J Supercomput (2011) 58:226–234
DOI 10.1007/s11227-009-0367-5

TRASGO: a nested-parallel programming system

Arturo González-Escribano · Diego R. Llanos

Published online: 10 December 2009

Abstract Programming models of pure nested-parallelism are appealing due to their ease of programming and good analysis and debugging properties. Although their simple synchronization structure is appropriate to represent abstract parallel algorithms, it does not take into account many implementation issues. In this work we present TRASGO, a programming system based on high-level, nested-parallel specifications. We show how it allows to easily express complex combinations of data and task parallelism with a common scheme, hiding the layout and scheduling details. The approach allows the development of a modular compiler where automatic transformation techniques may exploit lower level and more complex synchronization structures, unlocking the limitations of pure nested-parallel programming. This article presents an overview of the features of TRASGO, and its architecture. We present some performance results using well-known parallel algorithms, and a roadmap of improvements and new features to be added to TRASGO.

Keywords High-level programming models · Parallel compilers

1 Introduction

Many current high-performance scientific applications do not fully exploit their different levels of inherent parallelism, combining particular parallelization strategies. The diversity and complexity of modern parallel platforms make it very difficult to efficiently develop parallel applications in terms of the low-level concurrent programming model provided by the target machine. Important decisions in the implementation trajectory, such as choosing a scheduling scheme or a data-layout, become

A. González-Escribano · D.R. Llanos (✉)
Departamento Informática, Universidad de Valladolid, Valladolid, Spain
e-mail: diego@infor.uva.es

A. González-Escribano
e-mail: arturo@infor.uva.es

extremely difficult to optimize. Programmers need languages or tools that support unified, simple and combinable parallel specifications to express them. Tools should allow to capture design decisions and rely on the compiler and run-time system to do the associated complex mapping.

Message-passing portable APIs (e.g. MPI, PVM) are widely used in high-performance environments, as they propose an abstraction of the machine architecture, still obtaining good performance. However, programming directly with these unrestricted coordination models can be extremely error-prone and inefficient, as the synchronization dependencies that a program can generate are complex and difficult to analyze by humans or compilers [4].

More abstract and restricted programming models, such as nested-parallelism, are becoming an important trend in parallel programming, especially for multicore and other shared-memory platforms. Nested-parallel models represent a good trade-off between expressiveness, complexity and ease of programming [10]. They restrict the coordination structures and dependencies to those that can be represented by series-parallel (SP) task-graphs (DAGs). Due to the inherent properties of SP structures [11], they provide clear semantics and analyzability characteristics [5], a simple compositional cost model [8, 12] and efficient scheduling [2]. These properties can lead to automatic compilation techniques that increase portability and performance.

Previous research in our group has produced a highly-abstract XML intermediate representation for nested-parallel programs, named SPC-XML [3]. The sequential parts of the code are programmed in a convenient sequential language, such as C. These code pieces are programmed inside functions, specifying the input/output characteristic of each parameter. The functions have optional information of the asymptotic or average load of the code, based on the input sizes if needed. All this information simplifies the compiler data-dependence analysis and the runtime system load-balancing decisions.

The coordination algorithms are expressed by hierarchical XML tags. Recursive decompositions and parallel regions are easily expressed with clear semantics free of race conditions and deadlocks. Some attributes of the XML tags use generic names for mapping techniques which are provided as plug-ins in the system. Thus, they are highly extensible. Programming in SPC-XML reduces the development costs of parallel programs compared to directly using OpenMP or MPI. However, the quality of the automatic transformation system, and the implementation techniques included, is the key of the efficiency of the automatic-generated executables.

The extensibility properties of the framework give support to any specific set of compile-time scheduling and data-layout techniques, or generic runtime scheduling mechanisms, such as work-stealing. As an example, an extended version of SPC-XML has been successfully used for parallel stream-programming [14].

