J Supercomput (2013) 64:167–176
DOI 10.1007/s11227-012-0797-3

From latex specifications to parallel codes

Alejandro Acosta · Francisco Almeida · Ignacio Peláez

Published online: 26 June 2012

Abstract The advent of multicore systems, joined to the potential acceleration of the graphics processing units, has given us a low cost computation capability unprecedented. The new systems alleviate some well-known important architectural problems at the expense of a considerable increment of the programmability wall. Heterogeneity, at both the architectural and programming levels, poses a great challenge to programmers. As a contribution, we propose a development methodology for the automatic source-to-source transformation on specific domains. This methodology is successfully instantiated as a framework to solve Dynamic Programming problems. As a result of applying our framework, the end user (a physicist, a mathematician or a biologist) can express her problem through a latex equation and automatically derive efficient parallel codes for current homogeneous or heterogeneous architectures. The approach allows an easy portability to new emergent architectures.

Keywords Translators · Dynamic programming · Portability

1 Introduction

Current generation of computers is based on architectures based on multiple identical processing units composed of several cores (multicores) and it is expected that the number of cores per processor be incremented every year. It is also a well-known fact that the current generation of compilers is not being able to transfer automatically

A. Acosta (✉) · F. Almeida · I. Peláez
Department of Statistics and Computer Science, La Laguna University, S/C de Tenerife, Spain
e-mail: aacostad@ull.com

F. Almeida
e-mail: falmeida@ull.com

I. Peláez
e-mail: ignacio.pelaez@gmail.com

the capacity of the new processing units to the applications. The situation is further complicated given that current architectures are of heterogeneous nature, where this multicore systems can be combined, for example, with the capabilities of using GPU system as general purpose processing architectures. This fact constitutes a severe difficulty that appears in the form of a barrier to the programmability.

Many are the proposals to tackle with this problem. Leaving aside the proposals based on the development of new programming languages, due to inconveniences caused to the user (new learning effort and code reusability), many of the approaches are based in the source-to-source transformation of sequential code into parallel code, or on transforming parallel code designed for one architecture into parallel code designed for another [1–3]. Another different approach is based in the use of skeletons. The programmer is provided with a set of patterns already parallelized that constitutes a frame to develop parallel code, just by supplying sequential code [4–6]. It is worth also to mention the research on frameworks devoted to build the former source-to-source transformers [7–9].

Although technologically impressive, none of the projects based in skeletal parallel programming have achieved significant popularity in the wider parallel programming community. However, we claim that many of the developments made in the context of skeletal programming may play an important role in the automatic code generation based in source-to-source transformations. An important difficulty in the source-to-source transformation process is to transform sequential code sections into their parallel equivalent sections. That implies that the transformer must know in advance the sections to be parallelized, and how they should be translated, typically the user annotates the sections to be transformed.

An interesting feature of parallel skeletons is that they expose the parallelism exploitation pattern to the end user, while hiding (completely encapsulating) the implementation detail of the parallel pattern. New parallelizations (for new architectures for example) can be developed without any modification of the sequential code supplied by the user.

We have developed a source-to-source translator based on skeletons that generates code for many parallel architectures. The main goal is that the end user may obtain parallel code, without any knowledge in programming, just by defining her problem using a more natural language as the mathematics. An advantage of our approach is that, in general, a source-to-source transformation from sequential code to sequential code is semantically easier to develop that a transformation from sequential to parallel code. That is one of the fundamentals of our project, we automatically fill the sequential gaps in a parallel skeleton starting from a very user friendly specification. The parallelism is automatically provided by the skeleton and can be very easily extended. Since many parallel skeletons have been already developed and they work efficiently in current architectures, once the transformers have been developed, the level of productivity in terms of parallel code generated is highly increased.

As a proof of concept we apply the methodology to the dynamic programming technique, this technique is frequently applied to many research areas such as Control Theory, Operations Research, Biology, etc. [10–12]. As a result of this research it raises a specification language for Dynamic Programming problems that also constitutes a contribution of this work.

