High-Level Parallel Programming and the Efficient Implementation of Numerical Algorithms

Von der Mathematisch-Naturwissenschaftlichen Fakultät der
Rheinisch-Westfälischen Technischen Hochschule Aachen
zur Erlangung des akademischen Grades eines Doktors
der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Diplom-Informatiker

George Horaţiu Botorog

aus Bukarest (Rumänien)

Referent: Universitätsprofessor Dr. rer. nat. Klaus Indermark
Korreferent: Universitätsprofessor Dr. rer. nat. Friedel Hoßfeld

Tag der mündlichen Prüfung: 13. Januar 1998

D 82 (Diss. RWTH Aachen)

Aachener Informatik-Bericht 97–15

Series Editor: Fachgruppe Informatik
Aachen University of Technology
Ahornstr. 55
D–52074 Aachen
Germany

ISSN 0935–3232

Printed by: Lönnies & Schmitz Druckerei
Warmweiherstr. 22
D–52066 Aachen
Germany

To Edit and Andrei

and

to my parents

Contents

Preface

High-level programming languages generally enable less efficient programs than their lower-level counterparts do. This drawback has lead to a restricted use of high-level approaches in areas where efficiency is essential, such as parallel processing. On the other hand, parallel processing could benefit enormously from programming paradigms that offer a high abstraction level. The present thesis aims to demonstrate that the two opposite goals, high level and efficiency, can be unified into a good trade-off. Moreover, it shows that high-level programming can be employed successfully in solving real-world problems in parallel.

The approach described here is based on the concept of *algorithmic skeletons*, representing useful patterns of parallel computations that are embedded in a sequential programming language. Being implemented as polymorphic higher-order functions, skeletons are endowed with a high generality and flexibility. This thesis shows that, in spite of their high level, skeletons enable writing parallel programs that reach a high efficiency and scalability.

The first part of the dissertation deals with the sequential host language *Skil*, an imperative language enhanced with a series of functional features. The imperative character of *Skil*, together with a specially developed compile-time technique for the elimination of the functional features, leads to a highly efficient implementation of *Skil* programs. This is demonstrated by a series of parallel matrix applications, which run several times faster than previous skeleton-based implementations that use functional languages. At the same time, the *Skil* programs run only between 3% and 40% slower than the corresponding hand-coded C programs with message-passing.

The second part of the dissertation concentrates on the use of skeletons in solving real-world problems in parallel. The application area considered here is that of *multigrid methods*, a class of numerical algorithms that provides the fastest known solvers for systems of discretized partial differential equations. Apart from their importance in numerical mathematics, multigrid methods are generally suited for parallelization. Moreover, these methods fit naturally in the skeletal concept, as they use a small set of basic operations as building blocks. Run-time results show that the skeleton-based implementations of two multigrid applications achieve a high scalability, with an efficiency of up to 70% on 256 processors.

Acknowledgments

Throughout the years, several teachers have guided my scientific development. Three of them had a profound influence on my evolution, and I regard them as my mentors. They are—in chronological order—Prof. Constantin Ottescu, Prof. Dr. Vasile Brânzănescu, and Prof. Dr. Klaus Indermark. I owe special gratitude to each of them.

The work presented in this thesis was supervised by Prof. Dr. Klaus Indermark and by Prof. Dr. Friedel Hoßfeld. I am indebted to both of them for their help, which came promptly and extensively every time I needed it.

My research was funded by the *Graduiertenkolleg "Informatik und Technik"* at the Aachen University of Technology. In particular, I would like to thank its Speaker, Prof. Dr. Otto Spaniol, for sponsoring my participation in several international conferences.

Prof. Dr. Herbert Kuchen provided the initial concept, and found time for countless discussions on various aspects of this work. He carefully read several documents, including this dissertation, each time supplying valuable feedback.

Dr. Wolfgang Joppich supplemented and structured my knowledge in the field of multigrid methods, and gave me insider tips on the parallel implementation of these methods.

Dr. Holger Stoltze assisted me in one of the most difficult phases: crystallizing the goals and getting the work started.

My colleagues in the Department for Computer Science II have provided constant support. Special thanks are due to Dr. Markus Mohnen, who found solutions to nearly all my TEXnical problems, and to Olaf Chitil for numberless discussions on debugging English prose.

Part of the applications described in this thesis were run on the machines of the *Paderborn Center for Parallel Computing*. I would like to acknowledge the competent and prompt help of the PC^2 team.

Most important, my family has treated this endeavor with much understanding and patience, conceding priority to my work for long periods on end. I am profoundly grateful to my wife, who did her best to shelter me from collateral problems during this time, and to my son, whose charm and innocence brought me comfort in the work's most trying moments. Their moral support gave me the energy to bring forth this dissertation.

G. H. Botorog

Notation

$\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$, $\mathbb{B}$	The set of natural numbers, the set of integers, the set of real numbers, and the set of boolean values $\{$*false*, *true*$\}$, respectively.
$f(n) \in \mathcal{O}(g(n))$	Asymptotic behavior of functions (particularly time bounds): $\exists c \; \exists N_0$, such that $f(n) \leq c \cdot g(n)$, $\forall n > N_0$.
$f : A \to B$	A (partial) function with domain A and co-domain B.
$[A \to B]$	The space of (partial) functions from A to B.
$f \; [a \to b]$	Function update or enhancement: if $f : A \to B$, then $\overline{f} := f \; [a \to b]$ is defined as $\overline{f} : A \cup \{a\} \to B \cup \{b\}$, with $\overline{f}(x) :=$ if $x = a$ then b else $f(x)$.
$c \; [v/t]$	Substitution: $c \; [v/t]$ denotes the term c, in which all free occurrences of the variable v have been replaced by the term t, provided that v does not occur free in t. The same notation is employed for type substitutions, i.e., for substitutions, where c and t are types, and v is a type variable.
$T \; \langle A/B \rangle$	If A and B are given sets and T is a term algebra generated by the elements of A (using a given set of operators), then $T \; \langle A/B \rangle$ denotes the term algebra obtained by replacing A with B as generator of T.
$f, \; g, \; u$	Continuous functions: the right hand side of a partial differential equation, the right hand side of a boundary condition, and the solution, respectively.
$f_h, \; g_h, \; u_h$	Discrete counterparts of continuous functions.
$w_h, \; \tilde{e}_h, \; r_h$	Discrete functions: the approximation of the solution, the algebraic error, and the residual, respectively.
$G_h, \; G_H$	The set of grid points corresponding to a finer and to a coarser discretization level, respectively ($h < H$).

$\Omega,\ \partial\Omega$	A domain and its border.
$L,\ L_h$	A differential operator and its discrete counterpart.
$\Delta,\ \Delta_h$	The continuous and the discrete Laplace operator.
$I_h^H,\ I_H^h$	Inter-grid transfer operators: the restriction operator I_h^H maps fine-grid functions to coarse-grid functions; the prolongation operator I_H^h maps coarse-grid functions to fine-grid functions.
$\nu = \nu_1 + \nu_2$	Total number of relaxation sweeps: ν_1 pre-smoothing steps, and ν_2 post-smoothing steps.
γ	Number of recursive calls of the correction scheme at each grid level; determines the shape of the multigrid cycle, e.g., $\gamma = 1$ yields V-cycles, whereas $\gamma = 2$ yields W-cycles.

Chapter 1

Parallel Processing

Not only would [parallel] programming techniques be vastly different from the serial ones in the serial languages we now use, but they would be humanly impossible without machine intervention.

Saul Gorn (*CACM*, 1958)

Ever since the invention of serial computers, their speed has constantly increased, allowing to cope with more and more complex applications. Particularly in recent years, the improvements in performance were significant. However, physical limitations in space (given by the minimal size of a component of an IC) and time (given by the speed of light) make it impossible to continue improving the performance of such computers indefinitely. Moreover, although the performance-versus-cost curves for sequential machines have an initial linear growth, they generally reach a saturation point, beyond which small gains in performance are obtained only at prohibitive costs.

There is a consensus today that the best way to overcome these problems is *parallel processing*, i.e., spawning several processors, instead of only one, to execute a given task. For that, one of the main properties a parallel computer should have is *scalability*, meaning the ability to achieve performance proportional to the number of processors (Kumar et al. 1994). If this property is fulfilled, then both the performance limitations and the cost problem can be solved in principle by employing an appropriate number of processors.

Parallel processing is thus considered to be the main computational technology of the future (Trew and Wilson 1991). This is best illustrated by the progress in solving problems from a special class, called the *grand challenges*, representing "fundamental problems in science or engineering that have a broad economic and scientific impact, and whose solution could be advanced by applying high-performance computing techniques and resources" (Kumar et al. 1994). Whereas in the 1980's the first such problems —such as airfoil simulation, 48-hours weather forecast, and 2D-plasma modeling—were solved, employing a computational power of up to 1 GFLOPS, during the 1990's, with 10

GFLOPS to 1 TFLOPS available, an entire series of further problems could be solved, such as 72-hours weather forecast, 3D-plasma modeling, vehicle signature recognition, chemical dynamics, etc. Still, even more such problems are expected to be coped with in the near future, including climate modeling, fluid turbulence simulation, ocean circulation, viscous fluid dynamics, semi- and supra-conductor modeling, and quantum chromodynamics. It should be emphasized that only an insignificant number of these problems could have been handled without parallel computers.

In contrast to sequential computation, which is based on the simple and well-understood von Neumann model, parallel processing has more degrees of freedom. This is why several computation models have been proposed and a wide variety of architectures and programming paradigms are currently in use. In the following section, we outline the main such models, highlighting the one we have used in this work.

1.1 Models of Parallel Computers and Computations

Parallel computers differ in various respects, such as control mechanism, memory organization, interconnection network, etc. Out of the variety of existing classifications, perhaps Flynn's taxonomy is the most widely used one. Flynn divided computers according to whether they used single or multiple streams of data, and single or multiple streams of instructions (control) (Flynn 1972). From the point of view of parallel machines, there are two interresting classes in this taxonomy: SIMD (*s*ingle *i*nstruction, *m*ultiple *d*ata) and MIMD (*m*ultiple *i*nstruction, *m*ultiple *d*ata).

In an SIMD computer, a global control unit dispatches the instructions to all processing units, as depicted in Figure 1.1 (left). Each instruction is executed synchronously by all or some processors, however on different data. Experience has shown that SIMD computers are adequate for evenly distributed problems, e.g., in image processing, but inefficient for tasks that have an imbalanced load. Examples of SIMD parallel machines include the Illiac IV, the Distributed Array Processor (DAP), the Connection Machine CM-2, and the MasPar MP-1 (Trew and Wilson 1991).

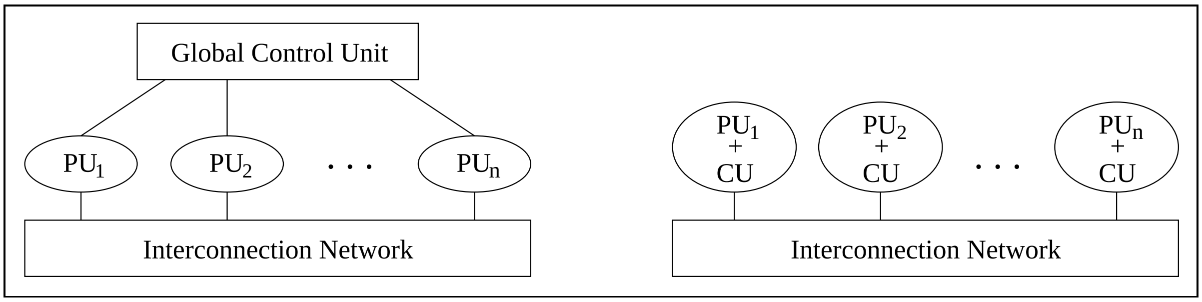


Figure 1.1: An SIMD architecture (left) and an MIMD architecture (right)

In MIMD architectures, the control is distributed among the processing units, as shown in Figure 1.1 (right). Each processor runs its own program independently, except for communication and synchronization steps with other processors. However, this paradigm grants a high degree of freedom, which usually makes parallel programming difficult. A simplification in this respect is the SPMD (*s*ingle *p*rogram, *m*ultiple *d*ata) programming style, where the processors execute the same program, but may follow different paths, or be at different points in the program at a given time. We have employed the SPMD style in the implementations presented in this work.

From the point of view of the memory organization in MIMD systems, all processors can access one or more common banks of memory, or each processor can access only its private memory[1]. In the first case, the machine is called *shared-memory* computer or multiprocessor, in the second case *distributed-memory* computer or multicomputer. The two architectures are depicted in Figure 1.2.

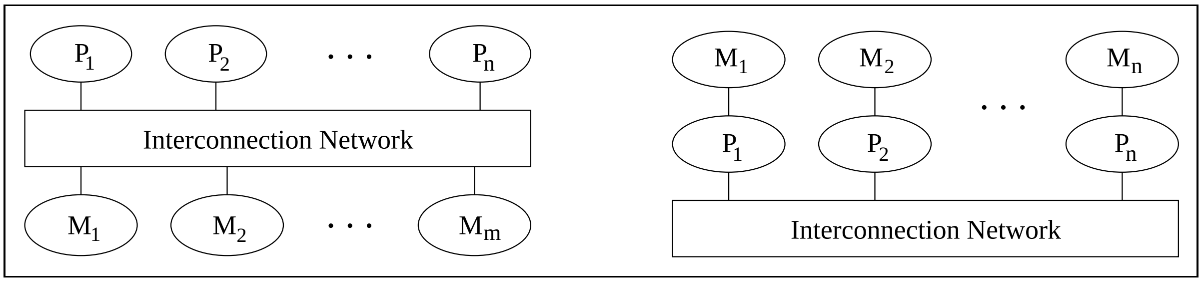


Figure 1.2: MIMD architectures: shared memory (left) and distributed memory (right)

Shared-memory machines have the advantage of being relatively simple to program, as their programming involves well-known techniques developed for multi-tasking computers, such as semaphores and monitors. The major drawback of these architectures is that the interconnection network must have a high bandwidth to ensure good performance. The reason is that, in each cycle, every processor may need to access the shared memory. In multi-level networks, a request has to pass through several stages of the network, before reaching its destination, thus raising the latency of memory accesses. For example, the butterfly interconnection network (Leighton 1992) has a latency that is logarithmic in the number of processing units. On the other hand, connecting the processors with the memory by a bus has the drawback of a limited scalability, because memory access becomes a bottleneck as the number of processors increases. This is why most of these architectures have up to 30 processors, often even fewer. Examples of shared-memory computers include the Alliant FX/2800, the Concurrent 3280 MPS, the Convex C2, the Sequent Symmetry (Trew and Wilson 1991), as well as the NYU Ultracomputer (Gottlieb et al. 1983).

Distributed-memory machines have a high scalability, as there is no central device that can become a bottleneck. This property, together with their relatively low cost, makes them appear today as the most promising parallel systems. As a consequence, distributed machines are the most wide-spread parallel systems, both in practice and in research. The work presented in this thesis was performed on such machines. Well known representatives of this class are the Intel iPSC and Paragon, the Cray T3D and T3E, the IBM SP2, the Fujitsu VPP700, the Thinking Machines CM-5, the Meiko Computing Surface, as well as transputer-based machines, such as the Parsytec GCel. Distributed-memory machines are also called massively parallel processors (MPPs).

A class of distributed systems with a different hardware, but conceptually similar to MPPs, are *networks of workstations* (NOWs). Hereby, a cluster of workstations or PCs are connected by a high-speed network, e.g., ATM (Anderson et al. 1995). The main

[1]There also exist architectures that can have both shared and private memory, e.g., the BBN TC2000.

advantage of NOWs is their lower cost compared to MPPs with the same computational power (Anderson et al. 1995).

More recently, heterogeneous systems were built as networks of computers, including, beside sequential machines, also MPPs. Such a hybrid system can be programmed, for example, using the PVM library (Geist et al. 1994), which runs on several platforms and has facilities both for data format conversions and for dynamic resource management.

The communication in a distributed system is achieved by *message passing*, i.e., by exchanging messages between processors via the interconnection network. A wide variety of network topologies has been employed, from the simplest ones, like the 2D-mesh (used for instance in the Intel Paragon, and in the Parsytec MC and GCel) and the 3D-torus (used in Cray T3D and T3E), to more complex ones, like the hypercube (used in the Intel iPSC and in the nCUBE), and the fat tree (used in the CM-5). Some examples of network topologies are depicted in Figure 1.3.

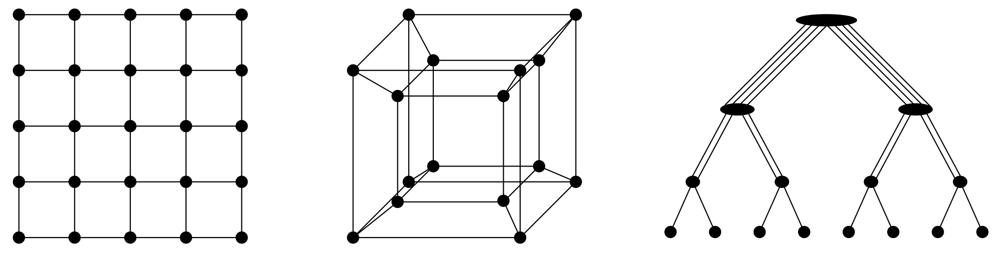


Figure 1.3: Examples of interconnection networks

Often, some topologies are more adequate than others to perform certain operations. For example, broadcasting on a parallel machine with p processors can be done in $\mathcal{O}(\log p)$ steps, if the network has a tree topology, but only in $\mathcal{O}(\sqrt{p})$ steps, if this topology is a 2D-mesh. Moreover, some operations may be easier to program assuming a certain topology. As the network topology is usually fixed for a given machine, some systems define *virtual (software) topologies*, which implement the routing of messages transparently to the user, offering him a view of the network according to the virtual topology. Of course, this does not change the complexity of operations, however, the embedding of the virtual topology in the hardware network is usually provably optimal (Röttger et al. 1994), therefore the resulting communication is also optimal for the given network. Virtual topologies have been intergrated for instance in MPI (Snir et al. 1996), in PCN (Foster et al. 1992), and in Parix (Parsytec Computer GmbH 1993; Röttger et al. 1994). In our work, we have made extensive use of the Parix virtual topologies.

We conclude this section by giving a classification of parallel algorithms. Depending on the kind of parallelism used, parallel algorithms and programs can roughly be divided into *process-parallel*[2] and *data-parallel* ones. In the first case, the program (dynamically) creates processes, which run concurrently, communicating and synchronizing from time to time. Hereby, each process consists of a different sequence of operations. In the second case, the program works on one or more distributed data structures, performing the same operations on all or some elements of these structures.

[2]Some authors employ the equivalent terms *task-parallel* or *control-parallel*.

Process-parallel programs seem to be naturally suited to MIMD computers, whereas data-parallel programs seem to fit the SIMD paradigm best. Although the first assertion is generally true, the second one applies only to regular problems. In case of unbalanced distributed data structures, executing different instructions on different parts of the data structure or, alternatively, performing a load redistribution, leads to poor performance on SIMD machines. On the other hand, running such programs on MIMD machines is always possible. Moreover, if the SPMD programming style is employed, then programming these machines for data parallelism can be done in a natural way.

Many algorithms contain sources for both process parallelism and data parallelism. However, the former is usually independent of the size of the problem, thus being limited. On the other hand, data parallelism increases with the problem size, leading to a higher scalability of implementations. Moreover, this form of parallelism is easier to control than its counterpart for large problem sizes and processor numbers. This is why data parallelism is perhaps today the most widespread parallel programming paradigm and at the same time yields the best results. The implementations presented in this thesis are all data-parallel.

1.2 Parallel Programming: The State of the Art

Although parallel and distributed systems gain more and more importance nowadays, the state of the art in the field of parallel software is far from being satisfactory. Not only is the programming of such systems a tedious and time-consuming task, but its outcome is usually machine-dependent and hence its portability restricted.

One of the main reasons for these difficulties is the low level of the parallel programming languages that are available. Whether the programmer uses shared variables, semaphores, etc. in the case of shared-memory machines, or message passing in distributed-memory systems, he has to account explicitly for all aspects of parallelism, i.e., for communication, synchronization, data and task distribution, load balancing, etc. This overhead places an additional burden on the programmer, who has to think not only about the *meaning* of his program, but also about its *behavior* on a given parallel computer.

Parallel programming is thus usually done on a low level, comparable to assembly languages in sequential computing. For instance, the *send* and *receive* primitives used in message passing to move data between processing units are conceptually similar to the assembler instruction *move*, which fetches data from the main memory into registers of a sequential machine.

However, tedious programming is not the only negative consequence of the low programming level. The next phase, that of program testing and debugging, is not easy, either. The programmer must now cope with problems such as deadlocks, starvation, and nondeterministic program runs. The latter is caused by processes running concurrently and is generally undesired, as it often makes program runs—especially erroneous ones—nonreproducible and therefore errors hard to detect and to isolate.

A further problem is debugging parallel programs. On the one hand, currently available debugging facilities have a restricted functionality. On the other hand, running the debugger process concurrently to the user program may change the behavior of the latter. As a consequence, errors that have occurred without the debugger may no longer occur in the presence of the debugger.

Finally, due to the nearness to the computer, low level parallel programs usually have a restricted portability, running only on a certain machine or on a restricted class of machines.

However, as the programmer has access practically to all levels of his program, from the highest one (that of the application) to the lowest one (e.g., message passing), he can obtain highly efficient parallel programs. This property is further supported by the possibility to select an optimal granularity of parallelism for a given application, as well as tune a program for efficiency[3].

Using the preceding analogy to assembly languages, the next step would be the design of higher-level languages for parallel programming. Indeed, there are several approaches following this direction; we describe the main ones in the next section.

1.3 Approaches to High-Level Parallel Programming

A series of attempts have been made to build high-level, structured languages, in which the user no longer has to handle parallelism explicitly, but can concentrate on his problem. Moreover, these languages often enable deterministic program runs, deadlock avoidance, and more advanced support for debugging. Such high-level approaches would naturally lead to less efficient programs than those written directly in low-level languages and the crux is to find a good trade-off between efficiency losses on the one hand, and gains in easiness of programming, as well as in safety and portability of programs on the other hand. Without trying to be exhaustive, we shall enumerate some of the most important high-level languages for parallel and distributed computation.

The *Parallel Virtual Machine (PVM)* (Geist et al. 1994) and the *Message Passing Interface (MPI)* (Snir et al. 1996), although including a series of higher-level primitives, actually belong to the low-level class of approaches. Their merit lies in that they offer standard interfaces for message passing, thus enabling higher code portability. They represent a step forward from the vendor-supplied, machine-dependent communication libraries. However, programs are usually still difficult to write, debug, and maintain.

At the other end of the scale lies the idea of *automatic parallelization*, where the user has to supply only a sequential program, all additional constructs necessary for parallelism being inserted by the compiler. Automatic parallelization was performed both in imperative and in declarative languages.

In the first case, the idea was to parallelize independent loops, or loops having only certain types of dependencies among the single iterations (Banerjee et al. 1993). The main problem thereby is that the dependency analysis—which is a static technique—delivers only an approximation of the actual dependencies among code segments at run time. This fact often leads to poor performance of the parallelized programs; as Banerjee and colleagues admit, "many real programs are not improved by existing compilers" (1993).

In the second case, declarative—mainly *functional*—languages were used, because of their implicit parallelism (Hogen et al. 1992). Hereby, the evaluation of (sub-)expressions can be done in any order, in particular in parallel, leading to the same result. Functional programs are high level and deterministic, but their implementations tend to be inefficient. The main reasons lie in the heuristic granularity of parallelism, as well as in the heuristic task and data distribution, that usually leads to high communication overheads.

[3]This tuning usually binds the program even tighter to the given architecture, however.

Along the Fortran line, a series of parallel dialects, such as Vienna Fortran (Chapman et al. 1992) and Fortran-D (Hiranandani et al. 1992) have been developed, culminating with *HPF* (High Performance Fortran Forum 1993). HPF enhances Fortran-90 (Adams et al. 1992) with constructs for declaring distributed arrays, for mapping them onto processors, and for specifying parallelizable loops. The main advantage of the Fortran-based languages is that they allow huge amounts of existing sequential code, especially in the area of scientific computing, to be parallelized with relatively little overhead. The drawbacks of HPF are its lack of support for process parallelism and the restriction of the distributed data structures to arrays. Therefore, irregular computations, as well as computations involving dynamic data structures cannot be expressed in HPF in a natural way.

In the *Shared Virtual Memory (SVM)* approach, the paging mechanism of the operating system is used to create a globally addressable virtual memory space on top of the distributed physical memory. Based on this idea, programming languages such as SVM-Fortran (Berrendorf et al. 1993) have been defined and implemented. One drawback of this approach is the overhead of sending and receiving entire pages, when only single data items are needed. Another problem is the phenomenon called 'false sharing', where two or more processors need different data items, which are, however, located on the same memory page, thus leading to an increase of the network traffic.

A related approach is *Linda* (Carriero et al. 1994), a language extension that can be included in different sequential languages, such as C or Fortran, and that provides the user with a global address space, the *tuple space*. The tuple space can be implemented both on shared-memory and (via message passing) on distributed-memory computers.

Gamma (Banâtre and Le Métayer 1991) is based on the model of chemical reactions and uses one data structure, the *multiset* or *bag*, and one operation called the Gamma scheme. Hereby, a computation is seen as a sequence of reactions, which consume elements of the multiset, producing new ones according to a set of given rules. The main problem of this approach is the exponential growth of the search space, which, even in the presence of optimizations, leads to poor performance.

Strand (Foster and Taylor 1990) is a concurrent logic programming language, which uses logical variables for communication and synchronization. Its principal drawback is the fine grained parallelism, which leads to a decrease of efficiency.

The *Bulk Synchronous Parallel (BSP)* model is based on the notion of the *superstep*, in which communication and synchronization are completely decoupled (McColl 1996; Valiant 1990). A BSP computation consists of a number of parallel supersteps, each superstep comprising a sequence of steps, followed by a barrier synchronization. During a superstep, every processor can perform several local computations, as well as send and receive a number of messages.

Last but not least, *algorithmic skeletons* (Cole 1989) are higher-order functions representing common parallelization patterns. They are integrated into a sequential language, thus allowing to structure and control the parallelism in application programs. The skeletal approach is discussed in detail in the following chapter.

Chapter 2

Algorithmic Skeletons

Perhaps the most important element in providing powerful changeable parts in a language is the availability of combining forms that can be generally used to build new procedures from old ones.

JOHN BACKUS

This chapter describes the skeletal approach to high-level parallel programming. First, the concept of algorithmic skeletons is defined and an example is given, illustrating the implementation of parallel algorithms with skeletons. This section is followed by an overview of the most important approaches to skeleton-based parallel programming. The final part of this chapter presents the main ideas of our approach, outlining the host language *Skil* and the classes of numerical applications considered in this thesis.

The notion of *algorithmic skeletons*, representing useful patterns of parallel computation, was introduced by Cole in the late 1980's (Cole 1989). Since then, the importance of this approach has constantly grown, such that today, there is an established and active 'skeletal' community. This fact is reflected not least in the increasing weight of this topic in renowned international meetings. Whereas until the early 1990's research on skeletons was usually presented embedded in wider topics, the International Conference *Euro-Par'96* had a dedicated workshop on parallel languages, programming, and high-level control (Bougé et al. 1996). This workshop was continued on the *Euro-Par'97*, which had even a special session on skeletons (Lengauer et al. 1997). Today, several important conferences—including conferences from other areas, such as the *International Conference on Software Engineering (ICSE)*—have special sessions or satellite workshops on high-level parallel programming. Moreover, a Dagstuhl Seminar on *Higher-Order Parallel Programming* was organized in 1997 (Cole et al. 1997).

2.1 Definition and Examples

A *skeleton* is an algorithmic abstraction common to a series of applications, that can be implemented in parallel (Cole 1989). Skeletons are embedded in a sequential host language, thus being the only source of parallelism in programs. Confining parallelism to skeletons makes it possible to structure and control it. Moreover, the user must no longer cope with parallel implementation details, which are not visible outside the skeletons. On the other hand, skeletons are efficiently implemented on a low level, for instance using message passing. At the same time, they offer an interface which abstracts from the underlying hardware, thus making programs portable. Classical examples of skeletons include *map*, which applies a given function to all elements of a data structure, *farm*, which models master–slave parallelism, and *divide&conquer*, which solves a problem by recursive splitting (Bacci et al. 1995; Cole 1989; Darlington et al. 1993).

The main difference between a skeleton and a parallel library function is that the former has a higher generality. A skeleton can be used not only to solve related problems, but also problems from different areas, that have a similar structure. For this, skeletons are usually implemented as *higher-order functions*, that is, as functions with functional arguments or result and have a *polymorphic* type. Thus, a skeleton can not only be applied to data structures having different types, but its functionality can be customized by functional arguments supplied by the user (Botorog and Kuchen 1996a).

The parameterization of skeletons with functions leads to a higher flexibility than that of library functions in imperative languages. This is illustrated by the following example. Consider the skeleton *divide&conquer* (`d&c`), which can be expressed in the functional language Haskell (Hudak et al. 1992) as follows[1]:

```
d&c ::  (a -> Bool) -> (a -> b) -> (a -> [a]) -> ([b] -> b) -> a -> b

d&c is_trivial solve split join problem =
  if (is_trivial problem)
   then (solve problem)
   else (join (map (d&c is_trivial solve split join) (split problem)))
```

This skeleton has four functional arguments: `is_trivial` tests if a problem is simple enough to be solved directly, `solve` solves the problem in this case, `split` divides a problem into a list of subproblems, and `join` combines a list of sub-solutions into a new (sub-)solution. The last argument is the problem to be solved. The function `map` applies a given function to all elements of a list, i.e., `map f [`x_1`, ... ,`x_n`] = [f `x_1`, ... ,f `x_n`]`.

Given this skeleton, the parallel implementation of an algorithm that has the structure of *divide&conquer* requires only the implementation of the four (sequential!) argument functions and a call of the skeleton. For instance, a *quicksort* procedure for lists can be implemented as follows:

```
quicksort lst = d&c is_simple ident divide concat lst
```

where `is_simple` checks if a list is empty or singleton, `ident` is the identity function and `divide` splits a list into three lists containing the elements that are smaller than a

[1] This definition of `d&c` only describes the overall functionality; it does not capture the parallel implementation.

given pivot element, the pivot element itself, and the elements greater than or equal to the pivot, respectively. Finally, `concat` concatenates two lists and `lst` is the list to be sorted.

However, other algorithms with the *divide&conquer* structure, such as Strassen's matrix multiplication, polynomial evaluation, numerical integration, FFT, etc. (Aho et al. 1974) can be implemented similarly, only by using different customizing argument functions. Skeletons can thus be viewed on a conceptual level as common parallelization patterns, and on a formal level as (polymorphic) higher-order functions.

Note that a similar flexibility cannot be achieved by first-order functions, i.e., by functions without functional arguments. Beside the elegance of this programming style, it is important that the skeleton is implemented in parallel, handling communication and synchronization, as well as distributed data internally. Thus, instead of error-prone communication via individual messages, there is a coordinated overall communication, which is guaranteed to work deadlock-free.

As the previous example shows, algorithmic skeletons can be represented in functional languages in a straightforward way as polymorphic higher-order functions. It would therefore seem appropriate to use a *functional* language as host. Indeed, as shown in the next section, most implementations of skeletons rely on functional hosts.

Depending on the kind of parallelism used, skeletons can be classified into *process-parallel* and *data-parallel* ones. In the first case, the skeleton creates a series of processes, which run concurrently. Some examples include *farm*, *pipe*, and *divide&conquer* (Bacci et al. 1995; Darlington et al. 1993). In the second case, the skeleton works on a distributed data structure, performing the same operations on some or all elements of this structure. Data-parallel skeletons, like *map*, *fold*, and *rotate*, are used in (Botorog and Kuchen 1998; Darlington et al. 1995; Kuchen et al. 1994).

2.2 Related Work

There are currently a number of research groups working on the design and implementation of parallel languages with algorithmic skeletons. Without trying to be exhaustive, we shall enumerate some of the most important ones.

Cole introduced in his book (1989) the skeletal concept, defining four general skeletons: *divide&conquer*, *task queue*, *iterative combination* (similar to the Gamma scheme sketched in Chapter 1), and *cluster* (solving problems by decomposition on a grid network). Cole introduced also the *skeletal abstract machine*, a refinement of abstract machines used in the implementation of functional languages.

Darlington's group at the Imperial College has been one of the most prolific in this field. Their first approach was to embed a set of general skeletons, including *pipe*, *farm*, *d&c*, and *ramp* ('reduce-and-map-over-pairs') in a purely functional language (Darlington et al. 1993).

This first attempt was followed by a refined approach, called *SPP(X)*, standing for *S*tructured *P*arallel *P*rogramming parameterized by a base language X. SPP(X) is a two-layer scheme, comprising a high-level, functional language, called the Structured Control Language (SCL), in which applications—containing skeleton calls—are written, and a low-level Base Language (at present Fortran) for efficient sequential code called from within the skeletons (Darlington et al. 1995). The skeletons presented in this context are both general ones, like *farm* and *SPMD*, and data-parallel ones working on distributed arrays, like *map*, *fold*, and *rotate*. The second category is similar to the skeletons described by Kuchen and colleagues (1994).

The most recent work of the Imperial College group handles algorithms that involve irregular computations, such as the Barnes–Hut method for solving N-body problems (Barnes and Hut 1986). Hereby, the concept of skeletons is extended to that of *Parallel Abstract Data Types*. A PADT consists of a data type definition of the (irregular) data structure and a set of parallel operators (skeletons) over this data type (Wu et al. 1997). This approach is closely related to our work presented in (Botorog and Kuchen 1995).

Several other approaches have used the functional framework to embed skeletons. We could mention the work of Bratvold, who integrated skeletons into Standard ML, yielding the SkelML system (Bratvold 1994), or that of Rabhi's group, who embedded their skeletons, such as the *Static Iterative Transformation (SIT)*, in Haskell (Schwarz and Rabhi 1996).

Last but not least, the Aachen skeleton group initially used functional host languages, too. The prototype implementation was a few hundred times more efficient than direct parallel implementations of functional languages (Stoltze 1994). This result was further improved in the following implementation by Kuchen, who embedded the data-parallel skeletons in Haskell, yielding DPFL (*Data-Parallel Functional Language*). This implementation approached the performance of hand-written parallel C code up to a factor between 6 and 10 (Kuchen et al. 1994; Kuchen 1995).

On the other hand, few attempts have been made to use an *imperative* language as host. One such attempt is the *Pisa Parallel Programming Language* (P^3L) (Bacci et al. 1995), which builds on top of C++, and in which skeletons are internal language constructs. The main drawbacks are the difficulty to add new skeletons and the restricted number of skeletons that can be used. Another imperative approach is that of Burkhart's group in Basel, consisting of a high-level (skeleton) library on top of MPI, called the *Programming Environment for the Message Passing Interface (PEMPI)* (Fang 1996).

Several researchers have concentrated on the theoretical aspects of algorithmic skeletons. The group around Pepper used data distribution algebras in the derivation and implementation of skeletons (Pepper and Südholt 1996). Skillicorn (1994) concentrated both on homomorphic skeletons, using the Bird–Meertens formalism (Bird 1989) to describe them, and on nonhomomorphic parallel operations on lists, which he called *compound list operations (CLOs)*. He further introduced *categorical data types*, a special case of PADTs, in which the parallel operations are homomorphic (Skillicorn 1994). The work of Lengauer's group in Passau handled homomorphisms from the level of their definition, using transformations of the *divide&conquer* paradigm, down to the implementation on particular architectures (Gorlatch 1996; Herrmann and Lengauer 1997). The skeleton group in Leeds modeled the performance of algorithmic skeletons using higher-order complexity functions (Deldarie et al. 1995).

Finally, we mention three approaches, where the parallel operations are not skeletons in the sense of our definition, but their functionality resembles that of certain skeletons.

