technology, Architecture, Programming*, McGraw-Hill, 1998.
[12] Y. Kwok and I. Ahmad, *Benchmarking and comparison of the task graph scheduling algorithms*, in Journal of Parallel and Distributed Computing, 59(3):381–422, 1999.
[13] Message Passing Interface Forum. The Message Passing Interface (MPI) standard.
[14] J. Nagle *Rfc 896: Congestion control in ip/tcp internerworks*, 1984.
[15] D. M. Nicol and J. H. Saltz, *Dynamic remapping of parallel computations with varying resource demands*, in IEEE Transaction on Computer, 37(9):1073–1087, 1988.
[16] M. Parashar and J. C. Browne, *Distributed dynamic data-structures for parallel adaptive mesh refinement*, in Proceedings of the International Conference on High Performance Computing, 1995.
[17] M. Parashar and J. C. Browne, *On partitioning dynamic adaptive grid hierarchies*, in Proceedings of the 29th Hawaii International Conference on System Sciences (HICSS'96) Volume 1: Software Technology and Architecture. IEEE Computer Society, 1996.
[18] G. L. Park, B. Shirazi, and J. Marquis, *Mapping of parallel tasks to multiprocessors with duplication*, in Proceedings of the 31st Hawaii International Conference on System Sciences (HICSS'98) Volume 7, page 96, Washington, DC, USA, 1998. IEEE Computer Society.
[19] P. Shirley and R. K. Morley, *Realistic Ray Tracing*, A. K. Peters, Ltd., Natick, MA, USA, 2003.
[20] K. Suffern, *Ray Tracing from the Ground Up*, A. K. Peters, Ltd., Natick, MA, USA, 2007.
[21] I. Wald, *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Saarland University, 2004.
[22] I. Wald, C. Benthin, A. Dietrich, and P. Slusallek, *Interactive Distributed Ray Tracing on Commodity PC Clusters – State of the Art and Practical Applications*, in Proceedings of EuroPar '03, Lecture Notes on Computer Science, 2790:499–508, 2003.
[23] I. Wald and V. Havran, *On building fast kd-trees for ray tracing, and on doing that in O(N log N)*, in Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing, pages 61–69, 2006.
[24] T. Whitted, *An improved illumination model for shaded display*, in Communications of the ACM, 23(6):343–349, 1980.
[25] L. Yang, J. M. Schopf, and I. Foster., *Conservative scheduling: Using predicted variance to improve scheduling decisions in dynamic environments*, in Proceedings of the 2003 ACM/IEEE conference on Supercomputing (SC'03), page 31. IEEE Computer Society, 2003.
[26] T. Yang and A. Gerasoulis, *Dsc: Scheduling parallel tasks on an unbounded number of processors*, in IEEE Transactions on Parallel and Distributed Systems, 5:951–967, 1994.

Edited by: Marek Tudruj, Dana Petcu
Received: February 5th, 2009
Accepted: June 28th, 2009

Scalable Computing: Practice and Experience
Volume 10, Number 2, pp. 189–199. http://www.scpe.org

ISSN 1895-1767
© 2009 SCPE

THE INFLUENCE OF THE IBM PSERIES SERVERS VIRTUALIZATION MECHANISM ON DYNAMIC RESOURCE ALLOCATION IN AIX 5L

MACIEJ MLYNSKI*

Abstract. This paper presents analysis of the influence of the virtualization mechanism of IBM pSeries servers on dynamic resource allocation in AIX 5L operating system. It raises issues of economic use of resources and distribution of processing power. Some innovative solutions are proposed as methods for dynamic resource allocation. These are: micro-partitioning and partition load manager utility. Some results of experiments are presented. The goal of these experiments was to estimate the influence of selected AIX 5L operating system adjustments on computation efficiency.

Key words: virtualization, AIX, DLPAR, dynamic resource allocation, PLM, VIO, hypervisor, micro-partitioning

1. Introduction. At present, the demand for server processing power begins to go beyond its processor capabilities. However, installing more processors and memory causes the resources to become more expensive [3, 14, 15]. Therefore research is being undertaken to investigate how to utilize the resources more efficiently. In this direction important research are grid and autonomic computing [3, 12, 17]. During the computer systems design phase we need to consider installing more than one application and database onto one individual server [2].

Many factors can influence multiple applications, such as splitting of mission critical applications and development, creating an environment for patch installation as well as computer systems for many different customers. A multi-instance environment could also be motivated by efficiency reasons [4, 5, 6]. Efficiency would also influence a *multi-instance* environment. Unfortunately, several conflicting instances of databases or applications on one server may create a problem with conflicting processing power resources. The problems can be resolved by adequate parameters modifications inside the operating system or server microcode.

The dynamic resource allocation mechanisms are based on well documented theory concerning mainly elements of the operational research such as queuing and resource management theory, Markov chains, analysis of time series and gradient methods [9, 16].

The paper discusses efficient use of resources and distribution of processing power in IBM pSeries servers. Three solutions for the improved dynamic resource allocation are being discussed: micro-partitioning, capped and uncapped processing power as well as the partition load manager utility. Results of experiments are presented, and the influence of virtualization technology on IBM pSeries server efficiency are analysed. The following are the main contributions of the paper:

- Efficiency problems in IBM pSeries servers have been experimentally analysed. The experiments have been done at customer sites in banks and others complex installations.
- Implementation of an efficient policy for partition load manager utility has been done.
- The set of virtual machine parameters has been analysed in concern with dynamic modification ability.
- It has been proved that to improve entire hardware efficiency by dynamic changes of the amount of assigned memory and the number of virtual processors, selected parameters in the operating system should be changed as well.

Dynamic resource allocation in operating systems involves a huge amount of parameters, but only their correct adjustment can give the expected results. The set of experiments presented in section four shows that increasing of processing resource in logical partition not always improve the overall system efficiency.

The goal of this paper is to indicate possibilities of improvement of dynamic resource allocation characteristics. In particular the attention was turned onto mechanisms of tuning operating systems and tuning the I/O subsystems.

2. The virtualization and workload management on AIX. A progress in methods for data transmission and information processing enforces changes in computer systems architecture. In the AIX 6.1 operating system modern virtualization mechanisms have been introduced. The mechanisms enable creating flexible configurations by administrators and systems architects. In the IBM pSeries servers environment *micro-partitioning* (dividing resources into small pieces), *IVE (Integrated Virtual Ethernet)*, *Shared Ethernet*, and *virtualization* have just been introduced.

*ASpartner Sp. z o.o., ul. Jaworowa 5, 05-816 Michalowice, Poland, (m.mlynski@aspartner.com.pl).

Conception of virtualization mechanism in IBM pSeries servers is similar to the concepts implemented in other virtual environments such as Xen and VMWare [19]. The conception assumes that physical resources like processors units, memory, I/O drawers and adapters can be shared. Fig. 2.1 shows the scheme of logical partitions on IBM pSeries architecture. [1] The new computer system architecture has introduced a number of new elements, which system architects should consider when designing information systems.

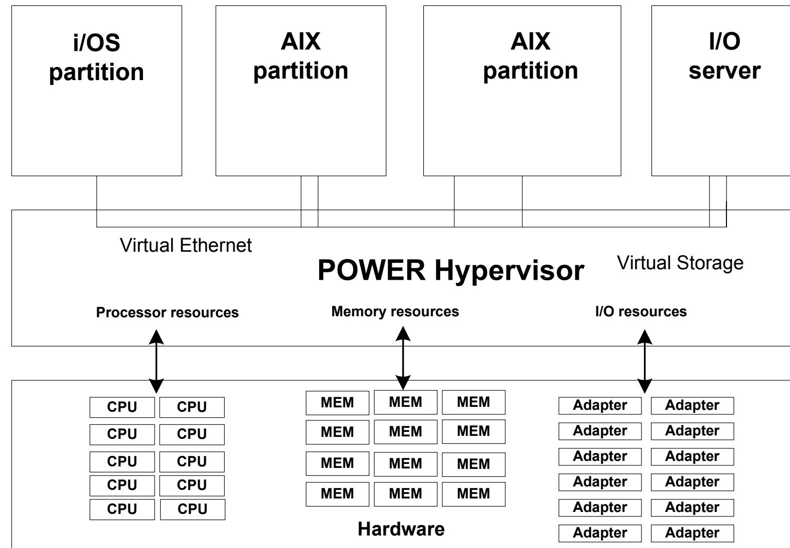


Fig. 2.1. *Scheme of logical partitions on IBM pSeries Systems server architecture. [1, 15]*

Another subject on which emphasis is placed today is dynamic resource allocation issue. Since not long time ago we know the *Work Load Manager* functionality on AIX operating system [8, 13, 14]. Now we can also use a new functionality called DLPAR (dynamic logical partitions). The functionality allows for on-line resources allocation between logical partitions e.g. processors, memory and I/O. Introduction of the DLPAR is possible because new servers have an intermediate layer called a *hypervisor* [7]. In this solution, memory is not addressed directly by the operating system. The hypervisor is serving a translation buffer role. Therefore these solutions allow for the use of a tool called *Partition Load Manager*. They also allow dynamic hardware manipulation. As a result, it is possible to configure a computer system so, that if required, we can dynamically add processors and memory to a particular partition. Virtualization technologies provide also abilities to switch resources between servers like WPARs (Workload partitions) which provides a single focal point for management of AIX 6.1 across an enterprise and *Live Application Mobility* which allows on-line realocations between servers. Hardware *hypervisor* is supporting a few kinds of operating systems installed together on one physical machine. On the *IBM pSeries System* hardware platform these operating systems, which can be installed in that way are *AIX*, *i/OS*, and *Linux* (*SLES*, *RHEL*). These operating systems can use shared and virtual resources.

In the partitioned environment, memory utilization is described by two parameters: the amount of allocated memory and the number of scans per second. Therefore, a partition load manager has its own knowledge database. In the database, the partition load manager stores information about system utilization. Resource allocation is realized on the basis of stored data and defined policy rules.

The process of making policy rules must be consistent with AIX specification. Memory space in AIX is divided into many sub spaces e.g. *working*, *client* and *persistent*. The working memory is a memory, which is used by program codes and libraries. Therefore, the client and persistent spaces are dedicated to storing data. How much memory to be assigned for a file cache and how much for a program code is adjusted by the *minperm* and *maxperm* parameters. On the basis of these parameters, the *page stealer* will decide which memory pages should be released.

Resource allocation between logical partitions is governed by the following rules: the hardware management console is sending a request to AIX for their release. After that, the operating system releases the resources

and sends information to the hardware management console that the resources were released. Fig. 2.2 shows a schematic of connections of hardware management console with particular logical partitions, partition load manager and resource manager. [1]

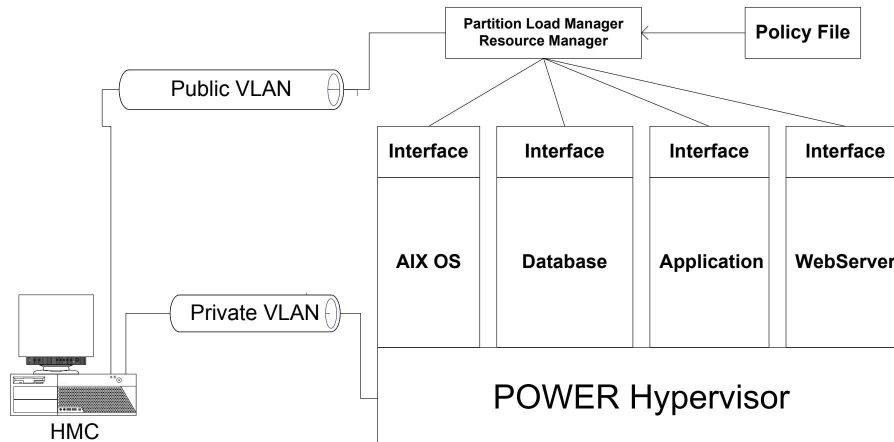


FIG. 2.2. *The scheme of physical connections to partition load manager. [1, 15]*

The policy rules of partition load manager must estimate and define minimum, maximum and guaranteed amount of resources, priority, weight and thresholds for every partition. Administrator should also define minimum and maximum thresholds when events will be generated for every resource. If the same partition sends requests for new resources, the system will release them according to the following rules:

- Resources available and un-assigned,
- Resources-allocated but not used,
- Resources from partitions with lowest priority.

From the above we can draw the conclusion, that the architects must plan the resources allocation before or during the design of a system, to ensure efficient resource allocation.

The design of policy rules has to respect both the architecture of a computer system and business application features. We have several system perspectives which should be respected on the basis of the P6 architecture, e.g.:

- Security, *CAPP/EAL4+* (controlled access protection profile and evaluation assurance level 4+)
- Architecture of central processing units e.g. *simultaneous multi threading*, *memory affinity*, *L3* sharing between processors,
- *RAS* architecture (reliability, availability and scalability),
- *Virtual* and *shared I/O*,
- *Virtual* and *shared Ethernet*,
- Capacity on Demand,
- Multiple operating systems.

Sometimes business applications and databases do not allow dynamic changes. Therefore, we must remember that applications and databases should allow for dynamic resource allocation changes and applications should be able to send requests to the hardware management console with demand for new resources. Applications and databases should also be able to reduce their own resources if the hardware management console needs it.

3. Policy rules for Partition Load Manager. If we want to create policy rules for a partition load manager we must first create new resources. In our environment, we have created two resource groups. For each of these resources groups we have defined the maximum and minimum amount of resources to be need. During this process we can also indicate if the processors are to be dedicated or shared. Fig. 3.1 shows the defined resources.

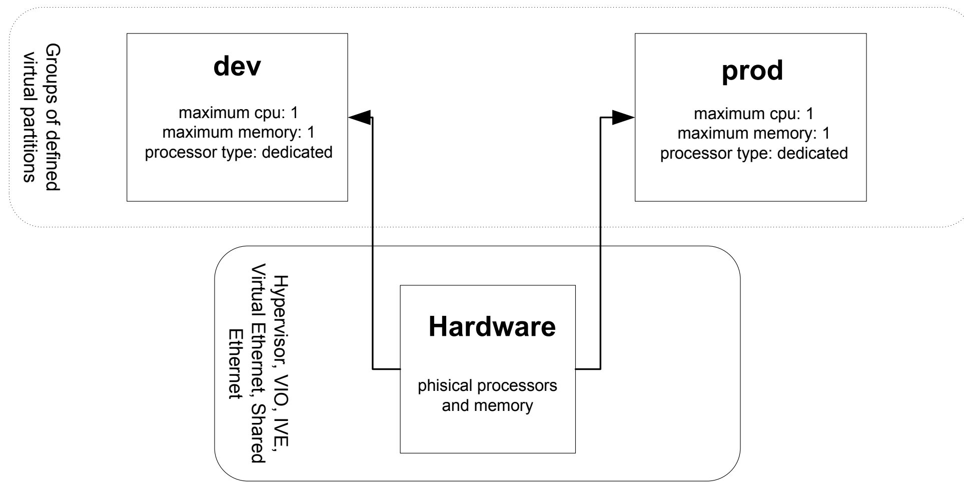


Fig. 3.1. *Resource group in Partition Load Manager.*

The *Partition Load Manager* has some parameters to define thresholds indicating when resources will be allocated between logical partitions. Frequent physical changes can cause low system stability. However, setting the thresholds to high values can cause that resources will never swap.

Next, we have assigned logical partitions for particular physical group. During that, we had to determine the *maximum*, *minimum* and *guaranteed resources* for all logical partitions. This operation concerns processors and memory.

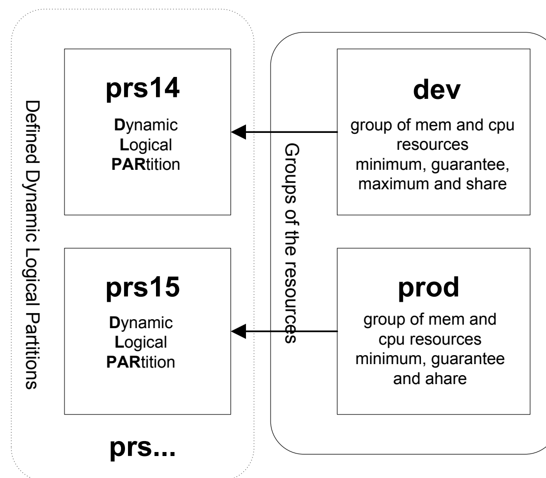


Fig. 3.2. *The definition of logical partitions in Partition Load Manager.*

Two logical partitions for this experiment were created. The first was called prs1 and had 1 CPU and 4 GB RAM. The second logical partition was called prs14 and had 4 CPU and 6 GB RAM. Fig. 3.2 shows the scheme in which one group is assigned to one logical partition. The logical partition called prs15 is assigned to a production group. Therefore the logical partition called prs14 is assigned to the development group.

For policy rules as well as a description of logical partitions, parameters that determine the behavior after crossing of thresholds should be specified. The essential parameter is a parameter which determines whether a partition load manager should release free resources automatically. In the case of a request for more resources to be sent by logical partition, the partition load manager will firstly give resources not yet allocated.

TABLE 3.1
Parameters of the logical partitions.

	Instance	Group	Defaults
Entitled capacity delta	10	-	10
Memory delta	1	1	1
CPU notisy intervals	5	6	6
Memory notisy intervals	5	6	6
CPU load average high threshold	0.9	1.00	1.00
CPU load average low threshold	0.1	0.40	0.40
Minimum entitlement per VP	0.4	-	0.50
Maximum entitlement per VP	0.4	-	0.50
Memory utilization high threshold	50	50	90
Memory utilization low threshold	90	90	50
Memory page steal high threshold	0	0	0
Immediate release of free CPU	yes	no	yes
Immediate release of free memory	yes	no	no

In cases when there are fewer resources, some should be released from the running partition. Then, they are released on condition that the logical partition is using more resources than was guaranteed. If all logical partitions need their own resources, then the partition load manager decides which resources should be released on the basis of partition weight. The weight of a logical partition is specified during the server partitioning.

Table 3.1 shows the parameters of a logical partition. All logical partitions can have different parameters. However, the group restrictions should be valid and proper i. e. precise match with expected requirements. Also tunable resources entitlements should be defined. Entitlements are a kind of resources limits.

Resources are re-allocated mainly based on these parameters. Resources between logical partitions are distributed in harmony with defined maximum, minimum and guaranteed values for memory and processors. In P5 and P6 environment it is possible to change the CPU allocation by 0.01 of their processing power. More precise explanation of the tuning parameters can be found in Table 3.2.

After the policy rules definition, we can run partition load manager. In our test environments, the system parameters were changed many times. After running the partition manager we were able to show statistics. Fig. 3.3 shows memory statistics for a logical partition prs15. Fig. 3.4 shows the processor utilization. Processing statistics show average processor utilization.

Show LPAR Statistics
CPU Statistics | Memory Statistics

Name	Minimum	Guaranteed	Maximum	Share	Current	Use %	Pagesteal
Server-9119-595-SN830EA8D			4096				
--- prod			4096				
------ prs15	1024	1024	4096	1	2048	80.56	0

FIG. 3.3. *Memory statistics for logical partitions.*

Show LPAR Statistics
CPU Statistics | Memory Statistics

Name	Minimum	Guaranteed	Maximum	Share	Current	Use %	Load average
Server-9119-595-SN830EA8D			1.00				
--- prod			1.00				
------ prs15	0.10	0.10	1.00	1	0.40	1.79	0.09

FIG. 3.4. *The CPU statistics for logical partition.*

The partition load manager can be run in passive or active mode. In passive mode only the analysis of policy rules is possible. The next stage, after analyzing the partition load manager, is to be run in active mode. In this case, the dynamic resource allocation is possible.

