Proceedings of the International Conference
on Computational and Mathematical Methods
in Science and Engineering, CMMSE 2009
30 June, 1–3 July 2009.

Evolution of a Nested-Parallel Programming System

Arturo Gonzalez-Escribano[1] and Diego R. Llanos[1]

[1] *Departamento de Informática, University of Valladolid (Spain)*

emails: arturo@infor.uva.es, diego@infor.uva.es

Abstract

Pure nested-parallelism programming models 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 key features of the evolution of a programming system based on high-level, nested-parallel specifications. They allow to easily express complex combinations of data and task parallelism with a common scheme, and to hide 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.

Key words: High-level programming models, parallel compilers

1 Introduction

Many current high-performance scientific applications may exploit several levels of parallelism with different strategies of parallelization. Programmers need languages or tools that support unified, simple and combinable parallel specifications to express them. They should allow to capture design decisions and rely on the compiler and run-time system to do the complex mapping associated.

The diversity and complexity of modern parallel platforms makes 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 an scheduling scheme or a data-layout, become extremely difficult to optimize.

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 [5].

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 represents a good tradeoff 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). Thanks to the inherent properties of SP structures [11], they provide clear semantics and analyzability characteristics [6], a simple compositional cost model [3, 9] and efficient scheduling [2]. These properties can lead to automatic compilation techniques that increase portability and performance.

2 Previous research

Previous research in our group has produced a highly-abstract XML intermediate representation for nested-parallel programs, named SPC-XML [4]. 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, and 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 run-time 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 dead-locks. 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. Some attributes of the XML tags use generic names for mapping techniques which are provided as plug-ins in the system. Thus, it is highly extensible. Programming in SPC-XML reduces the development costs of parallel programs comparing with directly using OpenMP or MPI. However, the quality of the automatic transformation system, and the implementation techniques included, are the key of the efficiency of the automatic-generated executables.

3 SPC-XML development

The SPC-XML compilation system is being completely redesigned. We discuss here key features of this new version and the related ongoing research:

Front-end language: We have developed an extension to C-language named cSPC. It supports all the features of the SPC-XML framework. A simple front-end transforms the cSPC programs to the XML intermediate representations. Figure 2 show a simple excerpt of cSPC code, with part of its associated XML representation below.

```
parallel( matrix.shape() ; blocks ; factors2D ) {
  parblock: doUpdate( matrix[ $P(0)-1:$P(0)+1 ][ $P(1)-1:$P(1)+1 ] );
}
```

```
<parallel shape="$matrix.shape()" layout="blocks" topology="factors2D">
  <parblock>
     <call name="doUpdate"><params>
        ...
     </params></call>
  </parblock>
</parallel>
```

Figure 1: Example of a cSPC piece of code for a parallel matrix computation and a excerpt of the associated XML representation

Three-tier parallel primitive: We have redesigned the simple and unified parallel primitive. It has three parameters: (1) A declaration of the amount of logical tasks to spawn in parallel, called a *shape*; (2) a layout function to map the logical tasks to processors; and (3) the name of a function to generate a virtual topology of processors. Inside the structure, the code associated to the logical tasks is specified. It supports replication of the same code on each logical process or different codes for each task. Thus, data and task-parallelism are supported by the same primitive.

This new scheme clearly splits the design decisions at three different levels of abstraction. It hiddens the low-level details of their implementation and potential dynamic decisions inside the layout and topology functions. The programmer never reasons in terms of the number of processors, and does not need to write complex formula to calculate data partitions or communications.

XML transformations: In previous versions we were expanding the whole application graph to apply data-flow analysis and detection of code structure. In the new compilation path we have substituted it by an expression-analyzer and modules for code transformations.

The transformation tools for the XML internal representation are guided by templates interpreted by a Xslt 2.0 processor. 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, propagations, loop transformations, etc. The transformations may detect opportunities to exploit low-level synchronization structures which are non-SP. For example: By barrier elimination, applying stencil-oriented skeletons, etc.

Back-end: The resulting XML document is translated to target code by a back-end.

Currently we provide a back-end which generates complete MPI programs in C language ready for the native compiler.

Layout and topology modules inject information in the XML representation which is translated as expressions or calls to run-time functions. They compute the subparts of data structures needed on each physical processor and the data which should be accessed synchronously or communicated across processors.

Run-time library: The generated code relies on a run-time library which has support for efficient hierarchical and cyclic tiling for multidimensional arrays, several functions for dynamic layout, virtual topologies, scheduling, communication of subarrays, etc.

Experimental work: To show the efficiency of the generated code, we are using cSPC to implement standard and well tested programs, such as the NAS Parallel Benchmarks. As their parallel structures are implemented with MPI, we may better compare the efficiency of the generated programs and the amount of lines of high-level code needed to exploit the parallelism.

4 Related work

Our supporting run-time library includes some similar features as other hierarchical tiled arrays libraries (see e.g. [1]).

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 [7] is a nested-parallel multi-threading 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 hidding 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 have access to shared-memory communications between grouped threads to skip the hard nested-parallel restrictions.

Intel Threading Building Blocks [8] present a conceptually similar approach, focused on data parallel intensive computations. It solves the introduction of non-SP structures 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-processes shape 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.

5 Conclusion

We have presented the key features of the current evolution of SPC-XML, 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 focus the programmer on design decisions. The system automatically detects code suitability for different static or dynamic scheduling techniques, and adapts the grain to the available processors. These features are key for the easy generation of efficient programs, for distributed or shared-memory environments, from the same portable code.

Acknowledgements

This research was 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" Programme.

References

[1] G. Bikshandi, J. Guo, D. Hoeflinger, G. Almasi, B.B. Fraguela, M.J. Garzarn, D. Padua, and C. von Praun. Programming for parallelism and locality with hierarchical tiled arrays. In *PPoPP'06*, pages 48–57. ACM Press, March 2006.

[2] R.D. Blumofe and C.E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proc. Annual Symp. on FoCS*, pages 356–368, Nov 1994.

[3] A.J.C. van Gemund. The importance of synchronization structure in parallel program optimization. In *Proc.* 11th *ACM ICS*, pages 164–171, Vienna, Jul 1997.

[4] A. González-Escribano, A.J.C. van Gemund, and V. Cardeñoso-Payo. SPC-XML: A structured representation for nested-parallel programming languages. In P.D. Medeiros J.C. Cunha, editor, *Euro-Par 2005, Parallel Processing*, volume 3648 of *LNCS*, pages 782–792. ACM, IEEE, Springer-Verlag, 2005.

[5] S. Gorlatch. Send-Recv considered harmful? Myths and truths about parallel programming. In V. Malyshkin, editor, *PaCT'2001*, volume 2127 of *LNCS*, pages 243–257. Springer-Verlag, 2001.

[6] K. Lodaya and P. Weil. Series-parallel posets: Algebra, automata, and languages. In *Proc. STACS'98*, volume 1373 of *LNCS*, pages 555–565, Paris, France, 1998. Springer-Verlag.

[7] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *ACM Queue*, 6(2):40–53, Mar 2008.

[8] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly, Jul 2007.

[9] D.B. Skillicorn. A cost calculus for parallel functional programming. *Journal of Parallel and Distributed Computing*, 28:65–83, 1995.

[10] D.B. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2):123–169, Jun 1998.

[11] J. Valdés, R.E. Tarjan, and E.L. Lawler. The recognition of series parallel digraphs. *SIAM Journal of Computing*, 11(2):298–313, May 1982.