The *Program Composition Notation (PCN)* employs *cells* and *templates* as a means to define reusable parallel program structures (Foster et al. 1992). Moreover, it makes use of virtual topologies and mappings onto the real processors. However, the language contains also lower-level features, such as *definitional variables*, by which processes can communicate, but which can lead to problems such as deadlocks.

In the language ILIAS (Loyens and Moonen 1994), arithmetic and logic operators are extended to global operators that can be applied pointwise to the elements of matrices.

NESL (Blelloch 1995) is a first-order functional, polymorphic, data-parallel language, which supports nested parallelism. However, NESL's parallel operations are less general than algorithmic skeletons, for example the operations `sum`, `count`, `max_index` and `min_index` can be implemented as instances of the `fold` skeleton.

2.3 Our Approach

The main goal of our work was to show that algorithmic skeletons are adequate for parallel programming *in practice*. This goal amounted to the following requirements:

1. The host language should enable a *high programming level*, allowing the straightforward integration of skeletons.

2. The implementation should be *efficient*, with only minimal performance losses compared to direct low-level parallel implementations.

3. The skeletons should have a certain amount of *generality*, i.e., they should be usable in solving different, more or less related problems.

4. The skeletons should be usable in solving *real-world* problems in parallel.

In the following, we consider each of these goals. Hereby, the first two goals are conflicting, therefore we had to find a good trade-off between them.

High Programming Level

We first consider the host language, i.e., the language in which the skeletons are embedded. As mentioned before, most skeletal approaches employ a *functional* host, because this choice allows a natural integration of the skeletons as higher-order functions. However, using a functional language leads to a decrease of the efficiency of resulting programs, which can become critical in applications where performance is important, like for instance in scientific computing.

We have therefore used an *imperative* language, namely C, as the basis of our host language. This approach has several advantages. First, the sequential parts of the program are more efficient than in the case of a functional language. Second, as both host and skeletons are implemented in an imperative language, the overhead of the context switch implied by the use of a functional host, has no longer to be taken into account. Third, imperative languages offer mechanisms for local accessing and manipulating data, which have to be simulated in functional languages (Kuchen et al. 1994). We have called the language *Skil*, as an acronym for *Sk*eleton *I*mperative *L*anguage (Botorog and Kuchen 1996b).

On the other hand, to allow skeletons to be integrated in their full generality, we have provided *Skil* with the following functional features:

- *Higher-order functions*, i.e., functions with functional arguments or result

- *Partial applications*, meaning the application of an n-ary function to $k \leq n$ arguments, yielding a new $n-k$-ary function

- *Conversion of operators to functions*, which allows passing operators as functional arguments or results, as well as partial applications of operators

- A *polymorphic type system*, which allows skeletons to be reused in solving similar problems on different data

- A construct for defining *distributed data structures*

The main problem arising here is the translation of the functional features and of polymorphism, as they have a crucial influence on the performance of resulting programs. Because a classical implementation by closure techniques (Field and Harrison 1988) is too inefficient, we have used an instantiation method related to Wadler's higher-order macro expansion (Wadler 1990) instead. This technique translates a (polymorphic) higher-order function into one or more specialized first-order monomorphic functions. Moreover, partial function applications are transformed to total applications. This transformation leads to considerable gains in efficiency, bringing the performance of our language close to that of low-level parallel imperative languages with message passing. However, as this elimination is done at compile time, a restriction had to be made regarding the functional arguments of higher-order functions. The language *Skil* and its implementation are described in detail in Chapter 3.

Efficiency

The performance of skeleton-based parallel implementations is determined on the one hand by the efficiency of the host language and on the other hand by the efficiency of the skeletons. The first problem was tackled by employing an imperative host language, anb by the compile-time elimination of the functional features, as presented in Chapter 3. The second problem, namely the efficient implementation of the skeletons, is handled in Chapter 4. Hereby, we demonstrate the overall efficiency of our approach. We present run-time results for sample array applications, and compare them with the results obtained by similar implementations in the functional language with skeletons DPFL (Kuchen et al. 1994; Kuchen 1995), and in parallel C with message passing. Our run times are 6 to 10 times faster than those of the DPFL implementation and only between 3% and 40% slower than those of the direct, low-level C implementation.

Generality

As we have already mentioned, skeletons should have a certain amount of generality. The question is, what is the adequate level of generality? The first skeleton-based approaches (Bacci et al. 1995; Cole 1989; Darlington et al. 1993) used only a small set of general skeletons. However, such skeletons may not be appropriate for expressing certain algorithms, or may lead to implementations that are not efficient. On the other hand, too specialized skeletons may degenerate to plain parallel library functions. Hence, more recent approaches—including ours—are somewhere in between these two extremes, more precisely, they try to design skeletons that can be used in solving related problems, or problems from different areas, but having a similar structure (Botorog and Kuchen 1995; Botorog and Kuchen 1998; Darlington et al. 1995; Wu et al. 1997).

We have thus designed our skeletons, such that they are applicable to certain *classes* of problems. For instance, the array skeletons presented in Chapter 4 were used to implement a series of matrix algorithms, like matrix multiplication or Gauss–Jordan elimination, but also a Monte Carlo method for solving a partial differential equation. Moreover, the multigrid skeletons presented in Chapter 5 can be used to implement most multigrid methods (Briggs 1987; Hackbusch and Trottenberg 1982; Joppich and Mijalković 1993).

In this work, we have concentrated on data-parallel skeletons. A study on the integration of process-parallel skeletons in *Skil* and their use in solving the *Salishan Problems* (Feo 1992) has been done by Freiberg (1998).

Real-World Applications

The final goal of this work was to show that skeletons can be used to solve real-world problems in parallel. We chose as our class of applications *multigrid methods*. There are several reasons for this choice. First, multigrid methods are nontrivial numerical applications. Second, there is an entire class of multigrid methods, which use, however, the same building blocks, thus fitting naturally in the skeletal concept. Third, the multigrid skeletons can be used in the implementation of other classes of algorithms on hierarchically structured data, such as N-body methods (Wu et al. 1997).

The multigrid skeletons and two applications implemented with these skeletons are presented in Chapter 5. Run-time results show that the skeleton-based implementation achieves a remarkably good scalability.

Chapter 3

The Language *Skil*

On the other hand,
we cannot ignore efficiency.

JON BENTLEY

This chapter presents *Skil*, an imperative language enhanced with a series of functional features, mainly higher-order functions, partial applications and polymorphism. These enhancements enable the high programming level needed for the integration of algorithmic skeletons. On the other hand, to obtain an efficient implementation, the functional features are translated at compile time into constructs of the imperative base language. After describing the main features the language and outlining the implementation the *Skil* compiler, we concentrate on the central aspect of our approach, namely the *instantiation* of the functional features. We formulate a static criterion that ensures the termination of the transformation and prove the correctness of the instantiation. Concluding this chapter, we present the main optimization technique employed by the *Skil* compiler, namely *inlining*. Results show that the code expansion caused by instantiation remains moderate, whereas the code growth due to inlining can be steered by the user.

3.1 Description of the Language

As already mentioned, *Skil* aims at combining a high programming level with an efficient implementation. We have therefore started with an *imperative* language, namely C[1] and

[1]This choice is only a pragmatic one, based on the widespread use of C, for which mature compilers are available, as we use a C compiler as a back end in our implementation. However, other imperative languages could be used equally well instead.

enhanced it with a series of functional constructs. The goal was to allow skeletons, which are polymorphic higher-order functions, to be specified as naturally as in functional languages. For example, consider the function `map`, which applies a given argument function to a given list of items, yielding a list of results. In Haskell (Hudak et al. 1992), this function can be defined as follows:

```
map :: (a -> b) -> [a] -> [b]
map f []    =  []
map f x:xs  =  f x : map f xs
```

`map` is a higher-order function with the functional argument `f` and has a polymorphic type, determined by the type variables `a` and `b`.

We shall now present the language enhancements that are necessary to make such function definitions possible. As *Skil* is based on C, it mostly uses the C syntax (Kernighan and Ritchie 1988), except for the functional features, where an extended syntax is necessary.

3.1.1 Higher-Order Functions

Skil allows the definition of higher-order functions (HOFs), i.e., of functions with functional arguments or result. Although this feature could be simulated in C by passing and returning pointers to functions, respectively, this simulation would not allow the use of functional arguments created dynamically by partial applications. Hence, we allow the explicit definition of HOFs.

As HOFs with functional arguments and HOFs with functional result differ both in the implementation technique and in the restrictions that apply, we present them in the following separately.

Functions with Functional Arguments

Skil allows functions to be passed as arguments. For instance, (a for the time being monomorphic version of) the function `map`—working on lists of integers—can be defined in *Skil* as follows, provided the standard list functions `nil`, `cons`, and `empty` are available:

```
intlist map (int f (int), intlist l) {
    if (empty (l))
        return (nil ()) ;
    else
        return (cons (f (l->elem), map (f, l->next))) ; }
```

Although functions can be arguments of other functions, they cannot occur in data structures and certain expressions. More exactly, the following restrictions apply:

- No element of a data structure may have a functional type. Pointers to functions are nevertheless allowed for compatibility to C.
- Expressions with functional type may occur only in function applications and in `return` statements. Equivalently, in the following expressions and statements, none of the (sub)expressions e_1 and e_2 may have a functional type (the syntax of C expressions and statements is described by Kernighan and Ritchie (1988)):

e_1 `=` e_2	(*assignment expression*)
e `?` e_1 `:` e_2	(*conditional expression*)
e_1 `,` e_2	(*sequence expression*)
`(`τ`)` e_1	(*casting*)
`sizeof (`e_1`)`	(*sizeof expression*)
`for (`e_1`;` e`;` e_2`)`	(*for loop*)
e_1 `;`	(*expression statement*)

Functions with Functional Result

Skil allows the definition of HOFs with functional result. As an example, consider the following HOF, which selects one of its functional arguments, assumed to be unary functions:

```
float select (float f (int), float g (int)) (int) {
    if (f (0))
        return (f) ;
    else
        return (g) ; }
```

For efficiency reasons, HOF definitions of this kind are expanded at compile time. More exactly, instead of the reduction semantics usually employed in the implementation of functional languages (Field and Harrison 1988), HOFs with functional result are considered here as *shorthand notation* for the corresponding η-expanded definitions with nonfunctional result. This compile-time technique leads to a higher efficiency of programs than its run-time counterpart, because it avoids the use of closures (Field and Harrison 1988). Thus, the preceding HOF definition stands for:

```
float select (float f (int), float g (int), int x) {
    if (f (0))
        return (f (x)) ;
    else
        return (g (x)) ; }
```

Notice the additional parameter `x` in the header of the function and the full applications in the `return` statements.

This meaning for HOFs with functional return type makes it necessary to handle such HOFs with care in the presence of side effects. Consider for instance the following HOF, which writes a string on standard output and then returns its argument function:

```
int idfun (int f (int)) (int)
    { printf ("hello") ;  return (f) ; }
```

and the functions `twice`, which applies a function twice to its argument, and `succ`, which computes the successor of an integer number:

```
int twice (int f (int), int x)
    { return (f (f (x))) ; }

int succ (int x)
    { return (x + 1) ; }
```

Then, the call

```
twice (idfun (succ), 0) ;
```

outputs the string "`hello`" twice, rather than once, as it would have if evaluated by a graph reduction machine (Field and Harrison 1988). Thus, in case of HOFs with functional result that have side effects, these effects occur each time the function is called. The programmer can nevertheless steer explicitly the number of side effects that occur, e.g., by means of global counters.

3.1.2 Currying

Closely related to higher-order functions is *function currying*. This means that an n-ary function[2] is considered to be a function with one argument returning a new, $(n-1)$-ary function, and so on. *Skil* functions are implicitly curried, i.e., the call $f\ (e_1, \ldots, e_n)$ is internally considered to represent $f\ (e_1)\ \ldots\ (e_n)$, whereby function application is left associative. However, functions can be curried explicitly also; moreover the two styles can be mixed freely, e.g., $f\ (e_1, e_2)\ (e_3, e_4)$.

Currying is important in *Skil*, because it allows *partial function application*, i.e., the application of an n-ary function to $k \leq n$ arguments. Partial applications are useful in creating new functions at runtime. As an example, suppose we want to compare all elements of a list `lst` with a given threshold value, using the binary function *less than* (`lt`). The desired boolean list can be generated by calling the function `map`, with a partial application of `lt` to the threshold value as its functional argument: `map (lt (thresh), lst)`.

3.1.3 Polymorphism

Consider again the definition of the function `map` given in the beginning of this chapter, which has the polymorphic type `(a -> b) -> [a] -> [b]`. Although polymorphism can be simulated in C by casting and void pointers, this simulation eludes the type checking performed by the compiler, leading to error prone programs. We have therefore chosen to support classical, Hindley–Milner polymorphism (Milner 1978) and perform a polymorphic type checking, which leads to safer programs.

A further reason for using polymorphism is that in C, void pointers are compatible to any *pointer* type, but not to any other type. Therefore, to fit a nonpointer type into a void pointer, the former has to be 'boxed' in a pointer type, leading to an additional indirection and hence to a decrease of efficiency. Our solution, however, avoids this overhead.

Polymorphism is achieved by enhancing the type system by *type variables*, which have in *Skil* the syntax `$`t, where t is an identifier. Type variables may be used in any type declaration, and can be instantiated with arbitrary types, with the exceptions described next. A type is called *polymorphic* iff it contains at least one type variable.

To simplify the declaration of instances of polymorphic structures, unions, and distributed data structures (see Section 3.1.5), the type variables appearing inside their components have to be listed as parameters in the overall definition, enclosed between angle brackets (`<>`). Thus, the declaration of polymorphic structures, unions, and distributed data structures is syntactically similar to the `data` declarations in functional

[2]The arity $ar\,(f)$ of a function f is defined as the number of formal parameters in the definition of f.

languages (Hudak et al. 1992), where type variables occurring on the right hand side must be listed on the left hand side (i.e., as parameters of the type constructor). For instance, a polymorphic list can be declared in *Skil* as follows[3]:

```
typedef struct _list <$t> { $t elem;  struct _list <$t> *next; }
        *list <$t> ;
```

From this generic list type, a monomorphic instance can be created by supplying a type for the parameter, similarly to functional languages (Hudak et al. 1992). For example, a list of integers can be declared as `list <int>`.

We can now give the polymorphic definition of the HOF `map` in *Skil*. Notice that the body is identical to the one in the monomorphic version, described in Section 3.1.1, however the functions `empty`, `nil` and `cons` must now be polymorphic.

```
list <$t2> map ($t2 f ($t1), list <$t1> l)
    { ... /* body remains unchanged */ ... }
```

The scope of a type variable is the block in which it appears, or the entire function definition, if it appears in the header of a function. Notice that this scoping is consistent with the *let*- and λ-bound polymorphism in functional languages (Leroy 1993; Milner 1978). We illustrate the difference between these bindings with an example. Consider the following polymorphic function definition:

```
int f ($t x)
    { $t y ; ... }
```

`f` can be applied to arguments of *any* type, i.e., `$t` is universally quantified in the type definition of `f`: $\forall$ `$t`, `f` : `$t` $\rightarrow$ int (see Section 3.1.6 for the specification of types). In particular, the `$t` in the argument list of `f` may be bound to different types in different occurrences, as it is the case in the following expression:

```
f (1) + f ("hello")
```

In the first call, `$t` is bound to int and in the second call to ref char (Section 3.1.6). This binding corresponds to the *let*-bound polymorphism in functional languages (Leroy 1993; Milner 1978). On the other hand, the `$t` in the body of `f` is no longer generic, i.e., it can be bound only to the type to which the first `$t` was bound. For instance, the statement

```
y = 1 ;
```

in the body of `f` would cause a type checking error in the case when `$t` is bound to ref char, whereas the statement

```
y = x ;
```

is always correctly typed. This typing corresponds to the λ-bound polymorphism in functional languages (Leroy 1993; Milner 1978).

In *Skil*, the following restrictions apply in the use of polymorphism:

[3]This type is the imperative counterpart of the functional list type `[a]`, defined here as a product type, rather than a sum-of-products type, as the predefined `NULL` pointer is used instead of the `[]` constructor.

- No type variable may be instantiated with `void` or with a functional type. This restriction is necessary in the translation of HOFs.

- No type variable may be instantiated with a `par` type (see Section 3.1.6), if the former is part of another data structure. This restriction is a special case of the condition that no `par` types may occur inside other data structures, as the current implementation of *Skil* does not support nested parallelism.

- If the type of the left hand side of an assignment is polymorphic, then it must be identical with the type of the right hand side. Thus, an assignment cannot lead to a type instantiation. The reason for this restriction is that polymorphism is unsound in the presence of assignments (Leroy 1993), as the following example shows:

```
$t x ;
x = 1 ;
x = "hello" ;
```

  In this case, it is not clear what type `x` should have, `int` or `ref char`. All other operators involving assignments (`+=`, `*=`, ... , `++`, `--`) take only monomorphic arguments.

- As a consequence, no global variables with polymorphic type are allowed, because global variables get their values only by assignment.

3.1.4 Conversion of Operators to Functions

A further feature of *Skil* is the conversion of operators to functions, which is denoted by enclosing the operator between brackets, i.e., `(`*op*`)`. This conversion allows passing operators as functional arguments to HOFs as well as partial applications of operators. For example, the expression `fold((+),lst1)` computes the sum of all elements of the list `lst1`, whereas `map((*)(2),lst2)` multiplies each element of the list `lst2` by 2.

The set *FOp* of operators[4] that can be converted to functions is (see also the *Skil* syntax in Appendix A):

$$FOp \;:=\; \{\ \texttt{||},\ \texttt{\&\&},\ \texttt{|},\ \texttt{\^{}},\ \texttt{\&},\ \texttt{==},\ \texttt{!=},\ \texttt{<},\ \texttt{>},\ \texttt{<=},\ \texttt{>=},\ \texttt{<<},\ \texttt{>>},\ \texttt{+},\ \texttt{-},\ \texttt{*},\ \texttt{/},\ \texttt{\%},\ \texttt{\~{}},\ \texttt{!}\ \}$$

We call a function that originates from an operator, *operator function*. The arity of an operator function is identical with that of its operator:

$$ar\,(\texttt{(}fop\texttt{)}) \;:=\; ar\,(fop) \;=\; \begin{cases} 2, & \text{if } fop \in FOp \setminus \{\texttt{!},\ \texttt{\~{}}\}, \\ 1, & \text{if } fop \in \{\texttt{!},\ \texttt{\~{}}\} \end{cases}$$

3.1.5 Distributed Data Structures

Apart from the need to cope with distributed functionality, a parallel language should allow the definition of distributed data structures. This aspect has been addressed in most languages with skeletons by considering the distributed data types (usually arrays) as internally defined (Darlington et al. 1993; Kuchen et al. 1994; Stoltze 1994).

[4] *Skil* inherits the C operators, no new operators are added.

As one of our aims is flexibility, we want to allow *any* distributed data structure to be defined, as long as it is homogeneous, in the sense that it is composed of identical data structures placed on each processor. This definition is done by means of the `pardata` construct, which is similar to C's `typedef` construct, but introduces a *distributed* (*parallel*) data structure. The syntax of this construct is:

`pardata` *name* `<`$\$t_1$`, ... ,` $\$t_n$`>` *implem* ;

where $\$t_1, \ldots, \$t_n$ $(n \geq 0)$ are type parameters and *implem* is a (possibly polymorphic) type, representing the data structure to be created on each processor. The distributed data structure is then the entirety of all local structures and is identified by *name*. Thus, a pardata can be seen logically as an array with p elements, where p is the number of processors, each element being placed on one processor. However, the type of the single elements (*implem* in the preceding declaration) is not subject to any restrictions, in particular, it can denote a recursive data structure, like a list or a tree. No distributed data structures may be nested, in particular the type arguments of a pardata construct cannot be instantiated with other pardatas.

Thus, the type arguments of a pardata can be instantiated with arbitrary types, except functional and distributed ones. However, the use of *dynamic* (i.e., pointer-based) data types may lead to problems. In this case, skeletons that move elements of the pardata from one processor to another should not move pointers as such, but the data pointed to by them. For that, such skeletons get additional functional arguments that account for this data marshalling. We have handled this problem by supplying additional functional arguments to skeletons (Botorog and Kuchen 1996c); the skeletons presented here are given in a simplified syntax, without these additional arguments.

Finally, the functions `$$par` and `$$dyn` test if a data structure is distributed (i.e., a pardata) and dynamic, (i.e., if it contains pointers), respectively. The latter is needed for the efficient implementation of skeletons that move elements with dynamic type between processors. Further details are given in Botorog and Kuchen (1996c).

3.1.6 Formal Specification

We now give a formal specification of the *Skil* language constructs, concentrating on expressions, types, and function definitions, because this is where the most important differences between *Skil* and C occur. We assume thereby the existence of a set *Fun* of function symbols, a set *Var* of variables, and a set $Arg \subset Var$ of formal parameters in function definitions.

As *Skil* is widely based on C, we shall use the structures of C to define *Skil*'s structures. We denote the set of C expressions by *CExpr*, the set of C applicative expressions (i.e., function calls) by *CApExpr*, the set of C primitive expressions by *CPrimExpr*, the set of C statements by *CStmt*, the set of C function definitions by *CFunDef*, and the set of C programs by *CProg*. These structures are described in Kernighan & Ritchie's C language reference book (1988).

Moreover, assuming A and B are given sets and T is a term algebra generated by the elements of A (using a given set of operators), we denote by $T\ \langle A/B \rangle$ the term algebra obtained by replacing A with B as the generator of T (i.e., the algebra where the atomic terms are elements of B, instead of elements of A).

Skil Expressions

There are two subsets of expressions, where differences between *Skil* and C occur: the primitive and the applicative expressions. The set *PrimExpr* of *Skil* primitive expressions enhances the set *CPrimExpr* of C primitive expressions (which comprises identifiers, constants, and expressions enclosed between brackets) by the set *TypeTstExpr* of type test expressions:

$$\begin{aligned} \mathit{TypeTstExpr} &:= \{\ \mathit{typetst}\ (\tau) \mid \mathit{typetst} \in \{\$\$\mathsf{par},\ \$\$\mathsf{dyn}\},\ \ \tau \in \mathit{Type}\ \} \\ \mathit{PrimExpr} &:= \mathit{CPrimExpr} \cup \mathit{TypeTstExpr} \end{aligned}$$

The set *ApplExpr* of *Skil* applicative expressions consists of:

- Applications of function symbols or formal arguments to a number of expressions not necessarily equal to the arity of the function (because of partial applications, and of HOFs with functional results, which can be over-applied)
- Applications of operator functions. The latter can be partially applied, but not over-applied, because they have a nonfunctional result type.
- Explicitly curried function applications

$$\begin{aligned} \mathit{ApExpr} := &\ \{\ \varphi\,(e_1,\dots,e_n) \mid \varphi \in \mathit{Fun} \cup \mathit{Arg},\ \ e_i \in \mathit{Expr},\ \ i=1,\dots,n,\ \ n \geq 0\ ^{5}\ \} \\ \cup &\ \{\ (\mathit{fop})\,(e_1,\dots,e_n) \mid \mathit{fop} \in \mathit{FOp},\ \ e_i \in \mathit{Expr},\ \ i=1,\dots,n,\ \ n \leq \mathit{ar}\,(\mathit{fop})\ \} \\ \cup &\ \{\ \mathit{apex}\ (e_1,\dots,e_n) \mid \mathit{apex} \in \mathit{ApExpr},\ \ e_i \in \mathit{Expr},\ \ i=1,\dots,n,\ \ n \geq 1\ \} \end{aligned}$$

Notice that the application $e(e')$ is a valid expression only in the previously enumerated cases, in contrast to functional languages, where e can be an arbitrary expression with functional type.

The set of *Skil* expressions is defined as a modification of the set of C expressions, where the C applicative and primitive expressions are replaced by the *Skil* applicative and primitive expressions, respectively:

$$\mathit{Expr} := \mathit{CExpr}\ \langle \mathit{CApExpr}/\mathit{ApExpr},\ \mathit{CPrimExpr}/\mathit{PrimExpr} \rangle$$

Skil Types

We now turn to the formal specification of the types. *Skil* has the same primitive types as C:

$$\mathit{PrimType} := \{\ \mathsf{char},\ \mathsf{int},\ \mathsf{short},\ \mathsf{long},\ \mathsf{signed},\ \mathsf{unsigned},\ \mathsf{float},\ \mathsf{double},\ \mathsf{void}\ \}$$

The set *Type* of *Skil* types is defined over the primitive types and the type variables (taken from a set *TypeVar*), as the closure with respect to the application of the C type constructors *array*, *pointer*, *structure*, *union*, and *function* (Kernighan and Ritchie 1988), and of *Skil*'s *pardata* constructor. We denote these type constructors respectively by arr, ref, $\times$, $+$, $\rightarrow$, and par. Moreover, the expression $\tau\ (\theta_1,\dots,\theta_m)$ denotes the parameterization of the type τ by the types $\theta_1,\dots,\theta_m$ (see Section 3.1.3). Formally, *Type* is the minimal set for which:

[5] Notice that n may be smaller than, equal to, or greater than $\mathit{ar}(f)$. If $n=0$, the brackets may be omitted.

1. $PrimType \subset Type$ (*primitive types*)
2. $TypeVar \subset Type$ (*type variables*)
3. $\tau, \tau_1, \ldots, \tau_n \in Type \setminus \{\mathsf{void}, \rightarrow, \mathsf{par}\}$[6], $n \geq 1$,
   $\theta_1, \ldots, \theta_m \in Type \setminus \{\mathsf{void}, \rightarrow, \mathsf{par}\}$, $m \geq 0 \quad \Rightarrow$
   a) $\mathsf{arr}\ \tau \in Type$ (*array*)
   b) $(\tau_1 \times \ldots \times \tau_n)\ (\theta_1, \ldots, \theta_m) \in Type$ (*structure*)
   c) $(\tau_1 + \ldots + \tau_n)\ (\theta_1, \ldots, \theta_m) \in Type$ (*union*)
   d) $\mathsf{par}\ \tau\ (\theta_1, \ldots, \theta_m) \in Type$ (*pardata*)
4. $\tau \in Type \setminus \{\mathsf{par}\} \Rightarrow \mathsf{ref}\ \tau \in Type$ (*pointer*)
5. $\tau_1, \tau_2 \in Type \Rightarrow \tau_1 \rightarrow \tau_2 \in Type$ (*function*[7])

Similarly to C, *Skil* allows the definition of *type synonyms*, by means of the `typedef` construct. An example is the definition of the type `list` in Section 3.1.3. However, type synonyms do not introduce new types, but merely denote already existing types, that were defined using the constructors defined previously. Therefore, we have not considered type synonyms in our formalisms, but have assumed that any occurrence of such a synonym has been replaced by the type it denotes.

Based on the sets of *Skil* language constructs defined previously, we introduce three auxiliary functions, which we shall use in the instantiation process (Section 3.3). The first of these functions, *type*, delivers the type of an expression, as determined during type checking:

$$type : \ Expr \rightarrow Type$$

The other two functions test, whether a type is functional and polymorphic, respectively. They are defined as follows:

$$functional, polymorphic : \ Type \rightarrow \mathbb{B}$$

$$functional\ (\tau) = \text{if } \tau = \tau_1 \rightarrow \tau_2 \text{ then } true \text{ else } false$$

$$\begin{aligned} polymorphic\ (\tau) = \ & \text{if } \tau \in PrimType \\ & \quad \text{then } false \\ & \text{else if } \tau \in TypeVar \\ & \quad \text{then } true \\ & \text{else if } \tau = \mathsf{arr}\ \tau' \text{ or } \tau = \mathsf{ref}\ \tau' \\ & \quad \text{then } polymorphic\ (\tau') \\ & \text{else if } \tau = (\tau_1 \times \ldots \times \tau_n)\ (\theta_1, \ldots, \theta_m) \text{ or} \\ & \qquad \tau = (\tau_1 + \ldots + \tau_n)\ (\theta_1, \ldots, \theta_m) \text{ or} \\ & \qquad \tau = \mathsf{par}\ \tau'\ (\theta_1, \ldots, \theta_m) \\ & \quad \text{then } polymorphic\ (\theta_1) \text{ or } \ldots \text{ or } polymorphic\ (\theta_m) \\ & \text{else if } \tau = \tau_1 \rightarrow \tau_2 \\ & \quad \text{then } polymorphic\ (\tau_1) \text{ or } polymorphic\ (\tau_2) \end{aligned}$$

[6]The notation "$\tau \neq \rightarrow$" ("$\tau \neq \mathsf{par}$") means that the top constructor of τ is not "$\rightarrow$" ("par").
[7]We consider here, for simplicity, only curried functions. However, this formalism can be straightforwardly generalized to uncurried functions.

functional returns *true*, if the constructor at the top of the type term is $\rightarrow$; *polymorphic* returns *true*, if the type term contains at least one type variable. Notice that in the case of parameterized types ($\times$, $+$, par), the polymorphism of the overall type depends solely on the polymorphism of the type arguments. The reason is that all type variables occurring in a parameterized type must explicitly appear as parameters (see Section 3.1.3). Thus, if all type arguments that are supplied for the parameters are monomorphic, then the overall type is monomorphic, otherwise it is polymorphic.

Finally, we define *equivalence* on types (denoted by $\simeq$) as structural equality, i.e., as identity of the top constructors and recursive equivalence of their parameters. However, as our language is an enhancement of C, we also observe the C typing rules (two structures are distinct iff they have different names, void pointer matches any pointer, etc.).

Further *Skil* Structures

The *Skil* statements are similar to the C statements, but built on the *Skil* expressions. A subset of the statements are the *compound statements* or *blocks*:

$$\begin{array}{lcl} \mathit{Stmt} & := & \mathit{CStmt}\ \langle \mathit{CExpr}/\mathit{Expr} \rangle \\ \mathit{CompStmt} & := & \mathit{CCompStmt}\ \langle \mathit{CExpr}/\mathit{Expr} \rangle \end{array}$$

Analogously, the set of *Skil* function definitions is defined as:

$$\mathit{FunDef}\ :=\ \mathit{CFunDef}\ \langle \mathit{CStmt}/\mathit{Stmt},\ \mathit{CExpr}/\mathit{Expr} \rangle$$

A *Skil* function definition is build similarly to a C function out of a header, that contains the type and the formal parameters, and a body, which is a compound statement (Kernighan and Ritchie 1988). Formally, a function definition $\mathit{fundef} \in \mathit{FunDef}$ has the form $\tau\ f\ (\tau_1\ a_1, \ldots, \tau_n\ a_n)\ \mathit{body}$, where $\tau, \tau_1, \ldots, \tau_n \in \mathit{Type}$, $a_1, \ldots, a_n \in \mathit{Arg}$, and $\mathit{body} \in \mathit{CompStmt}$.

The set of *Skil* programs is defined as $\mathit{Prog}\ :=\ \mathit{CProg}\ \langle \mathit{CFunDef}/\mathit{FunDef} \rangle$.

3.2 Implementation of the Language

In this section, we present the structure of the *Skil* compiler and give an overview of the translation of the functional features. The core of this translation, the instantiation procedure, is described in detail in Section 3.3, whereas the main optimization, inlining, is presented in Section 3.4.

3.2.1 The *Skil* Compiler

Although *Skil* can be used for general-purpose programming (both sequential and parallel), we have designed it to act as host language for algorithmic skeletons. Therefore, we outline in the following the translation of parallel *Skil* programs with skeletons. The skeletons are thereby assumed to be available, similarly to library functions, however in *source* code, because the *Skil* compiler does a source-to-source transformation. The concept of the compilation of *Skil* programs with skeletons is depicted in Figure 3.1.

The input of the *Skil* compiler is twofold: on the one hand, the application program, which is written in a sequential style, i.e., which contains skeleton calls, but no *explicit*

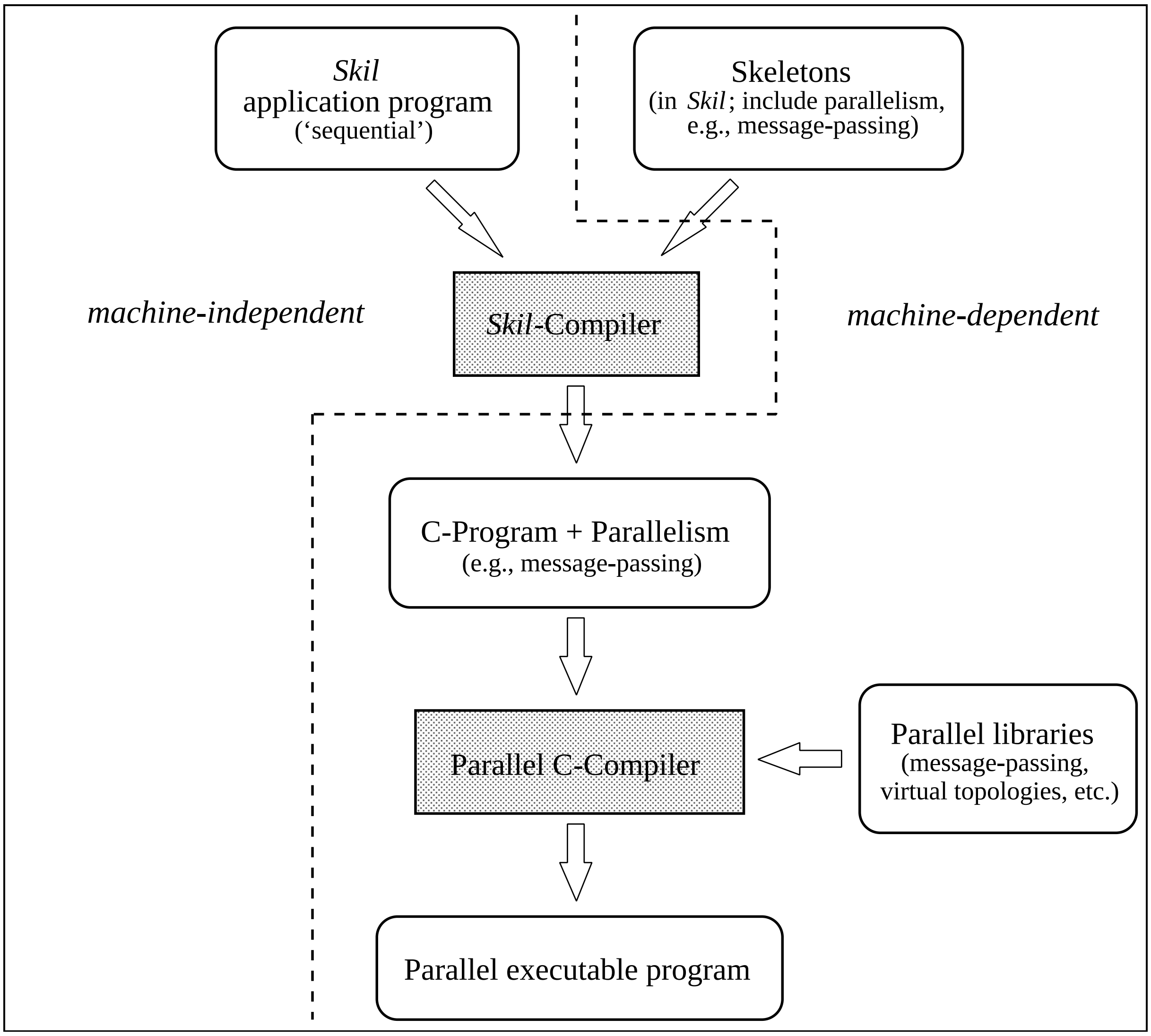


Figure 3.1: The *Skil* concept

parallelism (e.g., message passing on distributed systems), and on the other hand the skeletons, which contain explicit parallelism. The *Skil* compiler merges the two codes into a single program, translates the functional features into C constructs and generates parallel C code, based on the parallelism initially contained in the skeletons. The resulting program must then be translated by a parallel C compiler and linked with the appropriate libraries, such as message passing, virtual topologies, etc.

Apart from enabling a sequential programming style, skeletons offer an interface that abstracts from the underlying hardware, thus making programs portable. Hence, when moving to a different machine, only the skeletons would have to be reimplemented[8], whereas the application program could in principle remain unchanged. Of course, the parallel C compiler employed as back end and the parallel libraries linked to the program are also machine dependent, as shown in Figure 3.1.