Not all applications support dynamic resource allocation. Applications which do not support dynamic resource allocation often use bind processors and use *pinned memory*. *Pinned memory* is a memory whose segments are placed always at the same address and can not be swapped into the paging space. *Pinned memory* is used by subsystems of databases like *aio* (asynchronous I/O). In such applications, dynamic resource allocation

TABLE 3.2
CPU-related parameters [2].

Tunable	Description
Entitled capacity delta	The amount of CPU entitled capacity to add or remove from a shared processor partition. The value specifies the percent of the partition's current entitled capacity to add or remove.
CPU notify intervals	The number of 10 second sample periods that a CPU-related sample must cross a threshold before the reallocation will take effect.
CPU load average high threshold	The processor load average high threshold value. A partition with an average load above this value is considered to need more processor capacity.
CPU average load low threshold	The CPU average load low threshold value. A partition with a average load below this value is considered to have unneeded CPU capacity.
Minimum entitlement per virtual processor	The minimum amount of entitled capacity per virtual processor. This attribute prevents a partition from having a degraded performance due to too many virtual processors relative to its entitled capacity. When entitled capacity is removed from a partition, virtual processors will also be removed if the amount of entitled capacity for each virtual processor falls below this number. Default value is 0.5. Minimum value is 0.1. Maximum value is 1.0.
Maximum entitlement per virtual processor	The maximum amount of entitled capacity per virtual processor. This attribute controls the amount of available capacity that may be used by an uncapped (uncapped means that amount of resources is not strictly limited) shared processor partition. When entitled capacity is added to a partition, virtual processors will be added if the amount of the entitled capacity for each virtual processor goes above this number. Increasing the number of virtual processors in an uncapped partition allows the partition to use more of the available processor capacity.
Immediate release of free CPU	Indicates when processor capacity not needed by a partition is removed from the partition. The value of no indicates unneeded CPU capacity remains in the partition until another partition has a need for it. The value of yes indicates unneeded CPU capacity is removed from the partition when the partition does not need it any longer.

can cause a crash. The physical resource allocation have also a big impact on system efficiency. The strength of the influence it has on I/O subsystems is discussed in the following paragraph.

4. The analysis of the influence of disk subsystem parameters on operating system efficiency. We distinguish sequential and random disk transfers. In random disk transfers, administrators do not have much freedom to tune it, but in sequential transfers there is a bigger possibility [10, 11]. The most important parameters are *minpagereadahead* and *maxpagereadahead*. How the mechanism works is shown in the following examples, assuming that *minpagereadahed=2* and *maxpagereadahead=16*.

For this setup, in the case of sequential reading, first transfer from disk will read two pages. If system recognizes that the pages, which were read are sequential, the next transfers will read 4 pages, next one 8, 16 etc. All other transfers will read 16 blocks in one reading cycle. A maximum size of transfer is always equal to *maxpagereadahead* value. For the system to be efficient, it should have enough memory for every read. The system should have *maxpagereadahead* free pages. How many free pages will be adjusted by *minfree* and *maxfree* tuning parameters? The number of free pages should never fall below *minfree*. If the numbers of free pages reach *minfree*, then the *page stealer* process will release pages until the amount equals *maxfree*. *Maxfree* value is defined by the following formula:

$$maxfree = minfree + maxpgahead \tag{4.1}$$

Table 4.1
Influence comparison of system values on their efficiency.

	Min free	Max free	Max page read ahead	Mem pools	File size	Trans. type	Time
1	960	1088	8	1	1GB	read	1.14s
2	960	1088	16	1	1GB	read	1.07s
3	960	1088	32	1	1GB	read	1.01s
4	960	968	8	1	1GB	read	1.10s
5	960	968	16	1	1GB	read	1.15s
6	960	968	32	1	1GB	read	1.18s
7	960	1088	8	1	1GB	r/w	20.49s
8	960	1088	16	1	1GB	r/w	16.44s
9	960	1088	32	1	1GB	r/w	17.60s
10	960	968	8	1	1GB	r/w	19.16s
11	960	968	16	1	1GB	r/w	19.45s
12	960	968	32	1	1GB	r/w	20.02s

The minfree value is determined by experiments. It is usually equal to the size of one of system processes, which should be run fast. We must remember that the memory below *minfree* value is wasted. The experiment was carried on in highly loaded computer systems. To carry out the experiment, one should also have correction on the number of processors and memory pools. After the corrections, the *maxfree* is defined by the following formula:

$$maxfree = minfree + (maxpgahead * nCPU) \tag{4.2}$$

For small computer systems, the *mempool* value is equal one. However, in quite big computers, the value is adjusted during an installation process. For example, in one of tested computers that has 40 GB RAM, the number of *memorypools* was equal 5. The value can be changed. After a correction for *mampool*, the *maxfree* will be defined by the following formula:

$$maxfree = minfree + (maxpgahead * nCPU/mempools) \tag{4.3}$$

From this formula we see that during the dynamic memory allocations we should also modify the *Virtual Memory* values. The latest implementations of AIX require to restart the operating system after changes of the *memools* value. In the future, when we change the *mempools* value without rebooting, the *mempools* value should be changed automatically during dynamic resource allocations. To check how big influence on system efficiency the parameters of *maxfree*, *minfree* and *maxpagereadahead* have, experiments were carried out. During the experiments some files were copied. The experiment was carried out on one of logical partition on p595 server. The logical partition has 0.4 CPU (POWER5 1.9 GHz) and 2047 MB RAM. In Table 4.1 the results have been presented.

The experiments (1-6) confirm that an increase of *maxpagereadahed* increases system efficiency when we have enough free memory for pages to be allocated. The second part of this table shows the results of parallel read and write. The experiments 7-12, show that when we have small number of pages, the increase of *maxpagereadahead* value results in longer files copying time. It means that we should be careful about memory free pages. System will start running the *memory stealer* process when we do not have enough free pages, and so, it will result in weak system efficiency.

To better understand the dynamic resource partitioning another test was executed. The experiment was done on the p5 server with 11 virtual partitions and virtual VIO server. All eleven virtual partitions had no physical I/O adapters. All physical I/O were connected to the VIO server. The physical resources were divided and exported to particular partitions. Processing power was divided in *shared* and *uncapped* mode i. e. all partitions could use all CPU resources. The amount of utilized memory for each partition could be settled dynamically between 1 and 16 GB.

To measure the I/O efficiency the following tests have been executed: a) write the data to disk; b) read from disks; c) dynamic reallocating memory to particular partition during I/O operation.

The disk efficiency experiment showed that maximum transfer rate for the I/O for one partition was 320 MB/sec. During the experiment the 1MB block size was used. In Fig. 4.1 we can see partition utilization as a

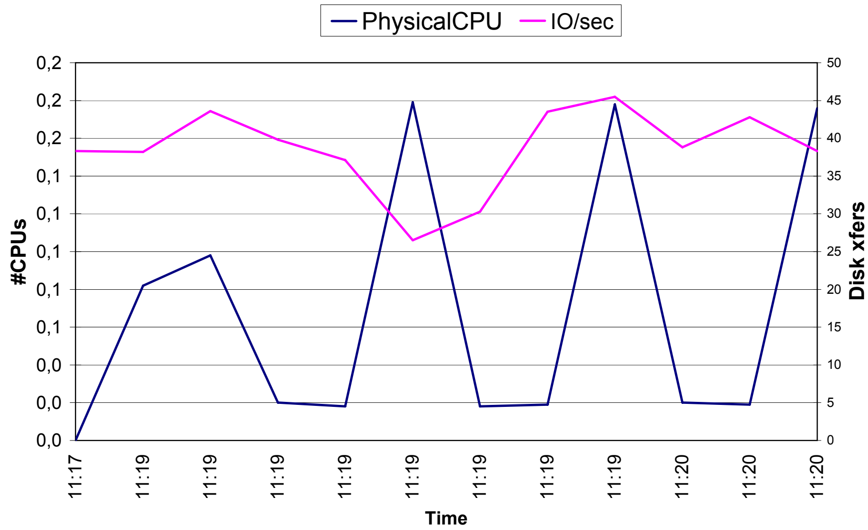


Fig. 4.1. *Lpar utilization during disk writing.*

function of time. Most of processing power spent on communication between external storage and the server was used by Virtual I/O server. Therefore on the LPAR only 0.2 CPU was used.

The second experiment showed the system behavior during dynamic memory reallocating. For one of partitions, the amount of memory was increased from 1 GB to 5 GB. As we can see in Fig. 4.2, the percentage of memory usage was decreased slightly. The dynamic memory reallocation works in the following way: the hardware management console is communicating with the hypervisor, if there is free memory than allocate it, then HMC is sending the message to operating system which information containing the memory address that might be used. Fig. 4.3 shows the same experiments which are shown in Fig. 4.2. The The numbers of *forks* and *execs* decreased in Fig. 4.2 at the same time when the amount of used memory changed in Fig. 4.3. For current version it is possible to decrease, increase and relocate memory simultaneously between two partitions. When decreasing an amount of memory from a partition all applications must release particular memory segments.

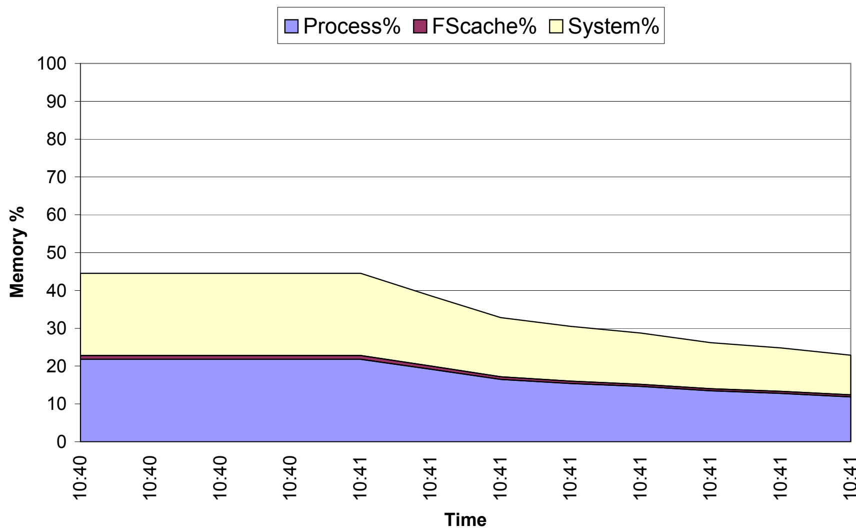


Fig. 4.2. *Memory usage.*

More memory caused that the amount of forks and exec slightly decreased, also the numbers *pswitch* and *syscalls* were decreased automatically during the memory allocation.

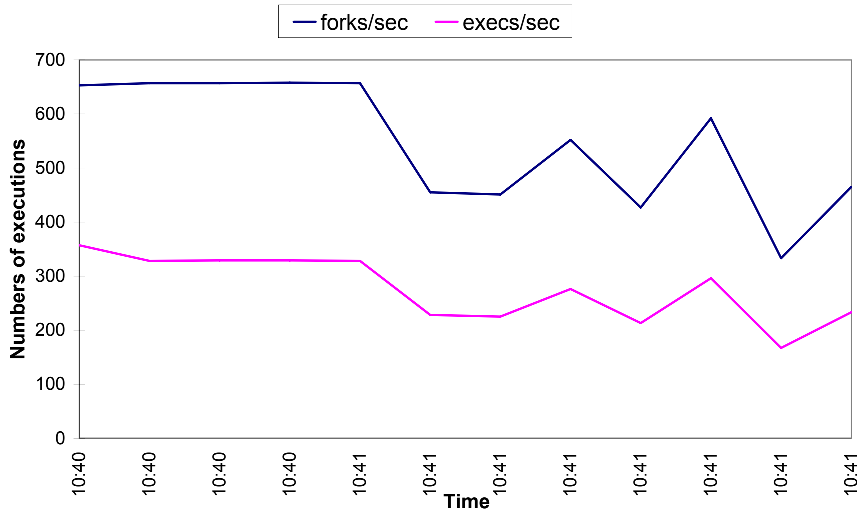


FIG. 4.3. *Processes.*

The system statistics show that when the amount of memory was increased the utilization of kernel read/write system calls, *hypervisor poolidle* has decreased. The parameters show that the capacity of the partition should be better and disk transfers should increase. Then it shows that the disk transfers slow down after the dynamic changes.

Next experiments show the system behavior when the numbers of processors and amount of memory has been changed. These experiments were taken in P5 server with 1.6MHZ CPUs with *uncapped* mode. The external devices were the following: (1) external storage DS4700 (2) logical volume created on RAID5 (3) fiber channel switches with 4Gbps adapters. Tests were generated by using the tools from *iozone* [18]. The first experiment has been done by using 0.1 CPU and 1GB RAM. The results are in Fig. 4.4.

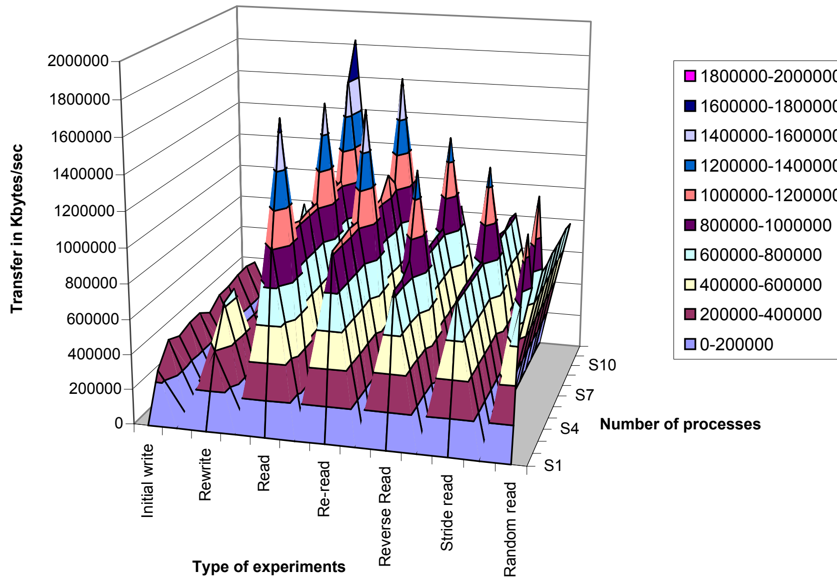


FIG. 4.4. *Transfers with 1GB 0.1 CPU.*

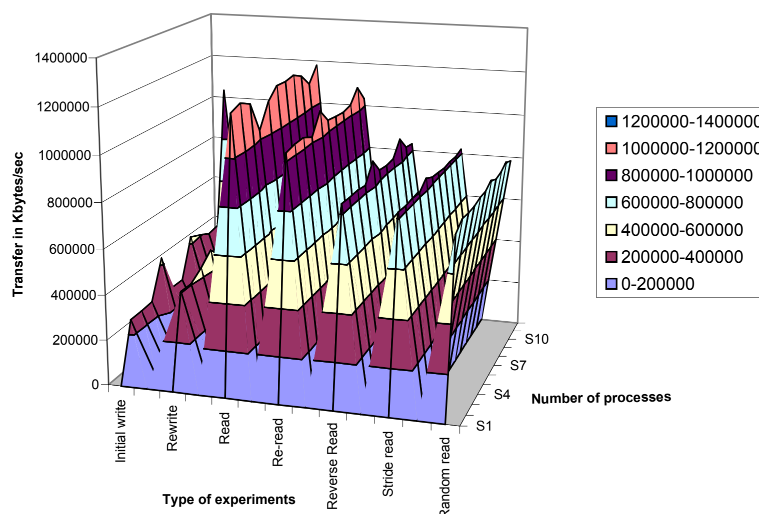


Fig. 4.5. *Transfers with 6GB and 3CPU.*

Because the intention of the experiments was to show an environment similar to a real system, the figures show series of experiments such as read, write, re-read, reverse read, stride read and random read. All of the tests have been done for various ranges of methods i. e. write, read, re-read, reverse read. The initial setup of this program was the following: minimum of processes equal 10, the maximum of processes equal 20. The generated plot (Fig. 4.4) shows S1 as the minimum numbers of processes, S10 as the maximum. Second experiment (Fig. 4.5) in the series has been done with 3 CPUs and 6 GB RAM.

The above experiments have shown that a logical partition with more memory and processing power (Fig. 4.5) has the ability to proceed with the I/O requests in a more stable way, because the transfer rate is the same for 10 and 20 processes. The Fig. 4.4 shows that average transfers are worst and also some undeterminism has been found. Because the transfer rate for read reached 1800000 kBytes/sec this is more than for a read with 6 GB RAM i. e. 1200000 kBytes/sec.

The experiments have shown that the influence of processor frequency on system performance is less important now than it was many years ago. For high frequency processors system parameters describing process scheduling have less influence on performance, because the parameters describing access to memory and storage are more important for this purpose these days.

5. Conclusions. Considering the fact that computer science and computer parallel systems architecture are still evolving, the problem of tuning dynamic resource allocation is a novelty. Continuously changing environment enforces the dynamic resource allocation mechanisms to improve.

This article concerns innovative techniques like *virtualization* and *partition load management*. Thanks to them, we can dynamically manage processing efficiency. The mechanisms make it possible to dynamically move resources between logical partitions on a server. Therefore the computer systems design can be more adequate to demands of new applications and databases. Consequently, we can say that it is possible to prepare methods for tuning operating system properties in order to improve suitability for dynamic resource allocation.

A reduction of the resource pool during normal system operation requires important changes in the operating system tuning values. The conclusion of the analysis and experiments is that in this respect even small changes in hardware can have big influence on system efficiency.

REFERENCES

[1] M. Mlynski, *The analysis of influence of IBM pSeries servers' virtualization mechanism on dynamic resources allocation in AIX 5L*, Proceedings for IEEE CS International Conference ISPDC, 2008.

[2] A. B. Blank, M. Gieparda, J. Haust, O. Stadle, D. Szersi, *Advanced POWER Virtualization on IBM @server p5 Servers: Introduction and Basic Configuration*, IBM ITSO, Austin, 2005.

[3] *Autonomic Computing*, http://www.research.ibm.com/autonomic

[4] P. Bari, P. Cassier, A. Defendi, J. Hutchinson,A. Maneville, G. Membrini, C. Ong, *System Programmer's Guide to Workload Manager.*, IBM ITSO, Austin, 2005.

[5] P. Bari, D. Dillenberger, H. Morrill, *System Programmer's Guide to Workload Manager.*, IBM ITSO, Austin, 2005.

[6] T. Cederlof, A. Klark, T. Herlin, T. Ostaszewski, *IBM Certification Study Guide AIX Performance and System Tuning.*, IBM ITSO, Austin, 2000.

[7] A. Dembeter, S. Dutta, A. Roll, S. Son, *AIX 5L Differences Guide Version 5.3 Edition.*, IBM ITSO, Austin, 2004.

[8] D. Clitherow, S. Herzog, A. Salla, V. Sokal, Trethewey J., *OS/390 Workload Manager Implementation and Exploitation.* , IBM ITSO, Austin, 2004.

[9] T. Czachorski, *Analitical Queuing Model for Performance Evaluation of Computer Systems and Computer Networks.*, ProDialog, no. 16, Poznan 2003.

[10] *IBM AIX Technical Documentation: Performance Management Guide.*, *VMM page replacement tuning.*, 2008.