This remaining of the paper has been structured as follows: in Sect. 2 we present the methodology that we propose to broach the problem, in Sect. 3 we raise the framework developed in the context of Dynamic Programming problems and in Sect. 4 we include some computational results obtained from our tool and point out the high productivity achieved by the approach while keeping the efficiency at the same time. Finally we end the paper with some concluding remarks and future lines of work.

2 The methodology

Usually, source-to-source translators are used to make easier the work of developers. The source language use to have a higher abstraction level than the target language. Many translators have been developed and they typically follow the common structure that operates in two different phases, the Front-end and the Back-end. This division provides a high flexibility to the translator since the Front-end depends on the input language and is independent on the architecture, and the Back-end depends on the platform, and is independent from the input language. That allows reusing the same Back-end to generate output code starting from several input languages, if different Front-ends are used. At the same time, the same Front-end can be used to generate output codes for different architectures, if Back-ends adapted to the target platforms are used. Typically the Front-end and the Back-end use an intermediate language as an intermediate layer for a better management of this independence.

The skeleton-based programming use to have two types of code section: a generic code section that is valid for all the problems and an specific code section that is particular to each problem. Listing 1 shows a simple example of this programming model for synthetic problems that traverse matrices. The general code section for this type of problem could be the nested loop evaluating each element of the matrix, thus a simple skeleton will be formed by these loops. However, the operation to be made in each cell of the matrix may be different, depending on the problem to be solved, in this case the specific code section for each problem is represented by the *Evaluate* function. Using different implementations of this function various different

Listing 1 Example of a Programming model based on skeletons

```
for (int i = 0; i < sizeA; i++) {      # General Traversing Loop
  for (int j = 0; j < sizeB; j++) {    # for a matrix
    evaluate(i,j);
  }
}

void evaluate (int i, int j) {         # Different evaluation methods
  C[i,j] = A[i,j] + B[i,j];            # of the matrix elements
}

void evaluate (int i, int j) {
  for (int k = 0; k < sizeC; k++) {
    C[i,j] +=  A[i,k] * B[k,j];
  }
}
```

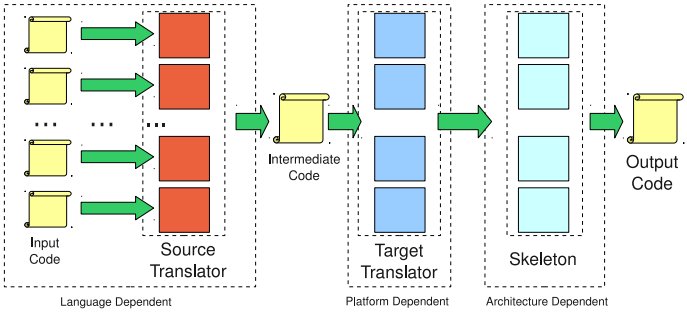

Fig. 1 Model for the proposed architecture

problems can be solved without modifying the code of the general skeleton. In this case the skeleton pattern is a sequential nested loop but different general patterns can be considered. A parallel version of this nested loop constitutes also a skeleton (the general code section), however, the specific code section (the *Evaluate* function) remains sequential.

In the skeletal based translation we propose to follow the basic source to source translation structure (Front-end/Back-end) but we introduce new layers providing an increased general abstraction view (Fig. 1). The proposal is close to that presented in [8]. In this case, we have a set of parallel programming skeletons that provide the general code for solving dynamic programming, the specific code for each problem is generated by the Back-End from a specification of the problem. This generated code is sequential, the parallelization is encapsulated within the skeleton, this enables the usage of different skeletons depending on the target architecture. New parallel patterns can be developed for new architectures and also the skeletons are suitable for static and dynamic optimizations. Note that we separate the source to source transformation from the parallelization. Transformations are only of sequential codes to sequential codes, while the parallelizations are abstracted into the skeletons.

