Program Optimization in the Domain of High-Performance Parallelism

Christian Lengauer

Fakultät für Mathematik und Informatik
Universität Passau, D–94030 Passau, Germany
lengauer@fmi.uni-passau.de

Abstract. I consider the problem of the domain-specific optimization of programs. I review different approaches, discuss their potential, and sketch instances of them from the practice of high-performance parallelism. Readers need not be familiar with high-performance computing.

1 Introduction

A program that incorporates parallelism in order to increase performance (mainly, to reduce execution time) is particularly difficult to write. Apart from the fact that its performance has to satisfy higher demands than that of a sequential program (why else spend the additional money and effort on parallelism?), its correctness is also harder to establish. Parallelism makes verification more difficult (one must prove the correctness not only of the individual processes but also of any overt or covert interactions between them) and testing less effective (parallel behaviour can, in general, not be reproduced).

Note that introducing parallelism for high-performance is a form of program optimization since, in high-performance computing, parallelism is never part of the problem specification. If one can get along without it – so much the better!

The present state of programming for high-performance parallel computers can be compared with the state of programming for sequential computers in the 1960s: to a large extent, one is concerned with driving a machine, not with solving a computational problem. One specifies, in the source program, the composition and synchronization of the parallel processes, and the communication between them. This can be viewed as assembly-level programming.

Thus, researchers in programming high-performance parallelism have been searching for commonly useful abstractions – much like researchers in sequential programming languages and their methodologies have been back in the 1960s and 1970s. Just as back then, their efforts are being resisted by the programmers in practice who feel that they need low-level control in order to tune program performance – after all, the performance is the sole justification for their existence! Just as back then, the price they pay is a lack of program structure, robustness and portability. However, one additional challenge in high-performance parallelism is the lack of a commonly accepted parallel machine model (for sequential programs we have the von Neumann model). One consequence is that, even if

the parallel program may be portable, its performance may not be but may differ wildly on different platforms.[1]

If we consider high-performance parallelism an application domain, it makes sense to view the programming of high-performance parallelism as a domain-specific activity. Thus, it is worth-while to assess whether techniques used in domain-specific program generation have been or can be applied, and what specific requirements this domain may have. To that end, I proceed as follows:

1. I start with a classification of approaches to domain-specific programming (Sects. 2–3).
2. I review some of the specific challenges in programming for the domain of high-performance parallelism (Sect. 4).
3. I review a variety of approaches to programming high-performance parallelism, place them in the classification of approaches to domain-specific programming and discuss their principles and limitations (Sects. 5–8). Here, I mention also all approaches to high-performance parallelism which are presented in this volume.
4. I conclude with a review of the main issues of raising the level of abstraction in programming parallelism for high performance (Sect. 9).

2 Domain-Specific Programming

What makes a language domain-specific is not clear cut and probably not worth worrying about too much. A debate of this issue would start with the already difficult question of what constitutes a domain. Is it a collection of users or a collection of software techniques or a collection of programs...?

For the purpose of my explorations we only need to agree that there are languages that have a large user community and those that have a much smaller user base, by comparison. Here I mean "large" in the sense of influence, manpower, money. I call a language with a large user community *general-purpose*. Probably undisputed examples are C and C++, Java and Fortran but, with this definition, I could also call the query language SQL general-purpose, which demonstrates that the term is meant in a relative sense.

A large user community can invest a lot of effort and resources in developing high-quality implementations of and programming environments for their language. A small user community has much less opportunity to do so, but may have a need for special programming features that are not provided by any programming language which other communities support. I call such features *domain-specific*. One real-world example of a language with a small user community, taken from Charles Consel's list of domain-specific languages on the Web, is the language Devil for the specification of device driver interfaces [1].

[1] On present-day computers with their memory hierarchies, instruction-level parallelism and speculation, a lack of performance portability can also be observed in sequential programs.

What if the small user community prefers some widely used, general-purpose language as a base, but needs to enhance it with domain-specific constructs to support its own applications? High-performance computing is such a community: it needs an efficient sequential language, enhanced with a few constructs for parallelism, synchronization and, possibly, communication. It has two options:

- It can view the combination of the two as a new, domain-specific language and provide a dedicated, special-purpose implementation and programming environment for it. However, this will put it at a disadvantage with respect to the large user community of the base language, if it is not able to muster a competitive amount of resources for the language development.
- It can attach a domain-specific extension to the base language [2]. In this case, the small community need not invest in the development of the general-purpose part of its language. Indeed, if the enhancement is crafted carefully, it may be possible to plug in a new release of the base language implementation, which has been developed independently, without too much trouble.

Let us look more closely at the latter approach for our particular purpose: program optimization.

3 Domain-Specific Program Optimization

Two commonly recognized benefits of domain-specific program generation are increased programming convenience and program robustness. But there is also the opportunity for increased program performance – via optimizations which cannot be applied by a general-purpose implementation. In many special-purpose domains, performance does not seem to be a major problem, but there are some in which it is. High-performance computing is an obvious one. Another is the domain of embedded systems, which strives for a minimization of chip complexity and power consumption, but which I will not discuss further here (see the contribution of Hammond and Michaelson [3] to this volume).

There are several levels at which one can generate domain-specifically optimized program parts. Here is a hierarchy of four levels; the principle and the limitations of each level are discussed further in individual separate sections:

Sect. 5: Precompiled Libraries. One can prefabricate domain-specific program pieces independently, collect them in a library, and make them available for call from a general-purpose programming language.

Sect. 6: Preprocessors. One can use a domain-specific preprocessor to translate domain-specific program pieces into a general-purpose language and let this preprocessor perform some optimizations.