[11] *IBM AIX Technical Documentation: Performance Management Guide.*, *Tuning Logical Volume Striping.*, 2008.

[12] J. Jann, L. M. Browning, R. S. Burugula, *Dynamic reconfiguration: Basic building blocks for autonomic computing on IBM pSeries server.*, IBM Systems Journal, vol. 42, no. 1, 2003 Autonomic Computing.

[13] M. Mlynski, *Dynamic Resources Allocation in AIX 5L, Real Time Systems - Design and Applications.*, XII Conference of Real-Time Systems , Ustron 2005.

[14] M. Mlynski, *Analysis of Using An AIX Dynamic Resource Allocation Mechanism to Describe a Utility Level of Server in Oracle Data Bases Environment*, Studia Informatica, vol. 26 no. 3(64), Gliwice 2005.

[15] C. Matthys, M. Mlynski, N. Tollet, G. Barbati, H. Chauhan, B. Dierberger, R. Marchini, H. Wittmann, *Planning, Installing and Using the IBM Virtualization Version 2.1*, IBM ITSO Poughkeepsie (USA)2006.

[16] S. Wegrzyn, *Informatics as Specific Domain on Motion and Processing of Information.*, ProDialog, no. 16, Poznan 2003.

[17] S. Wegrzyn, *Resource Management in Grids.*, ProDialog, no. 16, Poznan 2003.

[18] *IOzone*, http://www.iozone.org

[19] G. Vallee, T. Naughton, L. S. Scott, *System Management Software for Virtual Environment.*, Proceedings of the 4th international conference on Computing frontiers, 2007.

Edited by: Marek Tudruj, Dana Petcu
Received: February 7th, 2009
Accepted: June 28th, 2009

Scalable Computing: Practice and Experience
Volume 10, Number 2, pp. 201–216. http://www.scpe.org
ISSN 1895-1767
© 2009 SCPE

HETEROPBLAS: A SET OF PARALLEL BASIC LINEAR ALGEBRA SUBPROGRAMS OPTIMIZED FOR HETEROGENEOUS COMPUTATIONAL CLUSTERS*

RAVI REDDY†, ALEXEY LASTOVETSKY†, AND PEDRO ALONSO‡

Abstract. This paper presents a software library, called Heterogeneous PBLAS (HeteroPBLAS), which provides optimized parallel basic linear algebra subprograms for Heterogeneous Computational Clusters. This library is written on the top of HeteroMPI and PBLAS whose building blocks, the de facto standard kernels for matrix and vector operations (BLAS) and message passing communication (BLACS), are optimized for heterogeneous computational clusters. This is the first step towards the development of a parallel linear algebra package for Heterogeneous Computational Clusters.

We show that the efficiency of the parallel routines is due to the most important feature of the library, which is the automation of the difficult optimization tasks of parallel programming on heterogeneous computing clusters. They are the determination of the accurate values of the platform parameters such as the speeds of the processors and the latencies and bandwidths of the communication links connecting different pairs of processors, the optimal values of the algorithmic parameters such as the data distribution blocking factor, the total number of processes, the 2D process grid arrangement, and the efficient mapping of the processes executing the parallel algorithm to the executing nodes of the heterogeneous computing cluster.

We present the user interface and the software hierarchy of the first research implementation of HeteroPBLAS. We demonstrate the efficiency of the HeteroPBLAS programs on a homogeneous computing cluster and a heterogeneous computing cluster.

Key words: parallel linear algebra, ScaLAPACK, HeteroMPI, heterogeneous platforms

1. Introduction. This paper presents a software library, called Heterogeneous PBLAS (HeteroPBLAS), which provides optimized parallel basic linear algebra subprograms (PBLAS) for Heterogeneous Computational Clusters. This library is written on top of HeteroMPI [1], and PBLAS [2] whose building blocks, the de facto standard kernels for matrix and vector operations (BLAS [3]) and message passing communication (BLACS [4]), are optimized for heterogeneous computational clusters.

We start with presentation of related works, which have produced solely heterogeneous parallel algorithms/strategies. There are two strategies of distribution of computations that can be used to implement PBLAS for Heterogeneous Computing Clusters (HCCs) [5]:

1. HeHo: heterogeneous distribution of processes over processors and homogeneous block distribution of data over the processes. This was implemented in mpC language [6, 7] with calls to ScaLAPACK [8];
2. HoHe: homogeneous distribution of processes over processors with each process running on a separate processor and heterogeneous block cyclic distribution of data over the processes. This was implemented in mpC language with calls to BLAS and LAPACK [9].

The HeHo strategy is a multiprocessing approach that is commonly used to accelerate ScaLAPACK programs on HCCs. The strategies are compared using Cholesky factorization on a HCC. The authors [5] show that for HCCs, the strategies HeHo and HoHe are more efficient than the traditional homogeneous strategy HoHo (homogeneous distribution of processes over processors with each process running on a separate processor and homogeneous block cyclic distribution of data over the processes implemented in ScaLAPACK). The main disadvantage of the HoHe strategy is non Cartesian nature of the data distribution, which is the distribution where each processor has to communicate with more than four direct neighbors. This leads to additional communications that can be crippling in the case of large networks. Moreover, the non Cartesian nature leads to poor scalability of the linear algebra algorithms that are proven scalable in the case of Cartesian data distribution and networks allowing parallel communications.

Beaumont et al. [10] present a proposal motivating provision of PBLAS in the form of a library for HCCs. The authors discuss HoHe strategies to implement matrix products and dense linear system solvers for HCCs and present them as a basis for a successful extension of the ScaLAPACK library to HCCs. They show that extending the standard ScaLAPACK block cyclic distribution to heterogeneous 2D grids is a difficult task. However, providing this extension is only the first step. The ScaLAPACK software must then be completely redesigned and rewritten, which is not a trivial effort.

*The research was supported by Science Foundation of Ireland (SFI).

†School of Computer Science and Informatics, University College Dublin ({manumachu.reddy, alexey.lastovetsky}@ucd.ie).

‡Department of Information Systems and Computation, Polytechnic University of Valencia (palonso@dsic.upv.es). Partially supported by Vicerrectorado de Investigación, Desarrollo e Innovación de la Universidad Politécnica de Valencia, and Generalitat Valenciana.

The multiprocessing strategy (HeHo), however, is easier to accomplish. It allows the complete reuse of high quality software such as ScaLAPACK on HCCs with no redesign efforts and provides good speedups. It can be summarized as follows:

1. The whole computation is partitioned into a large number of equal chunks;
2. Each chunk is performed by a separate process;
3. The number of processes run by each processor is proportional to its speed.

Thus, while distributed evenly across parallel processes, data and computations are distributed unevenly over processors of the HCC so that each processor performs the volume of computations proportional to its speed.

There have been several research contributions illuminating the merits of this strategy. Kishimoto and Ichikawa [11] use it to estimate the best processing element (PE) configuration and process allocation based on an execution time model of the application. The execution time is modeled from the measurement results of various configurations. Then, a derived model is used to estimate the optimal PE configuration and process allocation. Kalinov and Klimov [12] investigate the HeHo strategy where the performance of the processor is given as a function of the number of processes running on the processor and the amount of data distributed to the processor. They present an algorithm that computes optimal number of processes and their distribution over processors minimizing the execution time of the application. Cuenca et al. [13] analyze automatic optimization techniques in the design of parallel dynamic programming algorithms on heterogeneous platforms. The main idea is to determine automatically the optimal values of a number of algorithmic parameters such as number of processes, number of processors, and number of processes per processor.

Therefore, there has been a proliferation of research efforts presenting approaches to implement parallel solvers for HCCs but scientific software based on these approaches is virtually nonexistent. However, one can conclude from these research efforts that any software aspiring to provide automatically tuned parallel linear algebra programs for HCCs must automate the following essential and complicated tasks, which are also described in [14]:

1. Determination of the accurate values of platform parameters such as speeds of the processors, latencies and bandwidths of the communication links connecting different pairs of processors. These parameters compose the performance model of the executing heterogeneous platform;

2. Determination of the optimal values of algorithmic parameters such as the data distribution blocking factor, the total number of processes, and the 2D process grid arrangement to be used during the execution of the linear algebra kernel, and

3. Efficient mapping of the processes executing the parallel algorithm to the executing nodes of the HCC.

In this paper, we present a software library, called Heterogeneous PBLAS (HeteroPBLAS), which provides optimized PBLAS for HCCs. The design of the software library adopts the multiprocessing approach and thus reuses the PBLAS software (a component of ScaLAPACK) completely. The library also performs the automations of the aforementioned tedious and error prone tasks. This can be seen as the first step towards the development of a parallel linear algebra package for HCCs.

The rest of the paper is organized as follows. We start with an overview of the software in the HeteroPBLAS package. We describe the different software components and building blocks of the first research implementation as well as the user interface. In §3, we present the performance model of one of the PBLAS routines. The writing of the performance models formed the core development activity of the project. In §4, we explain how the optimal values of the algorithmic parameters are determined. In §5, we present the experimental results of execution of HeteroPBLAS programs on a homogeneous computing cluster and a heterogeneous computing cluster. We conclude the paper by outlining our future research goals.

2. Overview of Heterogeneous PBLAS. The flowchart of the main routines of HeteroPBLAS software library is shown in Fig. 2.1. The high level building blocks of HeteroPBLAS are HeteroMPI and PBLAS. The Parallel BLAS (PBLAS) is a parallel set of BLAS, which perform message passing and whose interface is as similar to BLAS as possible. The design goal of PBLAS was to provide specifications of distributed kernels, which would simplify and encourage the development of high performance and portable parallel numerical software, as well as providing manufacturers with a small set of routines to be optimized. These subprograms were used to develop parallel libraries such as ScaLAPACK, which is a well known standard package providing high performance linear algebra routines for distributed memory message passing MIMD computers supporting PVM and/or MPI.

ScaLAPACK uses block partitioned algorithms to ensure good performance on MIMD distributed memory concurrent computers, by minimizing the frequency of data movement between different levels of the memory hierarchy. The most fundamental building blocks of ScaLAPACK are the sequential BLAS, in particular the Level 2 and 3 BLAS, and the BLACS, which perform common matrix oriented communication tasks. The PBLAS are used as the highest level building blocks for implementing the ScaLAPACK library and provide the same ease of use and portability for ScaLAPACK that the BLAS provide for LAPACK.

HeteroMPI is an extension of MPI for programming high performance computations on HCCs. The main idea of HeteroMPI is to automate the process of selection of a group of processes, which would execute the parallel algorithm faster than any other group.

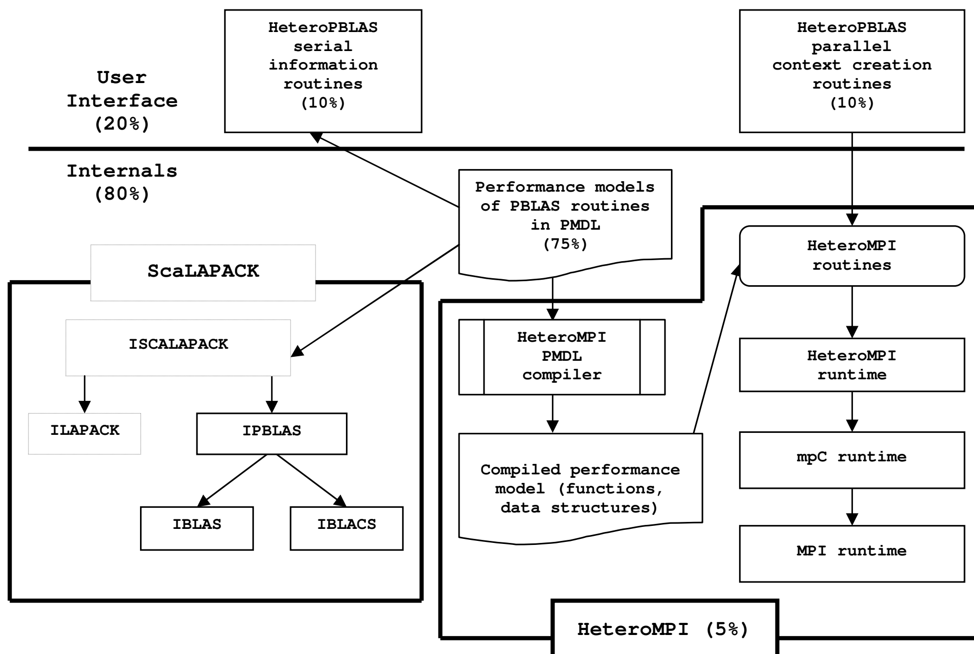


Fig. 2.1. *Flow of the HeteroPBLAS context creation routine call. The percentages give the breakup of HeteroPBLAS development efforts.*

The first step in this process of automation is the writing/specification of the performance model of the parallel algorithm. Performance model is a tool supplied to the programmer to specify his high level knowledge of the application that can assist in finding the most efficient implementation on HCCs. This model allows description of all the main features of the underlying parallel algorithm that affect the execution performance of parallel applications on HCCs. These features are:

1. The total number of processes executing the algorithm;
2. The total volume of computations to be performed by each of the processes in the group during the execution of the algorithm;
3. The total volume of data to be transferred between each pair of processes in the group during the execution of the algorithm;
4. The order of execution of the computations and communications by the parallel processes in the group, that is, how exactly the processes interact during the execution of the algorithm.

HeteroMPI provides a dedicated performance model definition language (PMDL) for writing this performance model. The model and the PMDL are borrowed from the mpC programming language. The PMDL compiler compiles the performance model written in PMDL to generate a set of functions, which make up the algorithm specific part of the HeteroMPI runtime system. These functions are called by the mapping algorithms of HeteroMPI runtime system to estimate the execution time of the parallel algorithm.

A typical HeteroMPI application contains HeteroMPI group management function calls to create and destroy a HeteroMPI group, and the execution of the computations and communications involved in the execution of the parallel algorithm employed in the application by the members of the group. The principal group constructor

routine HMPI_Group_auto_create determines the optimal number of processes and the optimal process grid arrangement, thereby relieving the application programmers from having to specify these parameters during the execution of the parallel application.

During the creation of a group of processes, the HeteroMPI runtime system solves the problem of selection of the optimal set of processes running on different computers of the HCC. It should be noted that this problem of mapping, in general, is NP-complete. The mapping algorithms used to solve the problem of selection of processes are explained in [1, 7]. The solution is based on the following:

1. The performance model of the parallel algorithm in the form of the set of functions generated by the compiler from the compilation of the performance model written in PMDL;

2. The performance model of the executing network of computers, which reflects the state of this network just before the execution of the parallel algorithm. This model considers the executing heterogeneous network as a multilevel hierarchy of interconnected sets of heterogeneous multiprocessors [7]. This model takes into account the material nature of communication links and their heterogeneity.

The principal routines in HeteroPBLAS software library are the context creation functions for the PBLAS routines. There is a context creation function for each PBLAS routine. It provides a context for the execution of the PBLAS routine but most importantly, performs the critical work of automating the difficult optimization tasks.

All the context creation routines have names of the form hscal_pxyyzz_ctxt. The second letter, x, indicates the data type. For example, d would mean double precision real data. The next two letters, yy, indicate the type of matrix (or of the most significant matrix). For example, ge would represent a general matrix. The last three letters zz indicate the computation performed. For example, the context creation function for the PDGEMM routine has an interface, which is shown below:

```
int hscal_pdgemm_ctxt(char* transa, char* transb,
    int * m, int * n, int * k, double * alpha, int * ia, int * ja,
    int * desca, int * ib, int * jb, int * descb, double * beta,
    int * ic, int * jc, int * descc, int * ictxt)
```

This function call returns a handle to a HeteroMPI group of MPI processes in ictxt and a return value of HSCAL_SUCCESS on successful execution. It differs from the PBLAS PDGEMM call in the following ways:

1. It returns a context but does not actually execute the PDGEMM routine;
2. The input arguments are the same as for the PDGEMM call except
 (i) The matrices A, B and C containing the data are not passed as arguments;
3. The output arguments differ as follows:
 (i) An extra return argument, ictxt, which contains the handle to a group of MPI processes that is subsequently used in the actual execution of the PDGEMM routine;
 (ii) A return value of HSCAL_SUCCESS indicating successful execution or otherwise an appropriate error code.

The function call is a collective operation and must be called by all the processes running in the HeteroPBLAS program. The context contains a handle to a HeteroMPI group of MPI processes, which tries to execute the PBLAS routine faster than any other group of processes. This context can be reused in multiple calls of the same routine or any routine that uses similar parallel algorithm as PDGEMM. The reader is referred to the HeteroPBLAS programmer's manual for more details of the user interface.

The HeteroPBLAS context creation/destruction routines call interface functions of HeteroMPI runtime system (the main routines being HMPI_Recon, HMPI_Timeof, HMPI_Group_auto_create). The HeteroPBLAS information functions calculate the total number of computations (arithmetical operations) performed by each process and the total number of communications in bytes between a pair of processes involved in the execution of the PBLAS routine. These routines are serial and can be called by any process. They do not actually execute the corresponding PBLAS routine but just calculate the total number of computations and communications involved. The block IPBLAS ('I' standing for instrumented) represents the instrumented code of PBLAS, which reuses the existing code base of PBLAS completely. The instrumentations made to it are (a) Wrapping of the parallel regions of the code in mpC par loops recognized by the PMDL compiler and (b) Replacement of the serial BLAS computation routines and the BLACS communication routines by calls to information functions, which return the number of arithmetical operations performed by each process and number of communications in bytes between different pairs of processes respectively.

The first step in the implementation of the context creation routine for a HeteroPBLAS routine is the writing of its performance model using the PMDL. The performance model definitions contain the instrumented code components. The HeteroMPI compiler compiles this performance model to generate a set of functions. During the creation of the context, the mapping algorithms of HeteroMPI runtime system call these functions to estimate the execution time of the PBLAS routine.

The writing of performance models of all the PBLAS routines has been the most laborious and intricate effort of this project. The key design issues were (a) *accuracy*, to facilitate accurate prediction of the execution time of the PBLAS routine, (b) *efficiency*, to execute the performance model in reasonable execution time, (c) *reusability*, as these performance models are to be used as building blocks for the performance models of ScaLAPACK routines, and (d) *preservability*, to preserve the key design features of underlying ScaLAPACK package. In the next section, we present one such performance model that demonstrates the tedium and accuracy of the writing process.