[8]We envisage the most common skeletons being implemented on different machines, but having the same interface, similarly to the way standard communication libraries such as PVM or MPI are implemented on different platforms.

The skeletal approach becomes advantageous with respect to portability, if the number of application programs is (much) larger than the number of skeletons that have to be implemented. Notice that this condition is consistent with the requirement that skeletons should have a certain amount of generality (Chapter 2), thus being usable to implement different algorithms from the same class or from related classes of problems.

We now turn to the description of the structure of the *Skil* compiler. The compiler performs a source-to-source translation with C as its output language.

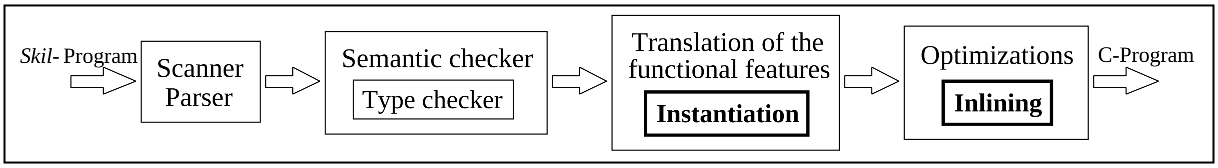


Figure 3.2: Organization of the *Skil* compiler

As shown in Figure 3.2, the *Skil* compiler has four main components:

1. The scanner and the parser, which are written using the tools *lex* and *yacc*, respectively (Levine et al. 1992). The *Skil* grammar supplied to *yacc* is a simplified version of the ANSI-C grammar, as given in Kernighan & Ritchie (1988), augmented with the necessary functional features, type variables, the `pardata` construct, etc. The grammar, as well as an overview of the differences between *Skil* and C are given in Appendix A.

2. The semantic checker, whose main task is to perform a Hindley–Milner polymorphic *type checking* (Milner 1978). Because *Skil*, like C, requires that all variables and functions be typed, the type checker does not have to perform a type *inference*, like in the case of most functional languages (Field and Harrison 1988), but has only to *check* the declared types. Apart from the type check, other semantic checks are done as well, for instance, whether a `case` statement is inside a `switch` statement, whether a `break` statement is inside a loop or a `switch`, whether a label used in a `goto` statement actually exists, etc.

3. The translation of the functional features into C constructs, which is the main transformation performed by the *Skil* compiler. The overall transformation is outlined in Section 3.2.2, whereas its core, the *instantiation* procedure, is presented in detail in Section 3.3.

4. The optimizer, which works on intermediate code (i.e., C) level. The most important optimization, the *inlining* of functions, is presented in Section 3.4.

Concluding this section, we briefly describe the usage of the *Skil* compiler. The compiler can be called by:

`skil` *src* [`-o` *dest*] [`-O`[*n*]] [`-x`ε] [`-v`]

whereby the parameters enclosed between brackets are optional. The meaning of the parameters is as follows:

src	name of the source file
`-o` *dest*	name of the output file (default "`a.out`")
`-O`[n]	optimization level n; n can be 0, 1 or 2, whereby: level 0: only basic optimizations (default) level 1: level 0 + cautious inlining level 2: level 1 + aggressive inlining
`-x`ε	maximal code expansion factor due to inlining, default 0.1; relevant only for optimization level 2
`-v`	verbose; the compiler signals the completion of each of its phases

3.2.2 Translation of the Functional Features

The main task of the *Skil* compiler is to eliminate the functional features, yielding efficient code. This elimination is done by translating each polymorphic higher-order function, possibly containing partial applications, to one or more monomorphic first-order functions, in which all applications are total. The following transformation $\mathcal{T}$ accounts for this translation.

$$\mathcal{T} : \mathit{Prog} \;\rightarrow\; \mathit{Prog}$$
$$\mathcal{T} \;:=\; \widetilde{\mathcal{O}} \circ \widetilde{\mathcal{U}} \circ \widetilde{\mathcal{I}} \circ \widetilde{\mathcal{H}}$$

where:

1. $\mathcal{H}$ η-transforms HOFs with functional result to functions with nonfunctional result. As $\mathcal{H}$ is defined for single functions (see next paragraph), we define $\widetilde{\mathcal{H}}$ as its homomorphic extension to programs, $\widetilde{\mathcal{H}} : \mathit{Prog} \rightarrow \mathit{Prog}$, that is, $\widetilde{\mathcal{H}}$ applies $\mathcal{H}$ to all functions in a program.

2. $\mathcal{I}$ instantiates HOFs with functional arguments, partial applications and polymorphism. As $\mathcal{I}$ is defined for expressions, we denote by $\widetilde{\mathcal{I}}$ its homomorphic extension to programs, $\widetilde{\mathcal{I}} : \mathit{Prog} \rightarrow \mathit{Prog}$. As $\mathcal{I}$ is the most complex transformation, it is presented in detail in Section 3.3.

3. $\mathcal{U}$ uncurries all curried function applications. Similarly to the previous transformation, we define $\widetilde{\mathcal{U}}$ as the homomorphic extension of $\mathcal{U}$ from applicative expressions to programs.

4. $\mathcal{O}$ transforms applications of operator functions (in prefix form) to applications of operators (in infix form). Again, $\widetilde{\mathcal{O}}$ denotes its homomorphic extension from applicative expressions to programs.

η-Transformation

HOFs with functional result are translated to functions with nonfunctional result by η-transformation, that is, by supplying additional arguments that account for the parameters of the result function. These parameters are inserted both in the header and in each `return` expression in the body of the function. As an example, consider the function `select` from Section 3.1.1. This HOF can be translated into a function with nonfunctional result by supplying an additional argument:

```
float select_1 (float f (int), float g (int), int x) {
    if (f (0)) return (f (x)) ;  else return (g (x)) ; }
```

Applications of the old function are then replaced by *partial* applications of the newly obtained one. For example, the call `map (select (sin, log), lst)` will be converted to `map(select_1(sin,log),lst)` whereas the partial application of `select_1` will be eliminated during instantiation, as shown in Section 3.3.

The η-transformation is performed by the function $\mathcal{H}$, which is applied to each function definition. The transformation $\mathcal{H}$ is presented in Figure 3.3.

$$
\begin{array}{l}
\mathcal{H} : \mathit{FunDef} \;\rightarrow\; \mathit{FunDef} \\
\mathcal{H}\;(\tau\; f\; (\tau_1\;\; a_1\texttt{,}\; \ldots\; \texttt{,}\; \tau_n\;\; a_n)\; (\tau'_1\texttt{,}\; \ldots\; \texttt{,}\; \tau'_k)\; \mathit{body}) \;:= \\
\quad \tau\; f\; (\tau_1\;\; a_1\texttt{,}\; \ldots\; \texttt{,}\; \tau_n\;\; a_n\texttt{,}\; \tau'_1\;\; a'_1\texttt{,}\; \ldots\; \texttt{,}\; \tau'_k\;\; a'_k) \\
\qquad \mathit{body}\,[\;\texttt{return}\;\; (e)\;/\;\texttt{return}\;\; (\eta\;(e,\;\{a'_1, \ldots, a'_k\}))\;] \qquad\qquad k \geq 0 \\
\quad \text{where}\;\; a'_1, \ldots, a'_k \in \mathit{Arg}\;\; \text{are new symbols} \\
\qquad\quad \eta : \mathit{Expr} \times \mathcal{P}\;(\mathit{Arg}) \;\rightarrow\; \mathit{Expr} \\
\qquad\quad \eta\;(f\;\;(e_1, \ldots, e_m),\; \{a'_1, \ldots, a'_k\}) \;:=\; f\;\;(e_1, \ldots, e_m,\, a'_1, \ldots, a'_k),\;\; m \geq 0
\end{array}
$$

Figure 3.3: The η-transformation

Notice that this translation implements exactly the intented meaning of HOFs with functional result, namely that of *shorthand notation* for their η-expanded counterparts. Thus, as shown in Section 3.1.1, if such a HOF contains a side effect, this effect occurs each time the function is called. Our η-transformation is not identical to the η-expansion of the λ-calculus (which is invalid in the presence of side effects), rather it is similar to the η-transformation employed by Serrano and Weis (1995).

Uncurrying

This transformation applies to all explicitly curried function calls, both coming from the source program and introduced by the instantiation procedure (Section 3.3.3 – function $\mathcal{S}$, line 19). It consists of flattening function applications, and is defined as follows:

$$
\begin{array}{l}
\mathcal{U} : \mathit{ApExpr} \;\rightarrow\; \mathit{ApExpr} \\
\mathcal{U}\;(f\;\;(a_{11}, \ldots, a_{1n_1})\;\ldots\;(a_{m1}, \ldots, a_{mn_m})) \;:=\; f\;\;(a_{11}, \ldots, a_{1n_1}, \ldots, a_{m1}, \ldots, a_{mn_m})
\end{array}
$$

Translation of Operator Functions

The translation of operator functions is trivial and we present it here only for completeness. For binary operator functions, it amounts to a conversion from prefix to infix form, whereas for unary operator functions it involves only an internal conversion. The transformation $\mathcal{O}$ is mapped to all applicative expressions in the program.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathcal{O} : \mathit{ApExpr} \;\rightarrow\; \mathit{ApExpr}} \\
\mathcal{O}\;((\mathit{fop})\;\;(e_1,\, e_2)) & :=\; (e_1\;\; \mathit{fop}\;\; e_2) & \text{if}\;\; \mathit{fop} \in \mathit{FOp} \setminus \{\texttt{!},\, \texttt{\~{}}\} \\
\mathcal{O}\;((\mathit{fop})\;\;(e)) & :=\; (\mathit{fop}\;\; e) & \text{if}\;\; \mathit{fop} \in \{\texttt{!},\, \texttt{\~{}}\} \\
\mathcal{O}\;(e) & :=\; e & \text{otherwise}
\end{array}
$$

The only pre-condition required by this transformation, is that the applications of operator functions are total, i.e., that all needed arguments are available. However, because $\mathcal{O}$ is performed *after* the instantiation $\mathcal{I}$, this condition is fulfilled, as shown in Theorem 2 b, in Section 3.3.5.

3.3 Instantiation

As one of our major goals is efficiency, we have avoided expensive run-time techniques, such as *closures*, i.e., pairs code–environment (Field and Harrison 1988), to implement the functional features of *Skil*, but used a compile-time technique based on *instantiation* instead.

The instantiation procedure builds for each higher-order or polymorphic function a number of first-order monomorphic specialized functions (*instances*), as needed in the calling program. The technique is related to Wadler's higher-order macro expansion (1990), but has the advantage of providing a unitary framework for translating higher-order functions with functional arguments, partial applications and polymorphism. The target language is the base language, i.e., C. Further related work is presented at the end of Section 3.3.3.

We first present a series of sample instantiations and derive from them the limitations of the concept. We then impose a constraint on the functional arguments of recursively defined HOFs to ensure the termination of the transformation. Finally, after the formal preliminaries, the actual instantiation procedure is presented.

In the following, unless otherwise stated, we denote expressions by e, e', ... , types by τ, τ', ... , function symbols by f, f', g, g', h, h', ... , and formal arguments by a, a', ...

3.3.1 Examples and Limitations of the Concept

We first concentrate on *higher-order functions*, whereas the concepts of partial applications and polymorphism are addressed later in this section. Consider the following (monomorphic) definition of `map`:

```
list <double> map (double f (double), list <double> l)
    { ... cons (f (l->elem), map (f, l->next)) ... }
```

The call `map (log, lst)` can be instantiated together with the definition, yielding the new call `map_1 (lst)` and the new definition:

```
list <double> map_1 (list <double> l)
    { ... cons (log (l->elem), map_1 (l->next)) ... }
```

The instantiation was performed by removing the functional parameter `f` from the header of the definition of `map` and replacing it by the actual argument `log` in the body. Instantiating the recursive call of `map` poses no problems, because the functional argument `f` of this call is just the formal parameter of the function. Thus, the transformation can be handled using a table of already made instantiations and comparing each new HOF to be processed with the instances in the table[9]. In this example, the instance `map_1` is created on encountering the call `map (log, lst)`, and used to instantiate the recursive call of `map` in the body of its definition.

There are nevertheless cases in which instantiation cannot be performed, because it would end up in a nonterminating computation. We construct such an example in the following. Let us first consider the composition functional $(f \circ g)\,(x) \;=\; f\,(g\,(x))$:

```
$c compose ($c f ($b), $b g ($a), $a x)
    { return (f (g (x))) ; }
```

Using it, we can define the following HOF[10]:

```
F (g, x)
    { ... F (compose (g, g), x) ... }
```

The transformation of this function generates a nonterminating sequence of instances, corresponding to the functional arguments g, $\mathtt{g}^2$, $\mathtt{g}^4$, etc. Such situations can be ruled out by imposing certain restrictions. Wadler (1990) sets the condition that no HOFs may have calls to higher-order functions as arguments in recursive calls. We use a refinement of this condition, which in addition allows the instantiation in cases like:

```
F (g, x)
    { ... F (compose (h1, h2), x) ... }
```

where `h1` and `h2` are the names of functions (not of functional formal arguments). We formulate the condition on the instantiation of HOFs as follows:

(C) *If a HOF is defined recursively, then in all its recursive calls (direct or indirect ones), no argument with functional type may contain an application* ***to*** *a functional parameter of the considered HOF.*

In case of direct recursion, this condition can be formally stated as follows.

Let $\tau_0\ f\ (\tau_1\ a_1, \ldots, \tau_n\ a_n)\ \mathit{body}$ be a HOF definition.
Then, for each application of the form $f\ (e_1, \ldots, e_k)$ in *body*
and $\forall i \in \{1, \ldots, k\}$ such that $\mathit{functional}\,(\mathit{type}\,(e_i))$ and $e_i \;=\; g_i\ (e'_{i,1}, \ldots, e'_{i,r_i})\,, \quad r_i \geq 0$
and $\forall l \in \{1, \ldots, r_i\}$ such that $\mathit{functional}\,(\mathit{type}\,(e'_{i,l}))$
and $\forall p \in \{1, \ldots, n\}$ such that $\mathit{functional}\,(\tau_p)$,
the following holds: $e'_{i,l}$ does *not* contain a_p.

However, because HOFs can be indirectly recursive, the check of a HOF call against its definition is not sufficient. Consider for example the following mutually recursive HOFs:

[9] Using this table also avoids the duplication of equivalent instances in the general case.

[10] In this and the following examples, we use a simplified syntax (omitting, for instance, types where they are irrelevant) and highlight only the important aspects.

```
F (g, x)
    { ... H (compose (g, g), x) ... }
H (g, x)
    { ... F (g, x) ... }
```

Supposing `F` is called with a functional argument `p`, we get the following nested HOF calls, obviously leading to a nonterminating instantiation:

```
... F (p, x) ...
  ↪ ... H (compose (p, p), x) ...
      ↪ ... F (compose (p, p), x) ...
          ↪ ...
```

The current HOF call must thus be compared with all *calls* of this HOF that are currently processed. For that, it is necessary to keep track of all currently nested instantiations, which we name *active instantiations.* Moreover, as we no longer compare a call with the corresponding definition, but with another call, and because functional parameters are inlined in the body of a HOF before its processing, we have to keep track, which of the actual arguments of a HOF call were initially formal parameters of the HOF definition. This tracking is done by marking these functional arguments (we use a * for this purpose), and checking the termination condition only for marked arguments.

As an example, consider the HOFs:

```
F (f) { ... F (compose (f, f)) ... }
G (f) { ... G (compose (p, p)) ... }
```

It can be easily verified that, if `p` is a function in the same program, then the call `F (p)` leads to a nonterminating instantiation, whereas the call `G (p)` does not.

The termination condition can thus be formulated as follows:

$$
\begin{array}{lll}
\textbf{(C')} & \multicolumn{2}{l}{\text{Let } f\ (e_1, \ldots, e_n) \text{ be a HOF call.}} \\
 & \multicolumn{2}{l}{\text{Then } \forall \text{ active instance } f\ (e'_1, \ldots, e'_n)} \\
 & \quad \text{and } \forall i \in \{1, \ldots, n\} \text{ such that} & \textit{functional}\ (\textit{type}\ (e_i)) \text{ and} \\
 & & e_i = g_i\ (\hat{e}_{i,1}, \ldots, \hat{e}_{i,r_i}), \quad r_i > 0 \\
 & \quad \text{and } \forall l \in \{1, \ldots, r_i\} \text{ such that} & \textit{functional}\ (\textit{type}\ (\hat{e}_{i,l})) \\
 & \quad \text{and } \forall j \in \{1, \ldots, n\} \text{ such that} & \textit{functional}\ (\textit{type}\ (e'_j)) \text{ and} \\
 & & e'_j = h_j\ (\hat{e}'_{j,1}, \ldots \hat{e}'_{j,s_j}), \quad s_j \geq 0 \\
 & \multicolumn{2}{l}{\text{the following holds: } \hat{e}_{i,l} \text{ does } \textit{not} \text{ contain } h_j{*}.}
\end{array}
$$

The condition **(C')** is in practice not too restrictive, because most HOFs used in real programs pass their functional parameters unchanged to their recursive calls, like the `map` function does. Moreover, as the HOFs in *Skil* are mainly the skeletons, the restriction is placed here on the skeleton implementor, rather than the user, and can therefore be regarded as internal.

We now turn to *partial applications.* Consider again the call `map(lt(thresh),lst)`, which compares all elements of a list with a given threshold value (Section 3.1.2). To instantiate it, together with the definition of `map`, the partial application of `lt` is replaced by the list of its arguments (here `thresh`) and inserted into the body of `map`, yielding the new call `map_2 (thresh, lst)` and the corresponding definition:

```
map_2 (x, l)
    { ... cons (lt (x, l->elem), map_2 (x, l->next)) ... }
```

Notice that we could *not* have applied here the same technique as in the case of plain functional arguments, i.e., eliminate the argument and insert it into the body of the HOF, because the call would have become `map_3 (lst)` and the corresponding definition:

```
map_3 (l)
    { ... cons (lt (thresh, l->elem), map_3 (l->next)) ... }
```

The problem is that, if the variable `thresh` is defined in another environment, then its insertion into the body of `map_3` brings it out of scope[11]. Hence, nonfunctional arguments have to be passed as arguments to HOF instances, unlike functional arguments, which are defined on the top level. The translation of partial applications thus includes a form of λ-lifting, i.e., unnesting of nested function definitions (Johnsson 1985), with the difference that not all parameters are added to the lifted function, but only the ones with local scope.

Should the functional arguments have functional arguments of their own, the latter have to be instantiated first. The instantiation procedure is thus applied innermost-to-outermost.

A *polymorphic* function is translated to one or more appropriate monomorphic instances. Recall the (polymorphic) definition of `map` from Section 3.1.3. On encountering a call that, say, computes the codes for a given list of characters (converting a `list<char>` to a `list <int>`), this function is instantiated to

```
list <int> map_4 (int f (char), list <char> l)
    { ... }
```

This instance can now be transformed further with respect to its functional arguments. In the procedure presented in Section 3.3.3, the instantiation of HOFs with functional arguments, partial applications and polymorphism is done simultaneously.

3.3.2 Formal Preliminaries

We now define a series of data structures and functions that we shall use in the instantiation procedure.

To avoid code duplication in case of equivalent HOF calls and as an aid to the instantiation of nested or recursive HOFs, a *table of instances* is defined. This table contains for each HOF a list of records of all instances created for that HOF. Each record contains the name of the function, the arity of the HOF call, and the type and name of each argument of the call. If the type of an argument is functional and the corresponding actual argument in the call has the form $g\ (e_1, \ldots, e_n)$, with $n \geq 0$, then the name field is set to g. In case of a nonfunctional argument the name field remains empty (ε). The types in the record are needed in the instantiation of polymorphic HOFs, as shown in the next section.

[11] Nevertheless, a possible optimization would be to use this technique for constants passed as arguments to partial applications, as constants are not bound to a certain scope.

$$\begin{aligned} \mathit{InstRec} \; &:= \; \{\, (f,\, k,\, \tau_1,\, g_1,\, \ldots\,,\, \tau_k,\, g_k) \;\mid\; f \in \mathit{Fun},\;\; k \in \mathbb{N},\;\; \tau_i \in \mathit{Type} \cup \{\varepsilon\}, \\ &\qquad\qquad g_i \in \mathit{Fun} \cup \{\varepsilon\},\;\; i = 1, \ldots, k \,\} \\ \mathit{Insts} \; &:= \; [\mathit{Fun} \to \mathcal{P}\ (\mathit{InstRec})] \end{aligned}$$

Notice that the instance record contains function names, but no formal arguments, i.e., $g_i \notin \mathit{Arg}$. The reason is that, by the time a function application is processed, the functional formal parameters of its enclosing function definition have been replaced by the actual function names throughout this definition (see procedure $\mathcal{I}$ in the next section).

Using the table of instances, the translation procedure can check whether an instance that is equivalent to the current one has been created before. Two HOF instances are considered to be *equivalent* (denoted by $\cong$) iff (1) they have the same arity, (2) the types of their arguments are pairwise equivalent, and (3) the functional arguments consist of calls or partial applications of the same functions, respectively. Because the instantiation can be performed more efficiently if an equivalent instance already exists, the check is done *before* transforming the current HOF call, by matching the *call* against the instance records previously created for that HOF. This check is done by the function $\mathcal{E}$, given in Figure 3.4 (hereby, f' is the name of an instance of f).

$$\mathcal{E} : \mathit{Expr} \times \mathcal{P}\ (\mathit{InstRec}) \;\to\; \mathit{InstRec} \cup \{\varepsilon\}$$

$$\begin{aligned} &\mathcal{E}\ (f\ (e_1, \ldots, e_k),\ \mathit{instrecs}) \;:= \\ &\quad \text{if } \exists\ \mathit{instrec} \in \mathit{instrecs} \text{ such that} \\ &\qquad\quad \mathit{instrec} = (f',\, k,\, \tau_1,\, g_1,\, \ldots\,,\, \tau_k,\, g_k) \text{ and} \\ &\qquad\quad \forall i \in \{1, \ldots, k\}\;\; \tau_i \simeq \mathit{type}\ (e_i) \text{ and} \\ &\qquad\qquad\qquad\qquad \mathit{functional}\ (\tau_i) \;\Rightarrow\; e_i \text{ has the form } g_i\ (e'_{i,1},\ \ldots,\ e'_{i,r_i}) \\ &\qquad \text{then result } \mathit{instrec} \\ &\qquad \text{else result } \varepsilon \end{aligned}$$

Figure 3.4: The test on equivalent instances

In addition to the instance table, a list of active instantiations has to be kept. This list helps to detect possible cases of nontermination, in directly or mutually recursive HOFs and consists of the records of all active instantiations: $\mathit{CrtInsts} := [\mathit{Fun} \to \mathcal{P}\ (\mathit{InstRec})]$. An instantiation of a HOF call terminates, if condition **(C')** holds. Practically, **(C')** is checked using instance records by the function $\mathcal{C}$ presented in the next section.

We further use a table that helps us to retrieve a function definition by its name: $\mathit{FunDefs} := [\mathit{Fun} \to \mathit{FunDef}]$. As a table is represented by a function $\mathit{fundefs} \in \mathit{FunDefs}$, its extension by a new entry fundef for the function f is denoted by $\mathit{fundefs}\ [f \to \mathit{fundef}]$.

The last definition we need, is that of the *substitution*. Substitutions are applied to *Skil* expressions, statements, and functions. To define substitutions, we have to introduce the notions of free and bound variables first. A variable is said to be *bound* in a given *Skil* language construct if either of the following holds:

- If the considered construct is a function definition, then a variable is bound, if it is a formal parameter of the function, or if it is declared on the top level in the body of the function, i.e., in the outermost block (these cases correspond to λ- and let-bound variables in functional languages, respectively).

- If the considered construct is a compound statement (block), then a variable is bound, if it is declared on the top level in that block (this case corresponds to let-bound variables in functional languages).

Variables that are not bound in a given construct are called *free*. Notice that a variable may occur free in a construct, but it may be bound in an inner block of that construct.

We define substitutions as idempotent transformations on *Skil* expressions, statements, and functions, where free variables are replaced by given expressions. Substitutions are written in postfix notation. Thus, $con\ [v/t]$ represents a variant of the construct con, in which all free occurrences of the variable v have been replaced by the construct t (which is usually an expression), provided v does not occur free in t. *Subst* denotes the set of all substitutions, and we use $\sigma, \sigma', \ldots$ for its elements.

Similarly, we denote by *TSubst* the set of *type substitutions*, that is, of substitutions applied to types, which replace type variables by types. In the instantiation procedure, we determine, for two given polymorphic types, their 'most general common sub-type'. For that, we introduce the notion of 'most general unifier'. Given the types $\theta_1, \theta_2, \ldots, \theta_n$, a *unifier* is a substitution σ, for which holds: $\theta_1\sigma = \theta_2\sigma = \ldots = \theta_n\sigma$. A unifier σ of a set of types is called *most general unifier (mgu)*, if, for any other unifier σ' of this set of types, there exists a substitution τ, such that $\sigma' = \sigma\tau$. If the mgu of a given set of types exists, then it can be computed using Robinson's unification algorithm (Robinson 1965). In the instantiation procedure, unification on pairs of types is performed by the auxiliary function *unify*.

$$unify : Type \times Type \rightarrow TSubst$$

3.3.3 The Instantiation Procedure

The instantiation of functional arguments, partial applications, and polymorphism is done by the procedure $\mathcal{I}$ shown in Figure 3.5, which is executed for all applicative expressions, that is, for all expressions that have the form $f\ (e_1, \ldots, e_k),\ \ k \geq 0$. These applications are run through starting from the main expression (which must be monomorphic and first order), recursively continuing with the body of each function called. As HOF calls may be nested, $\mathcal{I}$ is applied innermost-to-outermost, i.e., first to $e_1, \ldots, e_k$, and then to the overall call. For simplicity, we present here only the transformation of the overall call.

Apart from the expression to be instantiated, $\mathcal{I}$ gets as arguments the table *fdefs* containing all function definitions, including those of the created instances, the table *insts* with information on each HOF call that led to the creation of an instance, as well as the list *crtinsts* of active instantiations. It returns the new, instantiated expression, as well as the tables *fdefs* and *insts*, which it may have modified by adding new instance definitions and records, respectively. For more clarity, we have employed an imperative notation in the description of the transformation $\mathcal{I}$.

If the function f is neither higher-order nor polymorphic, then $\mathcal{I}$ applies itself to all applicative expressions in the body of f, leaving the initial expression e unchanged (lines 6–9). Otherwise, the function $\mathcal{E}$ is called to check if an equivalent instance of f has already been created (line 10). For the case that no such instance exists, a new instance record *instrec* is created with the arity set to k and all name and type fields set to ε (line 11). After that, each argument of the HOF call is processed by the step procedure $\mathcal{S}$ described later in this section (lines 13–15).

```
1   𝓘 : Expr × FunDefs × Insts × CrtInsts → Expr × FunDefs × Insts
2   𝓘 (e, fdefs, insts, crtinsts) :=
3       let e have the form f (e_1, ... , e_k),   k ≥ 0
4       fdef := fdefs (f)
5       ishof := ∃i ∈ {1, ... , k} such that functional (type (e_i))
6       if not ishof and not polymorphic (type (f))
7           then for all applicative expressions ê in the body of fdef do
8                   (ê, fdefs, insts) := 𝓘 (ê, fdefs, insts, crtinsts)
9                result (e, fdefs, insts)
10      eqinst := 𝓔 (e, insts (f))
11      instrec := (ε, k, ε, ε, ... , ε, ε)          (2k × 'ε' after 'k')
12      e' := e ;   i := 1 ;   σ := [ ]
13      for j := 1 to k do
14          (e', fdef, instrec, i, σ') := 𝓢 (e', fdef, instrec, i, j, eqinst)
15          σ := σ ∘ σ'
16      if eqinst ≠ ε
17          then let eqinst have the form (f', k, ...)
18          else if 𝓒 (instrec, crtinsts (f), insts, false)
19                  then error
20              fdef := fdef σ
21              let f' be a new name
22              fdef := fdef [f/f']
23              let instrec have the form (ε, k, τ_1, g_1, ... , τ_k, g_k)
24              instrec := (f', k, τ_1, g_1, ... , τ_k, g_k)
25              insts := insts [f → (insts (f) ∪ {instrec})]
26              crtinsts := crtinsts [f → (crtinsts (f) ∪ {instrec})]
27              for all applicative expressions ê in the body of fdef do
28                  (ê, fdefs, insts) := 𝓘 (ê, fdefs, insts, crtinsts)
29              crtinsts := crtinsts [f → (crtinsts (f) \ {instrec})]
30              fdefs := fdefs [f' → fdef]
31      result (e' [f/f'], fdefs, insts)
```

Figure 3.5: The instantiation procedure $\mathcal{I}$

In run j, $\mathcal{S}$ transforms the j^{th} argument of the HOF call, i.e., e_j (line 14). However, because each of the previous arguments $e_1, \ldots, e_{j-1}$ was replaced by 0, 1, or more new arguments (in case of partial applications), e_j was shifted to a new position in the argument list pointed to by the extra index i. In case no equivalent instance already exists, $\mathcal{S}$ also performs a stepwise update of the definition of the new instance (*fdef*) and of the instance record (*instrec*). Further, if *fdef* is polymorphic, $\mathcal{S}$ successively constructs the substitution that matches the type of *fdef* with that of the call of f.

After the function $\mathcal{S}$ was called for all arguments and if no equivalent instance already exists (line 16), then the termination condition **(C')** is checked. The function $\mathcal{C}$ delivers *true*, if a possibly nonterminating instantiation was detected (line 18), in which case, $\mathcal{I}$ exits with error. Otherwise, the polymorphic types in *fdef* are instantiated to monomorphic ones using the substitution σ (line 20). After that, the creation of the instance definition and of the instance record has to be completed. The definition header is provided with a fresh function name for the new instance (line 22), which is entered also into the record (line 24). In addition, the body of the definition must be instantiated, too. For that, the now completed record is added to the table of instances *insts* (line 25), as well as to the list of active instantiations *crtinsts* (line 26). The body of the definition is processed by performing the transformation $\mathcal{I}$ on all applicative expressions within (lines 27–28). After processing the body, the current instance record is removed from *crtinsts* (line 29), and the resulting instance definition *fdef* is added to the set of function definitions (line 30). Finally, the name of the HOF is replaced by the name of the instance in the transformed call (line 31).

$$
\begin{array}{l}
\mathcal{S} : \mathit{Expr} \times \mathit{FunDef} \times \mathit{InstRec} \times I\!N \times I\!N \times \mathit{InstRec} \;\rightarrow \\
\qquad \mathit{Expr} \times \mathit{FunDef} \times \mathit{InstRec} \times I\!N \times \mathit{TSubst} \\
\mathcal{S}\ (e,\ \mathit{fdef},\ \mathit{instrec},\ i,\ j,\ \mathit{eqinst}) \ := \\
\quad \text{let } e \text{ have the form } \ f\ (e_1,\ \ldots\ ,\ e_l) \\
\quad \text{and } \mathit{fdef} \text{ have the form } \ \tau_0\ f\ (\tau_1\ a_1,\ \ldots\ ,\ \tau_n\ a_n)\ \mathit{body} \\
\quad \text{and } \mathit{instrec} \text{ have the form } \ (\varepsilon,\ k,\ \tau_1,\ g_1,\ \ldots\ ,\ \tau_{j-1},\ g_{j-1},\ \varepsilon,\ \varepsilon,\ \varepsilon,\ \ldots\ ,\ \varepsilon) \\
\quad \tau \ := \ \mathit{type}\ (e_i) \\
\quad \mathit{instrec} \ := \ (\varepsilon,\ k,\ \tau_1,\ g_1,\ \ldots\ ,\ \tau_{j-1},\ g_{j-1},\ \tau,\ \varepsilon,\ \varepsilon,\ \ldots\ ,\ \varepsilon) \\
\quad \text{if } \ \mathit{polymorphic}\ (\tau_i) \ \text{ then } \ \sigma := \mathit{unify}\ (\tau_i, \tau) \ \text{ else } \ \sigma := [\ ] \\
\quad \text{if not } \mathit{functional}\ (\tau) \\
\qquad \text{then result } \ (e,\ \mathit{fdef},\ \mathit{instrec},\ i+1,\ \sigma) \\
\qquad \text{else let } e_i \text{ have the form } \ g_i^{(?)}\ (e'_{i,1},\ \ldots\ ,\ e'_{i,r_i}), \qquad r_i \geq 0 \\
\qquad\quad e' \ := \ f\ (e_1,\ \ldots\ ,\ e_{i-1},\ e'_{i,1},\ \ldots\ ,\ e'_{i,r_i},\ e_{i+1},\ \ldots\ ,\ e_n) \\
\qquad\quad \text{if } \ \mathit{eqinst} = \varepsilon \\
\qquad\qquad \text{then } \mathit{instrec} \ := \ (\varepsilon,\ k,\ \tau_1,\ g_1,\ \ldots\ ,\ \tau_{j-1},\ g_{j-1},\ \tau,\ g_i^{(?)},\ \varepsilon,\ \ldots\ ,\ \varepsilon) \\
\qquad\qquad\quad \text{let } \ y_{i,1},\ \ldots\ ,\ y_{i,r_i} \ \text{ be new variables} \\
\qquad\qquad\quad \mathit{fdef} \ := \ \tau_0\ f\ (\tau_1\ a_1,\ \ldots\ ,\ \tau_{i-1}\ a_{i-1},\ \mathit{type}\ (e'_{i,1})\ y_{i,1},\ \ldots\ , \\
\qquad\qquad\qquad\qquad \mathit{type}\ (e'_{i,r_i})\ y_{i,r_i},\ \tau_{i+1}\ a_{i+1},\ \ldots\ ,\ \tau_n\ a_n) \\
\qquad\qquad\qquad\qquad \mathit{body}\ [a_i\ /\ g_i{*}\ (y_{i,1},\ \ldots\ ,\ y_{i,r_i})] \\
\qquad\quad \text{result } \ (e',\ \mathit{fdef},\ \mathit{instrec},\ i + r_i,\ \sigma)
\end{array}
$$

Figure 3.6: The auxiliary step procedure $\mathcal{S}$

The step procedure $\mathcal{S}$ (Figure 3.6) instantiates the argument at the j^{th} position in the initial HOF call, which was shifted by the transformation of the first $j-1$ arguments to the i^{th} position, becoming e_i in the current state of instantiation of the call. The index i points also to the current parameter in the instance definition, because the parameters in the definition are replaced simultaneously with those in the call. On the other hand, j

is used as an index on the instance record, because the latter reflects the structure of the initial HOF call. Apart from these parameters, $\mathcal{S}$ gets as arguments the current instance definition *fdef* and current record *instrec*, as well as the equivalent instance *eqinst*. It returns the (possibly modified) expression, instance definition, and record, as well as the position of the next argument of the HOF call that will be processed, and the substitution yielded by matching the types of the current argument in the call and in the definition.

$\mathcal{S}$ first sets the type of the j^{th} item in the instance record to the type of e_i (lines 7–8). Further, it checks whether the type of the i^{th} parameter in the definition is polymorphic and if so, unifies it with the type of e_i yielding the type substitution σ (line 9). This unification succeeds, because the two types were found to be compatible by the type checker. After that, if the type of e_i is not functional, then the instantiation of this argument is finished (line 11). Otherwise, e_i is an application of a function, say g_i, to 0 or more arguments (line 12). The superscript $^{(?)}$ denotes that g_i may be marked by * or not. Then, e_i is replaced by the arguments of the application (line 13). Notice the unitary processing of both plain functional arguments ($r_i = 0$), and functional arguments resulting from partial applications ($r_i > 0$).