Sect. 7: Active Libraries. One can equip a library module for a domain-specific compilation by a program generator, which can derive different implementations of the module, each optimized for its specific call context.

Sect. 8: Two Compilers. One can use a domain-specific compiler to translate domain-specific program pieces to optimized code, which is in the same target

language to which the general-purpose compiler also translates the rest of the program. The separate pieces of target code generated by the two compilers are then linked together.

The domain specificity of the optimization introduces context dependence into the program analysis. Thus, one important feature that comes for free is that the optimization of a domain-specific program part can be customized for the context of its use. This feature is present in all four approaches, but it becomes increasingly flexible as we progress from top to bottom in the list.

4 The Challenges of Parallelism for High Performance

Often the judicious use of massive parallelism is necessary to achieve high performance. Unfortunately, this is not as simple as it may seem at first glance:

- The program must decompose into a sufficient number of independent parts, which can be executed in parallel. At best, one should be able to save the amount in time which one invests in additional processors. This criterion, which can not always be achieved, is called *cost optimality*; the number of sequential execution steps should equal the product of the number of parallel execution steps and the number of required processors [4]. It is easy to spend a large number of processors and still obtain only a small speedup.
- Different processors will have to exchange information (data and, maybe, also program code). This often leads to a serious loss of performance. In contemporary parallel computers, communication is orders of magnitude slower than computation. The major part of the time is spent in initiating the communication, i.e., small exchanges incur a particularly high penalty. As the number of processors increases, the relative size of the data portions exchanged decreases. Striking a profitable balance is not easy.
- In massive parallelism, it is not practical to specify the code for every processor individually. In parallelism whose degree depends on the problem size, it is not even possible to do so. Instead, one specifies one common program for all processors and selects portions for individual processors by a case analysis based on the processor number.[2] The processor number is queried in decisions of what to do and, more fundamentally, when to do something and how much to do. This can lead to an explosion in the number of run-time tests (e.g., of loop bounds), which can cause a severe deterioration in performance.

The user community of high-performance parallelism is quite large, but not large enough to be able to develop and maintain competitive language implementations in isolation [5]. Let us investigate how it is faring with the four approaches to domain-specific program optimization.

[2] This program structure is called *single-program multiple-data* (*SPMD*).

5 Domain-Specific Libraries

5.1 Principle and Limitations

The easiest, and a common way of embedding domain-specific capabilities in a general-purpose programming language is via a library of domain-specific program modules (Fig. 1). Two common forms are libraries of subprograms and, in object-oriented languages, class libraries. Typically, these libraries are written in the general-purpose language from which its modules are also called.

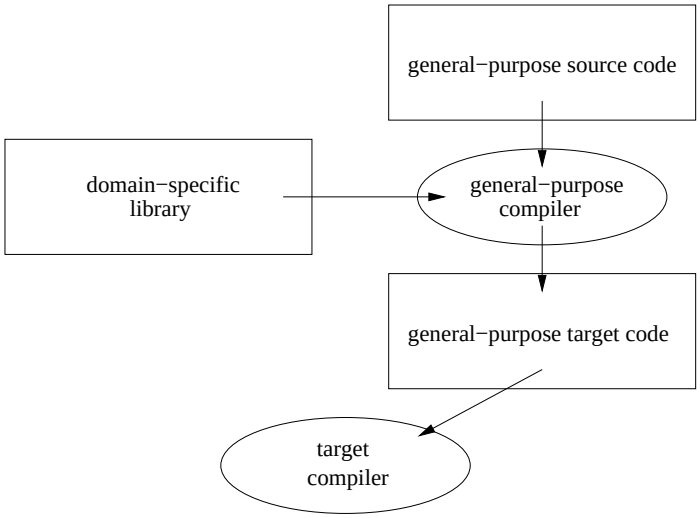

Fig. 1. Domain-specific library

In this approach, there is no domain-specific translation which could perform domain-specific optimizations. One way of customizing the implementation of a module for the context of its call is by asking the application programmer to pass explicitly additional, so-called structure parameters which convey context-sensitive information and which are queried in the body of the module.

The limitations of this approach are:

- Error messages generated in the library modules are often cryptic because they have been issued by a compiler or run-time system which is ignorant of the special domain.
- The caller of the module is responsible for setting the structure parameters consistently. This limits the robustness of the approach.
- The implementer of the module has to predict all contexts in which the module may be called and build a case analysis which selects the given context. This limits the flexibility of the approach.

5.2 Examples from High-Performance Parallelism

MPI. One example from high-performance parallelism is the communication library *Message-Passing Interface* (*MPI*) [6]. It can be implemented largely in C, but with calls to system routines for communication and synchronization. There exist MPI interfaces for Fortran and C – the popular general-purpose languages in high-performance computing.[3] The Fortran or C code is put on each processor in an SPMD fashion.

The routines provided by the library manage processor-to-processor communication, including the packing, buffering and unpacking of the data, and one can build a hierarchical structure of groups of processors (so-called communicators) which communicate with each other. There are over 100 routines in the first version of MPI and over 200 in the second. With such high numbers of routines it becomes difficult for programmers to orient themselves in the library and even more difficult for developers to maintain a standard.

Programming with MPI (or similar libraries like the slightly older PVM [7]) has been by far the most popular way of producing high-performance parallel programs for about a decade. As stated before, this could be likened to the sequential programming with unstructured languages or assemblers in the 1960s. Programmers have relatively close control over the machine architecture and feel they need it in order to obtain the performance they are expecting.

The library BSP [8] works at a slightly higher level of abstraction: the SPMD program is divided into a sequence of "supersteps". Communication requests issued individually within one superstep are granted only at the superstep's end. This alleviates the danger of deadlocks caused by circular communication requests and gives the BSP run-time system the opportunity to optimize communications.