Using this model, we propose a design structure where each one of the phases can be implemented as a source-to-source translator (Fig. 1). The Front-end may be seen as a source-to-source translator that generates an intermediate code, and the Back-end receives as input this intermediate source code and generates the output. By adding the skeletons to the translation structure, the model allows to develop a first *source translator* that generates intermediate code independent from the architecture, and also a second *target translator*, independent from the input intermediate language, that generates an output code adapted and optimized for different architectures.

This model is quite flexible since it allows the use the same *target translator*, and different *source translators*, to generate parallel code starting from various input languages. Or, at the same time, the use of the same *source translator*, and different *target translators*, to produce output code for different skeletal software platforms. The target code generated by the *target translator* can be used in many different parallel architectures just by combining the appropriate skeleton. New adapted skeletons can be developed for new emergent architectures without any change or new developments in the whole translation process.

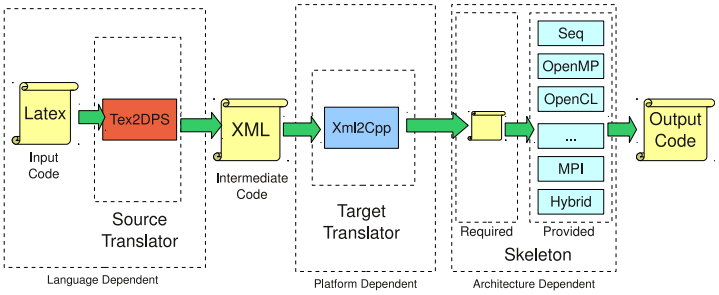

Fig. 2 The proposed software architecture

As a proof of concept we have implemented a source-to-source translator that follows this model (Fig. 2). The translator is directed to solve Dynamic Programming problems on parallel architectures. Of course, although the specific development of this paper is oriented to the Dynamic Programming technique, the same development model can be applied to other contexts.

3 The dynamic programming technique: a proof of concept

Dynamic Programming (DP) is an important problem-solving technique that has been widely used in various fields such as control theory, economy, biology and computer science [10–12]. In DP an optimal sequence of decisions is arrived at by making explicit appeal to the principle of optimality, this principle is the basis for the derivation of the dynamic programming recurrences as formalized, for example, in [13]. It is worth mentioning that a source of difficulties is the fact that the notation used changes substantially from one formalization to the other. In most of the cases, how to obtain the optimal policy providing the optimal solution is left outside of the formalizations, and usually remains expressed as a non-formalized, sometimes intuitive, procedure.

Most of the parallelizations presented for this technique are devoted to specific DP problems (see [14]) or are restricted to limited classes of recurrences. A unified parallel general approach was presented in [15] as an extension to the work of [13] but a strong theoretical effort is made in some cases. As far as we know, generic parallel approaches for DP are limited to classes of problems or they are not suitable to be easily assumed by a software component.

Analyzing the software approaches for DP, we found a group of general libraries for combinatorial optimization problems such as [16, 17]. They are used to supply interfaces for sequential and parallel executions but in most of the cases DP is not considered at all. Next, we can find specific DP sequential libraries such as [18], and interesting software approaches derived from laboratories that apply solvers to DP problems, following particular methodologies. In [19] we contributed with DPSKEL, a parallel skeleton where many efficient parallelizations for DP on different architectures are offered to the end user. The end user fills gaps on a C++ sequential code and the parallelism is automatically provided. In [20] we presented DPSPEC, a XML