If no equivalent instance already exists (line 14), then the instance record and the instance definition must be updated, too. The first update is done by setting the name of the j^{th} argument to g_i (line 15). Notice that, iff g_i was marked in the call (i.e., in e_i), then it also has to to be marked in *instrec* (denoted again by the superscript $^{(?)}$). The second update is performed both on the header and on the body of the function definition. In the header, the i^{th} parameter and its type are replaced by a list of new parameters ($y_{i,1}, \ldots, y_{i,r_i}$) having the types of the new arguments of the call (lines 17–19). In the body, each call of the eliminated formal parameter a_i is replaced by a call of the function g_i (marked, to show that it comes from an instantiated formal parameter), with the new parameters $y_{i,1}, \ldots, y_{i,r_i}$ as (additional) arguments (line 19). As a_i may have been applied to a number of arguments already, this substitution may create an explicitly curried application. This application is later uncurried by the transformation $\mathcal{U}$ (Section 3.2.2). Finally, the updated expression, instance definition and record are returned, together with the position of the next argument to be processed and the current substitution (line 20).

Concluding this section, we describe the function $\mathcal{C}$, which checks for possibly nonterminating instantiations. As the instantiation procedure $\mathcal{I}$ is applied innermost-to-outermost, a nested HOF call is implicitly flattened before processing, so that condition **(C')** cannot be checked directly. We therefore use the *instance records*, which reflect the structure of HOF calls.

There are two aspects in checking the condition **(C')** in this way. On the one hand, every active instantiation has created an instance record and inserted it into the list *crtinsts*. Thus, the check of the current instance against all active instances implies a check of the current record against all records in *crtinsts*. On the other hand, instance records reflect the flattened structure of a HOF call, therefore the check on a nested HOF call must be simulated by a transitive closure with respect to the records created by inner HOF applications.

The function $\mathcal{C}$, given in Figure 3.7, returns *true*, if a possibly nonterminating instantiation was detected. Furthermore, the parameter *nested* ensures that, in case of nested HOF calls, the check is done only for instance records corresponding to *inner* HOF applications. This restriction avoids recursive HOFs with unchanged functional parameter (such as `map`) to be reported erroneously as leading to nonterminating instantiations.

$$\mathcal{C} : \mathit{InstRec} \times \mathcal{P}\,(\mathit{InstRec}) \times \mathit{Insts} \times \mathbb{B} \rightarrow \mathbb{B}$$

$\mathcal{C}\ (\mathit{instrec},\ \mathit{crtrecs},\ \mathit{insts},\ \mathit{nested})\ :=$
  let *instrec* have the form $(f, k, \tau_1, g_1, \ldots, \tau_k, g_k)$
  for $i := 1$ to k do
    if $g_i \neq \varepsilon$
      then if $\exists\ \mathit{rec} = (g_i,\ \ldots) \in \mathit{insts}\,(g)$, for $g \in \mathit{Fun}$ such that
          $\mathcal{C}\ (\mathit{rec},\ \mathit{crtrecs},\ \mathit{insts},\ \mathit{true}) = \mathit{true}$
        then result *true*
      else if *nested* and g_i is marked (i.e., $g_i{*}$) and
          $\exists\ \mathit{crtrec} = (f', l, \tau'_1, g'_1, \ldots, \tau'_l, g'_l) \in \mathit{crtrecs}$ and
          $\exists\ j \in \{1, \ldots, l\}$ such that $g_i = g'_j$
        then result *true*
  result *false*

Figure 3.7: The termination check function $\mathcal{C}$

As an example, consider the nested HOF call `f (g (h (x)))`, where $ar\,(g) \geq 2$ and $ar\,(h) \geq 2$, i.e., the applications of `g` and `h` are partial. The application of `g` is processed first, yielding an instance, say `g1`, and a corresponding record, in which `h` occurs as functional argument. Then, the application of `f` is instantiated to, say `f1`, yielding a record, in which `g1` appears as functional argument. $\mathcal{C}$ checks the termination condition for the application of `f` by comparing the functional arguments in all active instance records of `f` not only with the arguments in the record of `f1`, i.e., with the function `g1`, but also with those in the record corresponding to `g1` (i.e., with the functional argument `h`), and so on, recursively.

Another example, which describes in detail how a possibly nonterminating instantiation is detected, is given at the end of the next section.

Related Work

Our instantiation approach is similar to that of Chin and Darlington (1996). In particular, their termination criterion based on *variable-only* parameters, which do not accumulate sub-terms through recursive calls, is equivalent to our termination condition **(C')**, ruling out the same cases of infinite instantiations.

Another related approach is that of Bell and colleagues (1997), where type-specialized *apply* functions and encodings of the functional arguments are used to perform the translation. This approach is more powerful than ours, being able to handle the cases of potential nontermination. On the other hand, it has the drawback that the execution of the *apply* functions to evaluate the encoded values imposes performance penalties (Bell et al. 1997). This inefficiency makes it less adequate for the use in the *Skil* framework.

However, as Bell and colleagues point out, an even better solution would result from combining the two kinds of approaches, i.e., from using instantiation (or, equivalently, Chin and Darlington's algorithm) to handle the unproblematic cases, and switching to type-specialized *apply* functions in the cases of potentially nonterminating instantiations.

3.3.4 An Example

We shall now illustrate the way the instantiation of HOFs with functional arguments, partial applications, and polymorphism works by means of an example. As most HOFs used in real programs are easy to instantiate—consider for example the function `map`—we have contrived the following (recursive) definitions of the HOFs `inner` and `outer`:

```
extern int p (int, int, int) ;
extern int q (int, int) ;
extern int t (int, int) ;

int inner (int f3 (int), int y)
    { return (f3 (y + 1)) ; }

$t outer ($t f1 ($t), $t f2 ($t), $t x, int n) {
    if (n == 1) return (f1 (x)) ;
    else return (outer (f2, f1, x, n - 1)) ; }
```

Consider further the nested higher-order function call

```
... outer (inner (p (1, 2)), q (3), t (4, 5), 6) ...
```

On comparing the calls of `p`, `q` and `t` with their prototypes, we see that `p` and `q` are partially applied, whereas `t` is totally applied.

We shall now describe the main steps of the instantiation. We start with the leftmost innermost HOF call, i.e., the call of `inner`. This HOF has as a functional argument the partial application of `p`, which is replaced by its arguments `1` and `2` (lines 12–13 in $\mathcal{S}$). Correspondingly, the definition of `inner` has to be adapted. This adaptation is done by replacing `f3` in the formal parameter list by two new arguments (say, `v1` and `v2`) (lines 17–18 in $\mathcal{S}$), and its call in the body by `p(v1,v2)` (line 19 in $\mathcal{S}$). The latter replacement yields `p(v1,v2)(y+1)`, which is later uncurried to `p(v1,v2,y+1)` by the transformation $\mathcal{U}$ (Section 3.2.2). After that, the body of the instance has to be processed (lines 27–28 in $\mathcal{I}$). As it contains no HOF calls, it remains unchanged. The body of `p` is processed during the translation of the module containing its definition[12]. The resulting call is thus `inner_1(1,2)` and the corresponding definition is:

```
int inner_1 (int v1, int v2, int y)
    { return (p (v1, v2, y + 1)) ; }
```

We now instantiate the outer HOF call, which has the form

```
... outer (inner_1 (1, 2), q (3), t (4, 5), 6) ...
```

The first two arguments are partial applications and are dealt with like in the previous case. The third argument is nonfunctional (total application) and therefore remains unchanged. After two steps we obtain the call `outer_1(1,2,3,t(4,5),6)` and the corresponding instance record:

[12] In case of separate compilation (see also the end of this subsection), the starting points of the instantiation are, apart from the main function, all exported functions, as these may be used in other modules.

$instrec_1$:= (`outer_1`, 4, int→int, `inner_1`*, int→int, `q`*, ε, ε, ε, ε)

At the same time, the following instance definition is created:

```
int outer_1 (int v3, int v4, int v5, int x, int n) {
    if (n == 1) return (inner_1 (v3, v4, x)) ;
    else return (outer (q (v5), inner_1 (v3, v4), x, n - 1)) ; }
```

We now have to process the body of this definition. This processing amounts to the instantiation of the call of `outer` in the `else`-branch and is done analogously to our initial expression, yielding the new call `outer_2 (v5, v3, v4, x, n - 1)`, the instance record

$instrec_2$:= (`outer_2`, 4, int→int, `q`*, int→int, `inner_1`*, ε, ε, ε, ε)

and the definition:

```
int outer_2 (int v6, int v7, int v8, int x, int n) {
    if (n == 1) return (q (v6, x)) ;
    else return (outer (inner_1 (v7, v8), q (v6), x, n - 1)) ; }
```

Again, the recursive call to `outer` has to be processed. This time, though, the function $\mathcal{E}$ finds $instrec_1$ to be equivalent to the instance that would be created for this call ($instrec_1 \in insts$ (`outer`)), so it only transforms this call to `outer_1 (v7, v8, v6, x, n - 1)` (line 13 in $\mathcal{S}$ and line 31 in $\mathcal{I}$). This transformation concludes the instantiation of our initial expression, the overall result being as follows:

```
/* p, g, and t unchanged */

int inner_1 (int v1, int v2, int y) {
    return (p (v1, v2, y + 1)) ; }

int outer_1 (int v3, int v4, int v5, int x, int n) {
    if (n == 1) return (inner_1 (v3, v4, x)) ;
    else return (outer_2 (v5, v3, v4, x, n - 1)) ; }

int outer_2 (int v6, int v7, int v8, int x, int n) {
    if (n == 1) return (q (v6, x)) ;
    else return (outer_1 (v7, v8, v6, x, n - 1)) ; }

 ... outer_1 (1, 2, 3, t (4, 5), 6) ...          /* the initial call */
```

Throughout the instantiation, the function $\mathcal{C}$ detects no potentially nonterminating instantiation. Had, however, `outer` been defined as follows:

```
$t outer ($t f1 ($t), $t f2 ($t), $t x, int n) {
    if (n == 1) return (f1 (x)) ;
    else return (outer (compose (f1, f1), f2, x, n - 1)) ; }
```

then, in the instantiation of the recursive call of `outer`, the application of `compose` would have been processed first, yielding the instance record:

$instrec_3$:= (compose_1, 2, int→int, inner_1*, int→int, inner_1*)

Then, the call of outer would have been instantiated, leading to the creation of $instrec_4$ (instead of $instrec_2$):

$instrec_4$:= (outer_3, 4, int→int, compose_1*, int→int, q*, ε, ε, ε, ε)

In this case, $\mathcal{C}$ would have considered first all argument functions occurring in $instrec_4$, then all argument functions appearing in $instrec_3$ (transitive closure over compose_1), etc. It would have found that inner_1, which occurs in $instrec_3$, occurs also in $instrec_1$ (which belongs to the list of active instance records, too) and it is marked. Therefore, $\mathcal{C}$ would have reported a potentially nonterminating instantiation.

Concluding this section, we consider the problem of separate compilation. Our instantiation technique normally requires monolithic compilation, because the application and HOF source codes must be available at the same time to the compiler. However, a restricted form of separate compilation is possible, if all HOF definitions are in the same module as their calls. In this case, first-order monomorphic functions can be used from other modules also. For instance, in the example given in this section, the functions p, q, and t were declared as external, their bodies being processed during the compilation of the modules containing their definitions.

3.3.5 Termination and Correctness of the Instantiation

We now address the aspects of termination and correctness of the instantiation. The termination is considered only with respect to the procedure $\mathcal{I}$, because the other transformations are simple, whereas the correctness is shown for the entire translation scheme $\mathcal{T}$, as different transformations within this scheme account for the translation of different features of *Skil*.

Theorem 1 (Termination) *The instantiation procedure, as defined by $\mathcal{I}$, terminates.*

Proof: The instantiation procedure starts with the main expression and continues with the bodies of the called functions and so on, recursively. To have a nonterminating instantiation, two conditions have to be met:

a) At least one HOF, say f, must be (directly or indirectly) recursive.

b) During the nested instantiation of f, a new instance has to be generated in each step, or, in the case of mutual recursion, a new instance must be generated at the latest after a finite number of steps. As the list *crtinsts* keeps track of the instances created during a nested instantiation, this condition can be formulated as follows: in each step of the nested instantiation of f, no equivalent record may exist in *crtinsts*.

We shall now show that condition **(C')** rules out a nonterminating transformation. An instance record *instrec* of f has the form:

$$instrec \;=\; (\hat{f}, k, \tau_1, g_1, \ldots, \tau_k, g_k)$$

where $\hat{f}$ is the name of the instance, k is the number of arguments of the call, τ_i is the type of the i^{th} argument, and g_i the name of the (applied) function, if the i^{th} argument is functional. Then $\forall$ *instrec'* $\in$ *crtinsts* (f), i.e., for all active instances of f, with *instrec'* $= (\hat{f}', k', \tau_1', g_1', \ldots, \tau_{k'}', g_{k'}')$ we have:

$$\begin{array}{lcl} \mathit{instrec} \not\cong \mathit{instrec}' & \Leftrightarrow & k \neq k' \quad \text{or} \\ & & \exists\ i,\ 1 \leq i \leq k, \ \text{ such that } \ \tau_i \not\simeq \tau_i' \ \text{ or } \ g_i \neq g_i' \end{array}$$

i.e., the new instance differs from the other active instances at least in one of the following items:

- The number of arguments of the call. However, this variation can lead to the creation of at most $ar(f)$ instances.
- The type of one of its arguments. However, as all types in a program can be determined statically, the number of instances that can be generated by type instantiation is finite.
- The applied function, if the argument is functional. As functions can be created dynamically, for instance by partial applications, the generation of an endless sequence of instances is possible in this case. We restrict ourselves therefore to the instantiation of functional arguments and further assume for simplicity that $k = k'$.

Consider a call of f. Because f is a HOF, at least one of its arguments, say the j^{th}, must be functional. The call of f must thus have the form:

$$f\ \ (e_1,\ \ldots\ ,\ e_{j-1},\ g\ (e_1',\ \ldots\ ,e_l'),\ e_{j+1},\ \ldots\ ,\ e_k)$$

where $l \geq 0$. We take a closer look at the functional argument and determine what will be entered in the current instance record at the j^{th} position (g_j). There are three cases, depending on the arguments $e_1', \ldots, e_l'$:

1. None of $e_1', \ldots, e_l'$ is functional. In this case, g is entered as such in the field g_j of the instance record. However, only a finite number of combinations, given by k and the total number of functions in the program, can thus be achieved, so that an endless instantiation is not possible in this case.
2. One of the arguments, say e_i', is functional, i.e., $e_i' = h_i\ (e_1'',\ \ldots\ ,\ e_m'')$, $m \geq 0$, but $\forall$ *instrec'* $\in$ *crtinsts* (f) with *instrec'* $= (\hat{f}', k, \tau_1', g_1', \ldots, \tau_k', g_k')$, we have: $g_s' \neq h_i$, $\forall s = 1, \ldots, k$. In this case, a new instance is generated for g, say g'. However, the next time this call has to be instantiated (due to recursion), its arguments $e_1', \ldots, e_l'$ will be the same, because they do not depend on the arguments of any active instantiation, and hence on the parameters of any enclosing function definition. Thus, g' is used again, no new instance being necessary. Consequently, the generation of an endless sequence of instances is not possible here, either.
3. An e_i' has the form $h_i\ (e_1'',\ \ldots\ ,\ e_m'')$, $m \geq 0$, and $\exists$ *instrec'* $\in$ *crtinsts* (f) with *instrec'* $= (\hat{f}', k, \tau_1', g_1', \ldots, \tau_k', g_k')$, and $\exists s \in \{1, \ldots, k\}$, such that $g_s' = h_i$. Here, a nonterminating instantiation may result, however this case is ruled out by condition **(C')**. Thus, the instantiation procedure, as defined by $\mathcal{I}$, terminates.

$\square$

Theorem 2 (Correctness[13]) *Given a Skil program that may contain higher-order functions, partial applications and polymorphic types, the following holds for the output of the Skil compiler:*

a) *It contains no higher-order functions.*

b) *It contains no partial function applications.*

c) *It may contain noninstantiable polymorphic types. An occurrence of such a type as argument of* `sizeof`, `$$dyn` *or* `$$par` *is regarded as a (fatal) program error. In all other cases, this type is irrelevant to the rest of the program and can thus be instantiated to an arbitrary monomorphic type, e.g., to* `int`.

Proof: The transformation $\mathcal{I}$ starts with the main function and processes all functions called herein, then all functions called in the bodies of these functions, and so on. The part of the code, for which there is no call path connecting it to the main function, remains unprocessed, and may therefore contain functional features at the end of the transformation. However, this code is unreachable, and discarded as such by the back end of the *Skil* compiler. We shall thus ignore the contents of this part of the code in our considerations, and derive our results only for remaining ('call reachable') part of the code.

a) In each phase of the transformation $\mathcal{T}$, a certain kind of HOF is translated:

1. HOFs with *functional result* are converted to functions with nonfunctional result in the η-transformation phase (Section 3.2.2). As *all* function applications and definitions are processed by the transformation $\widetilde{\mathcal{H}}$, the output program can contain no HOFs with functional result.
2. All HOFs with *functional arguments* (in the reachable part of the code) are processed by the transformation $\mathcal{I}$. Consider such a HOF call. If any of its arguments is again a HOF, it is processed first. Consider now the leftmost innermost such call. Let this call have the form f (e_1, ... ,e_n) and suppose e_i has a functional type. Then, if e_i is a function name, it is eliminated from the HOF call; if e_i is a partial application, then it is replaced by its arguments. However, these arguments are nonfunctional, otherwise e_i were a HOF call, which contradicts our assumption of having chosen an *innermost* HOF call. Thus, after processing all arguments (i.e., e_1, ... , e_n), the transformed call is first-order. We can now consider its enclosing HOF call, which, after a similar processing of all HOFs calls inside it, has become an innermost HOF call, and apply the same procedure to it. It can thus be shown by induction, that HOF calls nested up to an arbitrary depth are transformed to first-order calls. The same transformation is applied in parallel to the definitions of these HOFs.
3. *Curried* function definitions and calls—both initial ones from the source program, and those introduced by $\mathcal{S}$ (line 19)—are finally uncurried in the third stage of $\mathcal{T}$, by the transformation $\widetilde{\mathcal{U}}$ (Section 3.2.2).

[13]By *correctness*, we mean here the elimination of *all* functional features (with the possible exceptions outlined in the theorem), rather than the preservation of the semantics of the program.

b) Assume that a partial function application still exists after the instantiation was done by $\mathcal{I}$. There are three possibilities: (1) this partial application is an argument of a function call; (2) it is the argument of a `return` statement; (3) it occurs elsewhere in the program. In the first case, this occurrence would mean that the enclosing call is a HOF call, which contradicts item a). In the second case, the enclosing function definition would be a HOF definition (with functional result type), which again contradicts item a). In the third case, the partial application can either have come from the source program, or have been inserted by the transformation $\mathcal{I}$. If it was already in the source program, then it would have been rejected by the parser as a functional expression that does not comply with the specification of the language (Section 3.1.1). On the other hand, the transformation $\mathcal{I}$ does not insert *new* function calls in a program, but only modifies existing ones. This contradiction shows that all partial applications in the reachable code are eliminated by $\mathcal{I}$.

c) As the main expression is monomorphic, polymorphism can only enter the computation by local variables with polymorphic type or by calls of functions with polymorphic return type. Consider the first function definition (in the order of processing by $\mathcal{I}$), say `f`, in which a noninstantiable polymorphic type (call it `$t`) occurs. The only way in which `$t` could influence the computation outside of `f`, is through type-dependent computations. There are only three operators that take as argument a type and return an expression: `sizeof`, `$$dyn` and `$$par` (see Section 3.1). If the argument of one of these operators is a noninstantiable polymorphic type, then the result is not determined. This case is considered erroneous and disallowed as such. As an illustration, consider the following example:

```
int f (void) {
    $t y ;
    return (1 + sizeof ($t)) ; }
```

Here, the type `$t` cannot be instantiated, therefore the value of the expression `sizeof ($t)` is undefined. In all other cases, noninstantiable polymorphic types are irrelevant for the result of the program, and can therefore be instantiated to arbitrary monomorphic types. Examples of such noninstantiable polymorphic types are the declaration

```
$t x ;
```

where neither `x` nor `$t` are used anywhere in the block containing this definition, or the test

```
if (nil() == nil()) ...
```

where `nil` is the function returning the empty list (having the type `list <$t>`).

□

3.3.6 Code Size Expansion

As we have shown, the functional features of *Skil* are eliminated at compile time, by a transformation built around the instantiation procedure. This procedure creates new, type-specialized, first-order functions. As the transformation is static, an important question is, what is the extent of the resulting code size growth, and, in particular, whether this technique may lead to an explosion of the size of programs.

Obviously, the answer to these questions depends on the *Skil* programs that are considered, more specifically on the number of calls of each HOF with different argument functions or types. Both examples where the code size growth is irrelevant, and examples where this growth becomes dramatic can be constructed. We have thus decided to study this behavior for a series of real *Skil* programs. We have considered the applications presented in the following two chapters, i.e., array applications (matrix multiplication, shortest paths in graphs, Gauss–Jordan elimination, and parallel sorting by regular sampling (Botorog and Kuchen 1996c; Li et al. 1993)), and multigrid applications (different versions of the programs MG00 and MG01 (Stüben 1982; Stüben and Trottenberg 1984)). We have compared the sizes of the output of the *Skil* compiler with and without elimination of the functional features and obtained the results listed in Table 3.1.

Application	Code size without instantiation	Code size with instantiation	Relative code size expansion
matmult	64,128	86,266	1.35
shpaths	79,523	87,913	1.11
gauss	79,090	89,916	1.14
mcarlo	61,946	61,684	1.00
psrs	76,310	98,054	1.28
mg00j	143,839	158,342	1.10
mg00gs	143,492	161,349	1.12
mg00rbj	144,753	169,051	1.17
mg00lj	155,288	167,687	1.08
mg01a	140,217	157,632	1.12

Table 3.1: Code size expansion caused by instantiation

As we can see in Table 3.1, the programs are of average size (60 to 155 kB). The code expansion factors are mostly between 10 and 20 percent, with a worst case of 35 percent. This result is encouraging, because it shows that the main potential drawback of our approach, the explosion of the code size does not occur for any of the considered applications, on the contrary, in some cases, the code size growth is negligible. The explanation for this behavior is that, within a single application program, the variety of instances in which a skeleton appears is usually small, thus leading to the generation of a limited number of new (specialized) functions. The generality of skeletons usually comes to bear across a variety of different applications.

3.4 Inlining

We now turn to the final component of the *Skil* compiler, the optimizer. Although several optimizations are being performed, we present here only the most important one, which can lead to significant gains in performance, namely inline subprogram expansion, or simply *inlining*.

Inlining replaces a function or procedure call with a modified copy of the body of the function (procedure), thus being a transformation that trades code size for execution time. It improves the performance of a program by eliminating the overhead of function calls, i.e., of constructing the activation blocks, of saving and restoring the registers, and of transferring control to and from the called subprograms. Moreover, inlining allows subsequent optimizations, such as common expression elimination, constant propagation, dead code elimination, etc., to be applied to enlarged functions comprising caller and callees.

The importance of this optimization technique has been proved by various researchers. For example, Patterson and Sequin (1982) have reported that the call and return from a function is the most costly construct in high-level languages, arguing for an effective approach to reducing function call costs.

Inlining has a particular positive influence on the performance in the case of *functional* languages, because functional programs often contain many small functions, to improve the readability of the code. Appel (1992) demonstrates that inlining is the most efficient optimization of the SML/NJ compiler. This property of functional programs applies to *Skil* programs also, where instantiated argument functions of HOFs are among the main candidates for inlining.

Inlining has been used both in imperative languages (Chang et al. 1992; Davidson and Holler 1988) and in functional languages (Appel 1992; Jagannathan and Wright 1996; Serrano and Weis 1995; Serrano 1997). In the *Skil* compiler, inlining is performed *after* the elimination of the functional features, and thus on the level of the output language (C).

The two main problems that arise with respect to inlining are *which* functions to inline and *how* to perform the actual inlining. These aspects are handled in the next two sections, respectively. The concluding section presents the results obtained by our instantiation technique, namely performance gains versus code size expansion for several *Skil* applications. We make comparisons with other related approaches and their results.

3.4.1 Selection of the Candidates for Inlining

Our inlining technique has the properties described by Jagannathan and Wright (1996):

- *Generality:* All user-defined procedures are candidates for inlining. Inlining is not limited to system-defined procedures or specially marked procedures. However, the restrictions listed later in this section apply.

- *Selectivity:* Inlining decisions are selective. A procedure may be inlined at some call sites, but not at others, depending on the code size expansion it would cause.

Although inlining can lead to important efficiency gains, it has to be handled with care, because its unlimited use may lead to code size explosion. We have therefore decided to perform inlining only in the following cases:

- On functions that are called only once in the entire program, regardless of their size, as their inlining leads—at least theoretically—to no code expansion
- On functions whose body is small enough

Whereas the first condition is clear, the second one still needs to be quantified. For this, we estimate the size of a given code (i.e., of a program of of a function) by the size of its abstract syntax tree[14]. Although this estimate is not always accurate, it is enough to allow steering the inlining procedure. We put the second condition in concrete terms as follows:

> *A function is a candidate for inlining, if the sum of its size and the size of all previous inlinings, is not greater than ε times the initial size of the program (i.e., the size before inlining).*

In other words, the overall result of inlining should not enhance the code size by more than a factor ε. This condition is consistent with the selectivity property, because it considers the code expansion due to inlining of a function at each call site. Moreover, the candidates for inlining have to be considered in a certain order. We present the ordering criterion later in this section.

The question arising now, is what value should be chosen for the factor ε? For a higher flexibility, we have put this lever at the disposal of the user, allowing him to control the code size growth and, thus, the amount of inlining. Recall the description of the invocation of the *Skil* compiler from Section 3.2.1. The optimizing stage, and in particular inlining, can be steered by means of two compiler options, `-O` and `-x`. The first one sets the optimization level, whereas the second sets the code expansion factor ε, as follows:

`-O`	optimization level 0: only basic optimizations are performed (e.g., dead code elimination), but no inlining
`-O1`	optimization level 1: all optimizations from level 0 + inlining of functions that are called only once
`-O2`	optimization level 2: all optimizations from level 1 + inlining of further functions, up to the expansion factor ε, given as further option (`-x`ε). The default value for ε is 0.1.

Our inlining procedure hass the generality property (Jagannathan and Wright 1996), being able to process any user-defined function. However, inlining cannot be applied in the following special cases:

- To the main function, because this function is called by the system
- To directly recursive functions, because it is unclear, to what extent to unroll the recursion

[14]Throughout the remainder of this section, we shall denote by *size* (of a program or function) the size of the corresponding abstract syntax tree.

- To functions with variable number of parameters[15], as not all parameters may be known at compile time

- To functions passed through pointers as arguments to other functions (Kernighan and Ritchie 1988), as these are merely formal functional parameters

The inlining process employs three auxiliary data structures: two lists, calledonce and inlineable, and a graph, dgraph.

The elements of the list calledonce are the functions that are applied only once throughout the entire program. These functions are always inlined.

The list inlineable contains all other functions, for which inlining can be performed, but may contain also directly recursive functions. The reason is that directly recursive functions may emerge as a result of inlining mutually recursive functions, so that the test on direct recursion has to be done before each actual inlining. Each element of inlineable is a set of all calls of a certain function. The list inlineable is sorted by the product of the size and the number of calls of each function.

The functions in this list are inlined only if they do not lead to an enhancement of the code size of more than a factor ε or if, as a result of previous inlinings, the current call is the only one of this particular function remaining in the program (i.e., if this function has become a calledonce one).

After each inlining step, the list inlineable has to be updated. On the one hand, if all calls of a certain function have been inlined, then this function can be removed from the list. On the other hand, as a result of inlining, the size of the caller grows by the size of the callee, therefore its position in the list needs to be recalculated.

The graph dgraph reflects the static dependencies between the functions in a program. It is a directed weighted graph $G = (N,\ E,\ \mathsf{main})$, where the set of the nodes N consists of all function calls in the program. An edge $e \in E$ is drawn from f_1 to f_2, if a call of f_2 occurs in the body of f_1. e is weighted by the number of calls of f_2 in the body of f_1. main is a distinguished node in the graph, corresponding to the start function `main`.

dgraph enables an easy detection of direct recursion, as cycles comprising one node (i.e., as an edge from f to f). Cycles comprising several nodes correspond to mutual recursion and pose no problems. After each inlining step, the graph has to be updated, either by decrementing the weight of the edge between caller and callee, if the value is larger then 1, or by removing this edge, otherwise. If no more edges point to the callee node, than this node can be removed, too. Due to this removal of nodes and edges, cycles that initially comprise several nodes can degenerate during inlining to one-node cycles. Thus, as stated previously, the test on direct recursion has to be performed prior to each inlining step.

3.4.2 The Inlining Procedure

As we have mentioned, inlining is performed here on C, rather than *Skil* level, that is, after the functional features have been eliminated. The reason is on the one hand that the instantiated program has more functions than the initial program, and on the other

[15] We want to emphasize here that inlining is performed on the level of C programs. Functions with a variable number of parameters are *not* partially applied *Skil* functions—which have been eliminated at this stage—but C functions defined with ellipsis, e.g., `printf` (Kernighan and Ritchie 1988).

hand, that instantiation replaces formal functional parameters—which cannot be inlined—with actual functional arguments, on which inlining can be performed. Thus, the instantiated program offers more possibilities for inlining, and therefore generally leads to better results.

Inlining in an imperative language like C poses additional problems, which are not encountered in functional languages, the main cause being the rather complicated syntactic structures used in imperative languages. For instance, whereas in a functional language both a function call and a function body (right hand side) are expressions, thus allowing an easy inlining, in imperative languages function calls are expressions, but function bodies are statements, thus requiring a series of transformations prior to the actual inlining. Moreover, imperative languages like C contain commands that explicitly steer the control flow, such as `return`, `break`, or `goto`, which have to translated to `goto` statements. Further, static variables occurring in the body of a function to be inlined have to be eliminated. Finally, due to side effects, the order of function calls must be strictly kept, otherwise the semantics of the program might be altered. Thus, the inlining procedure consists of five steps:

1. Elimination of the static variables
2. Adaptation of the enclosing statement
3. Unnesting of the enclosing expressions
4. Insertion of the function body
5. Update of the dependency graph

The first three steps make the necessary preparations for the fourth step, which does the actual inlining, whereas the final step is needed to ensure the consistency of further inlinings. The five steps are presented in the following.

Elimination of the Static Variables

Static variables can be declared in C both globally and locally, i.e., in function bodies or blocks (Kernighan and Ritchie 1988). In the latter case, they retain their value across function calls. However, this property can lead to problems with respect to inlining. Consider, for example, a function `f`, in whose body the static variable `s` is declared. Assume further that `f` is called at two different sites and that both calls should be inlined. In this case, after inlining, we shall have two different variables `s`, one in each inlined body of `f`. The consequence is that the semantics of the program changes, because now two different locations are updated (and two different values retained across function calls), instead of only one.

The solution is to promote the local static variables to global ones, i.e., to pull their declaration out of the enclosing function definition. At the same time, these variables have to get new names, so as to avoid name clashes, e.g., with other promoted static variables having the same names.

Adaptation of the Enclosing Statement

The main problem with respect to inlining in an imperative language is that function calls are expressions, whereas function bodies are statements. Therefore, prior to replacing a function call by the corresponding body, the call and its enclosing expressions and statements have to be lifted to the level of statements. The goal of this transformation is to obtain an expression statement, i.e., a statement that consists of only one expression. This expression is either the function call, if the function has no return values, i.e., if it has the return type `void`, or an assignment with the function call on the right hand side, otherwise. These forms of statements can then be processed by the inlining procedure. As an example, consider the following statement:

```
return (f (x)) ;
```

Assume that we want to inline the call of `f(x)`. We cannot do this directly, because the body of the function `f` is a statement, whereas the argument of `return` has to be an expression. This is why we have to lift the call `f(x)` to an assignment statement. The `return` statement is thus transformed to:

```
_v = f (x) ;
return (_v) ;
```

Although the call of `f` apparently still cannot be replaced by its body, we show how this replacement is done in the actual inlining procedure (step four). On the other hand, the following statement

```
while (f (x))
   stmt ;
```

cannot be transformed directly by a similar lift, because `f(x)` has to be evaluated in each iteration. In this case, the `while` loop itself must first be transformed, the overall result —adequate for inlining—being:

```
_lab:  _v = f (x) ;
       if (_v) {
          stmt ;
          goto _lab ;
       }
```

Without giving a formal description of the transformation—which is simple in most cases—we outline the kinds of statements that need to be transformed:

- Labeled statements have the form `lab:` *stmt*, `case` *expr*`:` *stmt*, or `default` *stmt*. In these cases, the transformation reduces to that of the inner statement *stmt*, whereas the label must then be put in front of the first statement yielded by the transformation.
- `if` and `switch` statements are handled similarly to the preceding `return` example.
- `while` and `do` statements are unfolded, as shown in the second example.

- `for` statements have in C the syntax `for (`e_1`;` e_2`;` e_3`)` and represent a compact notation for `while` loops, where e_1 is the initialization, e_2 is the loop condition, and e_3 is an expression evaluated after each iteration, e.g., the incrementation of a counter (Kernighan and Ritchie 1988). Depending on which of the expressions e_1, e_2, or e_3 contain calls to be inlined, the following transformations are employed:
  - e_1: this expression is lifted in front of the `for` statement and promoted to an expression statement, thus being ready for inlining.
  - e_3: the `for` statement is transformed to a `while` statement. e_3 is promoted to an expression statement and, as it is in the body of the loop, it is ready for inlining.
  - e_2: the `for` statement is transformed to a `while` statement, however, in this case a further transformation of the `while` loop is necessary, as shown in the second preceding example, because e_2 is the loop condition.

  An example involving the third (and most complex) case is given in Figure 3.8.

- `break` statements occurring in loops or `switch` statements are replaced by `goto`s pointing to the first statement after the loop or switch. Similarly, `continue` statements are replaced by jumps to the first statement in the loop. The example in Figure 3.8 also contains the transformation of a `break` statement during inlining.

Unnesting of Enclosing Expressions

Function calls are not only part of statements, but they may also occur inside nested expressions. In this case, to be able to promote the function call as expression statement, the enclosing expressions must be promoted first. Due to the presence of side effects in imperative languages, the order of the promoted statements must be identical with the order in which the subexpressions are evaluated in the initial expression. As an example, consider the following statement, where the (inlineable) call of `f` occurs inside an expression sequence (*comma* expression), which, in its turn, occurs inside an arithmetic expression, the latter being part of an assignment (Kernighan and Ritchie 1988):

```
y = (++i, f(x)) * x ;
```

Here, the value of `i` is incremented, then `f(x)` is evaluated. The result of this evaluation —which is also the result of the comma expression—is the multiplied by `x`. The final result is then assigned to the variable `y`. The transformation for inlining yields:

```
_v1 = ++i ;
_v2 = f(x) ;
y = (_v1, _v2) * x ;
```

As the reader may notice from this example, the unnesting of expressions brings a certain overhead, caused by the additional assignments. The question arising is when this overhead exceeds the gains brought by inlining. Based on profiling information, we have come to the conclusion that inlining is usually not worthwhile for a nesting level beyond 3. Our inlining procedure thus takes into consideration only function calls that are nested at most 3 levels deep in other expressions.

Insertion of the Function Body

Once a function call has been unnested from its enclosing expressions and statements, it can be inlined, i.e., replaced by the function body. The transformation is as follows.

$$\texttt{u = f (}e_1\texttt{, ... , }e_n\texttt{) ;} \qquad \tau\ \texttt{f (}\tau_1\ x_1\texttt{, ... , }\tau_n\ x_n\texttt{)}\ \mathit{body}$$

$$\Downarrow$$

```
{
    register τ1 _v1 = e1 ;
     ...
    register τn _vn = en ;
    register τ _v ;
    body [x1/_v1, ... , xn/_vn, return (a)/{_v = a; goto _lab;}]
    _lab:  u = _v ;
}
```

We assume that the function call is part of an expression statement, more exactly the right hand side of an assignment. The statement is replaced by a block, which starts with the declarations of new, local variables, which get the values of the actual arguments, as well as a variable where the result will be stored[16]. After that, the body of the function is inserted, the formal parameters of its definition being replaced by the variables containing the values of the actual arguments. Finally, each `return` statement in the body is replaced with an assignment to the result variable followed by a jump to the end of the block. Hereby, '`a`' denotes any expression that occurs as argument of a `return` statement. If the function has no result, i.e., its return type is `void`, the declaration of the variable `_v` and the statements assigning values to it are no longer necessary.