One limitation of BSP is that there is no hierarchical structure of communicating processes (something which MPI provides with the communicator). All processors are potential partners in every communication phase.

The developers of BSP are very proud of the fact that their library consists of only 20 functions, compared with the hundreds of MPI. Still, programming with BSP is not very much simpler than programming with MPI. However, BSP has a very simple cost model – another thing its developers are proud of. The model is based on a small number of constants, whose values have to be determined by running benchmarks. They can differ significantly between different installations – and even for different applications on one installation! This introduces a certain inaccuracy in the model.

Collective Operations. For sequential programs, there are by now more abstract programming models – first came structured programming, then functional and object-oriented programming. For parallel programs, abstractions are

[3] There is also an MPI interface for Java, but a more natural notion of parallelism for Java is thread parallelism, as supported by JavaParty (see further on).

still being worked out. One small but rewarding step is to go from point-to-point communication, which causes unstructured communication code like the `goto` does control code in sequential programs [9], to patterns of communications and distributed computations. One frequently occurring case is the reduction, in which an associative binary operator is applied in a distributed fashion to a collection of values to obtain a result value (e.g., the sum or product of a sequence of numbers).[4] A number of these operations are already provided by MPI – programmers just need some encouragement to use them! The benefit of an exclusive use of collective operations is that the more regular structure of the program enables a better cost prediction. (BSP has a similar benefit.) With architecture-specific implementations of collective operations, one can calibrate program performance for a specific parallel computer [10].

One problem with libraries of high-performance modules is their lack of performance compositionality: the sequential composition of two calls of highly tuned implementations does, in general, not deliver the best performance. Better performance can be obtained when one provides a specific implementation for the composition of the two calls.

In general, it is hopeless to predict what compositions of calls users might require. But, at a comparatively low level of abstraction, e.g., at the level of collective operations, it is quite feasible to build a table of frequently occurring compositions and their costs incurred on a variety of parallel architectures. Gorlatch [11] and Kuchen [12] deal with this issue in their contributions to this volume.

Skeleton Libraries. As one raises the level of abstraction in libraries, it becomes common-place that a library module represents a pattern of computation and requires pieces of code to be passed as parameters, thus instantiating the pattern to an algorithm. Such a pattern of computation is also called a *template* or *skeleton*. One of the simplest examples is the reduction, which is one of the collective operations of MPI and which is also offered in HPF (see Sect. 6.2).

One major challenge is to find skeletons which are general enough for frequent use, the other is to optimize them for the context of their use.

Contributions to this volume by Bischof et al. [13] and by Kuchen [12] propose skeleton libraries for high-performance computing.

More Abstract Libraries. The library approach can be carried to arbitrarily high levels of abstraction.

For example, there are several very successful packages for parallel computations in linear algebra. Two libraries based on MPI are ScaLAPACK [14] and

[4] With a sequential loop, the execution time of a reduction is linear in the number of operand values. With a naive parallel tree computation, the time complexity is logarithmic, for a linear number of processors. This is not cost-optimal. However, with a commonly used trick, called Brent's Theorem, the granularity of the parallelism can be coarsened, i.e., the number of processors needed can be reduced to maintain cost optimality in a shared-memory cost model [4].

PLAPACK [15]. The latter uses almost exclusively collective operations. Both libraries offer some context sensitivity via accompanying structure parameters which provide information, e.g., about the dimensionality or shape (like triangularity or symmetry) of a matrix which is being passed via another parameter.

Another, increasingly frequent theme in high-performance parallelism is the paradigm of divide-and-conquer. Several skeleton libraries contain a module for parallel divide-and-conquer, and there exists a detailed taxonomy of parallel implementations of this computational pattern [16,17].

Class Libraries. One popular need for a class library in high-performance computing is created by the language Java, which does not provide a dedicated data type for the multi-dimensional array – the predominant data structure used in high-performance computing. Java provides only one-dimensional arrays and allocates them on the heap. Array elements can be arrays themselves, which yields a multi-dimensional array, but these are not allocated contiguously and need not be of equal length. Thus, the resulting structure is not necessarily rectangular.

One reason why the multi-dimensional array is used so heavily in scientific computing is that it can be mapped contiguously to memory and that array elements can be referenced very efficiently by precomputing the constant part of the index expression at compile time [18]. There are several ways to make contiguous, rectangular, multi-dimensional arrays available in Java [19]. The easiest and most portable is via a class library, in which a multi-dimensional array is laid out one-dimensionally in row- or column-major order.

6 Preprocessors

6.1 Principle and Limitations

A preprocessor translates domain-specific language features into the general-purpose host language and, in the process, can perform a context-sensitive analysis, follow up with context-sensitive optimizations and generate more appropriate error messages (Fig. 2).[5] For example, the structure parameters mentioned in the previous section could be generated more reliably by a preprocessor then by the programmer.

While this approach allows for a more flexible optimization (since there is no hard-wired case analysis), one remaining limitation is that no domain-specific optimizations can be performed below the level of the general-purpose host language. Only the general-purpose compiler can optimize at that level.

6.2 Examples from High-Performance Parallelism

Minimal Language Extensions. To put as little burden on the application programmer as possible, one can try to add a minimal extension to the sequential

[5] Compile-time error messages can still be cryptic if they are triggered by the general-purpose code which the preprocessor generates.