3. Performance model of PBLAS PDGEMM. The performance model definition of PDGEMM routine shown in Fig. 3.1 is used to demonstrate the complexity of the effort of writing a performance model. It describes the simplest case of parallel matrix–matrix multiplication of two dense square matrices A and B of size `n*n`. The reader is referred to [1, 7] for more details of the main constructs, namely `coord`, `parent`, `node`, `link`, and `scheme`, used in a description of a performance model. This definition is an extensively stripped down version of the actual definition, which can be studied from the package. The data distribution blocking factor `b` is assumed equal to the algorithmic blocking factor. Another simplification is that the matrices are divided such that `(n%(b*p))` and `(n%(b*q))` (see explanation of variables below) are both equal to zero.

```
/* 1 */ algorithm pdgemm(int n, int b, int bp, int bq, int p, int q) {
/* 2 */   coord I=p, J=q;
/* 3 */   node {I>=0 && J>=0:
/* 4 */         bench*((n/b)*((n/p)*(n/q)*(b))/((n/bp)*(n/bq)*(b)));};
/* 5 */   link (K=p, L=q)
/* 6 */   {
/* 7 */      I>=0 && J>=0 && J!=L:
/* 8 */        length*((n/p)*(n/q)*sizeof(double))
/* 9 */              [I, J]->[I, L];
/* 10 */     I>=0 && J>=0 && I!=K:
/* 11 */       length*((n/p)*(n/q)*sizeof(double))
/* 12 */             [I, J]->[K, J];
/* 13 */  };
/* 14 */  parent[0,0];
/* 15 */  scheme
/* 16 */  {
/* 17 */    int i, j, k;
/* 18 */    for(k = 0; k < n; k+=b)
/* 19 */    {
/* 20 */      par(i = 0; i < p; i++)
/* 21 */         par(j = 0; j < q; j++)
/* 22 */            if (j != ((k/b)%q))
/* 23 */              (100.0/(n/q)) %% [i,((k/b)%q)]->[i,j];
/* 24 */      par(i = 0; i < p; i++)
/* 25 */         par(j = 0; j < q; j++)
/* 26 */            if (i != ((k/b)%p))
/* 27 */              (100.0/(n/p)) %% [((k/b)%p),j]->[i,j];
/* 28 */      par(i = 0; i < p; i++)
/* 29 */        par(j = 0; j < q; j++)
/* 30 */          ((100.*(b/n)) %% [i,j];
/* 31 */    }
/* 32 */  };
/* 33 */ };
```

FIG. 3.1. *The performance model of the PDGEMM routine written in the performance model definition language.*

Line 1 is a header of the performance model declaration. It introduces the name of the performance model `pdgemm` parameterized with the scalar integer parameters `n`, `b`, `bp`, `bq`, `p`, and `q`. Parameter `n` is the size of square matrices A, B, and C. Parameter `b` is the size of the data distribution blocking factor. Parameters `bp` and `bq` are used in the execution of the benchmark code, which performs a matrix update of a matrix C_b of size `(n/bp)*(n/bq)` using A_b of size `(n/bp)*(b)` and B_b of size `(b)*(n/bq)`. The value of the parameter `bp` is the square root of the total number of processes available for computation. The value of parameter `bq` is equal to total number of processes divided by `bp`. Parameters `p` and `q` represent the number of processes along the row and the column in the process grid arrangement.

Line 3 is a *coordinate declaration* declaring the 2D coordinate system to which the processor nodes of the network are related. Line 4 is a *node declaration*. It associates the processes with this coordinate system to form a `p\times q` grid. It specifies the (absolute) volume of computations performed by each of the processes. The statement `bench` just specifies that as a unit of measurement, the volume of computation performed by some benchmark code be used. The benchmark code solves the core computational task, which is the matrix update representing the largest share of the computations performed by the processes at each step of the PBLAS routine. In other words, it is the costliest task performed by a process at each step of the PBLAS routine. In this case, the benchmark code used for estimation of speeds of processes performs a local DGEMM update of two dense `(n/bp)*b` and `b*(n/bq)` matrices where `b` is the optimal data distribution factor determined by the HeteroPBLAS runtime system (to follow in the next section). The total volume of computations performed by each process in the benchmark code is `(n/bp)*(n/bq)*b`. The total volume of computation performed by each process P_{IJ} at each step of the parallel algorithm is `(n/p)*(n/q)*b`. Since there are `n/b` steps in the parallel algorithm, the total volume would be `((n/b)*(n/p)*(n/q)*(b))`. The line 4 of node declaration describes the ratio of this total volume to the total volume of computations involved in the execution of the benchmark code. Lines 5–13 represent a link declaration. This specifies the links between the processes, the pattern of communication among the processes, and the total volume of data to be transferred between each pair of processes during the execution of the algorithm. Lines 7–9 of the link declaration describe horizontal communications related to matrix A. Obviously, processes from the same column of the process grid do not send each other elements of matrix A. Only processes from the same row of the process grid send each other elements of matrix A. Process P_{IJ} will send `(n/p)*(n/q)` number of elements of matrix A to process P_{IL}. So the total volume of data transferred (in bytes) from process P_{IJ} to process P_{IL} will be given by `(n/p)*(n/q)*sizeof(double)`.

Lines 10–13 of the link declaration describe vertical communications related to matrix B. Obviously, only processes from the same column of the process grid send each other elements of matrix B. In particular, process P_{IJ} will send all its elements of matrix B to all other processes from column J of the processor grid. That is, process P_{IJ} will send `(n/p)*(n/q)` number of elements of matrix B to process P_{KJ}. So the total volume of data transferred (in bytes) from process P_{IJ} to processor P_{KJ} will be given by `(n/p)*(n/q)*sizeof(double)`.

Line 15 introduces the *scheme declaration*. The `scheme` block describes how exactly processes interact during the execution of the algorithm. The scheme block is composed mainly of two types of units. They are computation and communication units. Each computation unit is of the form $e\%\%[i]$ specifying that e percent of the total volume of computations is performed by the process with the coordinates (i). Each communication unit is of the form $e\%\%[i[\rightarrow [j]$ specifying transfer of data from process with coordinates i to the process with coordinates j. There are two types of algorithmic patterns in the scheme declaration, which are sequential and parallel. The parallel algorithmic patterns are specified by the keyword `par` and they describe parallel execution of some actions (mixtures of computations and communications). The scheme declaration describes `(n/b)` successive steps of the algorithm. At each step `k`,

1. Lines 20–23 describe horizontal communications related to matrix A. A column of `b*b` blocks of matrix A is communicated horizontally. The number of elements transferred from P_{IJ} to P_{IL} at this step will be `(n/p)`. The total number of elements of matrix A, which process P_{IJ} sends to process P_{IL}, during the execution of the algorithm is `(n/p)*(n/q)`. Therefore, `(n/p)/((n/p)*(n/q))*100=(100./(n/q))` percent of all data that should be sent from process P_{IJ} to process P_{IL} will be sent at the step. The nested `par` statement in the main for loop of the `scheme` declaration specifies this fact. Again, we use the `par` algorithmic patterns in this specification to stress that during the execution of this communication, data transfer between different pairs of processes is carried out in parallel;

2. Lines 24–27 describe vertical communications related to matrix B. A row of `b*b` blocks of matrix B is communicated vertically. The number of elements transferred from P_{IJ} to P_{KJ} at this step will be `(n/q)`.

The total number of elements of matrix B, which process P_{IJ} sends to process P_{KJ}, during the execution of the algorithm is `(n/p)*(n/q)`. Therefore, `(n/q)/((n/p)*(n/q))*100=(100./(n/p))` percent of all data that should be sent from process P_{IJ} to process P_{KJ} will be sent at the step. The nested `par` statement in the main `for` loop of the `scheme` declaration specifies this fact. Again, we use the `par` algorithmic patterns in this specification to stress that during the execution of this communication, data transfer between different pairs of processes is carried out in parallel;

3. Lines 28–30 describe computations. Each process updates each of its `b*b` block of matrix C with one block from the pivot column and one block from the pivot row. At each of `(n/b)` steps of the algorithm, each process will perform `(100.*b/n)` percent of the volume of computations it performs during the execution of the algorithm. The third nested par statement in the main for loop of the scheme declaration just specifies this fact. The par algorithmic patterns are used here to specify that all processes perform their computations in parallel.

The simplest case of PDGEMM PBLAS routine just described demonstrates the complexity of the task of writing a performance model. There are altogether 123 such performance model definitions covering all the PBLAS routines. They can be found in the HeteroPBLAS package in the directory `/PBLAS/SRC` available at the URL [15]. The performance model files start with prefix `pm_` followed by the name of the PBLAS routine and have a file extension `mpc`.

```
int main(int argc, char **argv) {
   int nprow, npcol, pdgemmctxt, myrow, mycol, c__0 = 0;
/* Problem parameters */
   char *TRANSA, *TRANSB;
   int  *M, *N, *K, *IA, *JA, *DESCA, *IB, *JB, *DESCB, *IC, *JC,
        *DESCC;
   double *ALPHA, *A, *B, *BETA, *C;
/* Initialize the heterogeneous PBLAS runtime */
   hscal_init(&argc, &argv);
/* Get the heterogeneous PDGEMM context */
   hscal_pdgemm_ctxt(TRANSA, TRANSB, M, N, K, ALPHA, IA, JA, DESCA,
                     IB, JB, DESCB, BETA, IC, JC, DESCC, &pdgemmctxt);
   if (!hscal_in_ctxt(&pdgemmctxt))
      hscal_finalize(c__0);
/* Retrieve the process grid information */
   Cblacs_gridinfo(pdgemmctxt, &nprow, &npcol, &myrow, &mycol);
/* Initialize the array descriptors for the matrices A, B and C */
   descinit_(DESCA, \dots , &pdgemmctxt);  /* for Matrix A */
   descinit_(DESCB, \dots , &pdgemmctxt);  /* for Matrix B */
   descinit_(DESCC, \dots , &pdgemmctxt);  /* for Matrix C */
/* Distribute matrices on the process grid using user-defined pdmatgen */
   pdmatgen_(&pdgemmctxt, \dots );         /* for Matrix A */
   pdmatgen_(&pdgemmctxt, \dots );         /* for Matrix B */
   pdmatgen_(&pdgemmctxt, \dots );         /* for Matrix C */
/* Call the PBLAS pdgemm routine */
   pdgemm_(TRANSA, TRANSB, M, N, K, ALPHA, A, IA, JA, DESCA, B, IB,
           JB, DESCB, BETA, C, IC, JC, DESCC);
/* Release the heterogeneous PDGEMM context */
   hscal_free_ctxt(&pdgemmctxt);
/* Finalize the Heterogeneous PBLAS runtime */
   hscal_finalize(c__0);
}
```

Fig. 4.1. *HeteroPBLAS program employing PBLAS PDGEMM.*

4. Determination of the optimal algorithmic parameters. Fig. 4.1 shows the essential steps involved in calling the PBLAS PDGEMM routine in a HeteroPBLAS program. The HeteroPBLAS runtime is initialized using the operation `hscal- _init`. The heterogeneous PDGEMM context is obtained using the context constructor routine `hscal_pdgemm_ctxt`. The function call `hscal_in_ctxt` returns a value of 1 for the processes

chosen to execute the PDGEMM routine, otherwise 0. Then the homogeneous PBLAS PDGEMM routine is executed. The heterogeneous PDGEMM context is freed using the context destructor operation `hscal_free_ctxt`. When all the computations are completed, the program is exited with a call to `hscal_finalize`, which finalizes the HeteroPBLAS runtime.

The PDGEMM context constructor routine `hscal_pdgemm_ctxt` is the main function automating the difficult optimization tasks of parallel programming on HCCs. They are the determination of the accurate values of the platform parameters such as the speeds of the processors and the latencies and bandwidths of the communication links connecting different pairs of processors, the optimal values of the algorithmic parameters such as the data distribution blocking factor, the total number of processes, the 2D process grid arrangement, and efficient mapping of the processes executing the parallel algorithm to the executing nodes of the HCC.

4.1. Data distribution blocking factor. The context constructor routine `hscal_pdgemm_ctxt` uses the HeteroMPI function `HMPI_Timeof`, as shown below, to determine the optimal value of the data distribution blocking factor.

```
double time, min_time = DBL_MAX; void *model_params;
for (b = 1; b <= n; b++) {
   // Fill the parameters to the performance model
   model_params[0] = n;  model_params[1] = b;
   model_params[2] = bp; model_params[3] = bq;
   model_params[4] = p;  model_params[5] = q;
   // Refresh the speeds of the processors
   HMPI_Recon(&hscal_dgemm_benchmark, model_params);
   if (HMPI_Is_host()) {
      time = HMPI_Timeof(&hscal_model_pdgemm, model_params);
      if (time < min_time)
         opt_b = b; min_time = time;
   }
}
```

The function `HMPI_Timeof` is used to estimate the execution time of the algorithm on the underlying hardware without its real execution. This local operation can be called by any process, which is a member of the group associated with the predefined communication universe of HeteroMPI. The parameters to this function are the handle, `hscal_model_pdgemm`, generated from the compilation of the performance model of the PDGEMM routine, and the parameters to this performance model. As one can see from the code snippet, this estimation is performed for each possible set of values to the parameters to the performance model. Using the execution time predicted for each set, the optimal value can be determined, which would be the one resulting in minimum estimated execution time.

The estimation is based on the performance model of the PDGEMM routine and the performance model of the executing network of computers, which reflects the state of this network just before the execution of the PDGEMM routine. The function `HMPI_Recon` is used to update dynamically the estimation of processor speeds at runtime. This is a collective operation and must be called by all the processes running in the HeteroPBLAS application. The performance model of the executing network of computers is summarized as follows:

1. The performance of each processor is characterized by the execution time of the same serial code (takes place during the execution of `HMPI_Recon`)

(i) The serial code is provided by the application programmer;

(ii) It is supposed that the code is representative for the computations performed during the execution of the application;

(iii) The code is performed at runtime in the points of the application specified by the application programmer. Thus, the performance model of the processors provides current estimation of their speed demonstrated on the code representative for the particular application.

In this case, the serial code, `hscal_dgemm_benchmark`, performs a local DGEMM update of `(N/bp)`×`b` and `b`×`(N/bq)` matrices where `b` is the data distribution blocking factor;

2. The communication model [7] is seen as a hierarchy of homogeneous communication layers. Each is characterized by the latency and bandwidth. Unlike the performance model of processors, the communication

model is static, a shortcoming that would be addressed in our future work. Its parameters are obtained during the initialization of the Heterogeneous ScaLAPACK runtime and are not refreshed later.

The estimation procedure is explained in detail in [7] and is summarized here. The time of execution for each mapping, $\mu : I \rightarrow C$, where I is a set of the processes of the group, and $C = \{c_0, c_1, \ldots, c_{M-1}\}$ is a set of computers of the executing network, is estimated. The estimation time for the optimal mapping, which would ensure the fastest execution of the parallel algorithm, is returned. In general, for accurate solution of this problem as many as M^K possible mappings have to be probated to find the best one (here, K is the number of processes of the group). Obviously, that computational complexity is not acceptable for a practical algorithm that should be performed at runtime. Therefore, the HeteroMPI runtime system searches for some approximate solution that can be found in some reasonable interval of time, namely, after probation of $M \times K$ possible mappings instead of M^K.

Each computation unit in the `scheme` declaration of the form $e\%\%[i]$ is estimated. Each communication unit of the form $e\%\%[i] \rightarrow [j]$ specifying transfer of data from virtual processor with coordinates i to the virtual processor with coordinates j is estimated. Simple calculation rules are used to estimate the sequential algorithmic patterns in the `scheme` declaration. For example, the estimation of the pattern

```
for (e1; e2; e3) a
```

is calculated as follows:

```
for (T=0, e1; e2; e3)
   T += time taken to execute action a
```

The rules just reflect semantics of the corresponding serial algorithmic patterns.

The rule to estimate time for a parallel algorithmic pattern

```
par (e1; e2; e3) a
```

is more complicated and is explained in detail in [7]. This is the core of the entire mapping algorithm determining its accuracy and efficiency. It takes into account material nature and heterogeneity of both processors and network equipment. It relies on fairly allocating processes to processors in a shared memory multiprocessor normally implemented by operating systems for processes of the same priority (HeteroMPI processes are just the case). At the same time, it proceeds from the pessimistic point of view when estimating workload of different processors of that multiprocessor. Estimation of communication cost by the rule is sensitive to scalability of the underlying network technology. It treats differently communication layers serializing data packages and supporting their parallel transfer. The most typical and widely used collective communication operations are treated specifically to provide better accuracy of the estimation of their execution time. An important advantage of the rule is its relative simplicity and effectiveness. The effectiveness is critical because the algorithm is supposed to be multiply executed at runtime.

This procedure to determine the optimal value of the data distribution blocking factor is now performed during the installation of the HeteroPBLAS software rather than for every call of the context constructor routine for performance reasons.

4.2. Two dimensional process grid arrangement. The optimal value of this algorithmic parameter represents all of the following: the optimal values for the total number of processes, the number of process rows, the number of process columns, and efficient mapping of the processes to the executing computers of the HCC. The context constructor routine `hscal_pdgemm_ctxt` calls the HeteroMPI function `HMPI_Group_pauto_create` to determine the optimal values. Its operation employing HeteroMPI calls is shown below:

```
int i, k, pa, p, opt_p, q, opt_q, *a, terminate = 0;
double t, mint=DBL_MAX; void *model_params = NULL;
// The host process
if (HMPI_Is_host()) {
   // The total number of processes available for computation
   int np = HMPI_Group_size(HMPI_COMM_WORLD_GROUP);
   for (k=np; k>=1; k--) {
      // The possible two dimensional process arrangements (p,q)
      HMPI_Get_2D_process_arrangements(k, &pa, &a);
      // For each two dimensional process arrangement (p,q)
      for (i=0; i<pa; i++) {
         // Fill the parameters to the performance model
         // Estimate the execution time for each process arrangement (p,q)
```

```
            t = HMPI_Timeof(&hscal_model_pdgemm, model_params);
            if (t < mint)
               // A better process arrangement found, continue the algorithm
               terminate = 0; mint = t; opt_p = p; opt_q = q;
         }
         if (terminate) { break; }
         terminate=1;
      }
      // Create a HeteroMPI group of processes with the optimal values of
      // algorithmic parameters
      model_params[0] = n;  model_params[1] = opt_b;
      model_params[2] = bp; model_params[3] = bq;
      model_params[4] = opt_p;  model_params[5] = opt_q;
   }
   // Create a HeteroMPI group of processes
   HMPI_Group_create(gid, model_params);
   return HMPI_SUCCESS;
```

The algorithm can be summarized as follows: The number of steps of the algorithm is represented by the variable `k`. For `k=1`, the total number of processes available for computation, `np`, is determined. All the possible two dimensional process arrangements, (`p`,`q`), whose product is `np`, are obtained using the function, `HMPI_Get_2D_pro- cess_arrangements`. For example, if the total number of processes available is 25, then the possible two dimensional process arrangements whose product of the coordinates is 25 are {(1,25), (5,5), (25,1)}.

Each such process arrangement, (`p`,`q`), is filled into the array of parameters to the performance model of the PDGEMM routine. The function call `HMPI_Timeof` is then invoked to estimate the execution time of the algorithm. One of the inputs to this function call is the handle, `hscal_model_pdgemm`, which encapsulates all the features of the performance model in the form of a set of functions generated by the compiler from the description of the performance model of the PDGEMM routine. The function call `HMPI_Timeof` invokes the mapping algorithms of the HeteroPBLAS runtime system to select such a mapping that is estimated to ensure the fastest execution of the parallel algorithm for that process arrangement. The selection process is described in detail in [1, 7]. It is based on the performance model of the PDGEMM routine and the performance model of the executing network of computers, which reflects the state of the network just before the execution of the PDGEMM routine. During the selection process, the HeteroPBLAS runtime system searches for some approximate solution that can be found in some reasonable interval of time by probation of a subset of all possible mappings. From the execution times predicted for all the possible process arrangements, the process arrangement, (`opt_p`, `opt_q`), that results in the least estimated time of execution of the algorithm is determined.

For the next step, the total number of processes is decremented by one. The possible two dimensional process grid arrangements are again obtained and evaluated using the function `HMPI_Timeof`. The algorithm continues if a process arrangement is found for which the estimated execution time is less than the estimated execution time of the process arrangement (`opt_p`, `opt_q`) determined in the previous step. Otherwise, the algorithm terminates.