It is visible that further optimizations can be performed on the code resulted from this transformation. For example, the variable `_v` could be omitted and the value of any expression denoted by '`a`' assigned directly to the variable `u`. On the other hand, evaluating the actual arguments e_i before entering the body of the function `f` is mandatory; failing to do so would change the order of evaluation, and thus, possibly the semantics of the program. This is why none of the expressions e_i may be inserted directly into the body of `f`, even if that expression occurs only once therein.

An Inlining Example

We now give an example illustrating all steps of the inlining procedure. For that, we have considered that the inlineable call of a function `inc` occurs in a sequence expression inside the loop condition of a `for` statement. Moreover, we have assumed that the function `inc` contains a static variable. The input program is given in Figure 3.8 (left).

In the first step of the inlining procedure, the local static variable `nrcalls` is promoted to a global variable and renamed to `_v1_nrcalls`[17] both in the declaration and in all expressions inside the body of the function `inc`.

[16] For efficiency, these variables are declared as having the storage class `register`, thus advising the C compiler to allocate them machine registers, if possible (Kernighan and Ritchie 1988).

[17] We assume all variable and label names beginning with `_v` and `_l` to be new names.

```
int inc (int x) {
  static int nrcalls = 0 ;
  nrcalls++ ;
  return (x + 1) ;
}


void main (void) {
   ...
  for (i = 0 ;
       inc (y+z), i < N ;
       i++)
    if (i > z)
      break ;
   ...
}
```

⇒

```
static int _v1_nrcalls ;
void main (void) {
 ...
  {
    i = 0 ;
    {
      _l1:{
            int _v2, _v3 ;
            {
              register int _v4_x = y+z ;
              register int _v5 ;
              _v1_nrcalls++ ;
              {
                _v5 = _v4_x + 1 ;
                goto _l3 ;
              }
            }
            _l3:  _v3 = _v5 ;
          }
          _v2 = (_v3, i < N) ;
          if (_v2)
          {
            if (i > z)
              goto _l2 ;
            i++ ;
            goto _l1 ;
          }
      }
    _l2:  ;
  }
 ...
}
```

Figure 3.8: An inlining example

Then, the `for` statement containing the inlineable call is transformed to a loop controlled by a conditional test and a jump to the label `_l1`. At the same time, the `break` statement in the `for` loop is replaced by a `goto` to the end of the loop (to the label `_l2`). After that, the sequence expression enclosing the called to be inlined is unnested. The variables `_v2` and `_v3` serve thereby to store the intermediary results.

Finally, the call of `inc` is replaced with the function body. The variable `_v4_x` replaces the formal parameter `x` and is initialized with the actual argument `y+z`, whereas `_v5` serves to store the result of the function. The `return` statement is replaced by an assignment to `_v5` and a jump out of the function body to the label `_l3`. The output of the inlining procedure is given in Figure 3.8 (right).

Notice that a series of further optimizations may be performed on the output code, such as removing the statement `goto _l3`, as well as its innermost enclosing block, or replacing `_v5` by `_v3` and deleting the assignment between these variables. Performing these optimizations is the task of the C compiler used as back end in our implementation.

Update of the Dependency Graph

The main purpose of the dependency graph is to provide an efficient means to detect directly recursive functions, which are not eligible for inlining. As shown next, an indirectly recursive function can become directly recursive as a result of inlining, therefore this property must be checked prior to *each* inline expansion.

The update of the dependency graph is the final step of an inline expansion and it is outlined in Figure 3.9. Assume, for example, that a program consists of three functions, `main`, `f`, and `g`, where the latter two are mutually recursive, as shown in the left graph. Assume further, that the function `g` can be inlined in the body of its caller, `f`. The update of the dependency graph consists in removing the edge from `f` to `g` (because it has the weight 1), and in altering the edge from `g` to `f`, so that it goes from `f` to `f`. Moreover, as no edges point to `g` anymore, this node can be removed from the graph. At this point however, `f` can no longer be inlined, because it has become directly recursive.

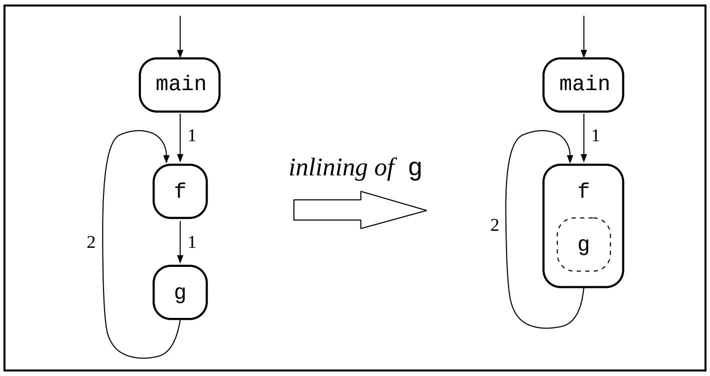


Figure 3.9: The contraction of the dependency graph as result of inlining

3.4.3 Performance

As stated at the beginning of this section, inlining is generally an effective optimization technique. We have investigated its behavior for the array applications described in Section 4.4. As the main problem of inlining is code size growth, we have studied the relation between performance gains and code expansion. Because the latter can be steered by compiler options (see Section 3.4.1), we have measured the relative speedups versus the relative code size expansion for different values of the compiler parameters `-O` and `-x`. The goal was to find, for each application, the best trade-off between performance and code size.

The first program we considered was shortest paths in graphs (Section 4.4.2). The absolute program sizes and run times, together with the relative speedups and code expansion factors are listed in Table 3.2. For more clarity, the dependency of the relative speedups from the code size expansion is depicted in Figure 3.10, in the left graphic.

Compiler options		Code size	Code size expansion	Run time	Relative speedup
`-O`	`-x`ε				
–	–	88,930	1.00	6.37	1.00
`-O`	–	87,934	0.99	6.37	1.00
`-O1`	–	120,254	1.35	6.20	1.03
`-O2`	`-x0.005`	121,603	1.37	6.20	1.03
`-O2`	`-x0.01`	124,635	1.40	4.67	1.36
`-O2`	`-x0.1`	144,056	1.62	4.60	1.38
`-O2`	`-x0.5`	203,703	2.29	4.60	1.38
`-O2`	`-x1`	231,734	2.60	4.67	1.36

Table 3.2: Code size expansion and performance gains due to inlining for *shortest paths*

The curve has a threshold, where the speedup jumps from 3% to 36%, and it is more or less constant in the rest. The explanation of the jump is that here, the function `min` is inlined in a loop that is run through $\mathcal{O}(n^3)$ times (see Section 4.4.2).

Notice the difference between the theoretical code expansion factors ε (given by the argument of the `-x` option) and the actual expansion factors listed in Table 3.2. The reason for this gap is that the compiler uses the size of the *abstract syntax tree* in its decisions, which differs from the actual program size. Moreover, we have made the assumption that the actual insertion of a function body does not enhance the code size, which is not true, as it can be seen in the example in Figure 3.8. On the other hand, an exact evaluation of the size after each inline expansion would be too costly. Nevertheless, the information provided by the static estimation of the program size is enough to allow steering the inlining process, and in particular finding the best trade-off between performance and code size growth, as the results given in this section show.

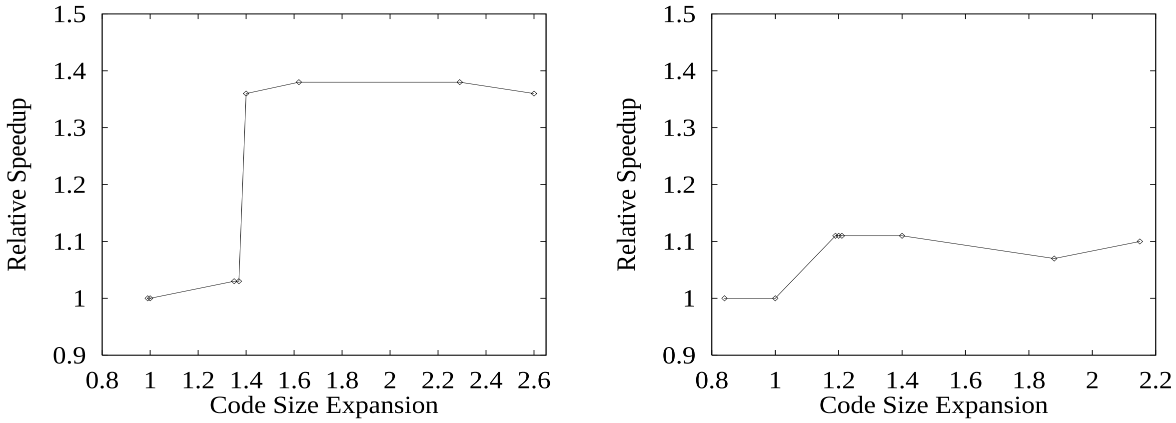


Figure 3.10: Performance gains versus code size expansion due to inlining for shortest paths (left) and matrix multiplication (right)

We have studied further array applications: matrix multiplication and two versions of Gauss–Jordan elimination, without and with partial pivoting (see Section 4.4). The results are synthesized in Figures 3.10 (right graphic) and 3.11, respectively.

Although the improvements were not so dramatic as in the case of shortest paths, we could still obtain relative speedups of more than 10 percent for matrix multiplication and more then 20 percent for both versions of Gauss–Jordan elimination.

Another important aspect is that all these curves have a similar shape, i.e., all have a threshold, where a considerable improvement takes place. The point is thus to find the threshold, where most of the efficiency gains are achieved, paying however the smallest price in terms of code size growth. Obviously, the threshold depends on the particular application considered and can be determined on an experimental basis. For our applications, it was always reached by choosing the value 0.1 for the factor ε[18]. This is why this value was set as default value for ε (see Section 3.4.1).

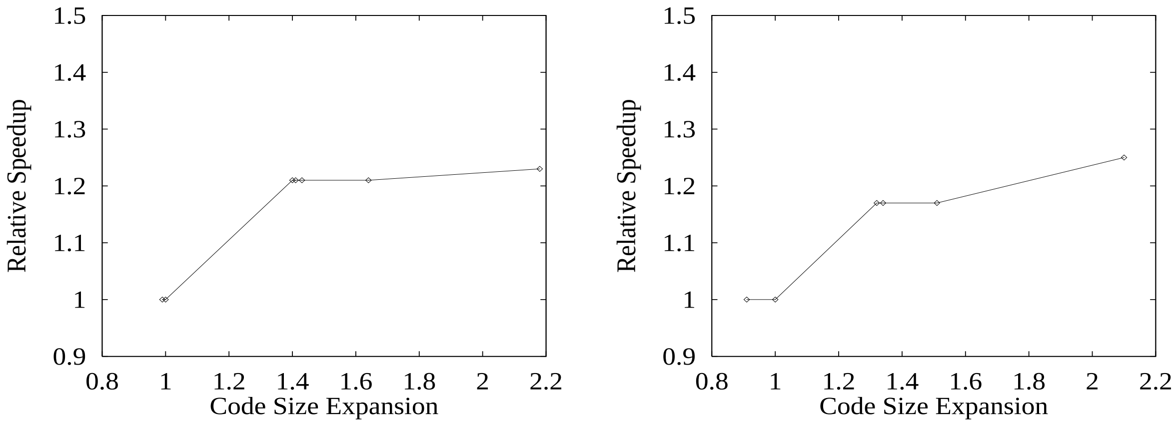


Figure 3.11: Performance gains versus code size expansion due to inlining for two versions of Gauss–Jordan elimination

Notice that, apart from the threshold, the relative speedup curves have slight variations, of less than 5 percent, both in the positive and in the negative sense. The slight drop occurring for example in the rightmost portion of the left graphic in Figure 3.10 is caused by the overheads introduced by the insertion of the function body, which exceeds the gains brought by inlining. Most of these overheads, however, can be removed by subsequent optimization techniques, such as code motion, dead code elimination, constant propagation, etc.

Although inlining usually brings considerable improvements of the performance, there are also cases where this property does not hold, like for instance for the Monte Carlo program presented in Section 4.4.4. Here, a slight drop of the relative speedups can be observed, as depicted in Figure 3.12. However, as the performance loss is merely 3 percent, we include it in the category of overheads that can be compensated by subsequent optimization steps.

Related Work

Inlining has been studied by a series of authors and included in several optimizing compilers. Without trying to be exhaustive, we shall enumerate some of these approaches

[18]Remember that ε is the value passed as argument to the `-x` option, representing the relative growth of the size of the *abstract* syntax tree, which is different from the *actual* code size expansion used in the graphics.

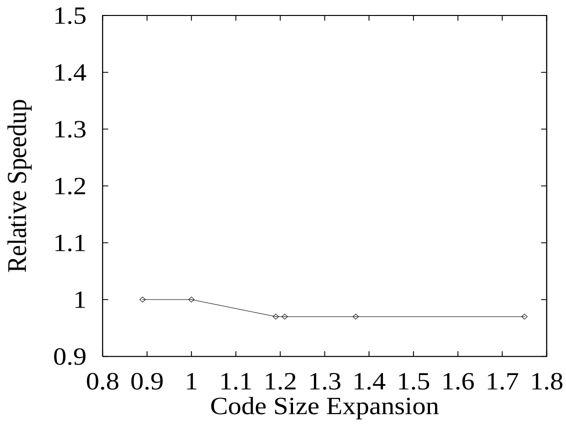


Figure 3.12: Performance gains versus code size expansion due to inlining for a Monte Carlo algorithm

and compare the results obtained by these implementations with ours.

The first two approaches perform inlining in the imperative language C. Davidson and Holler (1988) use only static information to decide what functions to inline, whereas Chang and colleagues (1992) employ profile information for this purpose. The other two approaches, that of Serrano (1995, 1997) and that of Jagannathan and Wright (1996), describe inlining in functional languages. Our approach is somewhere in between the two categories, because, on the one hand, it works on imperative (C) programs, but on the other hand, these programs were derived from *Skil* programs, which are usually written in a functional style, making intensive use of functions.

The most comprehensive results are those given by of Davidson and Holler, who have not only studied the highest number of applications (13), but also made measurements on four different machines. Serrano presents results for three inlining schemes; we have used the one with the highest speedups and lowest size expansions (*labels-inlining*) in the comparisons we have done. A difference between all four implementations and ours, is that our relative speedups are measured on a *parallel* computer, whereas the others are based on *sequential* machines.

The results obtained by these implementations are summarized in Table 3.3. We have listed the number of applications considered by each author, the code expansion ratio, and the relative speedups obtained as result of inlining. For the latter two items, we have given both the minimum and maximum values (in italics), and the average value (in bold).

The results of Davidson and Holler, as well as of Jagannathan and Wright, are more widely dispersed (whereby the best speedups were obtained for contrived examples), whereas the ranges of the others are narrower. The approach of Serrano (1997) is qualitatively different from the others: although most of the speedups are moderate, there is also no code expansion, on the contrary, in several cases the code size decreases.

On the other hand, the average speedups for the functional languages are higher than those for C. This fact is a 'quantitative confirmation' of the statement that functional programs are more adequate for inlining than their imperative counterparts, as they often contain many small functions, so as to improve the readability of the code.

From the measurements we have done, we have always selected the values immedi-

Approach	Language	# Appl's	Code expansion	Speedup
Davidson and Holler (1988)	imperative (C)	13	*0.97 – 1.88* **1.24**	*0.92 – 3.00* **1.12**
Chang et al. (1992)	imperative (C)	8	*1.00 – 1.32* **1.16**	*1.03 – 1.46* **1.11**
Serrano (1997)	functional	10	*0.80 – 1.18* **0.99**	*1.02 – 1.45* **1.16**
Jagannathan and Wright (1996)	functional	8	*0.83 – 2.36* **1.24**	*1.11 – 1.67* **1.25**
Botorog	imperative (C) (*'functional'*)	5	*1.19 – 1.40* **1.30**	*0.97 – 1.36* **1.17**

Table 3.3: Results for different implementations of inlining

ately above the efficiency threshold, because they represent the optimal trade-off between speedup and code growth. Our code expansion ratios are comparable to those obtained by the other implementations, whereas the average value is slightly higher. A possible explanation would be that subsequent optimizations steps, such as elimination of blocks and useless declarations and statements, as well as code motion, are not performed by the *Skil* compiler. On the other hand, the average relative speedup of 1.17 is approximately in the middle between those obtained for C and that of Jagannathan and Wright for functional programs and is comparable to that of Serrano. This result confirms the ranking of our approach between the imperative and the functional ones.

Chapter 4

Array Applications

Practise yourself, for heaven's sake, in little things;
and thence proceed to greater.

EPICTETUS

This chapter presents four simple, array-based applications using algorithmic skeletons. It begins with a presentation of the execution model used in the subsequent implementations. This presentation is followed by a description of the distributed data structure *array*, together with its basic operations. A series of skeletons working on distributed arrays are then presented. For each skeleton, its syntax, implementation and complexity are given. The final part of the chapter describes four array applications, together with their parallel implementation by means of skeletons. Run-time results point out that our implementation is faster than that of a functional language with skeletons, approaching the performance of hand-written C code with message passing.

4.1 The Execution Model

We first describe the execution model, which is the basis for the subsequent implementation of the skeletons. For that, we consider two models, a centralized and a decentralized one, and outline their advantages and drawbacks.

In the *centralized* model, a master processor executes the sequential (here, nonskeleton) operations and sends messages to the other processors telling them when to execute their share of the parallel (skeletal) computation. The slave processors are spawned by these requests and send acknowledgment messages to the master when they have concluded their work. The master has to perform a global synchronization before proceeding.

In the *decentralized* model, sequential operations are replicated on each processor. Pure computational skeletons such as `map` require no communication, because each processor knows when it has to start its part of the parallel computation. The two execution models are sketched in Figure 4.1.

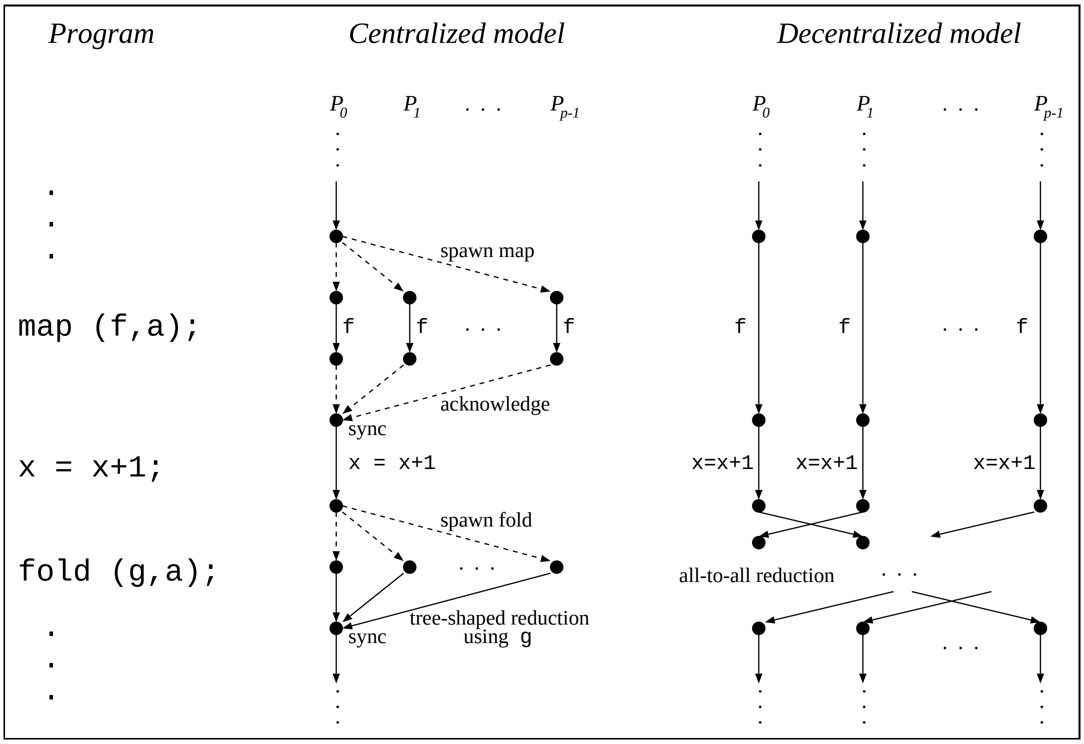


Figure 4.1: Control and message flow in the centralized and in the decentralized model

The main advantage of the decentralized model over its counterpart is that it leads to a simpler implementation of most parallel operations, because, on the one hand, actions do not have to be triggered by messages from the master, and on the other hand, data and functions do not have to be broadcast. At the same time, this model requires less communication (except maybe for the *fold* operation) and it is thus more efficient than its counterpart. Moreover, in the centralized model, the global startups and synchronizations have a negative influence on the scalability of programs.

We have thus chosen the *decentralized* model as the basis for our implementation. This choice was supported also by the use of Parix-C (Parsytec Computer GmbH 1993) in the implementation of the skeletons, as this language is based on the decentralized paradigm SPMD (*s*ingle *p*rogram, *m*ultiple *d*ata). Moreover, as the results in the concluding section of this chapter show, our implementation is considerably faster than the corresponding implementation in the language DPFL (Kuchen et al. 1994). This difference is not least due to the execution model, as DPFL uses the centralized model.

4.2 The Distributed Data Structure *array*

We now present the distributed data structure *array*, showing how it can be implemented in *Skil*.

Syntax:

A distributed array is declared in *Skil* by means of the `pardata` construct:

```
pardata array <$t> implem ;
```

where the type parameter `$t` denotes the type of the elements of the array and *implem* is the data structure created on each processor, as described in the next paragraph.

Implementation:

We consider p processors, but make no assumption about the topology in which they are connected. An array is d-dimensional and can be distributed in the current implementation only block-wise onto the processors[1]. Each block is represented internally as a record containing the (local) array elements, the dimension and total sizes of the array, as well as the bounds of all blocks (the values of the endmost three data structures are identical on all processors):

```
typedef struct _Block {
      $t *arr ;
      int dim ;
      Index totsize ;
      Bounds *bds ;
     } *Block ;
```

Hereby, `Index` is a vector with `dim` integer components, and `Bounds` is a data structure comprising two `Index` structures containing the lowest and highest index of a block, respectively. The lowest and highest indices of the block hosted by processor i can thus be addressed as `bds[i].lower` and `bds[i].upper`, respectively. The bounds of all blocks —including those hosted by other processors—are initialized at the creation of an array (by the skeleton `array_create`), and then used by skeletons that need to know where certain elements of the array are located (like, for instance, the *permute* skeletons).

We consider the set of indices to be ordered lexicographically and the function `next` to compute for a given index, the following index in this set. Moreover, we denote by `+`, `-`, etc. the operations on vectors induced by component-wise application of the corresponding scalar operators.

For more clarity, we shall access the elements of a block (placed in the component `arr`) by their *global* coordinates, i.e., with indices ranging from `bds[i].lower` to `bds[i].upper`, instead of from 0 to the number of elements of the considered block.

Basic operations:

Because the implementation of blocks is hidden, macros have been provided for accessing local array elements, the bounds of the own block, and the total sizes of the array. Array elements can be accessed by the macros:

[1] However, there is no problem in employing other distributions—such as cyclic, block-cyclic, etc. —as well. Of course, the implementation of the skeletons has to be enhanced with these other cases. In a future version, the array skeletons will be extended, to be able to handle the most common data distributions, like for instance those employed in HPF (High Performance Fortran Forum 1993).

```
$t array_get_elem (array <$t> a, Index ix) ;
void array_put_elem (array <$t> a, Index ix, $t newval) ;
```

which read and overwrite the array element having a given index, respectively. An important aspect is that these macros can be used only to access *local* elements, i.e., the index `ix` should be within the bounds of the array block currently placed on each processor. The reason for this restriction is that remote accessing of single array elements introduces additional communication that easily leads to inefficient programs. Nonlocal element accessing is still possible, however only in a coordinated way by means of skeletons.

Although the elements are local, for reasons of transparency, the indices used by the macros are global (i.e., indices of the overall array). However, in certain cases, the use of *local indices* (i.e., relative to the lower bound of a block) might lead to important efficiency gains (Botorog and Kuchen 1996a). For this, the following macros are provided:

```
$t array_get_local_elem (array <$t> a, Index local_ix) ;
void array_put_local_elem (array <$t> a, Index local_ix, $t newval) ;
```

The total sizes of an array are delivered by:

```
Index array_total_sizes (array <$t> a) ;
```

whereas the local bounds of the own block are accessible using the macro:

```
Bounds array_block_bounds (array <$t> a) ;
```

4.3 Skeletons for Arrays

We shall now present the skeletons for the distributed data structure `array`. As we have mentioned, skeletons should provide a certain generality and independence from the machine. In the case of array skeletons, this property amounts to the following requirements:

1. Array skeletons should be generic enough to be used in implementing different algorithms.

2. Their interface should hide the topology of the network used, only the number of processors (represented in our implementation by the global variable `netSize`) and the id of the current processor (`procId`) being available.

For each skeleton, its syntax, implementation, and complexity are given. The implementation is described in the SPMD programming model, where the same code is running on all processors.

The implementation of the *broadcast* skeleton is given also in terms of the SPMD model, using a virtual tree topology (Röttger et al. 1994). We assume thereby that the predicates `Leaf` and `Root`, as well as the functions `Parent` and `Children` (working on processor numbers and having the obvious meaning) are provided for this topology. In the implementation of the *fold* operation, we employ a virtual hypercube topology (Röttger et al. 1994) to build up an all-to-all communication.

In the expressions that we derive for the complexity of the skeletons, the complexities of computation and communication are considered separately, as follows:

- The complexity of one (local) operation is denoted by α.
- The complexity of communicating k data elements between two processors situated at the distance *dist* in the topology is given by $\beta\ (k,\ dist)$. The function β depends on the network topology, link bandwidth, etc.

Apart from this, we employ the following notations: d is the dimension of the array, n is the size of the array in one dimension (we assume here for simplicity that the arrays have the same size in each dimension, the number of elements of an array thus being n^d), p is the number of processors, and δ is the diameter of the topology ($\delta \in \mathcal{O}\,(\sqrt{p})$ for the 2-dimensional mesh, which we have used in our implementation). Moreover, as skeletons are higher-order functions, their complexity depends on that of their functional arguments (Deldarie et al. 1995). We shall denote the complexity of the functional argument f by t_f.

Finally, we need to characterize the embedding of a virtual topology in the given hardware network. For this, we introduce the notions of graph embedding and dilation as follows (Quinn 1994).

Given two graphs, $G = (V, E)$ and $G' = (V', E')$, an *embedding* of G in G' is a function ϕ from V to V'. The *dilation* of this embedding is defined as:

$$dil\,(\phi) \;=\; \max\,\{dist\,(\phi\,(u),\ \phi\,(v))\} \mid (u, v) \in E\}$$

where $dist\ (a, b)$ is the distance between the vertices a and b in G'.

We denote by τ the dilation of embedding a tree topology covering all nodes in the given hardware network, and by χ the dilation of embedding a hypercube topology of maximal size in the hardware network, possibly not covering all nodes. For the 2-dimensional mesh, τ and χ are both $\mathcal{O}\,(\sqrt{p})$ (Röttger et al. 1994).

As the skeletons were implemented on a distributed-memory machine, we have employed message passing in their implementation. We have used two kinds of communication primitives: synchronous or single-threaded ones, denoted by the primitives *send* and *receive*, and asynchronous ones, denoted by *asend* and *areceive*. In the first case, the computation is resumed only *after* the communication took place. Thus, an ordering can be imposed on an overall communication, like for instance in a top-down traversal of a tree topology. In the second case, calls of *asend* or *areceive* generate new threads, which run in parallel with the main computation. These threads are eventually terminated by the *async* primitive. This kind of communication can be employed to avoid deadlocks, as shown in Section 4.3.8, or to improve performance, as it leads to an overlap of computation and communication.

4.3.1 The *create* Skeleton

`array_create` generates a new, block-wise distributed array, and initializes its elements using a given function.

Syntax:

```
array <$t> array_create (int dim, Index size, Index blocksize,
                         Index lowerbd (int), $t init_elem (Index)) ;
```

Hereby:

- `dim` is the number of dimensions of the array (d in our notation). At present, only one- and two-dimensional arrays are supported, because in this case, the internal representation allows to achieve a higher efficiency. Arrays with an arbitrary number of dimensions can be also supported, however, in this case, accessing the array elements is not so efficient.

- `size` contains the global sizes of the array.

- `blocksize` contains the sizes of a block. Passing a zero or negative value for a component makes the skeleton fill in the default value $n/p^{1/d}$.

- `lowerbd` is a user-defined function, which gets a processor number and computes the lowest bound (i.e., the smallest index) of the block hosted by that processor. The skeleton uses this function to initialize on each processor the lower bounds of all blocks, as shown next. Thus, this function determines the mapping of array blocks onto processors.

- `init_elem` is a user-defined function that initializes each element of the array depending on its index.

Implementation:

Each processor first sets the size of the block either to the given value or to the default value $n/p^{1/d}$. Then, it initializes the bounds of all blocks by calling `lowerbd` for all processor numbers. These bounds are necessary for an efficient implementation of certain skeletons, such as *permute* (see Sections 4.3.8 and 4.3.9). After that, it creates a block with the starting index given by `lowerbd (procId)` and the given or default size. Finally, the array elements are initialized by the argument function `init_elem`. The implementation of this skeleton is outlined next (for more clarity, we have tagged the parameters of the skeletons with the labels *in*, *out*, or *inout*):

∀ **processor i:**

```
in dim, in size, in blocksize, in lowerbd, in init_elem, out a

bsize := if blocksize[i] > 0 then blocksize[i]
                              else size[i]/netSize^(1/dim)

for i := 0 to netSize - 1 do
   bds[i].lower := lowerbd (i)
   bds[i].upper := bds[i].lower + bsize

for ix := bds[procId].lower to bds[procId].upper do
   arr[ix] := init_elem (ix)

a := (arr, dim, size, bds)
```

Complexity:

If we denote the complexity of the function `lowerbd` by t_l and that of `init_elem` by t_i, then the complexity of the skeleton is $t(n) \in \mathcal{O}\left(t_l \cdot p + d \cdot p \cdot \alpha + t_i \cdot n^d/p\right)$, because we have to call `lowerbd` for the lower bound of all blocks (i.e., p times), then we have to compute the upper bounds (also p times, component-wise for vectors with d elements), and finally we have to call `init_elem` for each element of a block (each block comprising n^d/p array elements), all blocks being processed in parallel.

4.3.2 The *destroy* Skeleton

`array_destroy` deallocates an existing array, also removing all auxiliary data structures. It has constant complexity $t(n) \in \mathcal{O}(1)$.

Syntax:

```
void array_destroy (array <$t> a) ;
```

4.3.3 The *map* Skeleton

`array_map` applies a given function to all elements of an array and puts the results into another array. However, the two arrays can be identical; in this case the skeleton makes an in situ replacement.

Syntax:

```
void array_map ($u map_f ($t, Index), array <$t> from, array <$u> to) ;
```

The source and the target array must be identically distributed, but need not have the same element type (this fact is denoted here by the type variables `$t` and `$u`). The result is placed in another array rather than returned, because the latter would lead to the creation of a temporary distributed array, whereas our solution avoids this additional memory consumption. Note that this efficiency improvement is not directly possible in a functional host language, where side effects are not allowed[2].

Implementation:

The implementation of this skeleton is simple—*map* is, perhaps, the simplest form of data parallelism—each processor having to apply the argument function `map_f` independently to all elements of its block of the array:

$\forall$ **processor i:**

```
in map_f, in from, out to
for ix := from.bds[procId].lower to from.bds[procId].upper do
   to.arr[ix] := map_f (from.arr[ix], ix)
```

[2]A work-around might be *deforestation* (Wadler 1990), however this procedure is complicated for higher-order programs.

Complexity:

The complexity of this skeleton is $t(n) \in \mathcal{O}\left(t_m \cdot n^d/p\right)$, where t_m is the complexity of the argument function `map_f`.

4.3.4 The *fold* Skeleton

`array_fold` composes ('folds together') all elements of an array by means of a user-defined function. As the result should not necessarily have the same type as the array elements, the skeleton first applies the conversion function `conv_f` to all array elements in a *map*-like way[3]. Thus, our version of *fold* is actually a homomorphism, and it is more general than, for example, the `fold` function of Haskell (Hudak et al. 1992). Homomorphisms are characterized by Bird's Theorem (1989), as functions that can be factored into the composition of a *map* and of a *reduce* operation, i.e.: $hom = red\ (\oplus) \circ map\ (f)$.

Syntax:

```
$u array_fold ($u conv_f ($t, Index), $u fold_f ($u, $u), array <$t> a);
```

Implementation:

First, each processor converts the elements of its block using the function `conv_f` (which, in particular, can be the identity) and then folds all results with the function `fold_f`. After that, the results from all blocks (i.e., one from each processor) are folded together. As the order of composition is nondeterministic, the user should provide an associative and commutative folding function, otherwise the result is unpredictable. The inter-processor composition is implemented as an all-to-all communication performed on the edges of a virtual hypercube topology (Röttger et al. 1994), following the butterfly pattern: in the first step, the pairs of processors whose numbers—in binary representation—differ only in the least significant bit exchange their data, in the second step the processors whose numbers differ in the second least significant bit do the exchange, and so on, up to the most significant bit.

However, if p is not a power of 2—thus allowing the embedding of a virtual hypercube topology covering all processors—this procedure is modified as follows. The hypercube topology is constructed on q processors, where $q = 2^k$, $2^k \le p < 2^{k+1}$, i.e., the maximal hypercube fitting in the given network is built. In the first step, each of the $p-q$ processors that is not part of the hypercube sends its data to one processor of the hypercube, which folds the two results together. Because $p - q < q$, all such pairs of processors can be chosen to be disjoint, therefore this communication can be done in parallel in one step. After that, the folding is done on the hypercube as shown previously. The final step is the reverse of the first one, the processors on the hypercube sending the (now final) result to the processors not on the hypercube, such that the overall result is known eventually on all processors. The implementation of the *fold* skeleton is done as follows.

[3]This step could be done by a preliminary `array_map` also, but our implementation is more efficient.

```
∀ processor i:
    in conv_f, in fold_f, in a, out x

    ix := a.bds[i].lower
    x := conv_f (a.arr[ix])                                  (conversion)
    for ix := next (a.bds[i].lower) to a.bds[i].upper do
        x := fold_f (x, conv_f (a.arr[ix]))     (conversion and local folding)

    k := ⌊ lg₂ (netSize) ⌋
    q := 2^k

    if i ≥ q                     (communication non–hypercube to hypercube)
        send x to processor i - q
    else if i < netSize - q
        receive x' from processor i + q
        x := fold_f (x, x')

    for j := 0 to k - 1 do                       (folding on the hypercube)
        send x to processor bc (i,j)
        receive x' from processor bc (i,j)
        x := fold_f (x, x')

    if i < netSize - q           (communication hypercube to non–hypercube)
        send x to processor i + q
    else if i ≥ q
        receive x from processor i - q
```

Hereby, *bc* is a function that complements the j^{th} bit of the binary representation of its first argument (i), thus describing the connections in the j^{th} dimension of a hypercube:

$$
\begin{aligned}
&bc\,(\mathtt{i},\mathtt{j}) \;=\; (b_{k-1} \ldots \overline{b}_j \ldots b_0) \\
&\quad \text{where} \quad (b_{k-1} \ldots b_j \ldots b_0) \;=\; \mathtt{i}_{(2)}
\end{aligned}
$$

Complexity:

If t_c and t_f are the complexities of the conversion and folding function, respectively, then the complexity of the local computations is given by the initial conversion and the local (sequential) folding, and amounts to $\mathcal{O}\left((t_c + t_f) \cdot n^d/p\right)$. The complexity of communication and nonlocal computation is $\mathcal{O}\left((\log p + 2) \cdot (t_f + \beta\,(1,\ \chi))\right)$, because we have $\log p$ steps in the butterfly pattern, plus two steps with data transfers between the nodes of the hypercube and the nodes that are not on the hypercube. In each step, one data element is transferred at the maximal distance χ, and the folding function is called once. The overall complexity of *fold* is thus: $t(n) \;\in \mathcal{O}\left(t_c \cdot n^d/p \;+\; t_f \cdot (n^d/p + \log p) + \beta\,(1,\ \chi) \cdot \log p\right)$.