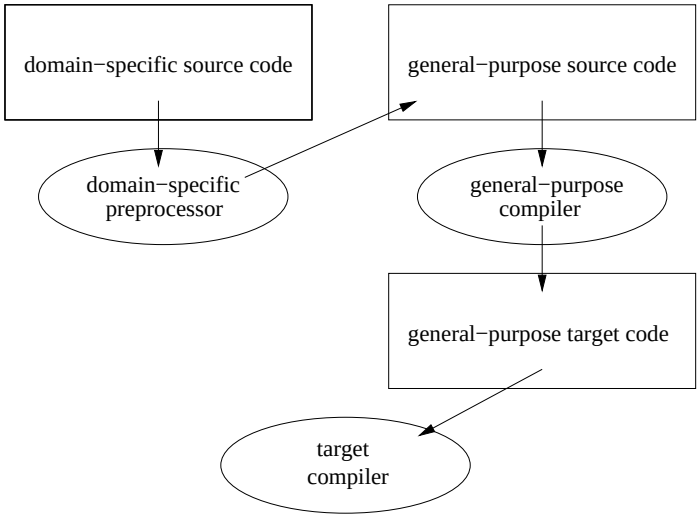

Fig. 2. Preprocessor

language. There are at least three commonly used languages which have been equipped this way. The internals of the compiler of the sequential language is not touched, but a parallel run-time system must be added. These language extensions are very easy to use. In particular, it is easy to parallelize existing programs which have been written without parallelism in mind.[6]

Cilk [20] is an extension of C for thread parallelism. It offers a good half dozen directives for spawning and synchronizing parallel threads and data access control. Cilk is based on a very smart run-time system for thread control and Cilk programs have won competitions and been awarded prizes (see the project's Web page).

Glasgow parallel Haskell (GpH) [21] is an extension of the functional language Haskell with just one function for spawning threads. Again, a special run-time system manages the threads.

JavaParty [22] is a preprocessor for Java to support the efficient computation with remote objects on clusters of processors. It adds one class attribute to Java: `remote`, which indicates that instances of the class can be moved to another processor. While Cilk and GpH are doing no preprocessing, JavaParty does. The preprocessor generates efficient code for object migration, finding a suitable location (processor), efficient serialization (coding the object for a fast transfer) and efficient garbage collection (exploiting the assumption of an unfailing network). An object may be migrated or cloned automatically, in order to increase the locality of accesses to it.

[6] These days, we might call them "dusty disk programs".

HPF. The most popular language extension in high-performance parallelism, supported by a preprocessor, is High-Performance Fortran (HPF) [23]. HPF is a modern version of sequential Fortran, enhanced with compiler directives, This cannot be considered a minimal extension, and one needs to have some expertise in parallelism to program HPF effectively. There are four types of directives: one for defining virtual processor topologies, one for distributing array data across such topologies, one for aligning arrays with each other, and one for asserting the independence of iterations of a loop.

HPF is at a much higher level of abstraction than Fortran with MPI. A program that takes a few lines in HPF often takes pages in Fortran with MPI:[7] the communication and memory management which is taken care of by the HPF compiler must be programmed explicitly in MPI.

Most HPF compilers are preprocessors for a compiler for Fortran 90, which add calls to a library of domain-specific routines which, in turn, call MPI routines to maintain portability. One example is the HPF compiler ADAPTOR with its communications library DALIB [25].

There was justified hope for very efficient HPF programs. The general-purpose source language, Fortran, is at a level of abstraction which is pleasingly familiar to the community, yet comparatively close to the machine. Sequential Fortran is supported by sophisticated compilers, which produce highly efficient code. And, with the domain-specific run-time system for parallelism in form of the added library routines, one could cater to the special needs in the domain – also with regard to performance.

However, things did not quite work out as had been hoped. One requirement of an HPF compiler is that it should be able to accept every legal Fortran program. Since the HPF directives can appear anywhere and refer to any part of the program, the compiler cannot be expected to react reasonably to all directives. In principle, it can disregard any directive. The directives for data distributions are quite inflexible and the compiler's ability to deal with them depends on its capabilities of analyzing the dependences in the program and transforming the program to expose the optimal degree of parallelism and generate efficient code for it. Both the dependence analysis and the code generation are still areas of much research.

Existing HPF compilers deal well with fairly simple directives for scenarios which occur quite commonly, like disjoint parallelism or blockwise and cyclic data distributions. However, they are quite sensitive to less common or less regular dependence patterns: even when they can handle them, they often do not succeed in generating efficient code. Work on this continues in the area of loop parallelization (see further on).

With some adjustments of the paradigm, e.g., a more explicit and realistic commitment to what a compiler is expected to do and a larger emphasis on loop parallelization, a data-parallel Fortran might still have a future.

[7] E.g., take the example of a finite difference computation by Foster [24].

OpenMP. Most of HPF is *data-parallel*: a statement, which looks sequential, is being executed in unison by different processors on different sets of data. There are applications which are *task-parallel*, i.e., which require that different tasks be assigned to different processors. The preprocessor OpenMP [26] offers this feature (more conveniently than HPF) and is seeing quite some use – also, and to a major part, for SPMD parallelism. It is simpler to implement than an HPF preprocessor, mainly because it is based on the shared-memory model (while HPF is for distributed memory) and lets the programmer specify parallelism directly rather than via data distributions. On the downside, if the shared memory has partitions (as it usually does), the run-time system has the responsibility of placing the data and keeping them consistent.

Loop Parallelization. One way to make an HPF compiler stronger would be to give it more capabilities of program analysis. Much work in this regard has been devoted to the automatic parallelization of loop nests.

A powerful geometric model for loop parallelization is the *polytope model* [27,28], which lays out the steps of a loop nest, iterating over an array structure, in a multi-dimensional, polyhedral coordinate space, with one dimension per loop. The points in this space are connected by directed edges representing the dependences between the loop iterations. With techniques of linear algebra and linear programming, one can conduct an automatic, optimizing search for the best mapping of the loop steps to time and space (processors) with respect to some objective function like the number of parallel execution steps (the most popular choice), the number of communications, a balanced processor load or combinations of these or others.