The optimal values of the blocking factor and the 2D process grid arrangement (`opt_p`, `opt_q`) are then passed as performance model parameters to the function call, `HMPI_Group_create`, which creates a HeteroMPI group of MPI processes that participate in the execution of the parallel application. These processes become members of the heterogeneous PDGEMM context. This function call is a collective operation and must be called by all the processes available for computation in the predefined communication universe of HeteroMPI.

Heuristics are used to reduce the number of process arrangements evaluated. For example, one dimensional arrangements are not evaluated in the case of matrix–matrix multiplication on Ethernet where it can be proved using simple formulas [16] that they perform poorly compared to two dimensional arrangements and do not minimize the objective function for networks with either sequential communications or parallel communications.

5. Experimental results. We present four sets of experiments. The first set of experiments is run on a homogeneous computing cluster 'Grig' (`https://www.cs.utk\-.edu/help/doku.php?id=clusters:grig`) consisting of 32 Linux nodes with 2 processors per node with Myrinet interconnect. The processor type is Intel EM64T. The software used is heterompi-1.2.0, MPICH-1.2.7, ScaLAPACK-1.8.0 and ATLAS, which is an optimized BLAS library.

We chose two level3 routines, PDGEMM and PDTRSM, for demonstration because they exhibit two different algorithmic patterns. In the case of PDGEMM, the size of the problem solved at each step of its execution, that is number of updates of the resulting matrix, is constant whereas in the execution of PDTRSM, the size of the problem (number of updates of the trailing submatrix) decreases with each step.

The speedup, which is shown in the figures, is calculated as the ratio of the execution time of the homogeneous PBLAS program and the execution time of the HeteroPBLAS program. Dense matrices of size $N \times N$ and vectors of size N were used in the experiments. The homogeneous PBLAS program uses the default parameters suggested by the recommendations from the ScaLAPACK user's guide, which are to (a) use the best BLAS and BLACS libraries available, (b) use a data distribution block size of 64, (c) use a square processor grid and (d) execute no more than one process per processor.

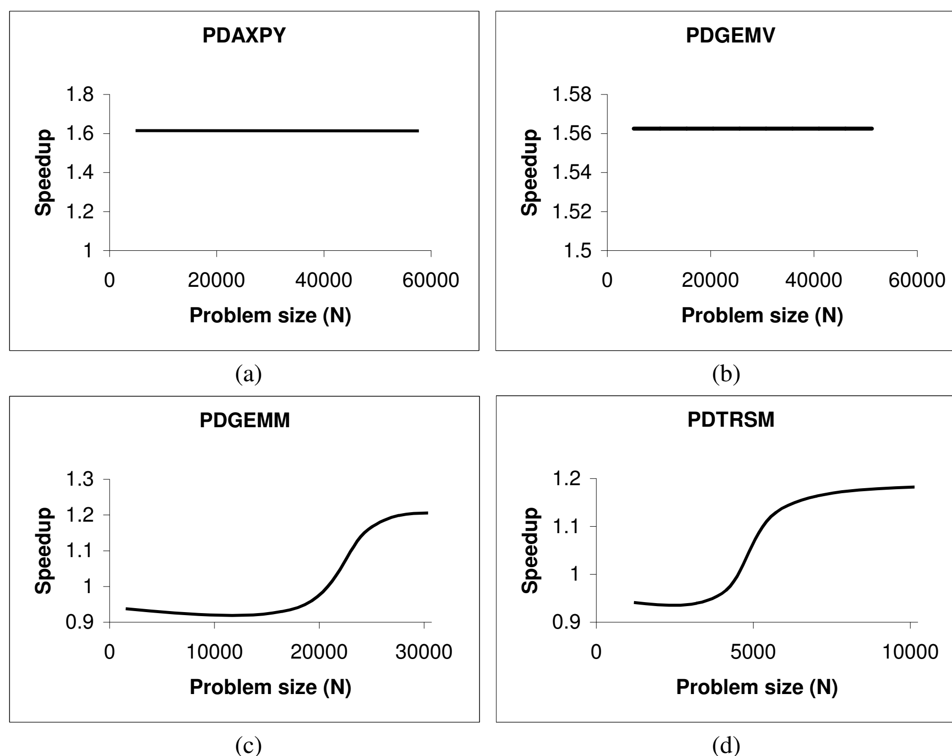


FIG. 5.1. *Experimental results on the homogeneous Grig cluster. N is the size of the vector/matrix. (a) Results of PDAXPY. (b) Results of PDGEMV. (c) Speedup of PDGEMM. (d) Speedup of PDTRSM.*

The first set of experiments is composed of two parts. Fig. 5.1 shows the experimental results of the first part. Fig. 5.1(a) and Fig. 5.1(b) show the experimental results from the execution of the PBLAS level1 routine PDAXPY and level2 routine PDGEMV on the homogeneous cluster. The homogeneous PBLAS programs use a 1×64 grid of processes (using one process per processor configuration). Fig. 5.1(c) and Fig. 5.1(d) shows the experimental results from the execution of the PBLAS level3 routines PDGEMM and PDTRSM, respectively. The homogeneous PBLAS programs use an 8×8 grid of processes (using one process per processor configuration). In the second part, we used the optimal data distribution blocking factor and the optimal process grid arrangement, determined by the HeteroPBLAS program, in the execution of the corresponding homogeneous PBLAS program. From both the parts, it was observed that there is no discernible overhead during the execution of HeteroPBLAS programs. The maximum overhead of about 7% incurred in the case of level3 routines occurs during the creation of the context. The execution times of HeteroPBLAS programs for level1 and level2 routines are the same if one process is executed per computer/node and not per processor. In the case of first part, one can notice that the HeteroPBLAS programs perform better than the homogeneous PBLAS programs. This is because the homogeneous PBLAS programs use the default parameters (recommendations from the user's guide) but not the optimized parameters whereas the HeteroPBLAS programs use the optimal algorithmic parameters such as the optimal blocking factor and the optimal process arrangement.

For equitable comparison, the PBLAS programs must be optimized for the HCC. However one of the goals of this paper is to show how the HeteroPBLAS software automates this procedure of optimization and therefore we use pure/unoptimized ScaLAPACK available online. However, as one must have observed by now, the process of optimization is not trivial.

The second set of experiments is run on a small heterogeneous local network of sixteen different Linux workstations (hcl01–hcl16) whose specifications can be read at the URL `http://hcl.ucd.ie/Hardware/Cluster+Specifications`. The network is based on 2 Gbit Ethernet with a switch enabling parallel communications between the computers. The software used is MPICH-1.2.5, ScaLAPACK–1.8.0, and ATLAS.

The HeteroPBLAS program uses the multiprocessing approach, where more than one process is run on each processor. The number of processes to run on each processor during the program startup is determined automatically by the HeteroPBLAS command line interface tools. A simple heuristic used is the heterogeneity of the network. The heterogeneity of the network due to the heterogeneity of the processors is calculated as the ratio of the absolute speed of the fastest processor to the absolute speed of the slowest processor. For example, consider the benchmark code of a local DGEMM update of two matrices 2560×128 and 128×2560, the absolute speeds of the processors hcl01–hcl16 in million flop/s performing this update are {10829, 9729, 11054, 11151, 11211, 11295, 10958, 10949, 7425, 7047, 10827, 11200, 9056, 11626, 9815, 12250}. As one can see, hcl16 is the fastest processor and hcl10 is the slowest processor. The heterogeneity in this case ≈ 2. The command line tools, therefore, would run two processes on each processor during the program startup.

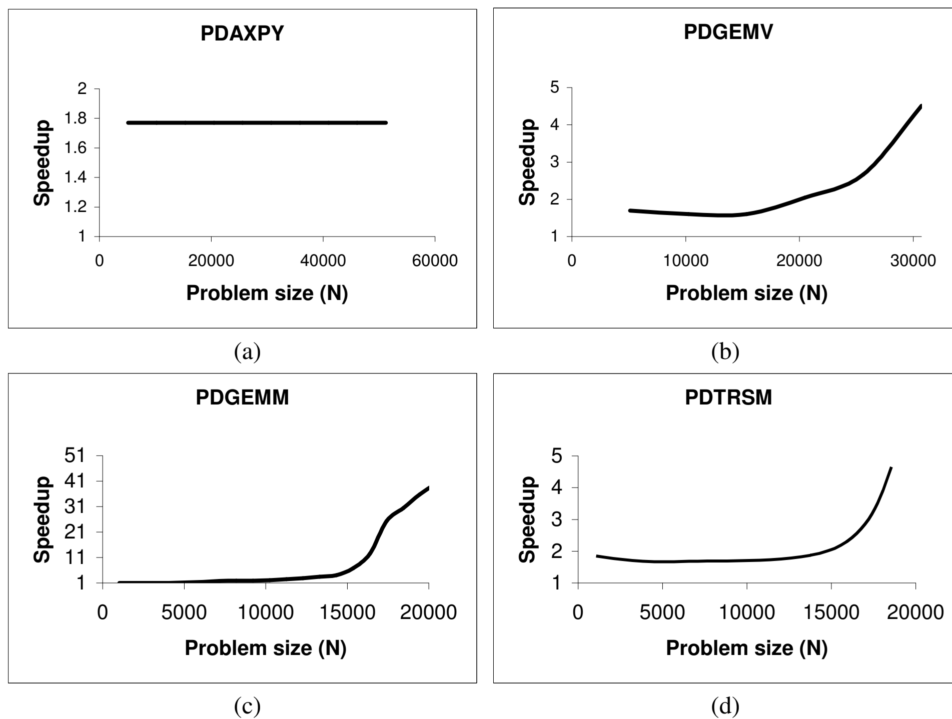


Fig. 5.2. *Experimental results on the heterogeneous cluster. N is the size of the vector/matrix. (a) Speedup of PDAXPY. (b) Speedup of PDGEMV. (c) Speedup of PDGEMM. (d) Speedup of PDTRSM.*

Fig. 5.2(a) and Fig. 5.2(b) show the experimental results from the execution of the PBLAS level1 routine PDAXPY and level2 routine PDGEMV. The homogeneous PBLAS programs use a 1×25 grid of processes (using one process per processor configuration). Fig. 5.2(c) and Fig. 5.2(d) show the experimental results from the execution of the PBLAS level3 routines PDGEMM and PDTRSM respectively. The homogeneous PBLAS program uses a 5×5 grid of processes (using one process per processor configuration).

There are a few reasons behind the superlinear speedups achieved in the case of PDGEMM and eventually for very large problem sizes in the case of PDTRSM not shown in the figure.

The first reason is the better load balance achieved through proper allocation of processes involved in the execution of the algorithm to the processors. During the creation of a HeteroMPI group of processes in

the context creation routine, the mapping of the parallel processes in the group is performed such that the number of processes running on each processor is as proportional to its speed as possible. In other words, while distributed evenly across parallel processes, data and computations are distributed unevenly over processors of the heterogeneous network, and this way each processor performs the volume of computations as proportional to its speed as possible. In the case of execution of PDGEMM on HCCs, it can be seen that for problem sizes larger than 5120, more than 25 processes must be involved in the execution to achieve good load balance. Since only 25 processes are involved in the execution of the homogeneous PBLAS program, good load balance is not achieved. However just running more than 25 processes in the execution of the program would not resolve the problem. This is because in such a case, the optimal process arrangement and the efficient mapping of the processes to the executing computers of the underlying network must also be determined. This is a complex task automated by HeteroMPI. Secondly, the optimal values of the algorithmic parameters that are automatically determined by the HeteroPBLAS library. One of the key algorithmic parameters is the data distribution blocking factor. There is a tradeoff between load imbalance and communication startup cost, which can be controlled by varying the blocking factor, b. The optimal values of the blocking factor determined by the HeteroPBLAS library are in the range ($128 \leq b \leq 256$) as shown in Fig. 5.3. It uses the value of 128. The procedure to determine the optimal block size is performed during the installation of the software.

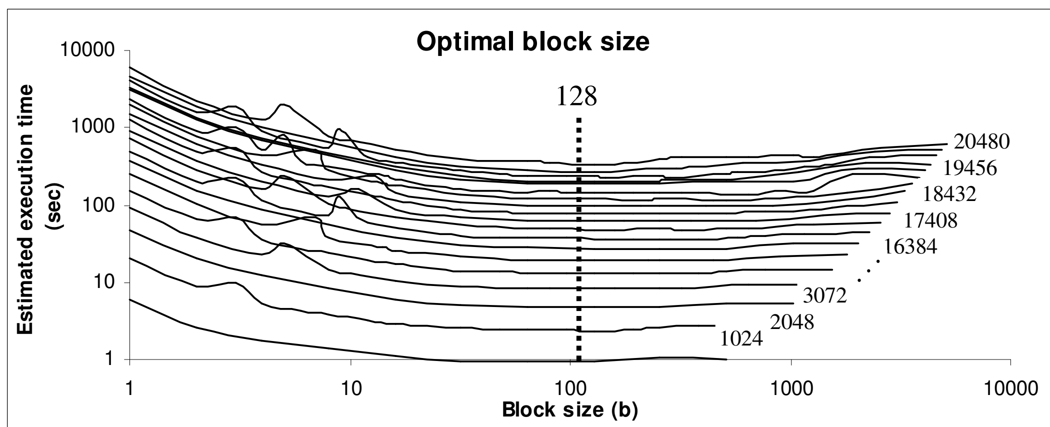


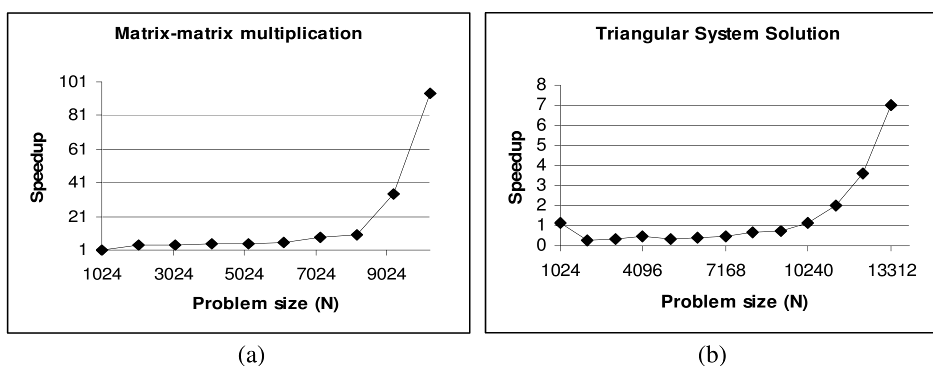
FIG. 5.3. *Optimal block size estimated by the HeteroPBLAS library for various problem sizes ($N \leq 20480$). The execution times are on a log scale.*

Finally, the optimal values of the 2D grid arrangement of processes (p,q) address the load imbalance caused by the computational "hot spots" where certain processes have more work to do between synchronization points than others. There is an optimal 2D process grid arrangement, (p,q), which depends on the communications characteristics of the network and determines the overlap of the communication with the computation. During the creation of a HeteroMPI group of processes in the context creation routine, the function `HMPI_Group_pauto_create` estimates the time of execution of the algorithm for each process arrangement evaluated. For each such estimation, it invokes the runtime mapping algorithm, which tries to arrange the processes along a 2D grid to optimally load balance the work of the processors. It returns the process arrangement that results in the least estimated time of execution of the algorithm. Table 5.1 shows the optimal 2D process grid arrangements determined by the context creation routines during the execution of the HeteroPBLAS programs. The total execution time of the HeteroPBLAS program is the sum of the context creation time and the algorithm execution time. It can be seen that for small problem sizes (for example, in the case for PDGEMM where $N \leq 4096$), the context creation times are large compared to the execution times of the parallel algorithms due to the evaluation of a large number of process arrangements. This number can be reduced using meaningful heuristics, which will be a subject for our future study.

Fig. 5.4 shows the execution times of sequential applications and HeteroPBLAS applications solving the same matrix-matrix multiplication and triangular system of equations. The sequential application calls optimized BLAS routines and is executed on the fastest processor (hcl16). The sequential applications performing matrix-matrix multiplication and solving triangular system of equations fail for problem sizes ($N \geq 10240$) and ($N \geq 13312$), respectively. It can be seen that the sequential program solving the triangular system

TABLE 5.1
Optimal 2D process grid arrangements (p,q) *determined during the execution of the context creation routines.*

Size of the matrix (N)	HeteroPBLAS PDGEMM Program			HeteroPBLAS PDTRSM Program		
	Predicted (p,q)	Context Creation Time (sec)	Execution Time (sec)	Predicted (p,q)	Context Creation Time (sec)	Execution Time (sec)
1024	(2,6)	1.8	0.34	(2,6)	0.5	0.27
2048	(2,6)	2	1	(2,7)	1	7
3072	(2,6)	2	3	(2,7)	1	21
4096	(2,6)	2	6	(2,7)	2	35
5120	(2,7)	2	11	(2,7)	2	80
6144	(2,7)	2	17	(2,7)	3	117
7168	(2,7)	2	27	(2,7)	3	160
8192	(2,7)	3	44	(2,7)	3	208
9216	(2,7)	3	48	(2,7)	4	307
10240	(2,7)	3	65	(2,7)	5	399
11264	(3,7)	11	106	(2,7)	5	491
12288	(3,7)	11	117	(2,7)	5	628
13312	(3,7)	11	164	(2,7)	6	737
14336	(3,7)	12	222	(2,7)	7	853
15360	(3,7)	12	385	(2,7)	8	919
16384	(3,7)	12	754	(2,7)	8	1285
17408	(3,7)	13	1204	(2,7)	9	1358
18432	(3,7)	13	1844	(3,7)	29	2033
19456	(2,14)	13	2729	(3,7)	33	2296
20480	(2,14)	28	4275	(3,11)	125	3017



(a) (b)

FIG. 5.4. *Speedup of the HeteroPBLAS programs over the sequential programs.*

(DTRSM) performs better than the HeteroPBLAS (and therefore PBLAS) program for the range of problem sizes ($N \leq 10240$). It is quite challenging work to find a procedure which automatically determines which program to use (either sequential or HeteroPBLAS) for this range of problem sizes. This is because HeteroPBLAS programs use performance models based on the PBLAS (ScaLAPACK) algorithms, which are clearly not accurate for small problem sizes. This is a subject for our future research.

The final set of experiments shown in Fig. 5.5 demonstrates the efficiency of the HeteroPBLAS program employing the level3 PDGEMM routine. Its efficiency is compared to that of the HeteroMPI program, which adopts the HoHe strategy using heterogeneous 2D block cyclic distribution of matrices [5]. We use the experimental approach to analysis of the performance of heterogeneous algorithms presented in [17]. The HeteroMPI program is close to optimal on the heterogeneous computing cluster. Since the execution time of the HeteroPBLAS program is practically the same as the HeteroMPI program, we can conclude that the efficiency of the HeteroPBLAS program is also close to optimal on this network.

6. Conclusions and future work. In this paper, we present a software library, called Heterogeneous PBLAS (HeteroPBLAS), which provides optimized parallel basic linear algebra subprograms for Heterogeneous Computational Clusters.