In this paper we present a complete development and compilation model called TRASGO (a Spanish name for a kind of goblin). This model introduces SPC-XML v0.7, a significant evolution of previous versions, as intermediate language representation. TRASGO performs expression analysis to build cost models and automatically generate the code needed for communications, instead of expanding complete task graphs as in previous SPC-XML systems. In SPC-XML v0.7 we have redesigned the unified `parallel` primitive in SPC-XML to better represent the mapping parameters in three levels, distinguishing logical processes, virtual topology and layout

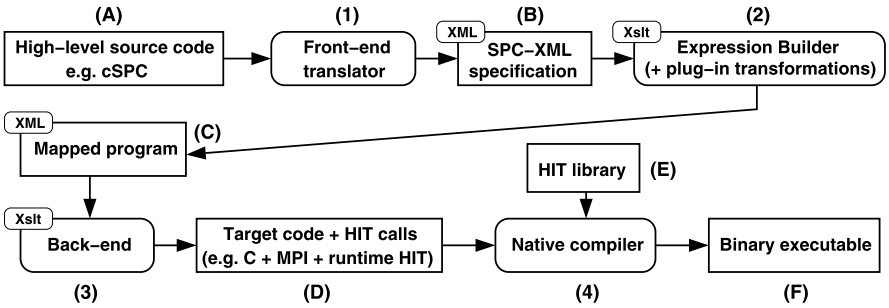

Fig. 1 TRASGO architecture

functions. Thus, the mapping modules are simpler and more composable, including simple, static data-layouts and dynamic load-balancing techniques. The programmer reasons in terms of logical (not physical) processes of any granularity. Task- and data-parallel programs are expressed with similar expressions and semantics.

The rest of the paper is organized as follows. Section 2 describes the TRASGO architecture, including the different stages of the compilation pipeline and the use of the SPC-XML intermediate representation. Section 3 shows some experimental results obtained using TRASGO to develop and execute an example application. Section 4 discusses some related work. Finally, Sect. 5 concludes the paper and presents some future work.

2 TRASGO architecture

We present a complete development and compilation model, called TRASGO. The core of TRASGO is the new SPC-XML v0.7 transformation system, together with a new automatic mapping approach, front-end languages and translators and an improved runtime system. We discuss here key features of TRASGO.

The TRASGO model architecture is depicted in Fig. 1. The input for the TRASGO model is an explicit, nested-parallel code that can be written in any traditional sequential language with some parallel extensions (A). As an example, we have developed an extension to C-language named CSPC, supporting all SPC-XML features. The particular language used is decoupled from the system through the use of a front-end translator (1) that generates SPC-XML intermediate code (B). Part of the power of the SPC-XML internal representation [3] is the existence of versatile XML tools, such as XPath and Xslt. These tools allow to detect structure properties of the document and to apply document transformations. TRASGO exploits them through an expression builder, and several plug-in transformation modules (2). They add annotations, new tags, and transform the initial SPC-XML document into a new one, including all the mapping and communication information needed (C). Finally, a back-end (3) translates this XML code into a target code (D), written in a given sequential language, which is linked against a standard communication library and a new runtime library (E), named HITMAP, developed by the authors as part of the TRASGO environment. Currently we provide a back-end which generates complete MPI programs