The polytope model comes with restrictions: the array indices and the loop bounds must be affine, and the space-time mapping must be affine.[8] Recent extensions allow mild violations of this affinity requirement – essentially, the permit a constant number of breaks in the affinity.[9] This admits a larger set of loop nests and yields better solutions.

Methods based on the polytope model are elegant and work well. The granularity of parallelism can also be chosen conveniently via a partitioning of the iteration space, called *tiling* [29,30]. The biggest challenge is to convert the solutions found in the model into efficient code. Significant headway has been made recently on how to avoid frequent run-time tests which guide control through the various parts of the iteration space [31,32].

Methods based on the polytope model have been implemented in various prototypical preprocessors. One for C with MPI is LooPo [33]. These systems use a number of well known schedulers (which compute temporal distributions)

[8] An *affine function* f is of the form $f(\boldsymbol{x}) = \mathbf{A}\,\boldsymbol{x} + \mathbf{b}$, where $\mathbf{A}$ is a constant matrix and $\mathbf{b}$ a constant vector. Affinity ensures the "straightness" of dependence paths across the iteration space of the loop nest, which allows the use of linear algebra and linear programming in their analysis.

[9] Dependence paths must still be straight but can change direction a fixed number of times.

[34,35] and allocators (which compute spatial distributions) [36,37] for the optimized search of a space-time mapping. Polytope methods still have to make their way into production compilers.

Preprocessing Library Calls. There have been efforts to preprocess calls to domain-specific libraries in order to optimize the use of the library modules. One straight-forward approach is to collect information about the library modules in a separate data file and use a preprocessor to analyze the call context in the program and exploit the information given by rewriting the calls to ones which give higher performance. Such a preprocessor, the Broadway compiler [38] was used to accelerate PLAPACK calls by up to 30% by calling more specialized library modules when the context allowed.

In this approach, the library itself is neither analyzed nor recompiled. The correctness of the information provided is not being checked by the preprocessor. On the positive side, the approach can also be used when one does not have access to the source code of the library – provided the information one has about it can be trusted.

7 Active Libraries

7.1 Principle and Limitations

The modules of an active library [39] are coded in two layers: the domain-specific layer, whose language offers abstractions for the special implementation needs of the domain, and the domain-independent layer in the host language. Thus, pieces of host code can be combined with domain-specific combinators.[10]

An active module can have several translations, each of which is optimized for a particular call context, e.g., a specific set of call parameters. The preprocessor is responsible for the analysis of the call context and for the corresponding translation of the domain-specific module code. The general-purpose compiler translates the pieces of host code which require no domain-specific treatment.

Arranging the program code in different layers, which are translated at different times or by different agents, is the principle of *multi-stage programming* (see the contribution of Taha [42] to this volume).

There is the danger of code explosion if a module is called in many different contexts.

7.2 Examples from High-Performance Parallelism

Active Skeletons. Herrmann has been experimenting with an active library approach for high-performance parallelism [41]. His domain-specific layer specifies the parallelism, synchronization and communication in the implementation

[10] With *combinators* I simply mean operators that combine program parts. Examples in our domain are parallelism (`||`) and interleaving (`|||`) from CSP [40], and disjoint parallelism (`DPar`) and communicating parallelism (`ParComm`) from our own recent work [41].

of a skeleton. The domain-specific language is specially crafted to express structured parallelism. It is defined as an abstract data type in Haskell, and the preprocessor is written in Haskell. We have started with Haskell as the host language but have switched to C for performance reasons. C is also the target language.

MetaScript. The system Metascript, being proposed by Kennedy et al. [43], implements the idea of *telescoping languages*, which is similar to the idea of multi-staged, active libraries. Additional goals here are to achieve performance portability between different types of machines, to reduce compilation times and to facilitate fast, dynamic retuning at run time.

The modules of a library are annotated with properties and optimization rules. These are then analyzed and converted by a script generator to highly efficient code. The script is run to obtain the target code of the module for each type of machine and context. The target code still contains code for retuning the module at load time.

The TaskGraph Library. In their contribution to this volume, Beckmann et al. [44] describe an active library approach in which the generation of pieces of parallel code from an abstract specification (a syntax tree), and the adaptation of the target code to the context in which it appears, go on at run time. The advantage is, of course, the wealth of information available at run time. With an image filtering example, the authors demonstrate that the overhead incurred by the run-time analysis and code generation can be recovered in just one pass of an algorithm that iterates typically over many passes.

8 Two Compilers

8.1 Principle and Limitations

In order to allow context-sensitive, domain-specific optimizations below the level of the host language, one needs two separate compilers which both translate to the same target language; the two pieces of target code are then linked together and translated further by the target language compiler (Fig. 3). Note that, what is composed in sequence in Fig. 2, is composed unordered here.

There needs to be some form of information exchange between the general-purpose and the domain-specific side. This very important and most challenging aspect is not depicted in the figure because it could take many forms. One option is a (domain-specific) preprocessor which divides the source program into domain-specific and general-purpose code and provides the linkage between both sides.