Table 1 Latex specification for the Knapsack problem

$$InputData \equiv \begin{cases} n \in N & \text{\# The number of objects} \\ C \in N & \text{\# The capacity of the Knapsack} \\ p_k \in N;\ k \in \{0 \ldots n-1\} & \text{\# The profit of object } k \\ w_k \in N;\ k \in \{0 \ldots n-1\} & \text{\# The weight of object } k \end{cases}$$

$$OutputData \equiv \begin{cases} x_k \in \{0,1\};\ k \in \{0 \cdots n-1\} & \text{\# The solution vector} \\ n-1, C & \text{\# The index solution} \end{cases}$$

$$DecisionDef \equiv \left\{ d_{k,c} \in \{0,1\};\ k \in \{0 \cdots n-1\};\quad c \in \{0 \cdots C\} \right. \quad \text{\# The decisions}$$

$$DPRecurrence \equiv f_{k,c} = \begin{cases} 0 \Rightarrow d_{k,c} = 0 & \text{if } c < w_k \\ p_k \Rightarrow d_{k,c} = 1 & \text{if } k = 0 \text{ and } c \geq w_k \\ \max\{f_{k-1,c} \Rightarrow d_{k,c} = 0,\ f_{k-1,c-w_k} + p_k \Rightarrow d_{k,c} = 1\} & \text{if } k \neq 0 \text{ and } c \geq w_k \end{cases}$$

$$FormerDecision \equiv \begin{cases} & \text{\# Assign solution } k \\ x_k = d_{k,c} & \text{if } k \geq 0 \text{ and } c \geq 0 \\ & \text{\# Next decision to assign} \\ k-1; c-(w_k * x_k); & \text{if } k > 0 \text{ and } c \geq (w_k * x_k) \end{cases}$$

specification for DP problems that could be used as an alternative instead of the C++ interface for the DP parallel skeletons. Although for a scientist (a biologist, a physician, or an economist) XML is easier to manage, it still remains as a nonnatural approach. On the other side, the problem of finding the optimal policy after the optimal value is computed remained unsolved at that moment.

As a contribution of this paper, we propose a new specification language for DP problems that integrates all the elements of the DP technique, including how to compute the optimal policy. DP problems using this specification can be transformed automatically into our parallel skeletons through our intermediate language DPSPEC. To achieve it, DPSPEC and the parallel skeletons have been conveniently extended.

The input defines a structure where the user can define the DP problem without any knowledge of programming, using a more natural language as the mathematics. The code for this structure is defined using the LaTeX processor, widely used by the scientific community. We define a template where the user can define the problem to be solved and its parameters. We illustrate this specification using the well-known DP approach for the Knapsack Problem (Table 1). As we can see the input data and solution for a problem are described at the *InputData* and *OutputData* sections respectively, the DP recurrence in the section *DPRecurrence*. Note that when this section is defined, the decisions $d_{k,c}$ are included so that the optimal policy can be obtained from the specification presented in section *FormerDecision*.

To transform from the LaTeX specification to DPSPEC code, we use the `TtM` (TeX to `MathML`) translator [21], which can be used to translate LaTeX mathematical equations to the `MathML` [22] standard recommended for mathematical equations on Internet. To generate the DPSPEC code we have developed an ad-hoc translator that parses the `MathML` document and transforms it into the DPSPEC specification. The use of LaTeX is just a proof of concept, however from the methodological point of view, any other transformer producing the intermediate DPSPEC code is valid. The transition from DPSPEC to C++ is done using the translation scheme presented in

[20] (XML2CPP in Fig. 2), which generates C++ specific code to each problem (Required sections in Fig. 2). This code is added to the skeletons, as discussed above, and contains the general code for all problems (provided sections in Fig. 2). The flexibility in our approach allows developing new translators from XML to another library of skeletons providing new or different functionalities.

Since skeletons are defined for different architectures, shared memory, message passing, hybrid (Fig. 2), the methodology provides a huge portability, particularly if one consider that extending the approach to a new emergent platform, just means to include a new parallel skeleton.

4 Computational results

To validate our methodology and the framework that we have developed, we tested with several DP problems (Table 2). Five different DP recurrences have been considered that are quite representative of a wide class of problems. Note that the data dependences are different in most of the formulas considered. That means that different parallel traverses of the DP table can be required. We just represented the DP problems using our LaTeX specification language and automatically generated the parallel codes. Five skeletons have been considered, the sequential one and parallel versions on OpenMP, MPI, Hybrid (MPI+ OpenMP) and MPI/ULL_CALIBRATE. This last version is a distributed memory MPI version combined with the ULL_CALIBRATE library [23] to optimize in run time through dynamic load balancing.