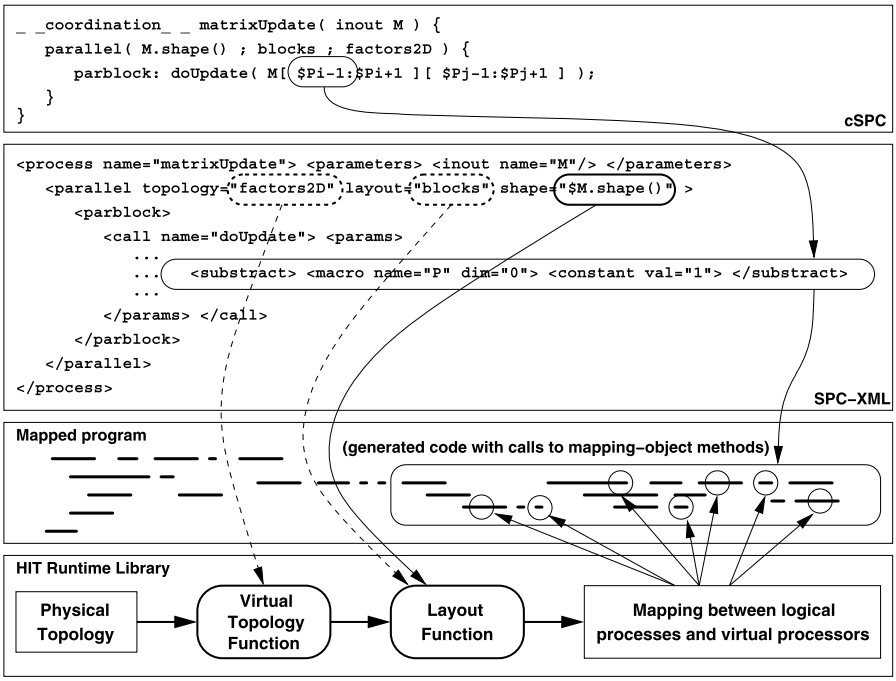

Fig. 2 Details of code transformation phases of TRASGO

in C-language, and a C-implementation of the HITMAP functionalities, ready for the native compiler (4) to produce the binary executable (F).

We show in Fig. 2 a simple excerpt of CSPC code and part of the associated XML representation. For simplicity, we omit the full tag tree for all the involved expressions. In CSPC there are two types of functions. The first type are classic C functions containing standard sequential C-code, but adding an `in`, `out` or `inout` modifier to each parameter. All data manipulation should be enclosed in one of these functions to define a clear interface behavior for the computational sequential code pieces. The second type of functions are preceded by a `__coordination__` modifier. Inside these functions only calls to other functions and coordination primitives are allowed, but not arbitrary data manipulation code. This clear separation of sequential code pieces and coordination primitives simplifies the parsing and recognition of the synchronization structure of the application in terms of the underlying SPC process algebra (see e.g. [12]).

In Fig. 2 we also show a brief example of the code generation process and how the generated target code interacts with the underlying runtime library. A key part of TRASGO is the unified `parallel` primitive. It allows to spawn independent parallel processes, map them automatically to the available processors, and build a global state at the end. This `parallel` primitive hides the details of the implementation and potential dynamic decisions that otherwise should be taken by the programmer.

It has the following format:

```
parallel( shape ; layoutFunction() ; topologyFunction() )
```

We have reordered the equivalent tag attributes in Fig. 2 for drawing clarity. This primitive, redesigned in SPC-XML v0.7, has three parameters which clearly split the design decisions at three different levels of abstraction: (1) a declaration of the amount of logical processes (computational units of any grain) to spawn in parallel, called `shape`; (2) a layout function name that will be used at runtime to map the logical processes to virtual processors; and (3) the name of a function that will generate, also at runtime, a virtual topology of processors. The last two parameters can be chosen from different options provided in the HitMap library, such as *blocks*, *cyclic*, or other classical layouts. Thus, the programmer never reasons in terms of the number of processors, and does not need to develop complex formula to calculate data partitions or communications. Getting rig of resource constraints allows to use the levels of granularity or decomposition most appropriate to express the algorithm semantics.

The code associated with the logical processes is specified inside the body, using one or more `parblock` statements. The `parallel` primitive supports replication of the same code on each logical process or different codes for each task. Thus, data- and task-parallelism are supported by a single primitive. The parameters of the `parallel` primitive are used to select the layout and virtual topology functions that will be invoked by the target code at runtime (see Fig. 2). These functions will use the available information about the physical topology to build a data-structure containing the mapping between logical processes and virtual processors. Expressions inside the `parblock` statements will be transformed to allow the use of this mapping information.