The main challenge in this approach is to maintain a clean separation of the responsibilities of the two compilers:

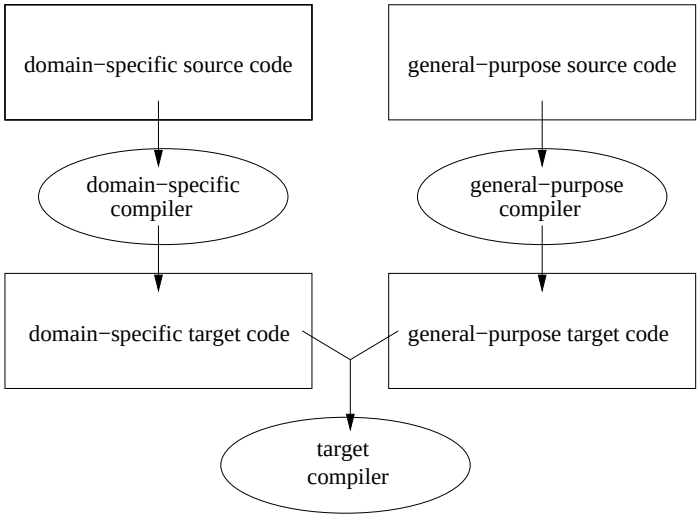

Fig. 3. Two compilers

- The duplication of analysis or code generation effort by the two compilers should be minimized. One would not want to reimplement significant parts of the general-purpose compiler in the domain-specific compiler.
- No special demands should be made on the general-purpose compiler; only the target code it generates should be taken. As a consequence, it should be possible to upgrade the general-purpose compiler to a newer version without any changes of the domain-specific compiler.

8.2 Examples from High-Performance Parallelism

At present, I know of no case in which this approach is being pursued in high-performance parallelism. The problem is that high-performance programmers still (have to) cling to the host languages C and Fortran. These are suitable target languages in Fig. 3 but, in all approaches pursued so far, they are also the general-purpose host language. Thus, there is no general-purpose compiler and the domain-specific compiler becomes a preprocessor to the target compiler, and the picture degenerates to Fig. 4.[11] We have tried to use Haskell as general-purpose source language [41] but found that, with the C code generated by a common Haskell compiler, speedups are hard to obtain.

This situation will only improve if production compilers of more abstract languages generate target code whose performance is acceptable to the high-performance computing community. However, in some cases, a more abstract host language may aid prototyping, even if it does not deliver end performance.

[11] Note the difference between Fig. 4 and Fig. 2: the level of abstraction to which the domain-specific side translates.

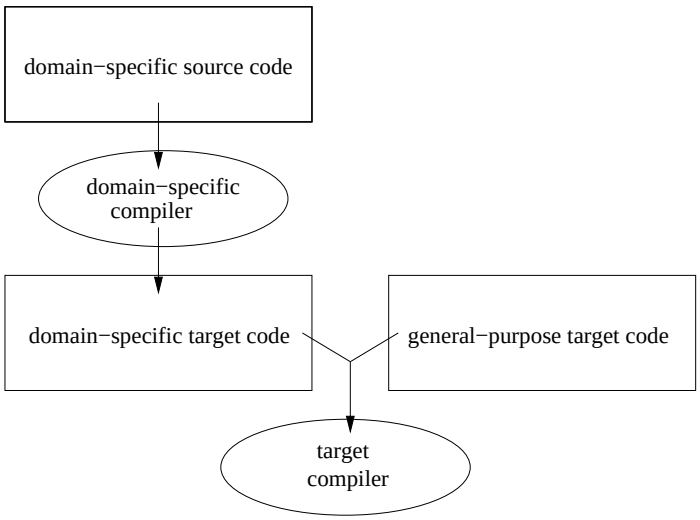

Fig. 4. Two compilers, degenerated

The domains of linear algebra and of digital signal processing have been very successful in generating domain-specific optimizers, but with a focus on sequential programs. One notable project in linear algebra is ATLAS [45]; in digital signal processing, there are FFTW [46] and SPIRAL [47]. FFTW comes close to the two-compiler idea and, since there is a parallel version of it (although this seems to have been added as an afterthought), I include it here.

FFTW. The FFTW project concentrates on the adaptive optimization of the fast Fourier transform. It follows a two-compiler approach, but with some departures from our interpretation of the scenario, as depicted in Fig. 3.

The idea of FFTW is to generate a discrete Fourier transform from a collection of automatically generated and highly optimized fragments (so-called *codelets*). The codelet generator corresponds to the general-purpose compiler in our picture. It is of a quite specialized form, so not really general-purpose. And it has been developed within the FFTW project, so it is not an external product. The source input it takes is simply an integer: the size of the transform to be calculated [48]. The codelet generator is written in Caml and produces platform-independent code in C. Actually, many users will just need to keep a bag of precompiled codelets and won't even have to install the codelet generator.

On the domain-specific side, codelet instances are selected and composed to a sequence by the so-called *plan generator*. The input to the plan generator consists of properties of the transform, like its size, direction and dimensionality. The search for the best sequence proceeds by dynamic programming, based on previously collected profiling information on the performance of the various

plans. The sequence is then executed by the FFTW run-time system, the so-called *executor*.

In the case of FFTW, the only information passed explicitly between the two sides in our figure is the size parameter for the codelet generator, which is passed from the left to the right.

A parallel version of the executor has been implemented in Cilk.

9 Conclusions

So far, the main aims of domain-specific program generation seem to have been programming convenience and reliability. The perception of a need for domain-specific program optimization is just emerging.

Even in high-performance parallelism, an area with much work on domain-specific program optimization, most programmers favour programming at a low level. The easiest approach for today's programmer is to provide annotations of the simplest kind, as in Cilk, GpH and JavaParty, or of a more elaborate kind with HPF and OpenMP. Imposing more burden, but also offering more control over distributed parallelism and communication is MPI.

The contributions on high-performance parallelism in this volume are documenting an increasing interest in abstraction. A first step of abstracting from point-to-point communications in explicitly distributed programs is the use of collective operations (as provided by MPI). The next abstraction is to go to skeleton libraries as proposed by Bischof et al. [13] or Kuchen [12]. One step further would be to develop an active library.