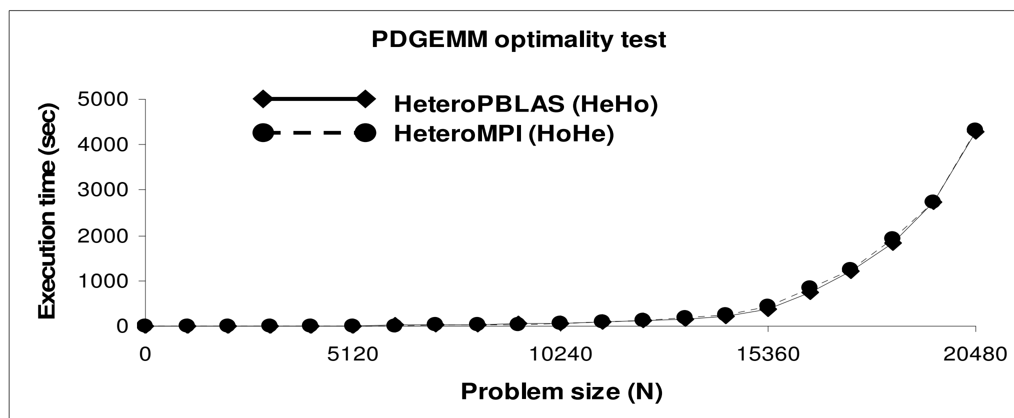


FIG. 5.5. *Execution times of the HeteroPBLAS and the HeteroMPI programs on the heterogeneous cluster. N is the size of the matrix. HeteroMPI program employs heterogeneous 2D block cyclic distribution of matrices.*

We show that the efficiency of the parallel routines is due to the most important feature of the library, which is the automation of the difficult optimization tasks of parallel programming on heterogeneous computing clusters. These tasks are the determination of the accurate values of the platform parameters such as the speeds of the processors and the latencies and bandwidths of the communication links connecting different pairs of processors, the optimal values of the algorithmic parameters such as the data distribution blocking factor, the total number of processes, the 2D process grid arrangement, and the efficient mapping of the processes executing the parallel algorithm to the executing nodes of the heterogeneous computing cluster.

We presented the user interface and the software hierarchy of the first prototype implementation of HeteroPBLAS. Our future work would involve testing of the software on multicore platforms. The software would then be integrated into the Heterogeneous ScaLAPACK package providing high performance routines for dense linear systems, least squares problems, and eigenvalue problems.

REFERENCES

[1] A. Lastovetsky and R. Reddy, *HeteroMPI: Towards a Message-Passing Library for Heterogeneous Networks of Computers*, Journal of Parallel and Distributed Computing (JPDC), Volume 66, No. 2, pp.197–220, Elsevier, 2006.
[2] *Parallel Basic Linear Algebra Subprograms* (PBLAS). `http://www.netlib.org/scalapack/ pblas_qref.html`.
[3] *Basic Linear Algebra Subprograms (*BLAS*)*. `http://www.netlib.org/blas`.
[4] *Basic Linear Algebra Communication Subprograms (*BLACS*)*. `http://www.netlib.org/blacs`.
[5] A. Kalinov and A. Lastovetsky, *Heterogeneous Distribution of Computations Solving Linear Algebra Problems on Networks of Heterogeneous Computers*, Journal of Parallel and Distributed Computing, Volume 61, No. 4, pp. 520–535, April 2001.
[6] A. Lastovetsky, D. Arapov, A. Kalinov, and I. Ledovskih, *A Parallel Language and Its Programming System for Heterogeneous Networks*, Concurrency: Practice and Experience, Volume 12, No. 13, pp. 1317–1343, November 2000.
[7] A. Lastovetsky, *Adaptive Parallel Computing on Heterogeneous Networks with mpC*, Parallel Computing, Volume 28, No. 10, pp. 1369–1407, October 2002.
[8] *Scalable* LAPACK. `http://www.netlib.org/scalapack/`.
[9] *Linear Algebra PACKage (*LAPACK*)*. `http://www.netlib.org/lapack/`.
[10] O. Beaumont, V. Boudet, A. Petitet, F. Rastello, and Y. Robert, *A Proposal for a Heterogeneous Cluster ScaLAPACK (Dense Linear Solvers)*, IEEE Transactions on Computers, Volume 50, No. 10, pp. 1052–1070, October 2001.
[11] Y. Kishimoto and S. Ichikawa, *An Execution–Time Estimation Model for Heterogeneous Clusters*, In 13th Heterogeneous Computing Workshop (HCW 2004), Proceedings of 18th International Parallel and Distributed Processing Symposium (IPDPS'04), IEEE Computer Society (2004).
[12] A. Kalinov and S. Klimov, *Optimal mapping of a parallel application processes onto heterogeneous platform*, 14th Heterogeneous Computing Workshop (HCW 2005), Proceedings of 19th International Parallel and Distributed Processing Symposium (IPDPS'05), IEEE Computer Society (2005).
[13] J. Cuenca, D. Giménez, and J. P. Martinez, *Heuristics for work distribution of a homogeneous parallel dynamic programming scheme on heterogeneous systems*, Parallel Computing, Volume 31, No. 7, pp. 711–735, Elsevier, 2006.
[14] A. Lastovetsky, *Scientific Programming for Heterogeneous Systems—Bridging the Gap between Algorithms and Applications*, Proceedings of the 5th International Symposium on Parallel Computing in Electrical Engineering (PARELEC 2006), Bialystok, Poland, IEEE Computer Society Press, pp. 3–8, 13–17 Sept 2006.
[15] *Heterogeneous* ScaLAPACK. `http://hcl.ucd.ie/project/HeteroScaLAPACK/`.

[16] O. Beaumont, V. Boudet, F. Rastello, and Y. Robert, *Matrix Multiplication on Heterogeneous Platforms*, IEEE Transactions on Parallel and Distributed Systems, Volume 12, No. 10, pp. 1033–1051, 2001.
[17] A. Lastovetsky and R. Reddy, *On Performance Analysis of Heterogeneous Parallel Algorithms*, In Parallel Computing, Volume 30, No. 11, pp. 1195–1216, 2004.

Edited by: Marek Tudruj, Dana Petcu
Received: February 7th, 2009
Accepted: June 28th, 2009

Scalable Computing: Practice and Experience
Volume 10, Number 2, pp. 217–228. http://www.scpe.org

ISSN 1895-1767
© 2009 SCPE

POWER-AWARE SPEED-UP FOR MULTITHREADED NUMERICAL LINEAR ALGEBRAIC SOLVERS ON CHIP MULTICORE PROCESSORS

JAYANTA MUKHERJEE* AND SOUMYENDU RAHA†

Abstract. With the advent of multicore chips new parallel computing metrics and models have become essential for redesigning traditional scientific application libraries tuned to a single chip. In this paper we evolve metrics specific to generalized chip multicore processors (CMP) and use them for parallel performance modeling of numerical linear algebra routines that are commonly available as shared object libraries tuned to single processor chip. The study uses a thread parallel model of parallel computing on CMPs. POSIX threads (pthread) have been used due to the wide acceptance and availability. The shortcoming of the POSIX threads for numerical linear algebra in terms of data distribution has been overcome by tuning algorithms so that a particular thread will operate on a specific portion of the matrix. The paper studies tuned implementations of the conventional a few parallel linear algebra method as examples on a generalized CMP model. For formulating a speed-up metric, this work takes into consideration the power consumption and the effect of memory cache hierarchy.

1. Introduction. Chip multicore processor (CMP) architectures are designed with the aim to boost performance by providing on-chip parallelism while reducing power consumption and heat output by integrating two or more processor cores on to a single chip. Power has become the most critical constraint in the design of many on-chip systems including the CMP systems. The analysis of both the software and hardware for overall power consumption is needed for CMPs and similar on-chip systems. The main objective of the present study is to introduce power as a parameter into a realistic speed-up metric for numerical linear algebra routines running on CMPs.

For embedded devices the software can be made power-aware by analyzing the pipeline and the data flow from the instruction set [32]. The estimation of power consumption is done using instruction level modeling and considering base cost, effect of circuit state and effect of cache misses. The software then can be tuned to consume less power based on this analysis. The basic idea behind the instruction level power models of a given processor is to study the energy consumption rate [33]. The energy cost of a program is then simply the product of its average power cost and its running time. These concepts can be applied to CMPs executing numerical linear algebra routines.

CMPs use communication fabrics in the form of networks-on-chips (NoCs) that scale to handle the communication demand among the cores [5, 24]. Thus a parallel computing model and speed-up metric for CMPs must take into account the power consumed in on-chip communication as well as that in computation.

We review some of the existing results and concepts leading to the derivation of our speed-up metric. Some duty cycle modulations are used in [4] to emulate performance asymmetry which is effective because of the unpredictability of performance. Also, [4] shows that limitations in scalability arise due to differences in computation power of the individual cores and not due to communication latencies. Thus load-balancing among the on-chip threads is an unexplored aspect of power and heterogeneity aware parallel speed-up metrics for CMPs. In [35] an experimental approach may be found for determining whether the differences in the implementation, design and scheduling of threads would produce significant differences in performance. But a more general approach will be to develop an analytical model that puts together parallel efficiency, granularity of parallelism, voltage/frequency scaling and the power consumption in a CMP. Such a general analytical model is discussed in [27] and the experimental work corroborating the model shows that the optimized parallel computing can bring significant power savings and still meet a given performance target provided granularity and voltage/frequency levels are chosen and combined judiciously. The particular choice, however, is dependent on the application's parallel efficiency curve and the process technology utilized. This analytical model does not take into consideration the cache hierarchy, the cache hit-miss statistics and cache power consumptions. The hierarchical cache traffic interacts with the spatial locality of data. In addition, static power consumption and leakage loss [12, 28] during waiting for synchronization of threads also must be modeled in a power aware speed-up metric.

Available speed-up metrics do not consider any extra overhead due to any check-pointing or fault-tolerance techniques adopted at the software or middleware level. For any application running for very long duration,

*Department of Computer Science, Thomas M. Siebel Center for Computer Science, University of Illinois at Urbana-Champaign, 201 N Goodwin Ave, Urbana, IL 61801, USA.

†(Corresponding Author) Supercomputer Education and Research Centre, Indian Institute of Science, Bangalore 560012, India; email:raha@serc.iisc.ernet.in

the roll-backs and recovery of computation which in turn involves power, communication and memory traffic overheads, becomes a necessary consideration.

In [23] recursive blocked algorithms are used to split the numerical linear algebra problems and thus the technique mainly improves on the temporal locality. Most of the work in the recursive algorithm is performed by efficient Basic Linear Algebra Subprograms (BLAS) library routines optimized for the calling application's host architecture. The approach uses OpenMP induced threads and produces the results for both uniprocessor and symmetric multi processing (SMP) platforms. Algorithms for efficient scheduling and synchronizing of threads on multiprogrammed SMPs have been developed in [3]. This work has modified the Linux kernel (version 2.2.15) in order to implement the scheduler as a run-time library. In [18] higher level BLAS and LAPACK-style [2] codes have been used for computing the pivoted Cholesky factorization routine for positive semi-definite matrices. Data distribution among different threads on a CMP can be done using block cyclic data distribution in the similar way as ScaLAPACK routines [10]. Work on new data structures for dense matrices and LAPACK design for chip multicore processors can be found in [8, 17]. More on important multicore architecture issues, programming models, algorithms requirements and software design for numerical library research and development can be found in [20]. Polyhedral models of data dependence and program transformations were considered in [6]. Multigrid applications can be found in [14].

The scalable implementation of BLAS and LAPACK [2] libraries on SMP systems [10] can be the basis for the modification needed in these routines for efficiently running on the CMPs. In this paper, data manipulation and computation for selected BLAS and LAPACK routines with POSIX threads [7] have been done with respect to the latency in physical and logical access to the processing element on the chip. Some of the implementations on a single chip multicore processor result in an SMP-like parallel computing. Further, the speed-up metrics discussed in the present paper lead to tuning the algorithm and implementation for power and memory access latency.

In its experimental part the present paper studies the performance of a few linear algebra subroutines on a simple CMP using multithreading. The computationally studied numerical methods are Jacobi Iteration and Cholesky factorization. The experiments compare the estimated actual speed-up against theoretical estimates obtained by the speed-up metric considering interconnect delay based communication latency, hierarchical cache miss rate and power (including leakage loss) consumed in communication, computation and waiting for synchronization.

2. Speed-up Metric. The parameters that affect the performance of a program on a CMP are on-chip interconnect delay, connection topology (e.g., hypercube, mesh) affecting switching and communication latency, power leakage and power dissipation (including thermal dissipation) and cache access time (as affected by cache hierarchy). Additionally, for a CMP densely packed with compute and other resources may experience transient faults and consequent maintenance of redundancy, check-pointing and roll-back also affect the performance.

A measure of the effectiveness of the availability of multiple processing units on the chip is given by the speed-up achieved over the same operation being done on a single processing unit [16]. The *speed-up*, S, is defined as the ratio of sequential to parallel execution time.

$$S = \frac{T_1(w_s)}{T_N(w_p)}, \tag{2.1}$$

where, w_s is the sequential and w_p is the parallel workload (instructions or computations). $T_1(w_s)$ is the sequential execution time or the amount of time needed to complete the workload w_s on a single processor and $T_N(w_p)$ is the parallel execution time or the amount of time to complete the workload on N processors. For a CMP system speed-up can be simply defined as the ratio of sequential execution time on a single core to the execution time using multiple cores in parallel:

$$S = \frac{T_1(w_s)}{T_n(w_p)}, \tag{2.2}$$

where, w_s is the sequential and w_p is the parallel workload which is obtained as combining the sequential workload at each node in the task graph and the overhead due to parallelization. We express $w_p = OF \times w_s$ where OF is the overhead factor which is greater than or equal to 1 and $T_n(w_p)$ is the execution time or the amount of time needed to complete the same operation on n cores. Within the parallel workload w_p, let PF be

the fraction of the workload that can actually be parallelized. Then, following [19, 21], the relationship between frequency, average number of cycles per instruction and the execution time using n cores is given as

$$T_n(w_p) = (1 - PF)w_p \frac{CPI_1}{f_1} + \frac{PF \times w_p}{DOP} \frac{CPI_n}{f_n}, \tag{2.3}$$

where, CPI is the average number of cycles per instruction and f is the operating frequency. The subscripts denote the number of cores for which the respective quantities are being considered. Here DOP is the *average* degree of parallelism which is defined as the average number of cores that can be busy computing a workload over $\frac{CPI_n}{f_n}$ units of time. The speed-up can be expressed as

$$\begin{aligned} S = \frac{T_1(w_s)}{T_n(w_p)} &= \frac{w_s \frac{CPI_1}{f_1}}{(1 - PF) \times w_p \times \frac{CPI_1}{f_1} + \frac{PF \times w_p}{DOP} \times \frac{CPI_n}{f_n}} \\ &= \frac{\frac{CPI_1}{f_1}}{(1 - PF) \times OF \times \frac{CPI_1}{f_1} + \frac{PF \times OF}{DOP} \frac{CPI_n}{f_n}}. \end{aligned} \tag{2.4}$$

The parallel efficiency [11] of an application running on N processors, $\epsilon(N)$, can be written as

$$\epsilon(N) = \frac{T_1}{NT_N}. \tag{2.5}$$

Parallel efficiency for any application running on a CMP with n cores, $\epsilon(n)$ can be written as

$$\begin{aligned} \epsilon(n) = \frac{T_1(w_s)}{nT_n(w_p)} &= \frac{w_s \frac{CPI_1}{f_1}}{n(1 - PF)w_p \frac{CPI_1}{f_1} + n \frac{PF \times w_p}{DOP} \times \frac{CPI_n}{f_n}} \\ &= \frac{\frac{CPI_1}{f_1}}{n(1 - PF) \times OF \times \frac{CPI_1}{f_1} + n \frac{PF \times OF}{DOP} \times \frac{CPI_n}{f_n}}. \end{aligned} \tag{2.6}$$

2.1. Power Consumption Model. We make same assumptions as in [27] regarding homogeneity, availability, simultaneous activity of the cores on a generalized CMP. It is also assumed that cores that are not being used are completely shut-down. Total power (P_{total}) consumption consists of three components: static power (P_{static}) which is the component required for biasing the circuit elements, active power (P_{active}) that is consumed during computation and leakage power ($P_{leakage}$) which is independent of any activity. Leakage power is consumed even if a core is not doing any computation. Considering all these components in a way similar to the approach in [27], we can write

$$P_{total} = P_{static} + P_{active} + P_{leakage}. \tag{2.7}$$

The static power can be expressed in terms of the biasing current and supply voltage:

$$P_{static} = V_{supply} I_{bias} = V_{DD} I_{static}, \tag{2.8}$$

where, V_{DD} or V_{supply} is the supply voltage and I_{bias} is the biasing current or static current (I_{static}). The leakage power dissipation varies with the CMOS process technology. For CMPs the leakage power is significant due to the smaller feature size of the CMOS process used and and depends on the number of processing entities (i. e., cores) on it. Also, as the CMOS process enters the sub-65 nm technology, the leakage component of power increasingly becomes significant [22, 27]. The active power consumption includes both dynamic power ($P_{dynamic}$) and short-circuit ($P_{shortcircuit}$) power. Active power is proportional to the instruction processing activities. It may be written as

$$P_{active} = P_{dynamic} + P_{shortcircuit},$$

so that

$$P_{total} = P_{static} + P_{dynamic} + P_{shortcircuit} + P_{leakage}. \tag{2.9}$$

Leakage current has five basic components: the sub-threshold leakage current (I_{sub}), the gated oxide leakage current (I_{gate}), the reverse biased PN junction current, the punch through current and the gate tunneling current [9]. We analyze the total leakage power by considering only the gated oxide leakage current I_{gate} and the sub-threshold leakage (I_{sub}) which are likely to be more significant. The sub-threshold leakage current (I_{sub}) is modeled as

$$I_{sub} = I_0 e^{\frac{V_{gs}-V_t-V_{off}}{n_{ds}v_T}}\left(1-e^{-\frac{V_{ds}}{v_T}}\right) \tag{2.10}$$

where I_0 is the leakage current at 25°C, V_t is the threshold voltage, V_{gs} is the gate to source voltage, V_{off} is an empirically determined model parameter [30], V_{ds} is the drain to source voltage and v_T is the thermal voltage obtained as $\frac{kT}{q}$, q being the electron unit charge, T, the temperature in Kelvin and k, the Boltzmann constant. The constant n_{ds} is experimentally determined for a particular technology. The dynamic power is computed as in [26]: $\alpha CV_{DD}^2 f$, α being the activity factor, C a capacitative constant and f, the switching frequency (taken same as the operating frequency) and V_{DD}, the supply voltage. The short circuit power dissipation is expressed as $P_{shortcircuit} = \frac{1}{2} I_{shortcircuit} V_{DD}$ [1], with $I_{shortcircuit}$ as the short circuit current. The gate oxide leakage current I_{gate} can be expressed as a sum of the gate to source leakage current i_{gs} and the gate to drain leakage current i_{gd} [25]. Thus, we can write,

$$P_{total} = I_{static}V_{DD} + \alpha CV_{DD}^2 f + \frac{1}{2} I_{shortcircuit}V_{DD} + V_{DD}(I_{gate} + I_{sub}). \tag{2.11}$$

We estimate

$$i_{gs} = \frac{127.04 \times L_{eff} \times e^{\left(5.60625\times\ V_{gs}^{-10.6}\times T_{ox}^{-2.5}\right)}}{2}, \tag{2.12}$$

$$i_{gd} = \frac{127.04 \times L_{eff} \times e^{\left(5.60625\times V_{gd}^{-10.6}\times T_{ox}^{-2.5}\right)}}{2} \tag{2.13}$$

following the commonly used gate-oxide leakage models [25] where L_{eff} is the effective gate width in nanometers, T_{ox} is the effective gate oxide thickness in nanometers and V is the effective supply voltage for the number of cores as designated in its subscript. The i_{gs} and i_{gd} are given in micro Ampere per micrometer of transistor width. Considering the above model for a single core processor, the estimate for power dissipation can be written as

$$P_1 = I_{static}V_1 + \alpha CV_1^2 f_1 + \frac{1}{2} I_{shortcircuit}V_1 + V_1(I_{gate} + I_{sub}). \tag{2.14}$$

For a CMP with n cores the power equation can be written as

$$P_n = nI_{static}V_n + n\alpha CV_n^2 f_n + n\frac{1}{2} I_{shortcircuit}V_n + nV_n(I_{gate} + I_{sub}). \tag{2.15}$$

For a multicore processor with n cores the power equation can be re-written in terms of voltage scaling:

$$P_n = nI_{static}\nu V_1 + n\alpha C\nu^2 V_1^2 f_n + n\frac{1}{2} I_{shortcircuit}\nu V_1 + n\nu V_1(I_{gate} + I_{sub}), \tag{2.16}$$

since $V_n = \nu V_1$, the scaled voltage for n cores with respect to a single core. As seen from the above model for CMPs the leakage power increases with increase of number of cores per chip. Hence apart from power actually consumed during computation (i. e., dynamic power) the leakage power plays an important role in on-chip parallel computing with a CMP.