The transformation of the XML internal representation is guided by templates interpreted by a Xslt 2.0 processor. The Xslt language has powerful tools to detect given properties in parts of the document, and to manipulate the XML code accordingly. Thus, many typical structural and expression transformations are easily programmed with Xslt: expression simplification, propagation, loop transformations, etc.

In previous versions we expanded the whole application graph to apply data-flow analysis and detection of code structure. In the new compilation path an expression builder, together with several plug-in transformation modules, analyzes the expressions in the SPC-XML document and uses them to derive multiple new expressions. They include calls to the runtime mapping information in order to schedule the selected logical processes' code to the right processors, and to derive communications.[1]

The result is another XML document named "mapped program." In the example in Fig. 2, the `$Pi-1` expression is rewritten in SPC-XML. This expression refers to the index of the *neighbor* of the logical process on the first dimension. The mapping mechanism will group logical processes in *neighborhoods* following the mapping

[1] Other plug-in modules may detect opportunities to exploit low-level synchronization structures which are non-SP (e.g. barrier elimination, applying stencil-oriented skeletons, etc.), but keeping the original SP semantics.

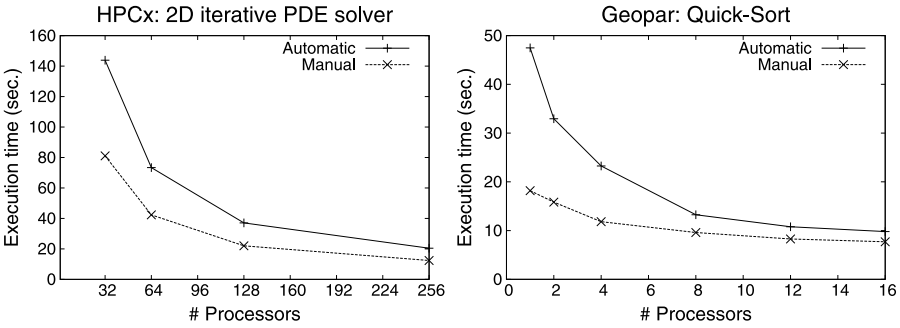

Fig. 3 Execution times of manually- and automatically-generated programs for a 2D iterative PDE solver (block partition), and a divide & conquer Quick-Sort algorithm

information provided by the runtime library, adjusting the communications accordingly. The HITMAP runtime library has support for hierarchical and cyclic tiling for multidimensional arrays, several functions for dynamic layout, virtual topologies, scheduling, communication of subarrays, etc.

3 Experimental results

In this section we present performance results obtained on real high-end machines to show the efficiency obtained by our automatically generated programs. We compare the execution times of reference MPI programs, manually developed and optimized, with those obtained by the codes automatically generated by TRASGO. Our example applications represent two important classes of programs: a 2D iterative PDE solver using block partition, based on data-parallelism; and a divide & conquer sorting algorithm, based on task-parallelism with a dynamic partition and load-balancing technique. The PDE solver is based on a 4-star stencil operation on each element of a 5000×5000 matrix. The program executes a fixed number of 5000 iterations. The Quick-Sort algorithm recursively balances the load across processors after each pivoting stage. We use an array of 50 millions of floating-point elements. In both cases, the high-level TRASGO programs use no more than three lines of code to express parallelism.

We present results obtained on two platforms. The first one is named HPCx. It is an IBM Power5 cluster formed by 16-processor nodes. The second one is named Geopar. It is a shared-memory machine formed by four Intel QuadCore boards. We show results for one example on each machine, but the following discussion also applies to results obtained on the other machine.