The main issues that have to be addressed to gain acceptance of a higher level of abstraction in high-performance parallelism are:

- How to obtain or maintain full automation of the generative process? Here, the regularity of the code to be parallelized is counter-balanced by the deductive capabilities of the preprocessor. In loop parallelization one has to push the frontiers of the polytope model, in generative programming with active libraries one has to find subdomains which lend themselves to automation – or proceed interactively as, e.g., in the FAN skeleton framework [49].
- How to deal with the lack of performance compositionality? Unfortunately, it seems like this problem cannot be wished away. If they have the necessary knowledge, preprocessors can retune compositions of library calls (à la Gorlatch [11] and Kuchen [12]).
- How to identify skeletons of general applicability? The higher the level of abstraction, the more difficult this task can become if people have different ideas about how to abstract. Communication skeletons at the level of MPI and of collective operations are relatively easy. Also, for matrix computations, there is large agreement on what operations one would like. This changes for less commonly agreed-on structures like task farms, pipelines [50] and divide-and-conquer [16].

10 Acknowledgements

Thanks to Peter Faber, Martin Griebl and Christoph Herrmann for discussions. Profound thanks to Don Batory, Albert Cohen and Paul Kelly for very useful exchanges on content and presentation. Thanks also to Paul Feautrier for many long discussions about domain-specific programming and program optimization. The contact with Paul Feautrier and Albert Cohen has been funded by a Procope exchange grant.

References

1. Réveillère, L., Mérillon, F., Consel, C., Marlet, R., Muller, G.: A DSL approach to improve productivity and safety in device drivers development. In: Proc. Fifteenth IEEE Int. Conf. on Automated Software Engineering (ASE 2000), IEEE Computer Society Press (2000) 91–100
2. van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: An annotated bibliography. ACM SIGPLAN Notices **35** (2000) 26–36
3. Hammond, K., Michaelson, G.: The design of Hume: A high-level language for the real-time embedded system domain (2004) In this volume.
4. Quinn, M.J.: Parallel Computing. McGraw-Hill (1994)
5. Robison, A.D.: Impact of economics on compiler optimization. In: Proc. ACM 2001 Java Grande/ISCOPE Conf., ACM Press (2001) 1–10
6. Pacheco, P.S.: Parallel Programming with MPI. Morgan Kaufmann (1997)
7. Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., Sunderam, V.: PVM Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing. MIT Press (1994) Project Web page: `http://www.csm.ornl.gov/pvm/pvm_home.html`.
8. Skillicorn, D.B., Hill, J.M.D., McColl, W.F.: Questions and answers about BSP. Scientific Programming **6** (1997) 249–274 Project Web page: `http://www.bsp-worldwide.org/`.
9. Gorlatch, S.: Message passing without send-receive. Future Generation Computer Systems **18** (2002) 797–805
10. Gorlatch, S.: Toward formally-based design of message passing programs. IEEE Transactions on Software Engineering **26** (2000) 276–288
11. Gorlatch, S.: Optimizing compositions of components in parallel and distributed programming (2004) In this volume.
12. Kuchen, H.: Optimizing sequences of skeleton calls (2004) In this volume.
13. Bischof, H., Gorlatch, S., Leshchinskiy, R.: Generic parallel programming using C++ templates and skeletons (2004) In this volume.
14. Blackford, L.S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., Whaley, R.C.: ScaLAPACK: A linear algebra library for message-passing computers. In: Proc. Eighth SIAM Conf. on Parallel Processing for Scientific Computing, Society for Industrial and Applied Mathematics (1997) 15 (electronic) Project Web page: `http://www.netlib.org/scalapack/`.
15. van de Geijn, R.: Using PLAPACK: Parallel Linear Algebra Package. Scientific and Engineering Computation Series. MIT Press (1997) Project Web page: `http://www.cs.utexas.edu/users/plapack/`.