The parallel platform used to execute our experiments is an AMD Opteron 6128 node (four processors, each processor composed of eight cores), with 32 cores sharing the memory. For the parallel executions, we conducted tests on all of the skeletons, varying the number of cores used from 2 to 32. For the hybrid skeleton the number of MPI processes was varied (from 2 to 32), as were the number of OpenMP threads launched by each process (from 1 to 16), restricting the total number of threads to no more than the number of cores in the system. To simplify the experience, the tests

Table 2 Dynamic Programming test problems

Problem	Recurrence
0/1 Knapsack KP	$f_{i,j} = \max\{f_{i-1,j}, f_{i-1,j-w_i} + p_i\}$
Resource Allocation MPP	$f_{i,j} = p_{1,j}$ if $i = 1$ and $j > 0$ $f_{i,j} = \max_{0 \le k < j}\{f_{i-1,j-k} + p_{i,k}\}$ if $(i > 1)$ and $(j > 0)$
Matrix Parentization RAP	$f_{i,j} = \min_{i \le k < j}\{f_{i,k} + f_{k+1,j} + (dim_i * dim_{k+1} * dim_{j+1})\}$
Triangulation Convex Polygons TCP	$f_{i,j} = cost_i * cost_{i+1} * cost_{i+2}$ if $(i = (j-2))$ $f_{i,j} = \min_{i<k<j}\{f_{i,k} + f_{k,j} + (cost_i * cost_k * cost_j)\}$ if $(i \neq j-1)$
Guillotine Cut GCP	$f_{i,j} = \max \begin{cases} \max_{0 \le k < \text{object}}\{\text{profit}_k\} \\ \max_{0 \le z \le i/2}\{f_{z,j} + f_{i-z,j}\} \\ \max_{0 \le y \le j/2}\{f_{i,y} + f_{i,j-y}\} \end{cases}$

Table 3 Running times in seconds for the sequential execution on the test problems

Size	1000	2000	5000
KP	0.4	1.63	10.2
RAP	20	162	2545
TCP	30	275	5128
MPP	31	282	5240
GCP	84	773	14033

have been developed using squared matrices of sizes 1000, 2000 and 5000. Note that according to the dependences of problems on Table 2, several traversing parallel approaches can be used to obtain the solution.

The parallel skeletons used compute rows in parallel in the case of the RAP and KP, the MPP and TCP are processed by computing the diagonals down-top in parallel and for the GCP the diagonals are computed in parallel top-down. The MPI/ULL_CALIBRATE skeleton was only implemented to solve problems where the matrices were accessed by rows.

Table 3 shows the running times of the sequential executions for all the proposed problems. All the times are expressed in seconds. This table provides a general view on the granularity of each problem. All the times are expressed in seconds. We can see as the KP is the problem with the finest granularity.

Figure 3 shows the speedups for instances of size 5000 for each problem. We can see as for fine grain problems (as the KP) the OpenMP skeleton provides the best results. When the granularity of the problem is increased (in the other of problems), the differences among the skeletons disappear, this is due to the low ratio computation/communication in fine grain problems that penalizes to the message passing skeletons. As for the hybrid skeleton, no significant differences emerge when the number of nodes and threads is varied for the same number of processes. The image shows the speedup of the best time obtained for all possible combinations. With the exception of KP, all the problems show increasing speedups with all the skeletons. The TCP and RAP reach an speedup close to 30 when using 32 cores. For the OpenMP skeleton, the speedup does not increase after 16 cores. Since this behavior is not observed with the MPI skeleton we suggest that this is likely due to the memory access conflicts, so the OpenMP skeleton still has some opportunities for improvement through optimizations on data alignment and cache access. The use of the mixed MPI/ULL_CALIBRATE (labeled HET in the figure) skeleton introduces a significant improvement in some of the problems. This skeleton takes advantage of the heterogeneous nature of the irregular recurrences, as that appearing in the RAP, to develop a dynamic load balancing of the rows. We can see the benefits obtained from the parallelization with the little effort of development imposed by our tool.