4.3.5 The *zip* Skeleton

`array_zip` combines the corresponding elements of two arrays pairwise, using the argument function `zip_f`, and places the results in another array.

Syntax:

```
void array_zip (array <$t> a, array <$t> b, $t zip_f ($t, $t),
                array <$t> res) ;
```

Implementation:

The implementation is similar to that of the *map* skeleton, with the difference that here *two* arrays are scanned simultaneously, rather than one. The two arrays must be distributed identically onto the processors.

$\forall$ **processor i:**

```
in zip_f, in a, in b, out res
for ix := a.bds[procId].lower to a.bds[procId].upper do
   res.arr[ix] := zip_f (a.arr[ix], b.arr[ix])
```

Complexity:

The complexity of `array_zip` is $t(n) \in \mathcal{O}\left(t_z \cdot n^d/p\right)$, because each pair of corresponding elements from the two arrays must be processed. Hereby, t_z is the complexity of the argument function `zip_f`.

4.3.6 The *copy* Skeleton

`array_copy` copies one array into another. The second array must already exist.

Syntax:

```
void array_copy (array <$t> from, array <$t> to) ;
```

Implementation:

Each processor copies locally the elements of the block of the source array to the block of the destination array (the two arrays must be distributed identically onto the processors). If the array blocks are represented as contiguous memory areas (as it is the case in our implementation), copying can be done more efficiently than by explicitly picking each element[4]. This efficiency consideration was the reason why this skeleton was implemented, instead of using `array_map` parameterized with the identity function for this purpose.

$\forall$ **processor i:**

```
in from, out to
memcpy (to.arr[to.bds[procId].lower],
        from.arr[to.bds[procId].lower],
        from.bds[procId].upper - from.bds[procId].lower)
```

[4]In *Skil*, this copying is done by the C function `memcpy`.

Complexity:

The theoretical complexity of `array_copy` is $t(n) \in \mathcal{O}\left(n^d/p \cdot \alpha\right)$, because each element of a block must be processed. However, as explained previously, this skeleton is more efficient than an appropriately parameterized `array_map`, which has the same theoretical complexity.

4.3.7 The *broadcast* Skeleton

`array_broadcast_block` sends one array block to all other processors. Each processor overwrites its block with the broadcast one. The block to be broadcast is the one containing the element with the given index `ix`.

Syntax:

```
void array_broadcast_block (array <$t> a, Index ix) ;
```

Implementation:

The skeleton `array_broadcast_block` works in two steps: first, the processor hosting the block to be broadcast (i.e., the one containing the element with the index `ix`) sends its block to the root of the virtual tree topology (Röttger et al. 1994).

```
∀ processor i:
   inout a, in ix

   if ix ∈ a.bds[i]
       send a.arr to root processor

   if Root(i) then                          (broadcast a.arr on the virtual tree)
      receive block from any processor j
      for ix' := a.bds[i].lower to a.bds[i].upper do
         a.arr[ix'] := block[ix' - a.bds[j].lower + a.bds[i].lower]
      for all k ∈ Children(i) do
         send a.arr to processor k

   else if Leaf(i) then
      receive block from Parent(i)
      for ix' ∈ a.bds[i].lower to a.bds[i].upper do
         a.arr[ix'] := block[ix'-a.bds[Parent(i)].lower+a.bds[i].lower]

   else
      receive block from Parent(i)
      for ix' ∈ a.bds[i].lower to a.bds[i].upper do
         a.arr[ix'] := block[ix'-a.bds[Parent(i)].lower+a.bds[i].lower]
      for all k ∈ Children(i) do
         send a.arr to processor k
```

In the second step, this block is broadcast along the edges of the tree to all other processors. In the implementation, the expression `block[ix' - a.bds[j].lower + a.bds[i].lower]` and the other similar ones represent the conversion of the indices of the received block (`block`) to indices of the local block (`a.arr`).

Complexity:

The complexity of this skeleton is given by the initial sending to the root processor and by the actual broadcast. As the distance between the processor hosting the block to be broadcast and the root processor is in the worst case equal to the diameter of the topology, the complexity of the first communication step is $\mathcal{O}\left(\beta\left(n^d/p,\ \delta\right)\right)$. The complexity of the actual broadcast is $\mathcal{O}\left(\beta\left(n^d/p,\ \tau\right)\cdot\log p\right)$, because a number of n^d/p elements has to be moved along the edges of the virtual tree topology, which has the depth $\log p$. Hereby, τ is the dilation resulting from the embedding of the virtual tree topology in the given hardware topology. The overall complexity of the skeleton is thus: $t(n)\ \in \mathcal{O}\left(\beta\left(n^d/p,\ \delta\right)+\beta\left(n^d/p,\ \tau\right)\cdot\log p\right)$.

In case of a 2-dimensional mesh hardware topology, this complexity is proportional to $\sqrt{p}$, as $\delta \sim \sqrt{p}$. Apparently, this result could be improved by using the edges of the virtual tree topology to perform the initial send, thus remaining in the logarithmic complexity class. However, this improvement is annihilated by the dilation τ, which is $\mathcal{O}\left(\sqrt{p}\right)$, so that the second term of the sum and hence the overall complexity remains in the order of $\sqrt{p}$. Of course, if the hardware topology contains a spanning tree (as in the case of trees or hypercubes with appropriate size), it would be more efficient—at least theoretically—to change the implementation of the first communication step.

4.3.8 The *permute_blocks* Skeleton

`array_permute_blocks` switches the blocks of an array using a given permutation function (`perm_f`). This function computes for each processor number the number of the processor to which the own block has to be sent (i.e., for each source processor, the corresponding target processor). The user must provide a bijective function from $\{0,\ldots,p-1\}$ to $\{0,\ldots,p-1\}$, otherwise a run-time error occurs.

Syntax:

```
void array_permute_blocks (array <$t> a, int perm_f (int)) ;
```

Implementation:

Each processor determines, to which processor it has to send its block and from which processor it has to receive a block. Because `perm_f`$^{-1}$ is not directly available, `perm_f`$^{-1}$`(i)` is computed indirectly, by checking `perm_f(`l`)` $\stackrel{?}{=}$ `i`, for all $l \in \{0,\ldots,p-1\}$. l is uniquely determined, because `perm_f` is bijective. Using this information, the communication structure is built and the blocks are exchanged between the processors.

$\forall$ **processor i:**

```
inout a, in perm_f
j := perm_f (i)
k := perm_f⁻¹ (i)
if i ≠ j then
    asend a.arr to processor j
    areceive block from processor k
    async with processor j
    async with processor k
    for ix := a.bds[i].lower to a.bds[i].upper do
        a.arr[ix] := block[ix - a.bds[k].lower + a.bds[i].lower]
```

Notice that the communication must be *asynchronous*, otherwise a deadlock may result. Further, because the function perm_f is bijective, the condition i $\neq$ j is equivalent to i $\neq$ k, i.e., if a processor gets its new block from another processor, then it also sends its own block to a different processor.

Complexity:

The complexity of this skeleton is given by the evaluation of perm_f(i) (of complexity t_p), by the evaluation of perm_f^{-1}(i) (of complexity $p{\cdot}t_p$), and by sending a block (comprising n^d/p elements) to another processor. The distance on the network between the source and the target processor is in the worst case $\mathcal{O}(\delta)$. The overall complexity of the skeleton is thus $t(n) \in \mathcal{O}\left(p \cdot t_p + \beta\left(n^d/p,\, \delta\right)\right)$.

4.3.9 The *permute_rows* Skeleton

array_permute_rows applies only to 2-dimensional arrays, i.e., to matrices. It permutes the rows of a matrix using a given permutation function, which computes the new position of a row based on its old position. The user must provide a bijective function from $\{0, 1, \ldots, n-1\}$ to $\{0, 1, \ldots, n-1\}$, otherwise a run-time error occurs. Moreover, for efficiency, the blocks of the array may contain only complete rows of the matrix (i.e., the matrix should be divided into horizontal slices).

Syntax:

```
void array_permute_rows (array <$t> a, int perm_f (int)) ;
```

Implementation:

Each processor uses the bounds of all blocks (a.bds) to determine the processors it has to send each of its rows to (denoted here by j) and the processors it has to receive a row from (denoted by h). Then, the communication structure is built. If the source and destination processors are distinct, then the rows are exchanged between processors, otherwise the rows are switched locally. In the first case, a message contains, apart from the elements of the row, the target row number, which tells the destination processor where to store the row. Notice that all received or copied rows must be buffered to avoid inconsistencies resulting from overwriting a row before sending it.

```
∀ processor i:
    inout a, in perm_f

    for k := a.bds[i].lower[0] to a.bds[i].upper[0] - 1 do
        let j be the processor number for which {perm_f(k),_}⁵ ∈ a.bds[j]
        let h be the processor number for which {perm_f⁻¹(k),_} ∈ a.bds[h]

        if i ≠ j then asend a.arr[k] to processor j
                      areceive rows[perm_f(k)] from processor h
                      async with processor j
                      async with processor h
                 else rows[perm_f(k)] := a.arr[k]
                      rows[k] := a.arr[perm_f⁻¹(k)]

    for k := a.bds[i].lower[0] to a.bds[i].upper[0] - 1 do
        a.arr[k] = rows[k]
```

Notice that each of a.arr[k] and rows[k] denotes a *row* of the matrix. As in the case of the previous skeleton, the communication must be asynchronous to avoid deadlocks.

An optimization that we have not included in this description for simplicity, is to pack all rows that have the same destination processor into a single message, instead of sending them one by one. This optimization, of course, does not change the worst-case complexity, but can lead to an improvement of the performance in practice.

Notice that this skeleton has a finer granularity than array_permute_blocks, because different rows of one block can be sent to different processors, or switched within the same block, but at the same time, it has a more restricted applicability, as it can be used only with two-dimensional arrays.

Complexity:

The complexity of this skeleton is given by the two loops, which are run through n/p times, by the computation of perm_f(k) and $\texttt{perm\_f}^{-1}\texttt{(k)}$ (which together amount to $(p+1)\cdot t_p$, where t_p is the complexity of f), by the communication caused by the exchange of the rows (in the worst case, all rows are moved to other—distinct—processors, each processor sending and receiving n/p messages of size n), and by the final copying of the new element from the temporary buffer into the array. The overall complexity of the skeleton is thus $t(n) \in \mathcal{O}\left(n/p \cdot (p \cdot t_p + \beta\,(n,\ \delta) + n \cdot \alpha)\right)$.

4.3.10 The *gen_mult* Skeleton

array_gen_mult composes two arrays of dimension 2 using the pattern of matrix multiplication, i.e., for each element of the result matrix it computes the 'dot product' of

[5] To avoid excessive details, we have used the pseudo-code notation {x,y} for the classical array with elements x and y. In this example, the second component is denoted by _, meaning that it is irrelevant, because a block contains all elements of a matrix row.

the corresponding row of the first matrix and the corresponding column of the second matrix. This composition is parameterized by the function that composes a row element with a column element (here `gen_mult`), and by the function that folds all partial results together (here `gen_add`). If the actual multiplication and addition are used, respectively, then we obtain the classical matrix multiplication. However, by using other argument functions, different algorithms with this communication pattern can be implemented, like for instance shortest paths in graphs (described in Section 4.4.2).

Syntax:

```
void array_gen_mult (array <$t> a, array <$t> b, $t gen_add ($t, $t),
                     $t gen_mult ($t, $t), array <$t> c) ;
```

Hereby, `a` and `b` are the arrays to be 'multiplied' and `c` is the result array.

Implementation:

The skeleton uses Cannon's distributed matrix multiplication algorithm (Kumar et al. 1994), in which local block multiplications alternate with block rotations among the processors. This algorithm, together with its implementation—which is similar to the implementation of the *gen_mult* skeleton—are described in Section 4.4.1.

A problem arises hereby due to the distribution of the matrices onto the processors. The rotations of the blocks are done horizontally for the first matrix and vertically for the second one, whereas the mapping of the result matrix remains unchanged. As the use of the same matrix in the role of both arguments, or in that of an argument and the result would lead to inconsistencies, we have imposed the condition that the matrices `a`, `b` and `c` are *pairwise distinct*, i.e., calls of the form `array_gen_mult(a,a, ... )` or `array_gen_mult(a, ... ,a)` are not allowed.

Complexity:

This skeleton consists of the initial rearrangement step, and of the loop comprising the alternating block multiplications and rotations (see Section 4.4.1). The complexity of the first step is $2\lfloor n/2 \rfloor \cdot \beta\,(n^2/p,\ 1)$, because the blocks of the two matrices are shifted cyclically at most $\lfloor n/2 \rfloor$ times between neighboring processors[6]. The loop is run through $\sqrt{p}$ times, each iteration comprising a multiplication of two blocks with $n/\sqrt{p} \ \times\ n/\sqrt{p}$ elements, and a shift of all elements of a block to a neighboring processor. The overall complexity of the *gen_mult* skeleton is thus $t(n) \in \mathcal{O}\left((n + \sqrt{p}) \cdot \beta\,(n^2/p,\ 1) + n^3/p \cdot \alpha\right)$.

4.4 Sample Array Applications

We shall now present four sample applications illustrating the way parallel programs can be written using algorithmic skeletons. The first program multiplies two matrices using Cannon's algorithm (Kumar et al. 1994). The second one uses the generic matrix

[6] A cyclical shift with k steps in one direction, where $\lfloor n/2 \rfloor < k < n$, is equivalent to a cyclical shift with $n - k$ ($\leq \lfloor n/2 \rfloor$) steps in the opposite direction.

multiplication skeleton to compute all-pairs-shortest-paths in graphs. The third program is a parallel implementation of the Gauss–Jordan elimination method for solving linear systems, and the final one implements a Monte Carlo method for the numerical solution of a partial differential equation (Kuchen et al. 1995). At the end of this section, we summarize the results and present comparisons with other implementations.

We have implemented the skeletons and applications on a Parsytec MC system with 64 T800 transputers connected as a 2-dimensional mesh and running at 20Mhz, under the Parix operating system (Parsytec Computer GmbH 1993). As only 1MB of memory was available per node, larger problem sizes could be fitted only into larger networks.

We have compared our results with those obtained for the same applications using the data-parallel functional language DPFL (Kuchen et al. 1994; Kuchen 1995) and the same or similar skeletons. Our run times are faster than those of DPFL, approaching in most cases those of hand-written message-passing C code. This result is due on the one hand to the efficiency of imperative languages, which is higher than that of their functional counterparts, and on the other hand, to the translation of the functional features by instantiation done by the *Skil* compiler (the implementation of DPFL is based on the closure technique). Detailed results are given throughout this section.

4.4.1 Matrix Multiplication

Algorithm Description:

The algorithm of Cannon multiplies two matrices distributed block-wise onto a two-dimensional mesh of processors. It first shifts the blocks of the first matrix placed on the i^{th} row of processors cyclically i times to the left and the blocks of the second matrix placed on the j^{th} column of processors j times upwards. After this restructuring, the blocks of the two matrices placed on the same processor can be multiplied, because their pairwise corresponding inner indices are equal (i.e., A_{ik} and B_{kj}). After this multiplication, the blocks of the first matrix are shifted cyclically one step to the left and those of the second matrix one step upwards. This shift leads to new index combinations, but these combinations have also equal inner indices, so that the blocks can again be multiplied with each other, and the result is added to that of the previous multiplication. After repeating this combination of computation and communication $\sqrt{p}$ times, where p is the size of the network, we obtain the product matrix of the two matrices. The algorithm is presented in detail by Kumar et al. (1994).

Implementation with Skeletons:

The implementation of this algorithm uses the `array_map` skeleton for the computation steps and the `array_permute_blocks` skeleton for the communication steps[7]. The program is implemented as shown next.

In the first line, the variables `a`, `b`, and `c` are declared as having the type 'distributed array with integer elements'. The arrays are then created with dimension 2, total size `n`$\times$`n`, and default block size by the *create* skeleton. The lower bounds (and hence the form of the blocks) are given by the argument function `lower_bd`. The elements of `a` and `b` are

[7]This implementation is different from the one presented in (Botorog and Kuchen 1996a), which used more specialized communication skeletons. We have given up these skeletons, because their applicability was too restricted.

initialized by the user-defined functions `init_f1` and `init_f2`, whereas those of the result matrix `c` are initially set to zero. The first calls of the *permute* skeleton perform the initial restructuring. The functions `left_i` and `upper_i` compute the numbers of the processors situated i positions to the left and upwards of the current processor, respectively.

```
void matmult (int n) {
   array <int> a, b, c ;

   a = array_create (2, {n,n}, {0,0}, lower_bd (n), init_f1) ;
   b = array_create (2, {n,n}, {0,0}, lower_bd (n), init_f2) ;
   c = array_create (2, {n,n}, {0,0}, lower_bd (n), zero) ;

   array_permute_blocks (a, left_i ) ;
   array_permute_blocks (b, upper_i) ;

   for (i = 0 ; i < sqrt (netSize) ; i++) {
      array_map (local_scal_prod (a, b), c, c) ;
      array_permute_blocks (a, left_1 ) ;
      array_permute_blocks (b, upper_1) ;
   }

   /* output array c */

   array_destroy (a) ;
   /* destroy b and c */
}
```

The local block multiplications are done by mapping the function `local_scal_prod` to every element of the result matrix `c`. This function computes the result of the scalar product of the corresponding row of the `a`-block and the corresponding column of the `b`-block and adds it to the current element of `c` (denoted here by `v`).

```
$t local_scal_prod (array <$t> a, array <$t> b, <$t> v, Index ix) {
   int k, a_size ;
   Bounds a_bds, b_bds ;
   Index local_ix1, local_ix2 ;

   a_bds = array_block_bounds (a) ;
   b_bds = array_block_bounds (b) ;
   a_size = a_bds->upperBd[1] - a_bds->lowerBd[1] ;

   local_ix1[0] = ix[0] - a_bds->lowerBd[0] ;
   local_ix2[1] = ix[1] - b_bds->lowerBd[1] ;

   for (k = 0 ; k < a_size ; k++) {
      local_ix1[1] = local_ix2[0] = k ;

      v += array_get_local_elem (a, local_ix1) *
           array_get_local_elem (b, local_ix2)  ;
   }

   return (v) ;
}
```

The function `local_scal_prod` function addresses the array elements by their local coordinates, because this addressing is more efficient than by global coordinates (Botorog and Kuchen 1996a). For that, it gets the lower bounds of the current block, and computes the local coordinates as offsets relative to these lower bounds.

`local_scal_prod` is called from within the skeleton `array_map`, which supplies the current array element `v` and its index `ix` as arguments (compare this call with the type of `array_map` in Section 4.3.3). However, it needs as further arguments the arrays `a` and `b`. Passing these arguments is done without altering the type of the functional argument of the *map* skeleton by *partially applying* `local_scal_prod` to these two arguments in the main procedure, thus creating a function that has the type expected by *map*.

Run-time Results:

We have measured the run times of the *Skil* program for matrix sizes between 100×100 and 800×800, on 4 to 64 processors[8]. The results are given in Table 4.1, where bold entries stand for absolute run times (in seconds), roman font entries denote speedups relative to the DPFL implementation, and entries in italics remaining slow-down factors with respect to the Parix-C[9] implementation.

n / $\sqrt{p} \times \sqrt{p}$	200	300	400	500	600	700	800
2×2	**14.37** 10.50 *1.30*	**47.84** 10.55 -	–	–	–	–	–
4×4	**4.08** 9.41 *1.38*	**12.99** 9.87 *1.33*	**29.96** 10.08 *1.30*	**57.63** 10.23 *1.29*	**98.58** 10.31 -	–	–
6×6	**2.29** 8.10 *1.56*	**6.49** 8.92 *1.49*	**14.73** 9.37 *1.38*	**28.10** 9.62 *1.35*	**46.42** 9.79 *1.33*	**73.12** - *1.31*	**108.70** - *1.30*
8×8	**1.55** 6.48 *1.87*	**4.38** 7.82 *1.60*	**9.10** 8.47 *1.50*	**17.22** 8.91 *1.44*	**28.06** 9.18 *1.41*	**44.11** 9.41 *1.38*	**63.61** 9.53 *1.36*

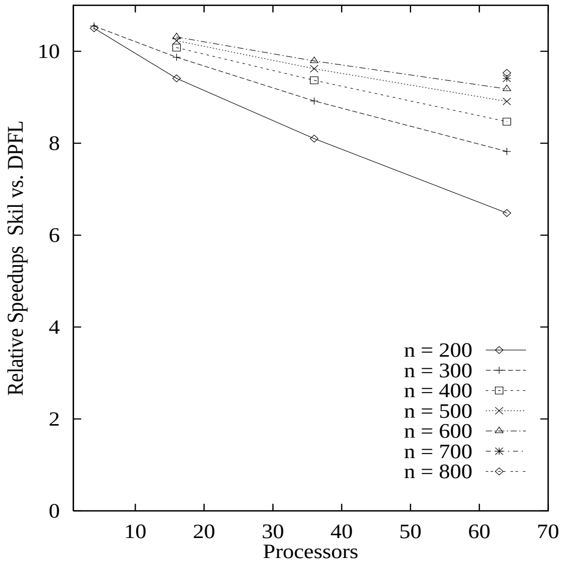
Table 4.1: Run-time results for matrix multiplication [**bold**: absolute times (in seconds); roman font: relative speedups DPFL versus *Skil*; *italics*: relative slow-downs *Skil* versus Parix-C]

For more clarity, we have plotted the speedups relative to DPFL and the slow-downs relative to Parix-C against the number of processors, for all matrix sizes and obtained the graphics in Figure 4.2. In the left graphic, most of the speedups relative to DPFL are grouped around the factor 9, whereas only a few go below 8. This result is reflected in the

[8] In the cases where $\sqrt{p}$ did not divide n, the next highest value divisible by $\sqrt{p}$ was taken, e.g., $n = 204$ for $\sqrt{p} = 6$).

[9] Parix-C is a parallel C dialect based on message passing (Parsytec Computer GmbH 1993).

right graphic also, where most of the slow-downs relative to C are around 1.4, whereas only few are situated above 1.5. The latter values correspond to the case when arrays with small sizes are distributed onto large networks, yielding small blocks. In this case, the communication overhead becomes important, leading to a drop of efficiency. Moreover, the use of the *permute* skeleton, which is more general than the *rotate* skeletons used by Kuchen (1995), also contributes to the slight performance drop in the lower left corner of Table 4.1. However, in most cases, the *Skil* version is only 30 to 40 percent slower than the direct implementation in C with message passing, which is acceptable in practical applications, considering the high level on which programs can be written.



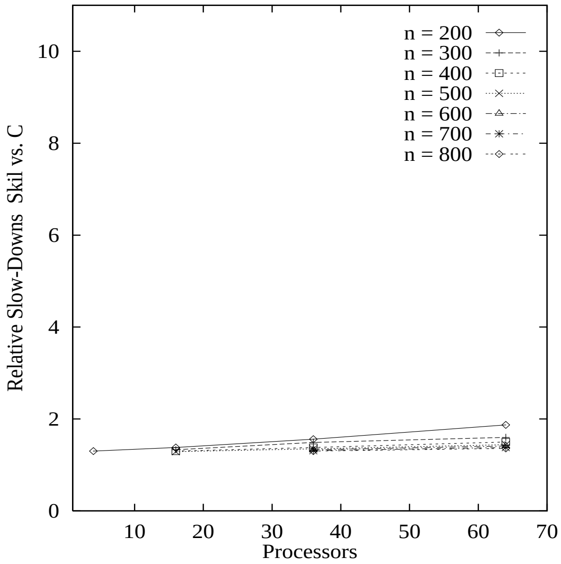


Figure 4.2: Comparison: *Skil* versus DPFL (left) and *Skil* versus Parix-C (right) for matrix multiplication

4.4.2 Shortest Paths in Graphs

Algorithm Description:

Let $G = (V, E)$, with $V = \{v_1, \ldots, v_n\}$ be a graph with nonnegative integer edge weights, and $A = (a_{ij})_{1 \le i,j \le n}$ be its distance matrix, with $a_{ii} = 0$, $a_{ij} = w_{ij}$, iff there is an edge between nodes v_i and v_j with the weight w_{ij}, and $a_{ij} = \infty$ otherwise. Then, the length of the shortest path between the nodes v_i and v_j is equal to the element c_{ij} of matrix $C = A^n$, where A^n is computed on the basis of matrix multiplication, by replacing scalar multiplication by addition and scalar addition by the minimum operation (Quinn 1994).

Implementation with Skeletons:

The implementation of the algorithm is given next. By successively computing A^2, A^4, $\ldots$, we need only $\log_2 n$ iterations to compute A^n (we assume for simplicity that n is a power of 2).

In the first line, the variables `a`, `b` and `c` are declared to have the type 'distributed array with elements of type unsigned integer'. The arrays are then created with dimension

2 and total size n×n by the *create* skeleton. The elements of array a are initialized by the user-defined function init_f, those of array b are set to 0, because b will hold a copy of a, and those of array c are set to the maximal integer value, which represents ∞. To avoid an arithmetic overflow when adding a value to ∞, the type *unsigned integer* was used for the array elements.

```
void shpaths (int n) {
  array <unsigned> a, b, c ;

  a = array_create (2, {n,n}, {0,0}, lower_bd (n), init_f) ;
  b = array_create (2, {n,n}, {0,0}, lower_bd (n), zero) ;
  c = array_create (2, {n,n}, {0,0}, lower_bd (n), int_max) ;

  for (i = 0 ; i < log2 (n) ; i++) {
     array_copy (a, b) ;
     array_gen_mult (a, b, min, (+), c) ;
     array_copy (c, a) ;
  }

  /* output array c */
  /* destroy arrays a, b, and c */
}
```

The rest of the preceding program contains the loop that computes A^n. The skeleton array_gen_mult is called with the minimum function in the role of the scalar addition and with the addition function in the role of the scalar multiplication. As the array arguments of array_gen_mult must be distinct, the array a is first copied into b. Finally, after outputting the result, the arrays are deallocated.

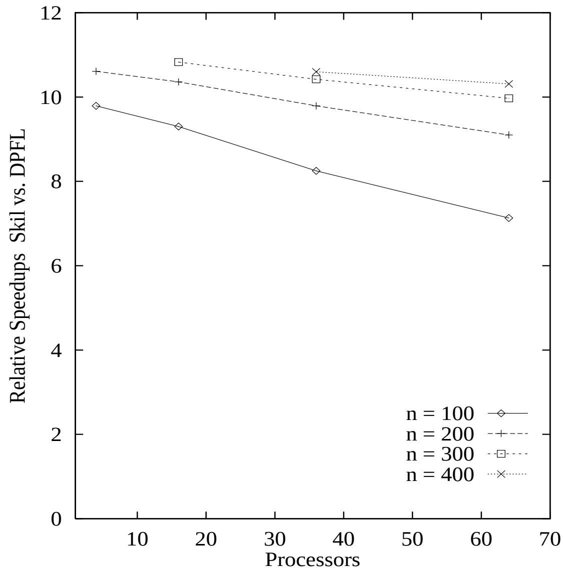
$\sqrt{p} \times \sqrt{p}$ \ n	100	200	300	400	500	600	700	800
2 × 2	**16.97**	**143.63**						
	9.79	10.61	–	–	–	–	–	–
	1.16	*1.09*						
4 × 4	**4.60**	**37.37**	**136.60**	**317.83**				
	9.30	10.36	10.83	–	–	–	–	–
	1.17	*1.08*	*1.09*	*1.08*				
6 × 6	**2.53**	**18.91**	**63.70**	**148.60**	**287.52**	**532.64**	**845.49**	
	8.25	9.79	10.42	10.60	–	–	–	–
	1.26	*1.15*	*1.12*	*1.10*	*1.09*	*1.08*	*1.08*	
8 × 8	**1.79**	**10.85**	**39.01**	**85.20**	**166.14**	**306.34**	**488.89**	**711.29**
	7.13	9.10	9.97	10.31	–	–	–	–
	1.38	*1.20*	*1.15*	*1.12*	*1.11*	*1.10*	*1.09*	*1.09*

Table 4.2: Run-time results for shortest paths [**bold**: absolute times (in seconds); roman font: relative speedups DPFL versus *Skil*; *italics*: relative slow-downs *Skil* versus Parix-C]

Run-time Results:

The shortest paths program was run for graphs with $n = 100$ to $n = 800$ nodes, on 4 to 64 processors. The absolute run times, together with the speedups relative to DPFL and the slow-downs relative to Parix-C are listed in Table 4.2.

We have again plotted the relative speedups and slow-downs, and obtained the graphics shown in Figure 4.3. The curves strongly resemble those from matrix multiplication, with a higher speedup relative to DPFL (nearly 10 on the average) and a lesser slow-down to Parix-C (of ca. 1.1)than in the first example. The reason is that here, the bulk of the computation is encapsulated in a single skeleton (`array_gen_mult`), thus allowing more effective optimizations than if several skeletons are combined.



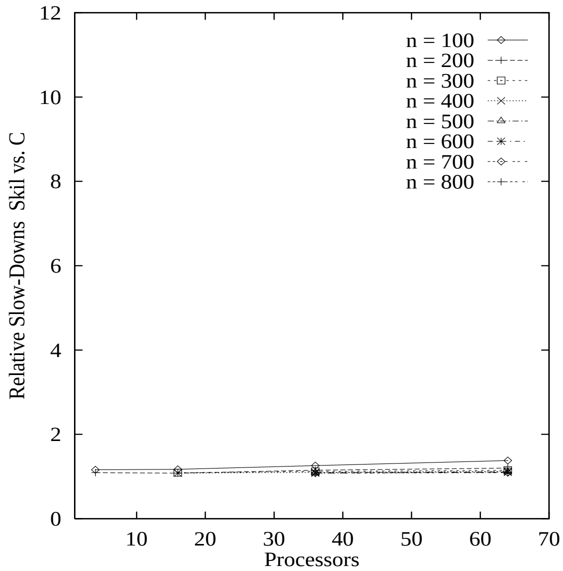


Figure 4.3: Comparison: *Skil* versus DPFL (left) and *Skil* versus Parix-C (right) for shortest paths

4.4.3 Gauss–Jordan Elimination

Algorithm Description:

Gauss–Jordan elimination is used to solve linear systems of the form $Ax = b$ or to invert a matrix A, if A is not singular. In the first case, the right hand side vector b is added to the matrix A as $n+1^{st}$ column. Using linear transformations and combinations of the rows of the extended matrix, such that the identity matrix is built in the first n columns, the (also transformed) $n+1^{st}$ column contains the solution vector x. Formally, the transformation can be written as follows.

$$
\begin{array}{l}
\textbf{for } k := 0 \textbf{ to } n-1 \textbf{ do} \\
\quad \textbf{for } i := 0 \textbf{ to } n-1 \textbf{ do} \\
\quad\quad \textbf{if } i \neq k \textbf{ then} \\
\quad\quad\quad \textbf{for } j := n \textbf{ downto } k \textbf{ do} \\
\quad\quad\quad\quad a_{ij} \ := \ a_{ij} - a_{ik} \cdot (a_{kj}/a_{kk}) \\
\textbf{for } i := 0 \textbf{ to } n-1 \textbf{ do} \\
\quad x_i := a_{in}/a_{ii}
\end{array}
$$

However, this algorithm is not complete, because it fails if the current pivot element (a_{kk}) is 0. This problem can be avoided by partial pivoting, i.e., by exchanging the current row with one where the k^{th} element is nonzero. For reasons of numerical stability, the row with the greatest absolute value of the k^{th} element is chosen as pivot row.

Implementation with Skeletons:

We shall now parallelize this algorithm. The outer loop is not adequate for parallelization, because it contains inter-loop dependencies, so we take the two inner loops and implement them with a *map* skeleton. However, the innermost loop must be run sequentially from n downto k, so as not to overwrite elements too soon. As the *map* skeleton does not process the array elements in a given order, we have to use two different arrays, one as source and one as target. The implementation of the algorithm is done as follows.

```
typedef struct _elemrec {float val; int row; int col;} elemrec ;

void gauss (int n) {
   array <float> a, b, piv ;
   elemrec e ;

   /* create arrays a and b, with size n × n+1 */
   /* create array piv, with size netSize × n+1 */

   for (k = 0 ; k < n ; k++) {
      e = array_fold (make_elemrec, max_abs_in_col (k), a) ;

      if (e.val == 0.0)
         error ("Matrix is singular") ;

      if (e.row != k)
         array_permute_rows (a, switch_rows (e.row, k)) ;

      array_map (copy_pivot (a, k), piv, piv) ;
      array_broadcast_block (piv, {k/(n/netSize), 0}) ;
      array_map (eliminate (k, a, piv), a, b) ;
      array_copy (b, a) ;
   }

   array_map (normalize (b), b, a) ;

   /* output array b */
   /* deallocate arrays a, b and piv */
}
```

We consider that the right hand side vector b has been appended to the matrix A, yielding a $n \times (n+1)$ matrix. This matrix is divided into p blocks (p is the number of processors and is stored in the global variable `netSize`), each block containing n/p rows (we assume for simplicity that p divides n).

The first problem is now to determine and to broadcast the pivot row. The determination part is done by means of the skeleton `array_fold`. The result we expect from this skeleton is twofold: on the one hand, the number of the row containing the pivot element with the maximal absolute value, and on the other hand, the value of the pivot element[10]. For that, we define the structured type `elemrec` that contains for each array element its value, its row and its column. `array_fold` first converts each array element to an `elemrec` and then folds them together, using the function `max_abs_in_col`, which computes the maximum only over those elements, whose column is k. The use of the `elemrec` data structure is possible, because of the *polymorphic* type of the skeleton `array_fold`.

Having found the would-be pivot row, we must exchange it with the current row. This exchange is done by calling the skeleton `array_permute_rows` with an argument function that, for each of the considered two rows, returns the number of the other one, and is the identity for every other row. Notice that, because the switching is done by a skeleton, which works globally on all rows, all processors, except two, have nothing to do. This is, however, *no* overhead, because no processor can continue before it gets the pivot row, so it would be idle anyway.

The next step is broadcasting the pivot row. Unlike row permutation, row broadcasting can be reduced to broadcasting blocks, if each block consists of exactly one row[11]. For this, the array `piv` is created with the size $p \times (n+1)$, each processor thus getting one row. Before broadcasting the pivot row, we have to get it from our array into `piv`. Because no single rows can be accessed, the copying is done by calling the skeleton `array_map`. The skeleton applies the function `copy_pivot` to all elements of the array `piv`, overwriting only those elements that correspond to the pivot row. The function `copy_pivot` is defined as follows.

```
$t copy_pivot (array <$t> a, int k, $t v, Index ix) {
   Bounds bds = array_block_bounds (a) ;

   if (bds->lowerBd[1] <= k  && k < bds->upperBd[1])
      return (array_get_elem (a, {k, ix[1]}) /
                array_get_elem (a, {k, k}));
   else
      return (v) ;
}
```

The function `copy_pivot` checks, whether the current block of the array `a` contains the pivot row, using the skeleton `array_block_bounds`. If this is true, then it returns the corresponding element, already normalized relative to the pivot element, i.e., a_{kj}/a_{kk}, otherwise the old value is returned. The skeleton `array_map` overwrites then the corresponding element of the array `piv`.

On checking the arity of this function against its call in the procedure `gauss`, we can observe that `copy_pivot` was *partially applied* to the array `a` and the row number `k` in the procedure `gauss`. The remaining two arguments, the array element `v` and its index `ix`, are supplied by the skeleton itself. Notice that we have addressed here the array elements by their global coordinates (i.e., using the macro `array_get_elem`), because this addressing is more simple than local addressing, and in this case it does not have a negative influence on the overall performance.