From equation (2.14) we can write the frequency for a single core processor in terms of power being consumed,

$$f_1 = \frac{P_1 - V_1(I_{static} + \frac{1}{2} I_{shortcircuit} + I_{gate} + I_{sub})}{\alpha CV_1^2} \tag{2.17}$$

$$=: \frac{P_1 - V_1\tilde{I}}{\alpha CV_1^2}.$$

From equation (2.15) we can derive the frequency for a CMP using n cores as,

$$f_n = \frac{P_n - nV_n(I_{static} + \frac{1}{2}I_{shortcircuit} + I_{gate} + I_{sub})}{n\alpha CV_n^2} \tag{2.18}$$
$$=: \frac{P_n - nV_n\tilde{I}}{n\alpha CV_n^2}.$$

Also, $\frac{f_1}{f_n} = n\nu^2 \frac{P_1 - V_1\tilde{I}}{P_n - n\nu V_1\tilde{I}}$. Putting the expression for the frequencies in equation (2.4) we can express power aware speed-up as follows.

$$S = \frac{1}{(1-PF)\times OF + n\frac{PF\times OF}{DOP}\frac{CPI_n}{CPI_1}\nu^2\frac{P_1 - V_1\tilde{I}}{P_n - n\nu V_1\tilde{I}}}. \tag{2.19}$$

We define $\vartheta := \frac{P_1}{V_1\tilde{I}}$ and $\varsigma = \frac{P_n}{P_1}$. Then, the speed-up can be written as

$$S = \frac{1}{(1-PF)\times OF + n\frac{PF\times OF}{DOP}\frac{CPI_n}{CPI_1}\nu^2\frac{\vartheta - 1}{\vartheta\varsigma - n\nu}}. \tag{2.20}$$

Power aware parallel efficiency, $\epsilon(n)$, for a CMP using n cores can be similarly expressed plugging in the expressions for frequencies in equation (2.6):

$$\epsilon(n) = \frac{1}{n(1-PF)\times OF + n^2\frac{PF\times OF}{DOP}\frac{CPI_n}{CPI_1}\nu^2\frac{P_1 - V_1\tilde{I}}{P_n - n\nu V_1\tilde{I}}}. \tag{2.21}$$

The *power optimization* (P_{opt}) can be expressed in terms of the energy consumption

$$P_{opt} := \frac{\text{Energy consumed in serial code execution}}{\text{Energy consumed in parallel code execution}} \tag{2.22}$$

So, the term Power optimization can be defined as follows.

$$\begin{aligned} P_1 &:= \text{Total execution time for serial code} \times (\text{Static Power} + \text{Leakage Power}) \\ &\quad + \text{System time for serial code} \times \text{Dynamic Power} \\ P_n &:= \text{Total execution time for parallel code} \times (\text{Static Power} + \text{Leakage Power}) \\ &\quad + \text{System time for parallel code} \times \text{Dynamic Power for all } n \text{ cores} \\ P_{opt} &= \frac{P_1}{P_n} \end{aligned} \tag{2.23}$$

When one thread is waiting for another thread to complete some computation, the static, leakage power and the short-circuit power will be consumed even though very little dynamic power is being consumed. In order to optimize a parallel implementation with respect to power consumption it is necessary to design the numerical algorithms in such a way that threads wait for each other a minimum at the barriers. For this the computational load among different cores must be balanced with respect to localization of the threads and the barrier calls.

2.2. Cache Hierarchy Effects. For cache miss, there is associated penalty in terms of loss of parallel efficiency. If cache is hit, let t_h be the time taken for fetching the data from the cache. For a cache miss, let the time taken be t_m where $t_m \gg t_h$. If for any algorithm p is the fraction of computation that has scored a cache hit, then time taken, t_{cache}, in cache operation for fetching the data can be written as $t_{cache} = pt_h + (1-p)t_m$. Then cache penalty C_p can be expressed as

$$C_p = \frac{t_{cache}}{t_h} = \frac{t_m}{t_h} + p\left(1 - \frac{t_m}{t_h}\right), \tag{2.24}$$

where if t_h is in the order of nanoseconds, then t_m is in the order of microseconds, as during cache miss data must be fetched from memory and memory access is a relatively slower process. Actual cache operation time is expressed as,

$$t_{cache} = C_p t_h. \tag{2.25}$$

Depending on cache hierarchy, t_h and p may vary. Time taken on a hit at the cache at core level is less than the time taken on a hit at cache per chip region each covering a group of cores, which in turn will be less than the time taken on a hit at the CMP's global cache. Let time taken for a cache hit from cache at core level be $t_{h.core}$, time taken for a cache hit at a cache for a group of cores be $t_{h.zone}$, and time taken for a cache hit at the global cache for the CMP be $t_{h.chip}$. Generally it would be realistic to assume that $t_{h.core} \leq t_{h.zone} \leq t_{h.chip}$. For a particular application, let there be cache hit at cache per core level p_1 fraction of times, cache hit at the group of cores level be p_2 fraction of times and the cache hit at the CMP's global cache be p_3 fraction of times. Then time taken for cache hit may be expressed as

$$t_h = p_1 t_{h.core} + p_2 t_{h.zone} + p_3 t_{h.chip}, \tag{2.26}$$

where, $p = p_1 + p_2 + p_3$. The speed-up, S, may be modified and written as

$$S = \frac{1}{C_p t_h \frac{f_1}{w_s \times CPI_1} + (1 - PF) \times OF + \frac{PF \times OF}{DOP} \frac{CPI_n}{CPI_1} \frac{f_1}{f_n}}, \tag{2.27}$$

in which frequency expressions from equations (2.18) and (2.19) may be substituted.

Depending on the architecture one has to modify the conventional algorithms to reduce the cache miss rate. So, for any application running on multicore processor, the algorithm should be designed in such a fashion that the amount of data handled or processed by a particular core is optimized, i. e., data driven parallelism is a better option.

2.3. Communication Overheads. The communication overhead including interconnect propagation and switching latency, t_c may be considered and the speed-up in equation (2.27) is re-formulated as

$$\begin{aligned} S_p &= \frac{1}{(C_p t_h + t_c)\frac{f_1}{w_s \times CPI_1} + (1 - PF) \times OF + \frac{PF \times OF}{DOP} \frac{CPI_n}{CPI_1} \frac{f_1}{f_n}} \\ &= \frac{1}{(C_p t_h + t_c)\frac{f_1}{w_s \times CPI_1} + (1 - PF) \times OF + \frac{PF \times OF}{DOP} \frac{CPI_n}{CPI_1} n\nu^2 \frac{\vartheta - 1}{\vartheta\varsigma - n\nu}} \end{aligned} \tag{2.28}$$

giving the power-aware speed-up considering computation, communication and cache hierarchy.

3. Numerical Linear Algebra Routines. The most widely used tools for scientific computing are the Basic Linear Algebra Subprograms (BLAS) and the LAPACK libraries [2, 15]. In addition to tuning the individual numerical linear algebra routine's serial algorithm to a single core's architecture, for a CMP, considerations for the overall tuning of the multithreaded parallel numerical linear algebra operation to the on-chip communication, cache and power architecture are essential. The present paper uses POSIX threads to implement ScaLAPACK [10] algorithms for observing the intended power-aware speed-up. For tuning purposes, data manipulation and computation are done according to physical and logical access of a single processing element (core) or a group of cores on the chip.

3.1. Jacobi Iterative Solver. For a real system of equations, $Ax = b$, the Jacobi iteration is a method of simultaneous corrections, as no component of an approximation $x^{(m)}$ is used until all the components of $x^{(m)}$ have been computed. The method replaces mth iterate $x^{(m)}$ by $x^{(m+1)}$ at once, before the beginning of next cycle. The algorithm may be written as $x^{(m+1)} = b + (I - A)x^{(m)}$. In the thread parallel version of the Jacobi iterative solver, the workload is distributed over all the threads across the rows. Every thread is operates on a block of adjacent rows to get the facility of spatial locality of reference of data. The data-distribution is shown in Figure 3.1. Matrices with $A_{i,j} = N/(i + j + 1)$, N being the matrix column dimension and all diagonal elements set at 10000 is used for computational experiments. The right hand side, $b_i = i$ and initial guess $x_i^0 = 4.0i$ have been used.

3.2. Cholesky Factorization. Symmetric positive definite real square matrix A has Cholesky factorization $A = LL^T$, L being the lower triangular matrix with positive diagonal entries. The linear system can be then solved by forward substitution in lower triangular system $Ly = b$, followed by the back substitution in upper triangular system $L^T x = y$.

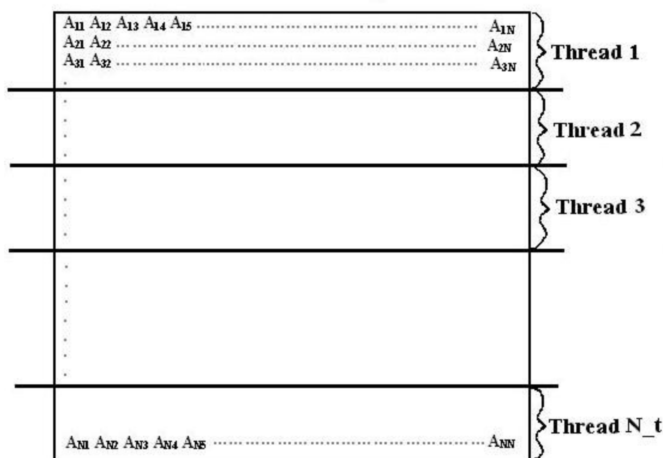


FIG. 3.1. *Threaded Parallel Jacobi Iterative Solver*

3.2.1. Threaded Parallel Algorithm. In the threaded parallel version of the Cholesky factorization the two basic operations, division (`cdiv`) and modifications (`cmod`), are parallelized. These two basic operations are based on the thread identity so that each thread will operate on some entries within a range of the matrix indices in a synchronized way. To achieve such synchronization blocking calls and thread schedulers available on POSIX thread library are used. We have used Round-Robin scheduling (`SCHED_RR`) for the threads setting minimum priority by using `sched_get_priority_min`(). Also, for Solaris®based systems the concurrencies can be set using `thr_setconcurrency`(). The threads were spawned using `pthread_create`() and `thread_create`() for Linux based and Solaris®based systems respectively. For synchronizing among threads `pthread_create`() (`thread_join`() for Sun Solaris®systems) was used. Scheduling parameters like the priority variable has been set by adding minimum priority and index to minimize thread waiting. By using `pthread_setschedparam`() and `pthread_getschedparam`() scheduling parameters, the Round-Robin Scheduling was configured. Data distribution has been done on the outer most loop in the block cyclic distributed sub-matrix format of Cholesky factorization. Each thread operates with in a range (like k_{start} to k_{end}) which depend on the thread identity or index. The range is proportional to the ratio of number of elements per row by the number of threads being spawned. With this, an attempt is made to keep computational load per thread almost equidistributed so that there is little skew in finishing time of the threads at a barrier. This in turn helps in meeting leakage power dissipation goals via synchronization objectives. The test cases used are symmetric positive definite matrices from [31] in dense format and the table 3.1 gives the list of test matrices used for computation. The schedulers add to the parallel overhead a little more (2-5 per cent) but avoids fine grained parallelization of the division (`cdiv`) and modifications (`cmod`) individually. The reduced granularity of parallelization improves the fraction to be parallelized.

4. Experiments and Results. In the speed-up equation (2.28), the voltage scaling ν and power scaling ς are determined from the various throttle settings attained by the particular CMP design. We take activity factor $\alpha = 1$ and estimate ϑ, which depends on the architecture and CMOS process of the CMP, with the following tools. Using Cacti [34] we have estimated cache access power characteristic using the cache size, number of banks, (set)-associativity and size of blocks as input. But to obtain leakage power characteristics we have used eCACTI [29] setting the parameters for applicable CMOS (90 nanometer in our experiments) technology parameter along with all the inputs required in Cacti. For the leakage power estimates we have used parameters of the cache hierarchy model of the host CMP in a generalized simplified framework as shown in figure 4.1. By using `g-cache-trace` modules the cache-statistics was obtained. The `g-cache-trace` modules also trace the operating system's cache statistics along with the code execution statistics. About 2 to 5 per cent better cache hit statistics in case of parallel code execution as compared to serial code execution was observed for the matrix computation routines experimented with. The simulator LUNA [12, 13] is used to obtain high level power and delay analysis for a generalized one-to-one link topology among the individual cores. The simulation using LUNA are done while accepting its limitation that the network representation, bandwidth for each link

Table 3.1
Example Matrices from [31]

Matrix Name	Dimension and Non-Zero Entries	Functionality
BCSSTK01	48×48, 224	BCS Structural Engineering Matrices : Small test problem
BCSSTK02	66×66, 2211	BCS Structural Engineering Matrices : Oil rig - statically condensed
BCSSTK03	112×112, 376	BCS Structural Engineering Matrices : Small test structure
BCSSTK04	132×132, 1890	BCS Structural Engineering Matrices : Oil rig—not condensed
BCSSTK05	153×153, 1288	BCS Structural Engineering Matrices : Transmission tower
BCSSTK06	420×420, 4140	BCS Structural Engineering Matrices : Medium test problem
BCSSTK08	1074×1074, 7017	BCS Structural Engineering Matrices : TV studio
BCSSTK09	1083×1083, 9760	BCS Structural Engineering Matrices : Square plate clamped
BCSSTK11	1473×1473, 17857	BCS Structural Engineering Matrices : Ore car—lumped mass
BCSSTK13	2003×2003, 42943	BCS Structural Engineering Matrices : Fluid flow

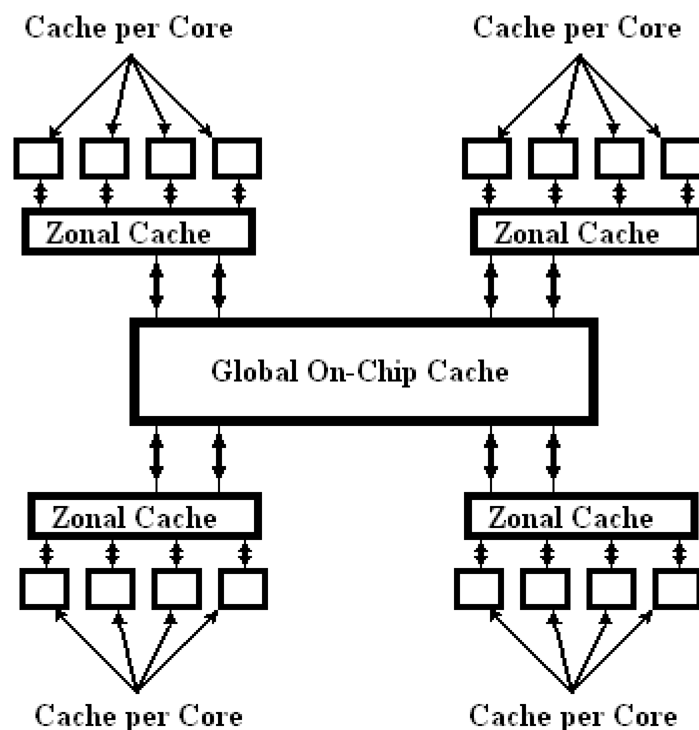


Fig. 4.1. *Cache hierarchy for a typical multicore system*

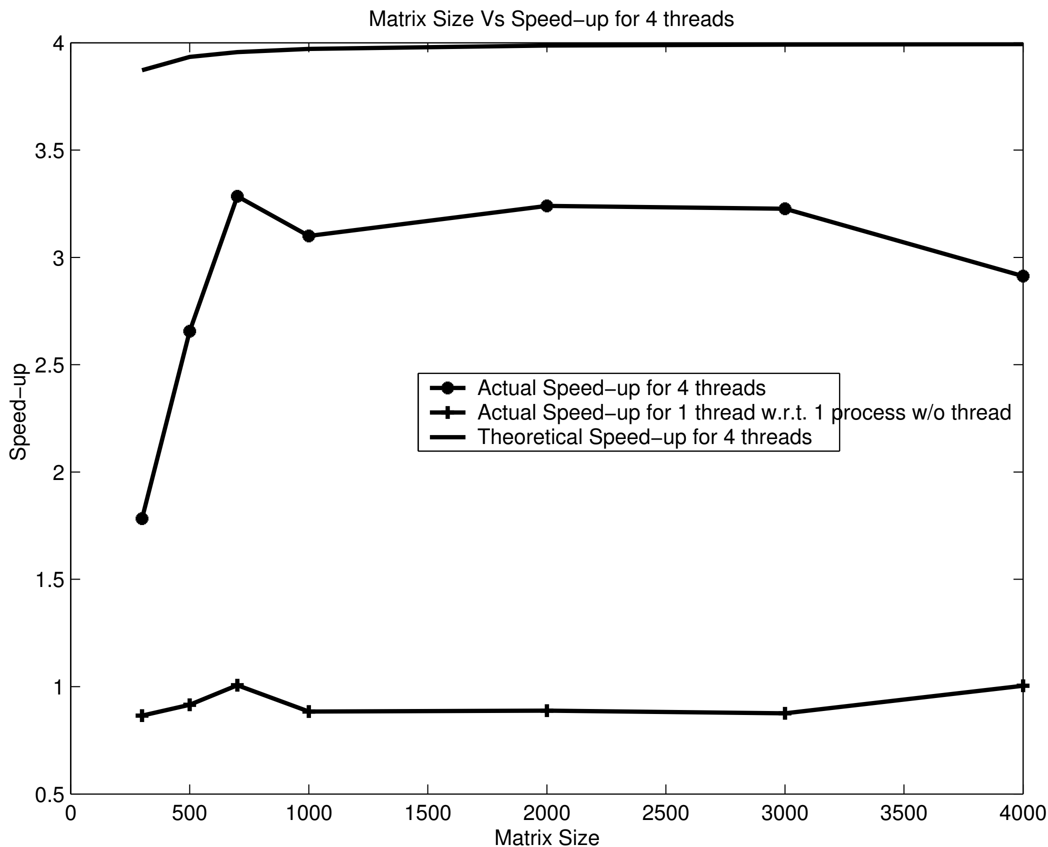


FIG. 4.2. *Speed-up for* 4 *threads for the Jacobi iterative solver*

and detailed data flow path may be very different in the actual CMP. Also, every application may not follow the same data flow path on the CMP every time. Thus LUNA gives only an idealized and statistically expected power behavior for the communication fabric.