5 Conclusion and future work

We propose a source-to-source transformation methodology based in skeletal programming. The model is quite flexible and allows high levels of code reusability,

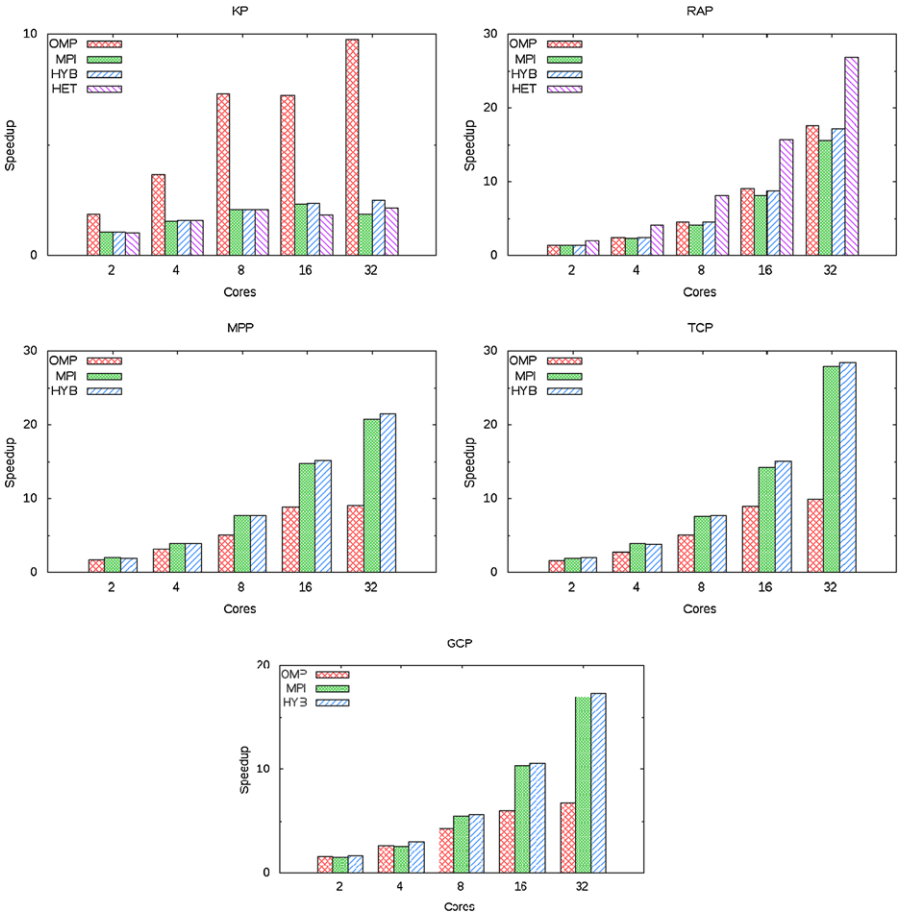

Fig. 3 Speedup for all the skeletons

portability and productivity. Since the parallel structure is decoupled from the translation model, no loss of efficiency is introduced by the approach. We applied the technique to Dynamic Programming problems. Several different problems expressed in LaTeX are automatically transformed into parallel programs that follow different parallelization implemented in various parallel libraries. The efficiency of the parallel code generated has been proved. For the near future we will be involved in two research directions. At the level of the skeletons we aim to extend DPSKEL with an OpenCL new skeleton, which would allow the portability of our methodology to GPUs. At the level of the Back-end translator, we propose to generate code for a new output language from the intermediate language. This output combined with the OpenCF [24] framework will provide web services interfaces so that the translators, and the parallel platforms, can be used transparently through web interfaces.

Acknowledgements This work has been supported by the EC (FEDER) and the Spanish MEC with the I+D+I contract number TIN2008-06570-C04-03 and TIN2011-24598, and by the Canary Government project SolSubC200801000307.