[10] The reason is that if this value is 0, then the matrix is singular.

[11] This method does *not* work for row permutation, in case two rows placed on the same processor are switched. Such clash cases would have to be handled by the application program, which should have access to single rows within the same block.

Now, the pivot row can finally be broadcast, and each processor can do the actual elimination on the rows of its block. The elimination is done by mapping the function `eliminate` to all elements of `a` and writing the results into the copy array `b`. The function gets its first three arguments, namely the number of the pivot row, the pivot array, and the old array by partial application. The elimination is done only for elements at the right of the pivot element, except for those in the pivot row itself, by computing $a_{ij} := a_{ij} - a_{ik} \cdot (a_{kj}/a_{kk})$ as shown next (`procId` is the number of the current processor, and is equal to the number of the row of the array `piv` mapped to this processor).

```
$t eliminate (int k, array <$t> a, array <$t> piv, $t v, Index ix) {
    if (ix[0] == k  ||  ix[1] < k)
        return (v) ;
    else
        return (v - array_get_elem (a, {ix[0], k}) *
                    array_get_elem (piv, {procId, ix[1]})) ;
}
```

Finally, because the pivot elements were not normalized to 1, each element of the rightmost column (i.e., of the result x) has to be divided by the pivot (i.e., diagonal) element of that row. The normalization is done by applying the function `normalize` to the elements of the array.

Run-time Results:

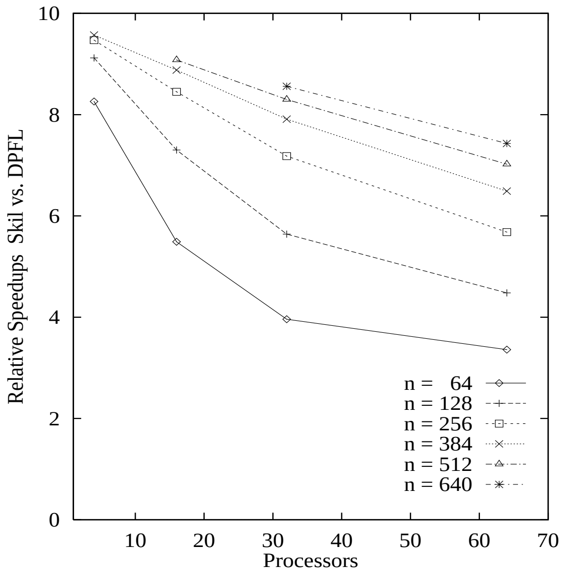
p \ n	64	128	256	384	512	640
4	**1.54**	**10.56**	**79.56**	**263.94**	–	–
	8.26	9.12	9.47	9.57		
	1.79	*1.79*	*1.83*	*1.85*		
16	**0.80**	**3.79**	**23.60**	**73.69**	**168.41**	**321.79**
	5.49	7.30	8.45	8.88	9.08	-
	1.38	*1.35*	*1.49*	*1.58*	*1.64*	*1.69*
32	**0.83**	**3.02**	**15.20**	**43.71**	**95.83**	**178.41**
	3.96	5.64	7.18	7.91	8.30	8.56
	1.22	*1.07*	*1.14*	*1.24*	*1.31*	*1.38*
64	**0.88**	**2.74**	**11.43**	**29.69**	**61.32**	**109.47**
	3.36	4.48	5.68	6.49	7.02	7.43
	1.07	*0.88*	*0.87*	*0.92*	*0.98*	*1.05*

Table 4.3: Run-time results for Gauss–Jordan elimination [**bold**: absolute times (in seconds); roman font: relative speedups DPFL vs. *Skil*; *italics*: relative slow-downs *Skil* vs. Parix-C]

The Gauss–Jordan elimination program was run for matrices with size $n \times n$, for different values of n between 64 and 640, on 4 to 64 processors. The first version of `gauss` was implemented *without* partial pivoting, i.e., corresponding to the initial version of the algorithm given in the beginning of this section. The reason was that this version had been implemented in DPFL (Kuchen 1995), and we wanted to make a fair comparison. Table 4.3 shows the absolute run times of the *Skil* program in seconds, its speedups relative to the DPFL implementation, as well as its slow-downs relative to Parix-C.

As for the previous examples, we have plotted the speedups relative to DPFL and the slow-downs relative to Parix-C against the number of processors, and obtained the graphics in Figure 4.4. In the left graphic, most of the speedups relative to DPFL are grouped around the factor 8, whereas only a few go below 6. Again, these values correspond to the case when arrays with small sizes are distributed onto large networks, yielding small blocks, and thus leading to a higher weighting of the communication overhead.

The slow-downs relative to C are between 1 and 1.85, in few cases even slightly less than 1, the average of 1.34 being comparable to the one obtained for matrix multiplication. On the whole, the results relative to message-passing C are good, in particular for large networks very good.



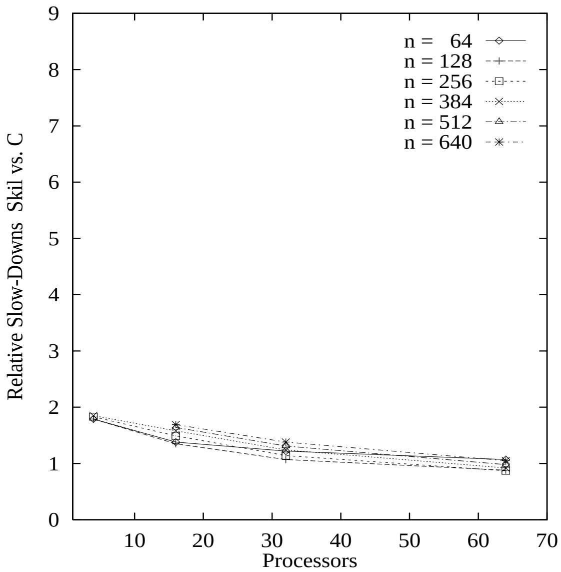


Figure 4.4: Comparison: *Skil* versus DPFL (left) and *Skil* versus Parix-C (right) for Gauss–Jordan elimination

The second version of `gauss` we tested was the one with partial pivoting. The run times were here about twice as long as in the first version, which is satisfactory, because it is visible from the description of the implementation of the pivot search and exchange, that this operation brings considerable communication overhead.

4.4.4 A Monte Carlo Method

Problem and Algorithm Description:

The final application we consider is a statistical ('Monte Carlo') numerical method for solving a partial differential equation (PDE). The idea is to find a random variable θ, such that its expectation $E\theta$ coincides with the solution of the problem. Having found such a θ, we need to generate n realizations of it $(\theta_1, \dots, \theta_n)$, and obtain the value of the solution as their arithmetic mean $E\theta \approx \frac{1}{n}\sum_{i=1}^{n}\theta_i$. An important advantage of this method is that the realizations θ_i are independent from one another, and can thus be computed in parallel with no communication in between. Communication is needed only in the final phase, for collecting the single results.

We consider the following partial differential equation:

$$\frac{\partial^2 u(x,y)}{\partial x^2} + \frac{\partial^2 u(x,y)}{\partial y^2} = 0$$

defined on the domain $[0,1] \times [0,1]$ with the boundary conditions

$$\begin{array}{lll} u(0,y) = y & \text{and} \quad u(1,y) = 1+y, & \forall y \in [0,1] \\ u(x,0) = x & \text{and} \quad u(x,1) = x+1, & \forall x \in [0,1] \end{array}$$

The domain is discretized to a rectangular mesh with step h and the PDE is discretized by finite differences to

$$u(x,y) = \frac{u(x+h,y) + u(x-h,y) + u(x,y+h) + u(x,y-h)}{4}$$

If (x_0, y_0) is the point where we want to know the solution of the PDE, then we must find a random variable θ, such that $u(x_0, y_0) = E\theta$. For that, we construct a random trajectory from our initial point to the boundary of the domain using the following Markov process:

$(x, y) := (x_0, y_0)$
while (x, y) is not on the boundary of the domain **do**
  $(x, y) := (x', y')$
  where (x', y') is selected from the set of neighboring points
    $\{(x-h, y), (x+h, y), (x, y-h), (x, y+h)\}$
    with the same probability $p = 1/4$

Let (x^*, y^*) be the final point on the boundary. Then, our random variable is $\theta := u(x^*, y^*)$. It can be proved that $u(x_0, y_0) = E\theta$, which validates the result. The algorithm is described in detail by Kuchen and colleagues (1995).

Implementation with Skeletons:

The *Skil* implementation of this Monte Carlo algorithm is simple, compared to its Parix-C counterpart presented by (Kuchen et al. 1995). Each of the p processors computes n/p random trajectories and places the results in its elements of a 1-dimensional distributed array of total size n. The trajectories are computed during the initialization phase of the array by the argument function `rand_traj`. The additional parameters needed by `rand_traj` (the initial point (x_0, y_0), the discretization step h, and the number of trajectories n) are supplied by partial application. After the random trajectories have been computed and the results placed in the distributed array, the single results must be collected and combined to yield the actual result $E\theta$. The gathering is done by the skeleton `array_fold`, which computes the sum of all elements of the array. Finally, the arithmetic mean is computed by dividing the result of the *fold* operation by n. The procedure is implemented as follows.

```
double monte_carlo (array <double> a, double x0, double y0, double h,
                    int n) {

    double res ;

    a = array_create (1, {n,0}, {0,0}, {-1,-1},
                      rand_traj (x0, y0, h, n/netSize)) ;
    res = array_fold (ident, (+), a) / n ;

    array_destroy (a) ;
    return (res) ;
}
```

Run-time Results:

We have run this application for different values of the discretization step h, and different numbers of trajectories n, on 3 to 63 processors. The starting point was in the middle of the domain, i.e., $(x_0, y_0) = (0.5, 0.5)$. Because the trajectories were generated randomly, each result was calculated as the average of at least 6 measurements. The absolute run times, together with the speedups relative to the DPFL implementation and the slow-downs relative to Parix-C, are given in Table 4.4.

h	0.05			0.02			0.005		
p \ n	1000	5000	10000	1000	5000	10000	1000	5000	10000
3	**0.94**	**4.58**	**9.12**	**5.19**	**26.06**	**51.65**	**85.22**	**414.33**	**826.82**
	5.09	4.41	5.29	6.16	5.22	6.14	5.57	4.90	5.73
	1.09	*1.06*	*1.10*	*0.91*	*0.92*	*0.93*	*0.99*	*0.97*	*1.98*
7	**0.44**	**1.99**	**3.98**	**2.33**	**11.29**	**22.48**	**36.65**	**180.86**	**357.64**
	5.05	4.73	5.65	6.14	5.38	6.27	5.58	4.86	5.72
	1.16	*1.10*	*1.11*	*0.90*	*0.92*	*0.91*	*0.94*	*0.97*	*0.98*
15	**0.23**	**0.98**	**1.88**	**1.17**	**5.39**	**10.61**	**18.15**	**86.77**	**169.19**
	5.61	5.36	6.61	6.27	5.71	6.81	5.39	4.81	5.76
	1.21	*1.10*	*1.08*	*0.92*	*0.91*	*0.92*	*0.92*	*0.97*	*0.97*
31	**0.15**	**0.52**	**0.98**	**0.65**	**2.73**	**5.22**	**9.72**	**42.68**	**84.66**
	5.00	5.27	6.61	6.26	6.05	7.46	5.05	4.74	5.62
	1.50	*1.21*	*1.14*	*1.00*	*0.92*	*0.90*	*0.96*	*0.96*	*0.99*
63	**0.10**	**0.29**	**0.51**	**0.38**	**1.47**	**2.68**	**4.91**	**22.31**	**42.27**
	7.40	6.69	7.29	6.03	6.39	6.70	5.33	4.76	4.88
	1.67	*1.26*	*1.16*	*1.00*	*0.94*	*0.91*	*0.82*	*0.97*	*0.95*

Table 4.4: Run-time results for the Monte Carlo algorithm [**bold**: absolute times (in seconds); roman font: relative speedups DPFL versus *Skil*; *italics*: relative slow-downs *Skil* versus ParixC]

We have again plotted the quotients DPFL versus *Skil* and *Skil* versus Parix-C and obtained the curves depicted in Figure 4.5. Again, the *Skil* implementation is faster than the DPFL one, with most of the relative speedups grouped around 6 (left graphic). On the other hand, the performance of *Skil* is comparable to that of C, most of the relative slow-downs being clustered around 1 (right graphic).

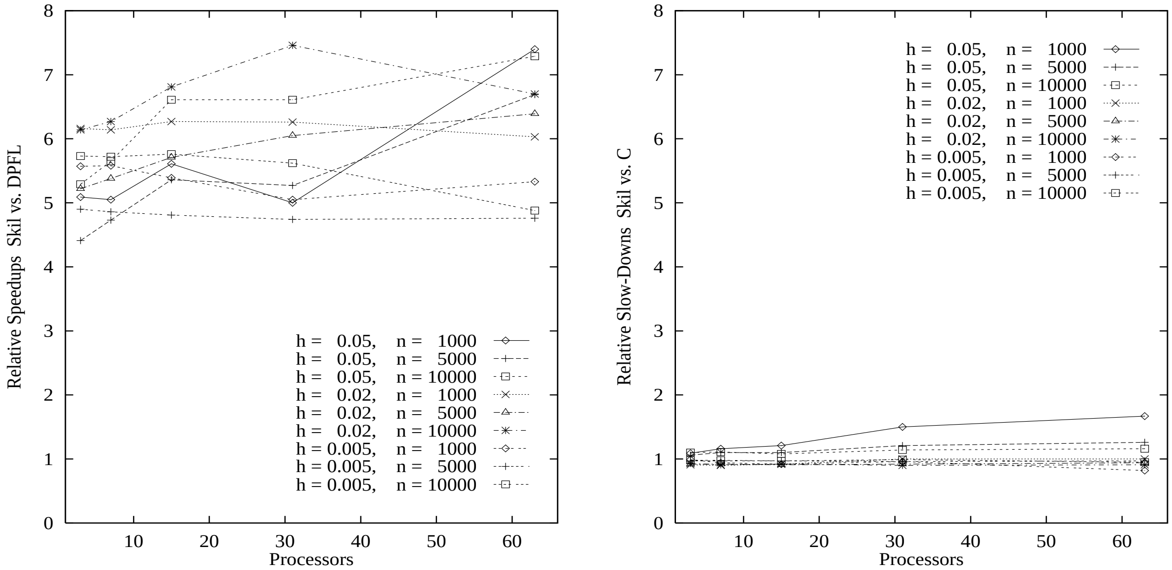


Figure 4.5: Comparison: *Skil* versus DPFL (left) and *Skil* versus Parix-C (right) for the Monte Carlo algorithm

4.4.5 Summary of the Results

The averages of the results of all applications that we have presented, are summarized in Figure 4.6. Hereby, `matrix mult1` is the implementation of the algorithm of Cannon presented in Section 4.4.1, whereas `matrix mult2` is the implementation of the same algorithm based on the skeleton `array_gen_mult`.

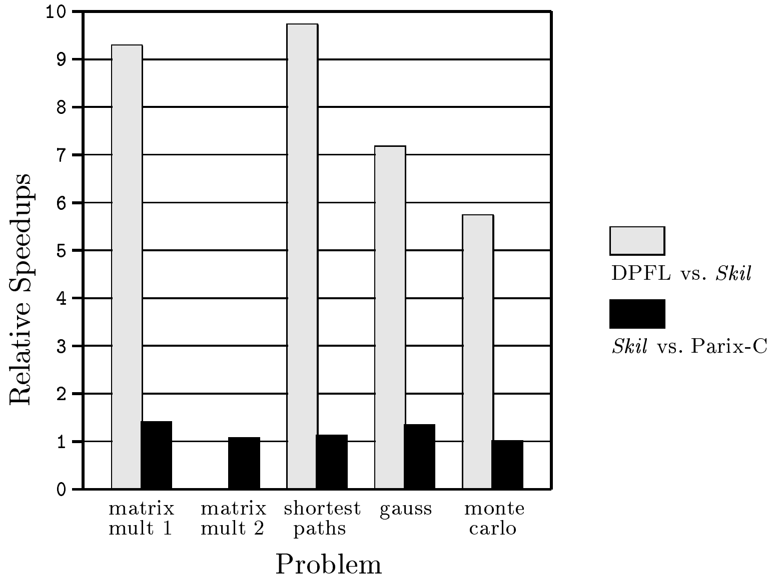


Figure 4.6: Averages of *Skil* results versus results of DPFL and Parix-C

The relative speedups of *Skil* versus DPFL are between 5.73 and 9.73, whereas the remaining slow-downs to C are relatively small, between 3% (for `monte carlo`) and 41% (for `matrix mult1`). On the whole, this overhead is a small price to pay for the considerable rise in programming convenience and safety, as well as in portability of programs offered by the skeletal approach.

The explanation of the quality of our results lies mainly in the compile-time instantiation technique employed for the elimination of the functional constructs, but also in the use of an imperative language, as well as in the decentralized computation model, described in the beginning of this chapter. All these features have a positive influence on the efficiency of the language.

The importance of these results lies in showing, that algorithmic skeletons can lead to efficient implementations, and thus become interesting for real-world applications.

Chapter 5

Multigrid Applications

The amount of computational work should be proportional to the amount of real physical changes in the computed system.

ACHI BRANDT

This chapter addresses the use of skeletons to solve real-world problems in parallel. We choose as application area a family of nontrivial numerical methods, namely *multigrid algorithms.* We study the data access and communication patterns occurring in these methods, and derive from them a set of skeletons, which can be used as the building blocks for most multigrid algorithms. The skeleton-based implementation of multigrid is illustrated by means of two applications. We then concentrate on a basic multigrid component, the relaxation, and describe how several relaxation procedures can be implemented in our framework. The results presented in the concluding section show that our implementations achieve remarkably good speedups.

After having presented in the previous chapter four simple applications that employ algorithmic skeletons, we now turn to a class of real-world applications, namely multigrid methods (Briggs 1987). There are several reasons for choosing these methods, both from the numerical point of view and from the point of view of the parallel implementation.

First, multigrid methods are the fastest known solvers for discretized (elliptic) partial differential equations. Moreover, a special subclass, *full multigrid* methods, has been proved to be (asymptotically) optimal, in the sense that the computational work required to achieve a given accuracy is proportional to the number of discrete unknowns. A short overview of the complexities of different iterative solvers is given next, whereby N denotes the number of unknowns.

Method	Complexity
Gaussian Elimination	$\mathcal{O}(N^3)$
Successive Over-Relaxation (SOR)	$\mathcal{O}(N^{\frac{3}{2}} \log N)$
Multigrid (MG)	$\mathcal{O}(N \log N)$
Full Multigrid (FMG)	$\mathcal{O}(N)$

Second, multigrid methods are suited for parallelization, in the sense that their parallel implementation usually leads to good speedups. However, parallelizing a multigrid method is a nontrivial task, especially if that method works with adaptive grid refinement (Bastian 1994).

Third, multigrid is an adequate application for algorithmic skeletons. On the one hand, it is a nontrivial real-world application. On the other hand, there is no single multigrid algorithm, but an entire class of such algorithms, which, however, build onto the same basic blocks (with respect to the computational structure), thus fitting naturally in the skeletal concept. Moreover, multigrid skeletons can be used in the implementation of other classes of hierarchically structured algorithms, such as N-body methods (Wu et al. 1997).

5.1 Introduction to Multigrid Methods

Multigrid methods are general and efficient solvers for (systems of) partial differential equations (PDEs). According to Stüben and Trottenberg (1984), these methods are based on three main ideas:

1. Error smoothing by relaxation
2. (Recursive) Corrections on coarser grids
3. Nested iteration

Although these ideas have been known or used separately for a long time, they needed to be combined first to yield the efficient multigrid methods. In this section, we outline these ideas and also present their aggregation into multigrid. Throughout our exposition, for more clarity, we restrict ourselves to the two-dimensional case.

5.1.1 Basic Notions

Let $\Omega \subset \mathbb{R}^2$ be a domain, $\partial\Omega$ be its border, and let f and g be two functions, defined as follows:

$$\begin{aligned} f &: \quad \Omega \to \mathbb{R} \\ g &: \quad \partial\Omega \to \mathbb{R} \end{aligned}$$

Assume further that L is a linear[1] elliptic differential operator on Ω. An elliptic boundary value problem with Dirichlet boundary conditions is defined by

$$\begin{aligned} Lu &= f \quad \text{on } \Omega \\ u &= g \quad \text{on } \partial\Omega \end{aligned}$$

[1] In Section 5.1.5, we lift this restriction, allowing L to be nonlinear, as well.

Hereby, $u = u(x, y)$ is the solution of the problem. As an example, consider the Poisson equation on the unit square, for the given functions f and g:

$$
\begin{aligned}
Lu = -\Delta u = -\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) &= f \quad \text{on } \Omega = (0,1) \times (0,1) \\
u &= g \quad \text{on } \partial\Omega
\end{aligned}
$$

To solve this problem numerically on a computer, the former has to be discretized first. Several discretization techniques can be employed, such as finite elements (Johnson 1987), finite volumes (Bastian 1994; De Keyser and Roose 1992), or finite differences (Joppich and Mijalković 1993; Stüben and Trottenberg 1984). We have chosen here the latter, because of its simplicity and appropriateness for our model problems.

The discretization by finite differences is done as follows. Let $h = (h_x, h_y)$ be a two-dimensional vector with $h_x, h_y > 0$. Then, the corresponding infinite rectangular two-dimensional grid $\mathcal{G}_h$ is defined as:

$$
\mathcal{G}_h \;=\; \{(x, y) \mid x = ih_x,\ y = jh_y,\ i, j \in \mathbb{Z}\}
$$

For a square grid, $h_x = h_y$, and the scalar quantity h is used instead. The discrete problem is defined on the domain $G_h := \Omega \cap \mathcal{G}_h$ and has the form:

$$
\begin{aligned}
L_h u_h &= f_h \quad \text{on } G_h \\
u_h &= g_h \quad \text{on } \partial G_h
\end{aligned}
$$

whereby the grid functions f_h and g_h are restrictions of f and g to G_h, respectively, and L_h is a discrete difference operator, which we shall describe by means of an example. For the Poisson equation, the discrete counterpart of the Laplace operator Δ is $L_h = \Delta_h$. At a given grid node (x, y), Δ_h is defined by approximating the partial derivatives by differences between values at neighboring grid nodes:

$$
\left(\frac{\partial u}{\partial x}\right)_{(x,y)} \;\approx\; \frac{u(x+h, y) - u(x, y)}{h}
$$

and, for second order derivatives:

$$
\left(\frac{\partial^2 u}{\partial x^2}\right)_{(x,y)} \;\approx\; \frac{u(x+h, y) - 2\,u(x, y) + u(x-h, y)}{h^2}
$$

Thus, the differential operator Δ is discretized at a given node as follows:

$$
\Delta \;\approx\; \Delta_h \;:=\; \frac{u(x+h, y) + u(x, y+h) - 4\,u(x, y) + u(x-h, y) + u(x, y-h)}{h^2}
$$

Instead of this rather complicated expression, we employ the following simpler and more suggestive notation:

$$
\Delta_h \;:=\; \frac{1}{h^2} \begin{bmatrix} & 1 & \\ 1 & -4 & 1 \\ & 1 & \end{bmatrix}
$$

This matrix, called *stencil* or *difference star* reflects the operation performed on a local neighborhood of the grid, whereby the central element corresponds to the current grid node and the missing stencil elements are zero. Thus, the meaning of applying the preceding stencil to a node of the grid is: *multiply the value of the current node with* -4, *then add to it the values of its northern, southern, eastern and western neighbors, and divide the result by* h^2, i.e., exactly the previous expression for Δ_h.

Depending on the differential operator L and its discretization, a stencil can have a higher or lesser number of elements, which may depend also on the current grid node. Frequently used stencils include the 9- and 25-point stars (Joppich and Mijalković 1993; Stüben and Trottenberg 1984).

Of course, it is possible to aggregate the equations bound to the single grid nodes into a large linear system and work with the system matrix. As this matrix is usually sparse, special efficiency-enhancing techniques may be employed thereby. However, we prefer the explicit use of grids and stencils, because they allow a *local* view of the operations, which fits into the concept of algorithmic skeletons of reducing the description of an entire system to local descriptions of its elements (which can then be passed, for example, as argument functions to skeletons).

Having set up the discrete problem, the corresponding linear system has to be solved. However, as the number of grid nodes and, hence, the number of equations is usually huge, obtaining the exact solution u_h is in general extremely expensive. Therefore, approximate solutions are computed instead. If $w_h^{(n)}$ denotes the approximate solution after n iterations, then the *algebraic error* $\tilde{e}_h^{(n)}$ is defined as

$$\tilde{e}_h^{(n)} \stackrel{\text{def}}{=} u_h - w_h^{(n)}$$

In general, the approximation $w_h^{(n)}$ does not satisfy the discrete equation. The deviation is called *residual* and is defined as

$$r_h^{(n)} \stackrel{\text{def}}{=} f_h - L_h w_h^{(n)}$$

Rearranging the first definition and inserting it into the second one yields

$$r_h^{(n)} = f_h - L_h \left(u_h - \tilde{e}_h^{(n)}\right) \stackrel{\text{lin}}{=} f_h - L_h u_h + L_h \tilde{e}_h^{(n)}$$

After reduction, we obtain

$$r_h^{(n)} = L_h \tilde{e}_h^{(n)}$$

which is called the *residual equation* and which is equivalent to the initial discrete equation. Thus, solving a given discrete equation can be reduced to solving its residual equation. This fact is used in the coarse grid correction, as shown in Section 5.1.3.

We want to emphasize that, in the derivation of the residual equation, the linearity of the operator L_h is essential. If this operator is not linear, then the residual equation is no longer valid. Performing the coarse grid correction for nonlinear operators is addressed in Section 5.1.5.

5.1.2 The Relaxation

The approximative solution of the discretized equation is computed by an *iterative method*, i.e., by starting with an initial guess of the solution and successively improving this approximation, until a certain convergence criterion is fulfilled. In each iteration, a procedure is applied to all elements of the current approximation of the solution, yielding the elements of the next approximation. The application of an iterative solver to a linear system is called *relaxation*.

There are several reasons for preferring iterative methods over their direct counterparts. On the one hand, as mentioned before, the number of linear equations is usually huge, an exact solution being prohibitively expensive. Moreover, as shown in the beginning of this chapter, iterative methods, such as SOR, are considerably faster than exact methods, such as Gaussian elimination. On the other hand, the discrete solution is anyway an approximation of the continuous solution of the initial problem, as it contains the discretization error. Therefore, if we succeed in making the algebraic error $\tilde{e}_h^{(n)}$ less than, or at least comparable to, the discretization error τ_h, then the overall result is as accurate as if an exact solver had been employed for the discrete system.

The main drawback of relaxation methods is their unsatisfactory convergence. More specifically, the convergence behavior is depicted in the left graphic in Figure 5.1, which shows the dependency of the norm of the error, $||e||$, on the number of iterations. It is visible from this graphic that relaxation is highly effective in the beginning, where it reduces the error considerably in few iterations. However, this effectiveness soon disappears, large numbers of iterations bringing only minimal improvements of the error.

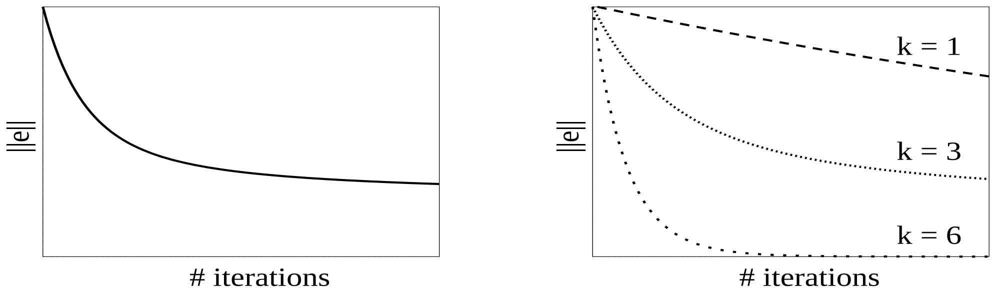


Figure 5.1: Convergence of relaxation

The explanation of this behavior can be found by means of Fourier analysis, i.e., by decomposing the error in its spectral components (also called frequencies or modes), and studying the convergence for each component. An example of three different frequencies is depicted in the right graphic in Figure 5.1. It is visible here that the higher frequencies (e.g., the one with the wave number $k = 6$) are damped fast, whereas the lower frequencies (e.g., for $k = 1$) are reduced extremely slow by relaxation. This is why relaxation methods are also called *smoothers*.

5.1.3 Coarse Grid Correction

Relaxation methods do not only converge slowly, but their convergence factor, given by the spectral radius ρ of the discrete operator L_h, depends on the discretization step h. This dependency has the form $\rho \in 1- \mathcal{O}(h^2)$ for the Jacobi and Gauss–Seidel relaxation, and $\rho \in 1- \mathcal{O}(h)$ for SOR (Joppich and Mijalković 1993). Thus, using a coarser grid G_H with $H > h$ would improve the convergence of the iterative solver. However, the finer grid cannot be discarded, because its mesh size h has been dictated by accuracy requirements imposed on the discretization. Instead, a combination of the two grids is employed, as shown next.

The idea is to find a link between the fine-grid problem and the coarse-grid problem, more exactly, between the residual equations on the two grids, $L_h\tilde{e}_h = r_h$ and $L_H\tilde{e}_H = r_H$. The coupling of the two problems is achieved by means of two linear operators: the *prolongation* operator I_h^H, which maps fine grid functions to coarse grid functions (i.e., from G_h to G_H), and the *restriction* operator I_H^h, which maps functions from G_H to G_h.

Assuming now that the number of nodes of G_H is much smaller than that of G_h, the coarse grid equation $L_H\tilde{e}_H = r_H$ can be solved much faster than its counterpart, even if, say, a direct method were used. This idea leads to the following algorithm (Joppich and Mijalković 1993), called the *coarse grid correction (CGC)*:

1. Residual computation $\quad r_h^{(n)} := f_h - L_h w_h^{(n-1)}$
2. Residual restriction $\quad r_H^{(n)} := I_h^H r_h^{(n)}$
3. Exact solution on the coarse grid $\quad L_H\tilde{e}_H = r_H^{(n)}$
4. Correction transfer $\quad \tilde{e}_h^{(n)} := I_H^h \tilde{e}_H^{(n)}$
5. Correction $\quad w_h^{(n)} := w_h^{(n-1)} + \tilde{e}_h^{(n)}$

Notice that we have used in this algorithm the residual equation, rather than the actual discrete equation. Although the two are equivalent, as shown in Section 5.1.1, the former has to be used, because the error and the residual can be transferred safely between grids. The reason is that these quantities are smooth, whereas for instance the right hand side of the initial equation, f, might not be. Transferring a quantity with high frequency modes to a coarser grid may lead to problems such as

- *Aliasing*, i.e., mapping two Fourier modes on the fine grid to the same mode on the coarse grid
- The *disappearance* of modes, meaning mapping a mode to the constant 0 function

On the other hand, the prolongation of a nonsmooth quantity to a finer grid may lead to high interpolation errors. Thus, only smooth quantities should be transferred between grids. The coarse grid correction is sketched in Figure 5.2.

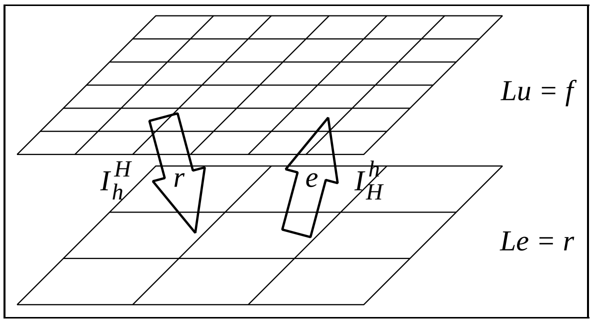


Figure 5.2: Coarse grid correction

To perform the coarse grid correction, we have to ensure that the residual and the error are smooth. This smoothing is done by relaxation, which, as we have shown in the previous section, has the smoothing property. Combining relaxation with coarse grid correction yields the *correction scheme (CS)*, also called *two-grid method* (Joppich and Mijalković 1993). The algorithm starts with ν_1 relaxations that eliminate the high frequency modes, followed by a coarse grid correction, and, optionally, by ν_2 post-smoothing steps[2], which reduce the high frequencies that might have been introduced by the prolongation. The correction scheme is sketched next, whereby the steps 1 – 5 remain unchanged, like in the coarse grid correction:

0.	Pre-smoothing	$\overline{w}_h^{(n)} := relax^{\nu_1}\,(w_h^{(n-1)}, L_h, f_h)$
1.	Residual computation	$r_h^{(n)} := f_h - L_h \overline{w}_h^{(n)}$
$\vdots$		$\vdots$
5.	Correction	$\tilde{w}_h^{(n)} := \overline{w}_h^{(n)} + \tilde{e}_h^{(n)}$
6.	Post-smoothing	$w_h^{(n)} := relax^{\nu_2}\,(\tilde{w}_h^{(n)}, L_h, f_h)$

5.1.4 The Multigrid Cycle

The question arising now is, what if the number of grid nodes on the coarser grid is still too large for a direct solution? The answer is to solve the residual equation on the coarse grid by the same procedure, i.e., to apply the correction scheme recursively for a certain number of levels, until a direct solution is possible[3]. The correction cycle thus obtained is called *multigrid cycle*. Depending on the number of recursive calls at each level—usually denoted by γ—different multigrid cycles result. The most commonly used ones are the V cycle (for $\gamma = 1$) and the W cycle (for $\gamma = 2$). The two cycles are depicted in Figure 5.3.

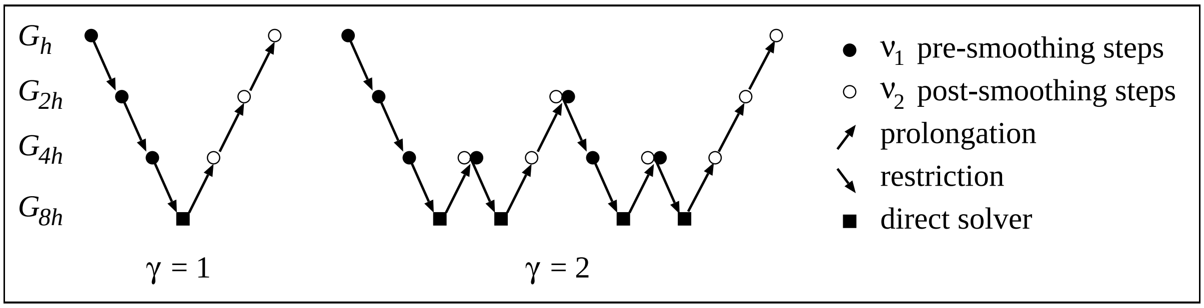


Figure 5.3: The V and W multigrid cycles

The idea behind multigrid cycles can be described from another angle as follows. The algorithm starts on the finest level, by performing a few relaxations that dampen the modes with the highest frequencies. Then, the error and residual, which have only smooth, i.e., lower frequency components, are transferred to the next coarser grid. Here, because of the coarser metric, some of these modes appear as having a higher frequency than on the fine grid. Thus, these frequencies can again be attenuated by relaxation (on the coarser grid). In the next step, the modes with the highest frequencies among the remaining modes are dampened on the next coarser grid, etc. The final remaining modes are eliminated by the exact solver on the coarsest grid.

[2] ν_1 and ν_2 are usually small, e.g., 1 or 2.

[3] Alternatively, an iterative solution on the coarse level could also be envisaged. This solution is more advantageous in the case of parallel implementations than a direct one; it is the solution we have adopted in our applications presented in Section 5.5.

5.1.5 The Full Approximation Scheme

The idea in classic multigrid methods is to have the approximate solution on the finest grid and corrections on all coarser grids. However, this idea uses the linearity of the differential operator, thus restricting the applicability of the method. Moreover, in certain applications it is desirable to have complete solution approximations on the coarser grids, too, instead of merely corrections.