4.1. Jacobi Iterative Solver. The speed-up increases nearly in linear fashion as the matrix size increases. In the parallel algorithm the amount of computation to be done in a single iteration has been distributed among different threads in such a manner so that all the threads are as much as possible equally loaded. There are some asymmetries in core performance in a CMP environment. But for our analysis leading to (2.28) we have neglected the performance asymmetry and take into account the delay due to blocking calls in our implementation of the multithreaded Jacobi iteration. These characteristics also implies that due to the unmodeled (in theoretical speed-up) overhead of parallelization in thread creation and blocking functions the actual speed-up is much less than the theoretical speed-up. The unmodeled overhead in thread-creation and blocking calls become relatively insignificant. The actual speed-up then approaches the theoretical speed-up for larger size of the matrix. The speed-up for one thread is less than 1 compared to the single process due to the thread overheads.

The actual execution time based speed-up for large matrices tends to follow the theoretical speed-up pattern for this example. The figure 4.2 shows the speed-up for 4 threads on a SUN Microsystem system based on a dual-core AMD Opteron®Model 2218 2.6GHz 90 nanometer SOI CMOS process CMP for the Jacobi iterative solver. The speed-up reaches more than 3 for square matrix with column dimension size greater than 600 using 4 threads.

The difference between the actual speed-up and theoretical speed-up is mostly due to synchronization overhead for different threads at the barrier calls and this overhead increases with the number of barrier calls as two threads are now localized on each core against the optimal situation of one per core. The theoretically unaccounted for skew due to the cores being busy with kernel threads also contribute to the shortfall with respect to the theoretical speed-up given by (2.28). However, we still get a good overall agreement with the

theoretical model for this BLAS level 2 rich algorithm. The active component of power is directly related to the time for which the cores are busy while the static and leakage components are proportional to the total execution time.

4.2. Cholesky Factorization. The speed-up of threaded parallel Cholesky factorization in total execution time for 2 and 4 threads on a SUN system with dual AMD Opteron®dual-core Model 2218 2.6GHz 90 nanometer SOI CMOS process CMPs is studied in this section. One thread was localized per core. The data distribution was a equal sized row block partitioning among the threads. For Cholesky factorization the speed-up does not change much for input matrix sizes beyond the square matrix dimension of 1083. For the actual speed-up a similar slope in the speed-up curve to that of the theoretical curve as per equation (2.28) is observed. The speed-up curve both in theory and in our experiments may be seen to have a steep slope for square matrix with column dimensions between 70 and 450, and are more or less in agreement in terms of the expected profile of the experimental data. For matrices of size smaller than 70×70 the overhead due to parallelization and synchronization is relatively significant and makes multithreading not useful. For matrices larger than 1083×1083 cache misses play a more important role in affecting the actual speed-up. Thus a drop in actual speed-up is expected. The theoretical speed-up was derived assuming an infinite availability of cache and bus which obviously does not hold good for beyond a certain data size since the resources on a single chip multicore processor is finite and fixed unlike a multi-processor system where availability of resources can be more flexible. The simple cache hit-miss model becomes inadequate beyond this point of saturation and the actual speed-up falls far short of the theoretical speed-up. The figure 4.3 shows the speed-up of threaded Cholesky factorization for 2 and 4 threads on the same SUN Microsystems machine based on dual dual-core AMD Opteron®CMPs. The theoretical speed-up increases as the matrix size increases and reaches close to 1.9 and 3.7 for 2 and 4 threads respectively. We note that this is a BLAS level 3 rich algorithm unlike the Jacobi iterative solver and hence the saturation point for the multi-threaded algorithm is reached faster. For large number of threads we observe that the power characteristic is not greatly sensitive to the number of threads. This makes threaded Cholesky Factorization on a CMP an algorithm relatively less sensitive to power aware optimizations.

4.3. Remarks. All the power characteristics are based on the estimates made on the processor models with only public domain manufacturer data using eCACTI and Cacti. This is plugged into the theoretical speed-up and compared with the actual system time and total time lapsed for running the codes. Jacobi iterative solver is easier to tweak for optimal speed-up due to the fact that in each iteration all the values needed are already being computed in the last iteration. This gives a possibility of getting $\Theta(n)$ speed-up where n is the number of cores for a single user system executing only this parallel program. Thread-scheduling do not play a significant role in tuning this solver. For Cholesky factorization, data-parallel model involves a lot more complex constraints. Thread-scheduling plays a significant role in reducing the use of blocking calls and in turn, static and leakage power dissipation related slow-down. Thread-scheduling, as before, plays a significant role. The blocking calls, however, cannot be avoided and is necessary to maintain the data-consistency. This in turn decreases the speed-up. The amount of blocking increases the static power consumption and the leakage loss. The data-parallel model induces a relatively higher cache hit rate which helps in getting better performance with lower power consumption. For Cholesky Decomposition, tuning based on simplistic cache and communication model makes it difficult to get the predicted speed-up for larger matrices.

5. Conclusions. The threaded parallel numerical linear algebra routines is a natural approach to tuning the BLAS and LAPACK/ScaLAPACK routines to a CMP. Power, on-chip cache traffic and on-chip interconnect behavior play a crucial role in getting the speed-up which is desired to be close to the order of the number of threads. Due to the umodeled overheads in thread creation and synchronization, we do not get significant speed-up for smaller matrices. Again for very large problems, cache memory related latencies come into the picture and affects speed-up. A possibility is that a CMP with a large number of cores can be utilized to reduce the cache-misses by understanding the order of data access in the matrix computation. Algorithmic level optimization of the power may be possible by reducing the static and leakage dominated phases of the implementation. By utilizing the cache-hierarchy to the maximum possible extent the overall waiting time can be reduced and consequently the overall execution time as lower power is dissipated. Efficient thread schedulers must be embedded into the tuned implementations to utilize the cores as symmetrically as possible.

As general CMPs with large number of cores and switched on-chip communication fabric become available, it would be possible to observe the power aspects of the speed-up metric in greater detail.

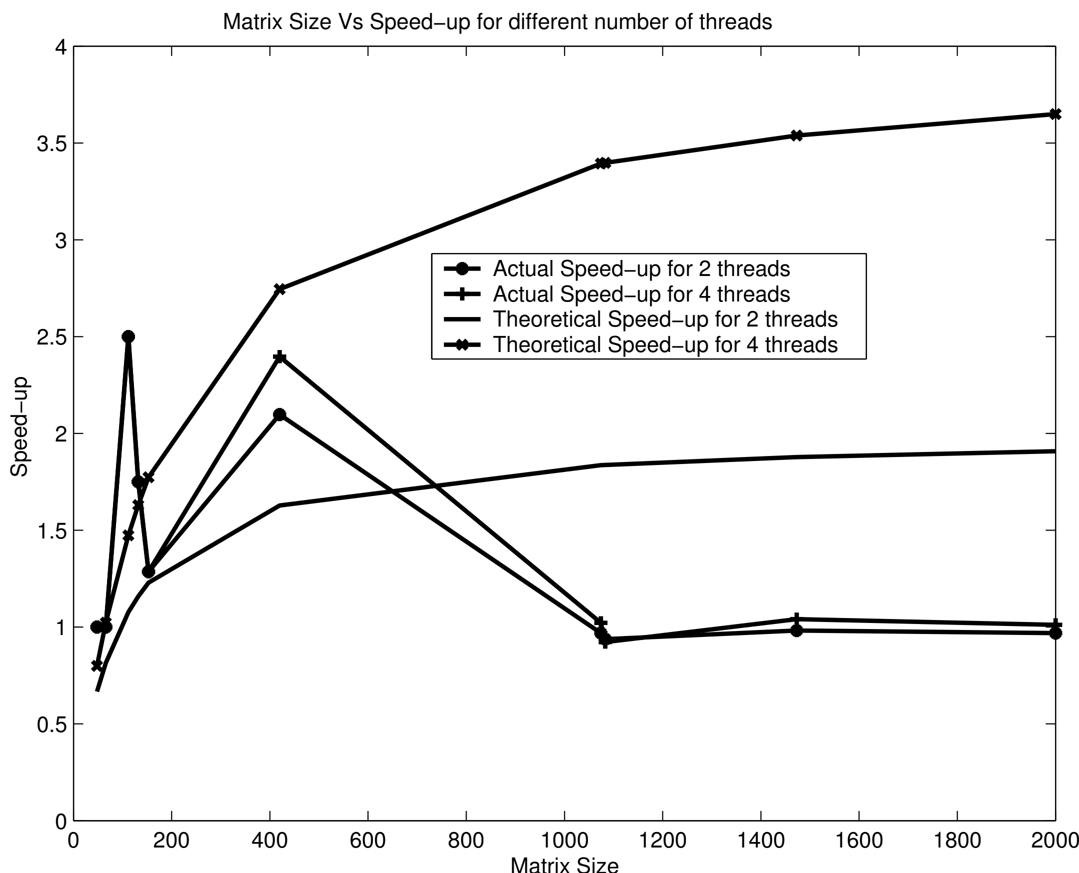


FIG. 4.3. *Speed-up for different thread numbers for the Cholesky factorization. It may be noted that our theoretical speed-up computation is in good agreement with the experimental data for dense matrix size of up to 450. For larger matrices the system becomes saturated and page faults and cache thrashing dominate. The theoretical model is based on the implicit assumption that infinite cache is available. Hence the disagreement beyond a certain matrix size. Unlike a multi-chip multi-processor system, a chip multicore processor displays this limitation more sharply. Also, for very small matrices the thread overheads dominate and both theoretical and actual speed-ups are less than* 1 *indicating that below a certain data size and work load it is not worth multithreading.*

REFERENCES

[1] EMRAH ACAR, RAVISHANKAR ARUNACHALAM, AND SANI R. NASSIF, *Predicting short circuit power from timing models*, in ASPDAC: Proceedings of the 2003 conference on Asia South Pacific design automation, New York, NY, USA, 2003, ACM, pp. 277–282.

[2] E. ANDERSON, Z. BAI, C. BISCHOF, L. S. BLACKFORD, J. DEMMEL, JACK J. DONGARRA, J. DU CROZ, S. HAMMARLING, A. GREENBAUM, A. MCKENNEY, AND D. SORENSEN, *LAPACK Users' guide (third ed.)*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999.

[3] C. D. ANTONOPOULOS, D. S. NIKOLOPOULOS, AND T. S. PAPATHEODOROU, *Informing algorithms for efficient scheduling of synchronizing threads on multiprogrammed SMPs*, in ICPP '02: Proceedings of the 2001 International Conference on Parallel Processing, Washington, DC, USA, 2001, IEEE Computer Society, pp. 123–130.

[4] SAISANTHOSH BALAKRISHNAN, RAVI RAJWAR, MIKE UPTON, AND KONRAD LAI, *The impact of performance asymmetry in emerging multicore architectures*, in ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture, Washington, DC, USA, 2005, IEEE Computer Society, pp. 506–517.

[5] TOBIAS BJERREGAARD AND SHANKAR MAHADEVAN, *A survey of research and practices of Network-on-chip*, ACM Comput. Surv., 38 (2006), p. 1.

[6] U. BONDHUGULA, M. BASKARAN, A. HARTONO, S. KRISHNAMOORTHY, J. RAMANUJAM, A. ROUNTEV, AND P. SADAYAPPAN, *Towards effective automatic parallelization for multicore systems*, Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on, (2008), pp. 1–5.

[7] DAVID R. BUTENHOF, *Programming with POSIX Threads*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[8] ALFREDO BUTTARI, JULIEN LANGOU, JAKUB KURZAK, AND JACK DONGARRA, *A class of parallel tiled linear algebra algorithms for multicore architecture*, in MIMS Preprint, LAPACK Working Note no. 191, 2007.

[9] Xuning Chen and Li-Shiuan Peh, *Leakage power modeling and optimization in interconnection networks*, in ISLPED '03: Proceedings of the 2003 international symposium on Low power electronics and design, New York, NY, USA, 2003, ACM, pp. 90–95.

[10] Jaeyoung Choi, Jack J. Dongarra, L. Susan Ostrouchov, Antoine P. Petitet, David W. Walker, and R. Clint Whaley, *Design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines*, Sci. Program., 5 (1996), pp. 173–184.

[11] David E. Culler, Anoop Gupta, and Jaswinder Pal Singh, *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[12] Noel Eisley and Li-Shiuan Peh, *High-level power analysis for on-chip networks*, in CASES '04: Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems, New York, NY, USA, 2004, ACM, pp. 104–115.

[13] Noel Eisley, Vassos Soteriou, and Li-Shiuan Peh, *High-level power analysis for multi-core chips*, in CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems, New York, NY, USA, 2006, ACM, pp. 389–400.

[14] C. Garcia, M. Prieto, and F. Tirado, *Multigrid smoothers on multicore architectures*, NIC Series Parallel Computing: Architectures, Algorithms and Applications, 38 (2007), pp. 270–286.

[15] G. H. Golub and C.F. Van Loan, *Matrix Computations*, The Johns Hopkins University Press, Baltimore and London, third ed., 1996.

[16] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar, *Introduction to Parallel Computing*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[17] Fred G. Gustavson, *The relevance of new data structure approaches for dense linear algebra in the new multi-core/many core environments*, in Exploiting Concurrency Efficiently and Correctly—(EC)2, CAV 2008 Workshop, 2008.

[18] Sven Hammarling, Nicholas J. Higham, and Craig Lucas, *LAPACK-Style codes for pivoted Cholesky and QR updating*, tech. report, MIMS EPrints United Kingdom, 2007.

[19] John L. Hennessy and David A. Patterson, *Computer Architecture, Fourth Edition: A Quantitative Approach*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[20] M A Heroux, *Design issues for numerical libraries on scalable multicore architectures*, Journal of Physics: Conference Series, 125 (2008), p. 012035 (11pp).

[21] Kai Hwang, *Advanced Computer Architecture: Parallelism,Scalability,Programmability*, McGraw-Hill Higher Education, 1992.

[22] ITRS, *ITRS Report* , tech. report, The International Technology Roadmap for Semiconductors, 2006.

[23] I. Jonsson, *Recursive blocked algorithms, data structures, and high-performance software for solving linear systems and matrix equations*, tech. report, Academic Archive On-line [http://www.diva-portal.org/oai/OAI] (Sweden), 2003.

[24] Rakesh Kumar, Victor Zyuban, and D.M. Tullsen, *Interconnections in multi-core architectures: understanding mechanisms, overheads and scaling*, in Proc. 32nd Annual International Symposium on Computer Architecture (ISCA'05), Los Alamitos, CA, USA, 2005, IEEE Computer Society, pp. 408–419.

[25] Dongwoo Lee, Wesley Kwong, David Blaauw, and Dennis Sylvester, *Analysis and minimization techniques for total leakage considering gate oxide leakage*, in DAC '03: Proceedings of the 40th conference on Design automation, New York, NY, USA, 2003, ACM, pp. 175–180.

[26] Jian Li and José F. Martínez, *Dynamic power-performance adaptation of parallel computation on chip multiprocessors*, High-Performance Computer Architecture, 2006. The Twelfth International Symposium on, (11-15 Feb. 2006), pp. 77–87.

[27] ———, *Power-performance considerations of parallel computing on chip multiprocessors*, ACM Trans. Archit. Code Optim., 2 (2005), pp. 397–422.

[28] Yingmin Li, K. Skadron, D. Brooks, and Zhigang Hu, *Performance, energy, and thermal considerations for* SMT *and* CMP *architectures*, High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on, (12-16 Feb. 2005), pp. 71–82.

[29] Mahesh Mamidipaka and Nikil Dutt, *eCACTI: An enhanced power estimation model for on-chip caches*, tech. report, University of California, Irvine, Irvine, CA 92697-3435, USA, Sept., 2004.

[30] M. Mamidipaka, Kamal Khouri, Nikil Dutt, and Magdy Abadir, *Leakage power estimation in SRAMs*, tech. report, CECS Technical report, TR 03-32, UC Irvine, Oct. 2003, 2003.

[31] NIST, *Matrix Market*, June 2004.

[32] V. Tiwari, S. Malik, and A. Wolfe, *Power analysis of embedded software: a first step towards software power minimization*, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2 (1994), pp. 437–445.

[33] Vivek Tiwari, Sharad Malik, Andrew Wolfe, and Mike Tien-Chien Lee, *Instruction level power analysis and optimization of software*, in VLSID '96: Proceedings of the 9th International Conference on VLSI Design: VLSI in Mobile Communication, Washington, DC, USA, 1996, IEEE Computer Society, pp. 326–328.

[34] S.J.E. Wilton and N.P. Jouppi, *CACTI: an enhanced cache access and cycle time model*, Solid-State Circuits, IEEE Journal of, 31 (May 1996), pp. 677–688.

[35] Fabian Zabatta and Kevin Ying, *A Thread Performance Comparison: Windows NT and Solaris on a Symmetric Multiprocessor*, 1998.

Edited by: Dana Petcu
Received: November 6th, 2008
Accepted: March 9th, 2009

AIMS AND SCOPE

The area of scalable computing has matured and reached a point where new issues and trends require a professional forum. SCPE will provide this avenue by publishing original refereed papers that address the present as well as the future of parallel and distributed computing. The journal will focus on algorithm development, implementation and execution on real-world parallel architectures, and application of parallel and distributed computing to the solution of real-life problems. Of particular interest are:

Expressiveness:
- high level languages,
- object oriented techniques,
- compiler technology for parallel computing,
- implementation techniques and their efficiency.

System engineering:
- programming environments,
- debugging tools,
- software libraries.

Performance:
- performance measurement: metrics, evaluation, visualization,
- performance improvement: resource allocation and scheduling, I/O, network throughput.

Applications:
- database,
- control systems,
- embedded systems,
- fault tolerance,
- industrial and business,
- real-time,
- scientific computing,
- visualization.

Future:
- limitations of current approaches,
- engineering trends and their consequences,
- novel parallel architectures.

Taking into account the extremely rapid pace of changes in the field SCPE is committed to fast turnaround of papers and a short publication time of accepted papers.

INSTRUCTIONS FOR CONTRIBUTORS

Proposals of Special Issues should be submitted to the editor-in-chief.

The language of the journal is English. SCPE publishes three categories of papers: overview papers, research papers and short communications. Electronic submissions are preferred. Overview papers and short communications should be submitted to the editor-in-chief. Research papers should be submitted to the editor whose research interests match the subject of the paper most closely. The list of editors' research interests can be found at the journal WWW site (`http://www.scpe.org`). Each paper appropriate to the journal will be refereed by a minimum of two referees.

There is no a priori limit on the length of overview papers. Research papers should be limited to approximately 20 pages, while short communications should not exceed 5 pages. A 50–100 word abstract should be included.

Upon acceptance the authors will be asked to transfer copyright of the article to the publisher. The authors will be required to prepare the text in $\LaTeX 2_\varepsilon$ using the journal document class file (based on the SIAM's `siamltex.clo` document class, available at the journal WWW site). Figures must be prepared in encapsulated PostScript and appropriately incorporated into the text. The bibliography should be formatted using the SIAM convention. Detailed instructions for the Authors are available on the SCPE WWW site at `http://www.scpe.org`.

Contributions are accepted for review on the understanding that the same work has not been published and that it is not being considered for publication elsewhere. Technical reports can be submitted. Substantially revised versions of papers published in not easily accessible conference proceedings can also be submitted. The editor-in-chief should be notified at the time of submission and the author is responsible for obtaining the necessary copyright releases for all copyrighted material.