16. Herrmann, C.A.: The Skeleton-Based Parallelization of Divide-and-Conquer Recursions. PhD thesis, Fakultät für Mathematik und Informatik, Universität Passau (2001) Logos-Verlag.
17. Herrmann, C.A., Lengauer, C.: $\mathcal{HDC}$: A higher-order language for divide-and-conquer. Parallel Processing Letters **10** (2000) 239–250
18. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers – Principles, Techniques, and Tools. Addison-Wesley (1986)
19. Moreira, J.E., Midkiff, S.P., Gupta, M.: Supporting multidimensional arrays in Java. Concurrency and Computation – Practice & Experience **13** (2003) 317–340
20. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the Cilk-5 multithreaded language. ACM SIGPLAN Notices **33** (1998) 212–223 Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'98). Project Web page: `http://supertech.lcs.mit.edu/cilk/`.
21. Trinder, P.W., Hammond, K., Loidl, H.W., Peyton Jones, S.L.: Algorithm + strategy = parallelism. J. Functional Programming **8** (1998) 23–60 Project Web page: `http://www.cee.hw.ac.uk/ dsg/gph/`.
22. Philippsen, M., Zenger, M.: JavaParty – transparent remote objects in Java. Concurrency: Practice and Experience **9** (1997) 1225–1242 Project Web page: `http://www.ipd.uka.de/JavaParty/`.
23. Koelbel, C.H., Loveman, D.B., Schreiber, R.S., Steele, Jr., G.L., Zosel, M.E.: The High Performance Fortran Handbook. Scientific and Engineering Computation. MIT Press (1994)
24. Foster, I.: Designing and Building Parallel Programs. Addison-Wesley (1995)
25. Brandes, T., Zimmermann, F.: ADAPTOR—a transformation tool for HPF programs. In Decker, K.M., Rehmann, R.M., eds.: Programming Environments for Massively Distributed Systems. Birkhäuser (1994) 91–96
26. Dagum, L., Menon, R.: OpenMP: An industry-standard API for shared-memory programming. IEEE Computational Science & Engineering **5** (1998) 46–55 Project Web page: `http://www.openmp.org/`.
27. Lengauer, C.: Loop parallelization in the polytope model. In Best, E., ed.: CONCUR'93. LNCS 715, Springer-Verlag (1993) 398–416
28. Feautrier, P.: Automatic parallelization in the polytope model. In Perrin, G.R., Darte, A., eds.: The Data Parallel Programming Model. LNCS 1132. Springer-Verlag (1996) 79–103
29. Andonov, R., Balev, S., Rajopadhye, S., Yanev, N.: Optimal semi-oblique tiling. In: Proc.13th Ann. ACM Symp.on Parallel Algorithms and Architectures (SPAA 2001), ACM Press (2001)
30. Griebl, M., Faber, P., Lengauer, C.: Space-time mapping and tiling – a helpful combination. Concurrency and Computation: Practice and Experience **16** (2004) 221–246 Proc. 9th Workshop on Compilers for Parallel Computers (CPC 2001).
31. Quilleré, F., Rajopadhye, S., Wilde, D.: Generation of efficient nested loops from polyhedra. Int. J. Parallel Programming **28** (2000) 469–498
32. Bastoul, C.: Generating loops for scanning polyhedra. Technical Report 2002/23, PRiSM, Versailles University (2002) Project Web page: `http://www.prism.uvsq.fr/ cedb/bastools/cloog.html`.
33. Griebl, M., Lengauer, C.: The loop parallelizer LooPo. In Gerndt, M., ed.: Proc. Sixth Workshop on Compilers for Parallel Computers (CPC'96). Konferenzen des Forschungszentrums Jülich 21, Forschungszentrum Jülich (1996) 311–320 Project Web page: `http://www.infosun.fmi.uni-passau.de/cl/loopo/`.
34. Feautrier, P.: Some efficient solutions to the affine scheduling problem. Part I. One-dimensional time. Int. J. Parallel Programming **21** (1992) 313–348

35. Feautrier, P.: Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. Int. J. Parallel Programming **21** (1992) 389–420
36. Feautrier, P.: Toward automatic distribution. Parallel Processing Letters **4** (1994) 233–244
37. Dion, M., Robert, Y.: Mapping affine loop nests: New results. In Hertzberger, B., Serazzi, G., eds.: High-Performance Computing & Networking (HPCN'95). LNCS 919. Springer-Verlag (1995) 184–189
38. Guyer, S.Z., Lin, C.: Optimizing the use of high-performance software libraries. In Midkiff, S.P., Moreira, J.E., Gupta, M., Chatterjee, S., Ferrante, J., Prins, J., Pugh, W., Tseng, C.W., eds.: 13th Workshop on Languages and Compilers for Parallel Computing (LCPC 2000). LNCS 2017, Springer-Verlag (2001) 227–243
39. Czarnecki, K., Eisenecker, U., Glück, R., Vandevoorde, D., Veldhuizen, T.: Generative programming and active libraries (extended abstract). In Jazayeri, M., Loos, R.G.K., Musser, D.R., eds.: Generic Programming. LNCS 1766, Springer-Verlag (2000) 25–39
40. Hoare, C.A.R.: Communicating Sequential Processes. Series in Computer Science. Prentice-Hall Int. (1985)
41. Herrmann, C.A., Lengauer, C.: Using metaprogramming to parallelize functional specifications. Parallel Processing Letters **12** (2002) 193–210
42. Taha, W.: A gentle introduction to multi-stage programming (2004) In this volume.
43. Kennedy, K., Broom, B., Cooper, K., Dongarra, J., Fowler, R., Gannon, D., Johnsson, L., Mellor-Crummey, J., Torczon, L.: Telescoping languages: A strategy for automatic generation of scientific problem solving systems from annotated libraries. J. Parallel and Distributed Computing **61** (2001) 1803–1826
44. Beckmann, O., Houghton, A., Mellor, M., Kelly, P.: Run-time code generation in C++ as a foundation for domain-specific optimisation (2004) In this volume.
45. Whaley, R.C., Petitet, A., Dongarra, J.J.: Automated empirical optimizations of software and the ATLAS project. Parallel Computing **27** (2001) 3–35 Project Web page: `http://math-atlas.sourceforge.net/`.
46. Frigo, M., Johnson, S.G.: FFTW: An adaptive software architecture for the FFT. In: Proc. IEEE Int. Conf. on Acoustics, Speech, and Signal Processing (ICASSP'98). Volume 3. (1998) 1381–1384 Project Web page: `http://www.fftw.org/`.
47. Püschel, M., Singer, B., Xiong, J., Moura, J.F.F., Johnson, J., Padua, D., Veloso, M., Johnson, R.W.: SPIRAL: A generator for platform-adapted libraries of signal processing algorithms. J. High Performance in Computing and Applications (2003) To appear. Project Web page: `http://www.ece.cmu.edu/ spiral/`.
48. Frigo, M.: A fast Fourier transform compiler. ACM SIGPLAN Notices **34** (1999) 169–180 Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'99).
49. Aldinucci, M., Gorlatch, S., Lengauer, C., Pelagatti, S.: Towards parallel programming by transformation: The FAN skeleton framework. Parallel Algorithms and Applications **16** (2001) 87–121
50. Kuchen, H., Cole, M.: The integration of task and data parallel skeletons. Parallel Processing Letters **12** (2002) 141–155