Figure 3 shows the execution times. In all cases we observe good scalability. However, the programs obtained by TRASGO present a performance penalty. The penalty is proportional to the number of sequential operations in the critical path, plus a small constant overhead. In the case of the PDE solver the number of sequential operations decreases proportionally to the number of processors. Thus, the penalty is a constant factor of the manual program execution time. The Quick-Sort algorithm executes

the sequential local stage in $O(n \log n)$ operations. As the number of processors increases, the number of sequential operations in the critical path reduces faster than n divided by the number of processors. Thus, the performance delay is also diminished. This effect is derived from: (1) constant inefficiencies on each data element access through the internal structures we use to manipulate the hierarchical tiling of arrays; and (2) bad optimization delivered by the native compiler when facing the data-partition expressions generated by TRASGO. Changing the problem data-sizes confirms the effects. The added constant overhead observed in all cases is generated by some inefficient buffer duplication and data management on the communication operations.

All these problems can successfully be solved by a new version of the HITMAP library and a simplified expression builder, which are currently being developed. The improvements include a simplification of the expressions generated for data-partition, and an evolution of the internal structures that handle matrix accesses, such as unnecessary pointer elimination. Despite these initial performance limitations, it is important to highlight that TRASGO provides excellent speedups at a modest fraction of the cost of a manual parallelization.

4 Related work

From the programmer's point of view, TRASGO provides the simple semantics, cost models, and implementation advantages of coarse-grain nested-parallel programming models, such as BSP or nested versions of BSP [9]. Nevertheless, it also supports automatic grain computation from simple fine-grain data-parallel expressions, like those used in HPF [13], avoiding complex decisions related to program mapping.

Pthreads and Java have nested *fork-join* mechanisms, but they are not particularly convenient for expressing data parallelism or automatically manipulate the computation grain. Cilk [2] was one of the first nested-parallel systems to fully exploit the efficient work-stealing scheduling technique, but this leads to similar grain problems. OpenMP targets only shared-memory. It provides different programming approaches, synchronized *parallel-for* structures, teams of coarse threads, and task-queue schedulings. However, specific non-SP coordination structures are difficult to be programmed efficiently without using the non-SP mechanisms provided by the language (like lock-variables, or sparse atomic operations).

CUDA [6] is a nested-parallel multithreading language for all-purpose programming on GPUs. The construction of parallel structures is somehow similar to our approach. The system automatically and dynamically balances the computations hiding the machine details. But the programmer should still reason with the number of threads spawned and with the computation grain. Lacking a powerful data-flow analysis, the structures of the code are not flexible and the synchronization barriers cannot be eliminated. Thus, the programmer also has access to shared-memory communications between grouped threads to skip the hard nested-parallel restrictions.

Intel Threading Building Blocks [7] present a conceptually similar approach, focused on data-parallel intensive computations. It solves the introduction of non-SP structures by adding a limited set of common simple patterns (like pipeline) as parallel constructors. The programmer needs to reason about more complex combinations

and structures, hindering further decomposition of parallelism under the same analyzability conditions. Concepts like our logical-process shapes and blocking-layout functions have equivalents on Intel TBBs. However, the layouts are more limited and the blocking size and grain level need to rely on programmer decisions or heuristics (in version 2.0). Too coarse-grained computations may not be exploiting all parallelism; but too fine-grained computations make the work-stealing scheduler work very inefficiently.

Finally, our supporting runtime library includes some features similar to other hierarchical tiled arrays libraries, such as [1].

5 Conclusion

We have presented the key features of TRASGO, a compiling system for pure nested-parallel programming. It is based on expressing parallelism with a simple and unified approach which hides low-level details, and focuses the programmer on design decisions. The system automatically applies transformations, and generates code which uses the static or dynamic information provided by the selected mapping techniques. The transformation system also adapts the grain of the computation to the available processors, simplifying the use of hierarchical task or data decompositions. TRASGO is an extensible system. Currently, it includes several plug-in transformation techniques, a runtime library supporting hierarchical tiling of dense arrays, and several predefined layout and scheduling techniques. The experimental results obtained show that TRASGO can derive efficient codes, from high-level and abstract specifications.