References

1. Schordan M, Quinlan DJ (2003) A source-to-source architecture for user-defined optimizations. In: JMLC, pp 214–223
2. Dooley I (2006) Automated source-to-source translations to assist parallel programmers. Master's thesis, Dept. of Computer Science, University of Illinois. http://charm.cs.uiuc.edu/papers/DooleyMSThesis06.shtml
3. Ueng SZ, Lathara M, Baghsorkhi SS, Hwu WMW (2008) Cuda-lite: reducing gpu programming complexity. In: LCPC08. LNCS, vol 5335. Springer, Berlin, pp 1–15
4. Bischof H, Gorlatch S (2002) Double-scan: introducing and implementing a new data-parallel skeleton. In: Proceedings of the 8th international Euro-Par conference on parallel processing, Euro-Par'02. Springer, London, pp 640–647
5. Cole M (2004) Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. Parallel Comput 30:389–406
6. González-Vélez H, Leyton M. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. Softw Pract Exper
7. ROSE. www.rosecompiler.org
8. Benkner S, Mehofer E, Pllana S (2008) Towards an intelligent environment for programming multicore computing systems. In: Proceedings of the 2nd workshop on highly parallel processing on a chip (HPPC 2008), in conjunction with Euro-Par 2008
9. Dave C, Bae H, Min S-J, Lee S, Eigenmann R, Midkiff SP (2009) Cetus: a source-to-source compiler infrastructure for multicores. IEEE Comput 42(11):36–42
10. Nascimento J, Powell W (2010) Dynamic programming models and algorithms for the mutual fund cash balance problem. Manag Sci 56:801–815
11. Erdelyi A, Topaloglu H (2010) A dynamic programming decomposition method for making overbooking decisions over an airline network. INFORMS J Comput 22:443–456
12. Huang K, Liang Y-T (2011) A dynamic programming algorithm based on expected revenue approximation for the network revenue management problem. Transp Res, Part E, Log Transp Rev 47(3):333–341
13. Ibaraki T (1988) Enumerative approaches to combinatorial optimization, part II. Ann Oper Res 11:1–4
14. Andonov R, Rajopadhye S (1997) Optimal orthogonal tiling of 2-D iterations. J Parallel Distrib Comput 45:159–165
15. Morales D, Almeida F, Rodríguez C, Roda J, Delgado A, Coloma I (2000) Parallel dynamic programming and automata theory. In: Parallel computing
16. Eckstein J, Phillips CA, Hart WE (2000) In: PIcO: an object-oriented framework for parallel branch and bound. Technical report, RUTCOR
17. Le Cun B (2001) Bob++ library illustrated by VRP. In: European operational research conference (EURO'2001), Rotterdam, p 157
18. Lubow BC (1997) SDP: generalized software for solving stochastic dynamic optimization problems. Wildl Soc Bull 23:738–742
19. Peláez I, Almeida F, Suárez F (2007) Dpskel: a skeleton based tool for parallel dynamic programming. In: Seventh international conference on parallel processing and applied mathematics, PPAM2007
20. Peláez I, Almeida F, González D (2006) An xml specification for automatic parallel dynamic programming. In: International conference on computational science (1), pp 872–875
21. TtM a TeX to MathML translator. http://hutchinson.belmont.ma.us/tth/mml/
22. W3C. W3c math home. http://www.w3.org/Math/
23. Galindo I, Almeida F, Blanco V, Badía JM (2008) Dynamic load balancing on dedicated heterogeneous systems. In: Lastovetsky A, Kechadi T, Dongarra J (eds) Recent advances in parallel virtual machine and message passing interface, EuroPVM/MPI 2009, pp 64–74
24. Santos A, Almeida F, Blanco V, Díez D, Regueira J, Sicilia E (2008) Towards automatic service generation and scheduling in the OpenCF project. Int J Web Grid Serv 4(4):367–378