TRASGO is currently being improved and developed in several ways. While it is straightforward to develop new front-ends, back-ends and runtime libraries for other programming models are a challenge to tackle new portability issues. Simplified expression management will allow better data-flow dependence analysis and workload modeling. The runtime library is being reworked to implement more efficient management of hierarchical tiling for dense arrays, eliminating performance penalties. Support for sparse and more complex data-structures may be also considered.

Acknowledgements This research is partly supported by the Ministerio de Educación y Ciencia, Spain (TIN2007-62302), Ministerio de Industria, Spain (FIT-350101-2007-27, FIT-350101-2006-46, TSI-020302-2008-89, CENIT MARTA, CENIT OASIS), Junta de Castilla y León, Spain (VA094A08), and also by the Dutch government STW/PROGRESS project DES.6397. Part of this work was carried out under the HPC-EUROPA project (RII3-CT-2003-506079), with the support of the European Community–Research Infrastructure Action under the FP6 "Structuring the European Research Area" program. The authors wish to thank Dr. Valentín Cardeñoso-Payo, and Prof. Arjan van Gemund, for their support in the early stages of this research; and Dr. Mark Bull, Dr. Murray Cole, Prof. Michael O'Boyle, Prof. Henk Sips, and Ana Lucia Varbanescu for many helpful discussions.

References

1. Bikshandi G, Guo J, Hoeflinger D, Almasi G, Fraguela BB, Garzarn MJ, Padua D, von Praun C (2006) Programming for parallelism and locality with hierarchical tiled arrays. In: PPoPP'06. ACM, New York, pp 48–57

2. Blumofe RD, Leiserson CE (1994) Scheduling multithreaded computations by work-stealing. In: Proc annual symp on FoCS, pp 356–368
3. González-Escribano A, van Gemund AJC, Cardeñoso-Payo V (2005) SPC-XML: A structured representation for nested-parallel programming languages. In: Medeiros PD, Cunha JC (eds) Euro-par 2005, parallel processing. LNCS, vol 3648. ACM, New York, pp 782–792
4. Gorlatch S, (2001) Send-Recv considered harmful? Myths and truths about parallel programming. In: Malyshkin V (ed) PaCT'2001. LNCS, vol 2127. Springer, Berlin, pp 243–257
5. Lodaya K, Weil P (1998) Series-parallel posets: Algebra, automata, and languages. In: Proc STACS'98, Paris, France. LNCS, vol 1373. Springer, Berlin, pp 555–565
6. Nickolls J, Buck I, Garland M, Skadron K (2008) Scalable parallel programming with CUDA. ACM Queue 6(2):40–53
7. Reinders J (2007) Intel threading building blocks: outfitting C++ for multi-core processor parallelism. O'Reilly, Sebastopol
8. Skillicorn DB (1995) A cost calculus for parallel functional programming. J Parallel Distrib Comput 28:65–83
9. Skillicorn DB (1996) miniBSP: A BSP language and transformation system. Technical report, Dept of Computing and Information Sciences, Queen's University, Kingston, Canada
10. Skillicorn DB, Talia D (1998) Models and languages for parallel computation. ACM Comput Surv 30(2):123–169
11. Valdés J, Tarjan RE, Lawler EL (1982) The recognition of series parallel digraphs. SIAM J Comput 11(2):298–313
12. van Gemund AJC (1997) The importance of synchronization structure in parallel program optimization. In: Proc 11th ACM ICS, Vienna, pp 164–171
13. VanderWiel SP, Nathanson D, Lilja DJ (1997) Complexity and performance in parallel programming. In: Proc HIPS'97, Geneva, Switzerland. IEEE Comput. Soc, Los Alamitos
14. Varbanescu AL, Nijhuis M, González-Escribano A, Sips H, Bos H, Bal H (2007) SP@CE-An SP-based programming model for consumer electronics streaming applications. Lect Notes Comput Sci 4382:33–48