The *full approximation scheme (FAS)* achieves these two goals. The idea is here to restrict not only the residual, but also the current solution approximation $w_h^{(n)}$ from finer to coarser grids. Using the equality $u_h = w_h + \tilde{e}_h$, the following equation holds on the fine grid:

$$r_h = f_h - L_h w_h = L_h\left(w_h + \tilde{e}_h\right) - L_h w_h$$

which is valid for both linear and nonlinear operators L_h (Joppich and Mijalković 1993). Transferring this equation to a coarse grid, we obtain:

$$I_h^H r_h = L_H\left(\hat{I}_h^H w_h + \tilde{e}_H\right) - L_H \hat{I}_h^H w_h$$

or, after introducing the coarse grid variable $u_H := \hat{I}_h^H w_h + \tilde{e}_H$:

$$L_H u_H = L_H \hat{I}_h^H w_h + I_h^H r_h$$

whereby the restriction operator applied to the solution, $\hat{I}_h^H$, is not necessarily equal to the restriction operator applied to the residual, I_h^H. FAS allows to solve approximations of the initial equation on the coarse grids, instead of residual equations. The algorithm is given in the following for a two-grid scheme; a multigrid scheme is obtained out of it by recursive nesting (Joppich and Mijalković 1993). The difference between FAS and the correction scheme from Section 5.1.3 lies in the restriction of the solution approximation $\overline{w}_h^{(n)}$ and in the computation of the correction, which are performed in steps 3 and 4, respectively:

0. Pre-smoothing	$\overline{w}_h^{(n)} := relax^{\nu_1}\left(w_h^{(n-1)}, L_h, f_h\right)$
1. Residual computation	$r_h^{(n)} := f_h - L_h \overline{w}_h^{(n)}$
2. Residual restriction	$r_H^{(n)} := I_h^H r_h^{(n)}$
3. Exact solution on coarse grid	$\boldsymbol{L_H u_H^{(n)} = L_H \hat{I}_h^H \overline{w}_h^{(n)} + r_H^{(n)}}$
4. Correction computation	$\boldsymbol{\tilde{e}_H^{(n)} := u_H^{(n)} - \hat{I}_H^h \tilde{e}_H^{(n)}}$
5. Correction transfer	$\tilde{e}_h^{(n)} := I_H^h \tilde{e}_H^{(n)}$
6. Correction	$\widetilde{w}_h^{(n)} := \overline{w}_h^{(n)} + \tilde{e}_h^{(n)}$
7. Post-smoothing	$w_h^{(n)} := relax^{\nu_2}\left(\widetilde{w}_h^{(n)}, L_h, f_h\right)$

It is important that the quantities transferred between grids are smooth. Thus, the prolongation is not done directly, e.g., by interpolating u_H, because this procedure could lead to interpolation errors. Instead, it is done via the error, i.e., by computing e_H on the coarse grid (step 4), interpolating it to e_h on the finer grid (step 5), and making the correction there (step 6).

Apart of allowing to handle nonlinear operators, FAS is useful in multigrid methods with adaptive grid refinement (Joppich and Mijalković 1993). Here, the grids are refined *locally*, depending on the norm of the error. To apply the residual equation, composite grids have to be used. These grids are nonhomogeneous and therefore difficult to handle. On the other hand, FAS makes it possible to work separately on sub-domains where the grids are homogeneous, with appropriate boundary conditions, thus simplifying the operations performed on the grids. Further details can be found in (Hackbusch and Trottenberg 1982; Joppich and Mijalković 1993; Stüben and Trottenberg 1984).

5.1.6 Full Multigrid

As we have seen, the correction schemes—both CS and FAS—are applied iteratively to solve a problem. The question arising, is, what should be the initial approximation used in this iterative process. Often, the initial guess is the null function, but this guess may not always be suitable. On the other hand, a good initial guess could bring a significant reduction of the number of iterations, and the point is here to obtain the start approximation cheaply, i.e., at a considerably lesser cost than that of the actual iterations.

The idea hereby is to use a grid hierarchy to compute good initial approximations. This process is called *nested iteration* and starts on the coarsest grid by solving the discrete equation. The solution (which may be approximative) is interpolated to the next finer grid, where it serves as initial guess, and so on, up to the finest grid.

Combining the nested iteration with multigrid correction cycles yields the *full multigrid (FMG)* method. FMG starts by solving the problem exactly on the coarsest grid, then interpolates the solution to the next finer grid. Here, it performs a multigrid correction cycle. The solution obtained eventually is interpolated to the next finer grid, where a new correction cycle takes place, and so on. An FMG algorithm with V correction cycles is depicted in Figure 5.4.

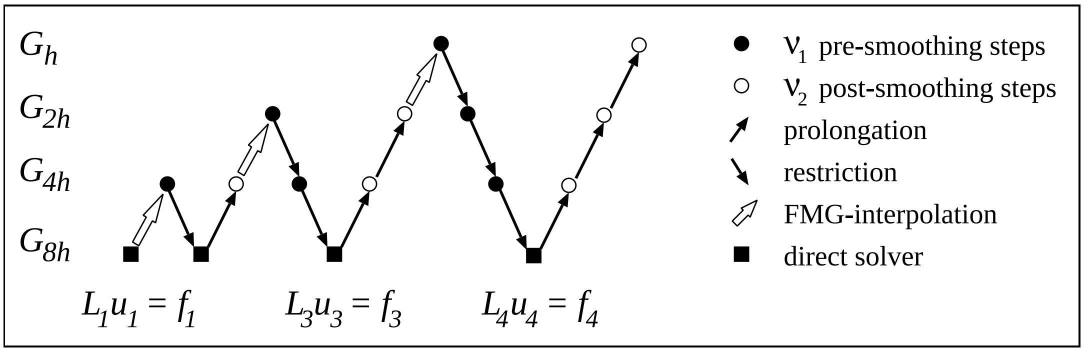


Figure 5.4: FMG with V correction cycles

The importance of full multigrid methods lies in their optimal complexity of $\mathcal{O}(N)$, where N is the number of nodes on the finest grid. The reason is that, to achieve an accuracy of the solution comparable to the truncation error, only $\mathcal{O}(1)$ iterations are necessary, an iteration having the worst-case complexity $\mathcal{O}(N)$ (Joppich and Mijalković 1993). As a comparison, the number of multigrid correction cycles—used as iterative solver—needed to reach the same accuracy is $\mathcal{O}\log(N)$, leading to an overall complexity of this method of $\mathcal{O}(N \log N)$ (Joppich and Mijalković 1993). Thus, as stated at the beginning of this chapter, full multigrid methods are the fastest known algorithms for solving (systems of) discretized partial differential equations.

5.2 The Representation of the Multigrid

The first problem that we consider in the design of skeletons for multigrid, is the internal representation of the grid hierarchy. For that, we shall describe the domains of the differential equations, the grids that are being employed, and their distribution onto the processors, including locality and data overlapping. Finally, we present the data structure `MGrid` and its parameterization by user-defined data.

5.2.1 The Domain and its Discretization

The partial differential equations we consider are defined on two-dimensional domains $\Omega \subset I\!R^2$. For simplicity, we have adopted the restriction employed by Stüben (1982), namely that the domain Ω is defined as

$$\Omega \;=\; \{\, (x,y) \in I\!R^2 \;|\; a \leq x \leq b, \quad f_{low}\,(x) \leq y \leq f_{hi}\,(x) \,\}$$

i.e., as the set of the points in the plane situated between two—not necessarily continuous—functions, f_{low} and f_{hi}, defined on the interval $[a, b]$. Thus, a domain is described by four parameters: the real values a and b, and the functions $f_{low}, f_{hi} : [a, b] \rightarrow I\!R$. This restriction serves only to simplify the implementation and can be lifted in principle, allowing the use of domains described by arbitrary curves in the plane, respectively surfaces in three dimensions.

The grids we use in the discretization are *structured* (rectangular) ones, with the same mesh size h in both dimensions. This property is again no restriction as a matter of principle, but serves only to simplify the implementation. Moreover, the type of the grid depends also on the discretization technique employed, for instance finite differences usually lead to structured grids, whereas finite elements yield unstructured grids, e.g., triangulations (Bastian 1994). On the other hand, as several authors show, the use of structured or—more generally—block-structured grids is sufficient for a wide variety of problems, including irregular and time-dependent problems (Brandt 1977; Hackbusch and Trottenberg 1982; Joppich and Mijalković 1993; Linden et al. 1992; Stüben 1982).

5.2.2 The Distribution of the Grids

Once the problem has been discretized, the next step is its parallelization. The standard approach is 'horizontal' parallelization, in which each grid level is distributed onto the processors. As a consequence, both operations on single levels, such as relaxation, and operations on pairs of neighboring levels, such as prolongation and restriction, are performed in parallel. As the grids are structured and uniform, parent and children nodes on neighboring grid levels are placed on the same processor[4], as shown in Figure 5.5. This property holds also for the nodes on the borders of the sub-domains placed on different processors, due to the overlap areas described next.

We have partitioned each level of the grid into *stripes* and mapped each stripe to one processor. The stripes can be vertical, horizontal, or diagonal. This one-dimensional partitioning has several advantages:

[4]This 'vertical locality' condition is, however, no longer trivially fulfilled in the presence of adaptive refinement (Bastian 1994; Joppich and Mijalković 1993).

- It leads to a simpler implementation.
- It allows the optimization of the communication for different relaxation methods, e.g., diagonal 'slicing' is adequate for Gauss–Seidel or SOR relaxation, where the data dependencies are wavefront-like (see Section 5.3.5).
- It reduces the number of messages needed to update the overlap areas so as to keep the data consistent.
- The processors can be organized as a pipe virtual topology, which can be embedded in most hardware topologies with dilation 1 (Röttger et al. 1994), thus optimizing the communication.

On the other hand, according to Kumar and colleagues (1994), stripe partitioning has the drawback of being less efficient and less scalable than block-checkerboard partitioning. However, we found the scalability to be in practice more than satisfying: we have obtained, for the applications we have considered, efficiency values between 92 percent on 64 processors and 70 percent on 256 processors (Section 5.5). The only difficulty we have encountered with the one-dimensional partitioning, is that the coarsest grid 'thins out'—i.e., some processors become idle—already for $\mathcal{O}(p^2)$ nodes, whereas for two-dimensional partitioning, this threshold is at $\mathcal{O}(p)$ nodes. The consequence is a limitation of the depth of the grid hierarchy, as shown in Section 5.5.

To reduce the amount of communication, *overlap areas* are defined. They represent extensions of the sub-domains placed on each processor, that contain copies of data items placed on the neighboring processors, as depicted in Figure 5.5. They serve to improve the efficiency of reading accesses to the neighborhoods of data items on the borders of sub-domains. Consider for instance a 2-dimensional structured grid. If an interpolation operator uses the values at the 4 adjacent grid nodes, then an overlap area of depth 1 is sufficient to allow performing this operation fully parallel, with no additional communication.

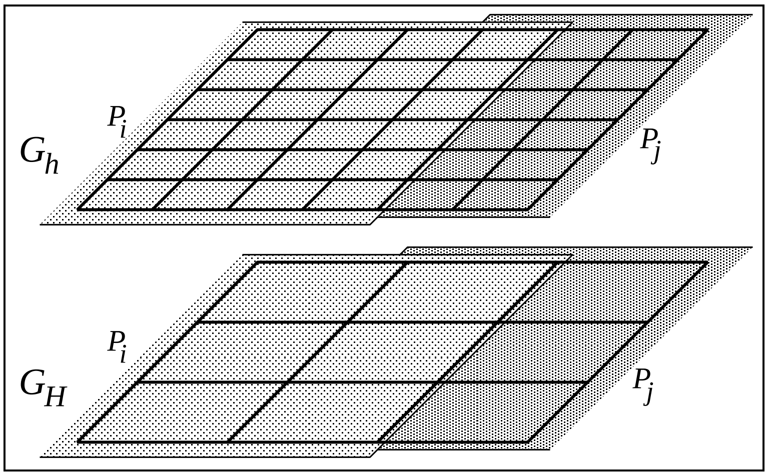


Figure 5.5: Grid distribution and overlap areas

If the original data values on a processor are altered by a skeleton, then this skeleton must also update the copies in the overlap areas of the other processors, to keep the data consistent. However, this update involves only one coordinated communication between neighboring processors, which is considerably cheaper than passing an extra message each time a datum is needed from another processor during computation. Moreover, if the data are only read-accessed, like for instance the data on the source grid in an inter-grid transfer, no update is necessary.

5.2.3 Data Structures

Concluding this section, we describe the data structures employed in our implementation. The distributed data structure `MGrid` is declared using the `pardata` construct:

```
pardata MGrid <$t> implem ;
```

where `$t` stands for the type of the user-defined data structure 'plugged in' at each node and *implem* is the data structure located on each processor, implemented as follows.

On each processor, a data structure is created, comprising the domain (characterized by the four parameters described in Section 5.2.1), the number of levels, the depth of the overlap areas, the slicing (vertical, horizontal, or diagonal), as well as the actual grid hierarchy. The latter consists of a sequence of levels, from which one is always the current level. A grid level is characterized by the discretization step h and by the total number of nodes, and consists of an array of lines. These lines are, depending on the slicing, columns, rows, or diagonals. A line may consist of one or more continuous sets of grid nodes, the *strings*.[5] Each node is identified by its Cartesian coordinates. At each node, the user can 'plug in' his own data structure, which is typically a record containing the approximation of the solution, the right hand side, the error, and the residual (see Section 5.5). The entire data structure is depicted in Figure 5.6.

This representation of the multigrid based on strings of nodes is compatible to the data structures used to represent *locally refined grids*, as described by Brandt (1982), or Joppich and Mijalković (1993). Our data structure may thus be used with no modification in the implementation of *adaptive* multigrid algorithms, too.

As the inner structures of the data structure `MGrid` are not visible to the user of the skeletons, we have provided two additional data structures that are accessible: `Line` represents a line of the grid, however, compactified as an array of node records, each record consisting only of the coordinates and the user data of that node. `Coords` is a record containing the x and y coordinates of a node. These two data structures are used in the skeletons presented in Section 5.4.

5.3 Data Dependencies and Access Patterns

In this section, we analyze the basic operations used in multigrid: grid refinement and coarsening, data prolongation and restriction, relaxation, and *forall* operations. For relaxation, we consider different algorithms, which lead to different data dependencies. Based on these data dependencies, we derive in the Section 5.4 the skeletons for multigrid.

[5]As an example, consider the domain delimited by the x axis and the parabola $y = x^2$ on an interval containing the origin. In case of horizontal slicing, each line comprises two strings, one in each half of the plane.

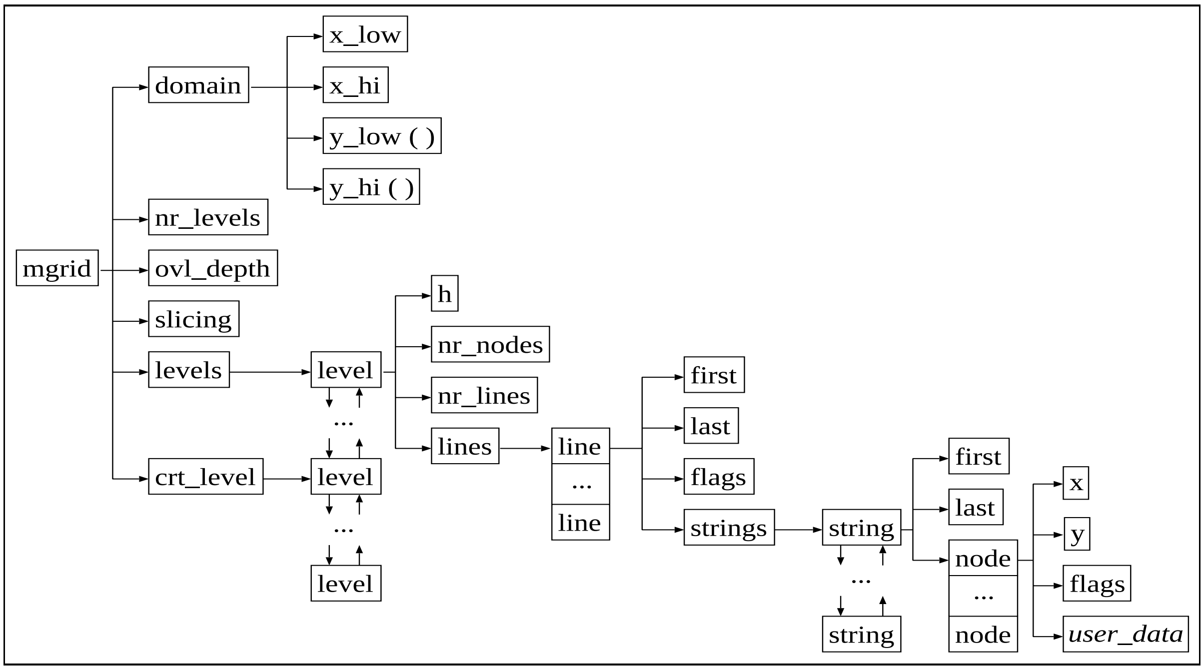


Figure 5.6: The internal representation of the multigrid

We are interested in a *local view* of each operation, because this view leads to a straightforward derivation of adequate skeletons: the local operators are passed as functional arguments to the skeletons, which work on an entire distributed grid (Botorog 1997). Thus, the tasks of implementing a solution to a problem (a numerical one in our case) and that of handling parallelism are decoupled, the application programmer having to deal only with the first one.

5.3.1 Grid Refinement

The first operation we consider, is generating a new grid by refining an existing grid. We employ a local point of view and look first at how a single node of the new grid is created. There are two possibilities: either several nodes of the old grid are used (for instance, the coordinates of the new node can be computed as the average of the coordinates of two or more parent nodes), or only one node of the old grid is used (in this case, the coordinates of the new node could be obtained, for instance, by adding and subtracting $h/3$ to the coordinates of the parent node, respectively, where h is the mesh size of the coarser grid). The two variants of refinement are sketched in Figure 5.7, where the nodes depicted in bold are the newly created ones. That both the old nodes and the new ones are hollow, suggests that *no* data are transferred between the grids, but only the new nodes—which can be viewed as containers for the user data—are constructed.

After all nodes of the new grid have been generated, the links between the new nodes, as well as the links between parents and children, have to be built. As with the new nodes, the newly created links are depicted in bold in Figure 5.7. In case of structured, rectangular grids, these links do not have to be constructed explicitly, because, given the regularity of the grid, it is clear from the coordinates of a node, which are its neighbors, and where its parents and children are placed. However, in case of unstructured grids,

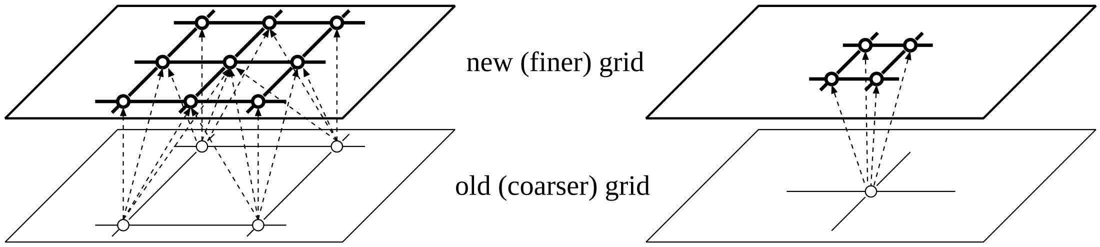


Figure 5.7: Grid refinement

like for instance triangulations (Bastian 1994), these links have to be stored and used explicitly.

Finally, as the new grid is also distributed, data along the borders of the sub-domains placed on neighboring processors must be replicated to create the overlap areas.

It is visible from this description, that generating a new grid requires special operations, therefore, a dedicated skeleton *create_level* is necessary at this point. To cover both refinement variants, the skeleton is applied *cluster-wise* to the nodes of the old level, i.e., to a small environment (neighborhood) of nodes at a time (see Section 5.4.3), whereas the variant of a single parent is treated as a special case in this framework.

5.3.2 Grid Coarsening

As mentioned in Section 5.1, coarser grids are usually viewed as correction grids. Hence, given a grid on which we want to solve a problem, we need a coarsening operation to yield the corresponding correction grids. Grid coarsening can be viewed as the dual of grid refinement, therefore its analysis is similar to the one in the previous subsection.

As with refinement, there are here also two possibilities regarding the creation of a new node: either it has only one parent (i.e., a selection of the old nodes has taken place), or it has several parents (the coordinates of the new node being derived e.g., as an average of the coordinates of the parents). The two variants are sketched in Figure 5.8, where we have kept the bold notation for new nodes and links, and that of hollow nodes, suggesting that no user data are transferred between the grids.

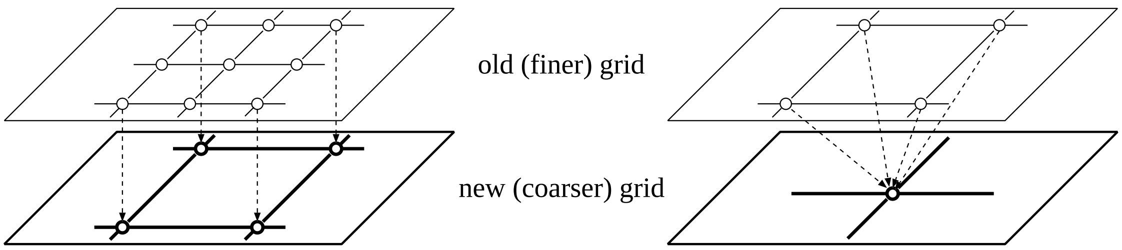


Figure 5.8: Grid coarsening

As the operations involved by grid coarsening are similar to the ones used in refinement (creation of new nodes, links, and overlap domains), the same dedicated skeleton *create_level* (Section 5.4.2) will be employed for this multigrid primitive, too.

5.3.3 Data Prolongation

Data prolongation is the transfer of data from a coarser to a finer grid. From a local point of view, prolongation generally uses the data at several parent nodes to compute the data at one child node, usually by an interpolation procedure. This operation is depicted in Figure 5.9, where on the one hand, the thin lines used for the links indicate that the grid structure does not change, and on the other hand, the full nodes represent the transfer of data at these nodes. Hereby, the gray nodes are the source nodes, which are read-accessed, whereas the black nodes are the targets, where the new data are written to.

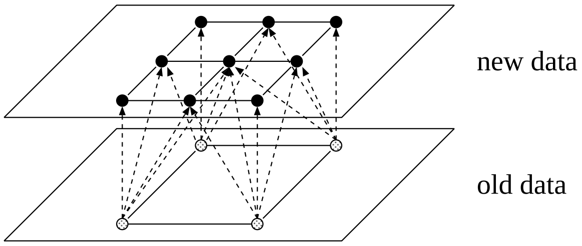


Figure 5.9: Data prolongation

Unlike grid refinement, the functionality of data prolongation is simpler, so that a more general, map-like skeleton, applied to all nodes of the new or of the old level can be used here. However, because interpolation requires data from more than one node at a time, we need an enhanced *map*, which has access to a group (*environment*) of nodes at one time. As shown in Section 5.4.3, this operation is the *mapenv* skeleton. Still, maps are bound to single levels of the grid. To achieve the inter-grid functionality, this skeleton has to be combined with special *inter-grid* operations, which compute for each node the set of its parents or children, respectively. The inter-grid operations are described in Section 5.4.5.

5.3.4 Data Restriction

Data restriction is the counterpart of data prolongation, i.e., it is the transfer of data from a finer to a coarser grid. However, unlike prolongation, restriction can be performed both point-wise (i.e., by injection) and cluster-wise (for instance, by weighting) (Hackbusch and Trottenberg 1982; Joppich and Mijalković 1993). The two variants are outlined in Figure 5.10.

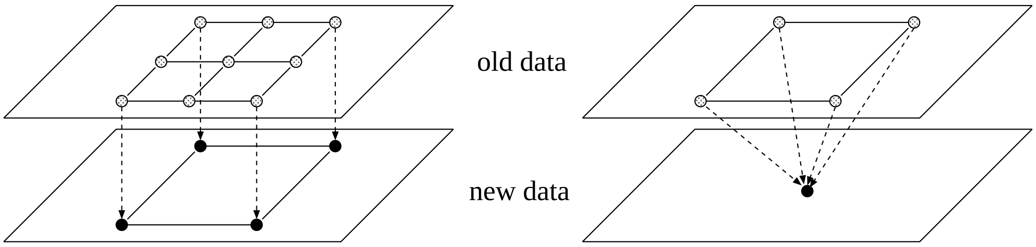


Figure 5.10: Data restriction

Similarly to prolongation, we need here the *mapenv* skeleton for the weighted restriction (right picture), however in case of injection (left picture), a classical *map* is sufficient. Moreover, in both cases, the additional use of inter-grid operations (Subsection 5.4.5) is necessary.

5.3.5 Relaxation

Relaxation is one of the basic components of multigrid and amounts to a number of iterations of a solver, such as Jacobi or Gauss–Seidel, performed on the nodes of a grid. Employing again the local view, we use *stencils* (see Section 5.1.1) to describe the data dependencies. A stencil prescribes how the value of its central element has to be altered, depending on the old value of that element and on the values of the neighboring nodes. For instance, the stencil

$$\frac{1}{h^2}\begin{bmatrix} & 1 & \\ 1 & -4 & 1 \\ & 1 & \end{bmatrix}$$

which represents the 5-point discretized Laplace operator, denotes

$$\forall i,j \qquad u_{i,j} := \frac{1}{h^2}\left(-u_{i-1,j} - u_{i,j-1} + 4u_{i,j} - u_{i+1,j} - u_{i,j+1}\right)$$

The data dependencies (for one stencil application) are depicted in Figure 5.11.

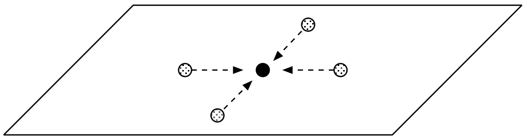


Figure 5.11: Relaxation

From this figure, it is clear that what we need here is again the *mapenv* skeleton. However, a map-like skeleton processes the nodes in a nondeterministic order, because of their distribution among the processors. This property causes a problem if the nodes have to be processed in a certain order, as shown next.

Consider first the case of the Jacobi relaxation, where in each step, the old values of the neighbors, i.e., the values from the previous iteration, are used (see Figure 5.12). For the discretized Poisson equation (see Section 5.1.1), the Jacobi relaxation method is described by the following formula:

$$u_{i,j}^{(n+1)} := \frac{1}{4}\left(u_{i-1,j}^{(n)} + u_{i,j-1}^{(n)} + u_{i+1,j}^{(n)} + u_{i,j+1}^{(n)} - h^2 f_{i,j}\right)$$

Hereby, the superscripts denote the number of the iteration. It is visible, that the new values can be computed independently in any order, in particular in parallel. This property poses no problems with respect to the implementation with the *mapenv* skeleton.

In the case of Gauss–Seidel relaxation, however, this property no longer holds. Here, the values of two of the neighbors have to be new, i.e., computed in the current iteration (Figure 5.13). For the Poisson equation, this relaxation method is described by the scheme:

$$u_{i,j}^{(n+1)} := \frac{1}{4}\left(u_{i-1,j}^{(n+1)} + u_{i,j-1}^{(n+1)} + u_{i+1,j}^{(n)} + u_{i,j+1}^{(n)} - h^2 f_{i,j}\right)$$

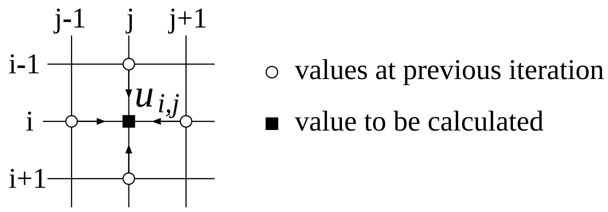


Figure 5.12: Jacobi relaxation

Hence, the nodes are processed in a wavefront-like manner, i.e., in a special order. The *mapenv* skeleton thus has to provide a means to specify the order in which the nodes must be processed. This mechanism is presented in Section 5.4.3.

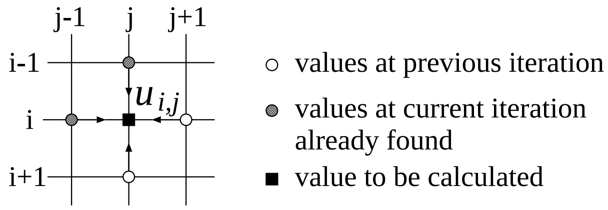


Figure 5.13: Gauss–Seidel relaxation

The relaxation methods mentioned so far were point-wise, in the sense that in each step, the value of a single node was updated. It is possible, however, to update the values of a group of nodes at a time by using a direct solver. For instance, in the line successive over-relaxation (LSOR) method, an entire line of points is updated simultaneously (Figure 5.14). For the Poisson example, we have the following system of equations:

$$u_{i-1,j}^{(n+1)} - 4u_{i,j}^{(n+1)} + u_{i,j-1}^{(n+1)} := h^2 f_{i,j} - u_{i+1,j}^{(n)} - u_{i,j+1}^{(n)}$$

whereby the variables on the left hand side are the unknowns. Thus, a tridiagonal system has to be solved for each column j (Modi 1990).

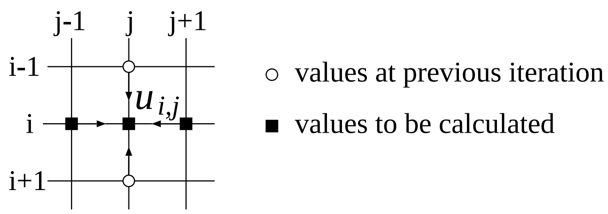


Figure 5.14: LSOR relaxation

From the skeletal point of view, handling such cases requires a special map-like skeleton, in which more than one node can be written to in a single step. This skeleton is *mapline*, presented in Section 5.4.3.

5.3.6 *forall* Operations

Finally, we consider procedures that execute a point-wise operation independently on all nodes of a grid level. As an example, consider the correction, which consists of adding the error to the current approximation of the solution: $u := u + e$. This contracted notation stands for $\forall i, j \;\; u_{i,j} := u_{i,j} + e_{i,j}$. As all point-wise operations can be executed independently in parallel, the *forall* procedure can be implemented straightforwardly using a *map* skeleton, as shown in Section 5.5.1.

5.4 The Multigrid Skeletons

Based on the data access patterns presented in the previous section, we now derive the skeletons for multigrid algorithms. These skeletons are constructors and destructors of a grid hierarchy, operations of single grid levels, *map*-like, and *fold* operations. Finally, we define inter-grid operations, which are not skeletons, but are bound to single grid nodes, parallelism being achieved by using them inside *map* skeletons.

For each operation, its syntax and informal semantics are given. In contrast to the presentation of the array skeletons in Section 4.3, we have not included here the description of the implementation or the complexity. The reason was that both the underlying data structure `MGrid` and the skeletons are considerably more complicated than their array counterparts, therefore they would have required far more extensive descriptions.

5.4.1 The Constructor and Destructor of a Multigrid

`mg_create` constructs a new grid hierarchy, which consists of one grid level. It gets as arguments the four parameters that describe the domain (`xlow`, `xhi`, `ylow`, and `yhi`), the initial discretization step `h0`, the depth of the overlap areas `ovl_depth`, the user-defined initialization function `init_elem`, and the type of grid partitioning used (`slicing`).

Each processor creates the lines of nodes that make up its stripe of the grid (see Section 5.2.3), as well as the extra lines that represent the overlap areas. The number of overlap lines is given by the parameter `ovl_depth`. The lines are vertical, horizontal, or diagonal, as indicated by the parameter `slicing`. In the next step, the nodes of this level are generated and grouped into strings, based on the description of the domain and the initial discretization step. After that, the user data at each node are initialized by the function `init_elem`. The syntax of the skeleton `mg_create` is:

```
MGrid <$t> mg_create (double xlow, double xhi, double (*ylow) (double),
                       double (*yhi) (double), double h0, int ovl_depth,
                       $t init_elem (Coords), int slicing) ;
```

For an efficient implementation of multigrid algorithms, the depth of the overlap areas (`ovl_depth`) should be greater than or equal to the maximum of all environment (neighborhood) depths in subsequent operations (stencils in relaxations, interpolations, restrictions, etc.). Usually, `ovl_depth` should be chosen at least 1.

The counterpart of `mg_create`, the skeleton `mg_destroy`, removes an existing grid hierarchy. It gets as argument the function `destroy_elem`, which is called for every node of every grid level, deallocating the user data structure placed at that node. After that, mg_destroy removes all strings, lines, and grid levels, as well as all auxiliary data structures of the multigrid.

```
void mg_destroy (MGrid <$t> mg, void destroy_elem ($t)) ;
```

5.4.2 Constructors and Destructors of Single Grid Levels

mg_create_level is a constructor that adds a new grid level to a grid hierarchy. If the current level is the finest one, and if the discretization step h passed as argument to the skeleton is smaller than the current mesh size, then the current level is refined. Otherwise, if the current level is the coarsest one, and if h is larger than the current mesh size, then the current level is coarsened. In the other cases, the skeleton reports an error. In addition to creating the new level, this skeleton makes it the current level of the grid hierarchy.

The functionality of this skeleton is similar to that of mg_create, however, restricted to the creation of one grid level. The argument function init_elem can be different for each level, whereas the parameters of the domain, the depth of the overlaps, and the type of slicing are common to the entire grid hierarchy. The syntax of this skeleton is:

```
void mg_create_level (MGrid <$t> mg, double h, $t init_elem (Coords)) ;
```

mg_remove_level is a destructor of single grid levels, needed for instance in solving time-dependent problems, where the grid hierarchy must be partially destroyed and then reconstructed to adapt to the evolution of the problem (Joppich and Mijalković 1993). This skeleton works similarly to mg_destroy, however, on a single level. The argument function destr_elem is used to remove the user data at each node.

```
void mg_remove_level (MGrid <$t> mg, void destroy_elem ($t)) ;
```

The operation mg_set_crt_level sets the current level of a multigrid. It gets as argument the number of levels by which the pointer to the current level should be moved (i.e., the level difference). If this number is positive, then the pointer is moved toward finer grid levels, otherwise it is moved toward coarser levels. The skeleton returns the number of levels actually skipped.

```
int mg_set_crt_level (MGrid <$t> mg, int level_diff) ;
```

5.4.3 *map*-like Skeletons

Map-like skeletons are used to perform the bulk of the computations on the multigrid, namely relaxations, *forall* operations, and data transfers (prolongation, restriction). They are applied to all nodes of one grid level, whereas inter-grid data transfers are achieved by using them in conjunction with inter-grid operations (Section 5.4.5). There are three kinds of maps: operating on single nodes (*map*), single nodes together with their immediate neighbors (*mapenv*), and lines of nodes (*mapline*), respectively.

The first skeleton, `mg_map`, applies its argument function to each node of the current grid independently, i.e., the argument function `map_f` can read and write the user data only at the current node. This skeleton is used for instance to add the error to the current approximation of the solution or—together with the inter-grid operations—to transfer data to a coarser grid (as illustrated in the example in Section 5.5.1).

```
void mg_map (MGrid <$t> mg, $t map_f ($t, Coords), Bool upd_ovl) ;
```

The second skeleton, `mg_mapenv`, also updates data at single nodes, but the argument function `mapenv_f` has also reading access to the neighboring nodes (environment) of the current node, up to the specified depth `env_depth`. This skeleton is used for the relaxations and for inter-grid data transfer (Section 5.5.1).

```
void mg_mapenv (MGrid <$t> mg, int env_depth, long dep_f (Coords),
                $t mapenv_f ($t *, Coords *), Bool upd_ovl) ;
```

Two problems arise in the use of this skeleton. The first one occurs in the case of relaxation. As pointed out in Section 5.3.5, the *mapenv* skeleton has to provide a means to specify the order in which the nodes must be processed. This goal is achieved by using a further argument function, `dep_f`, which computes for each node a bit-flags template corresponding to the environment of this node. Hereby, a set bit denotes that the *new* value at that node must be used. Using this local information, the skeleton sets up a dependency graph and processes the nodes as indicated by its edges. Examples of how different relaxation methods can be specified, are given in Section 5.5.1. Note that, because the user data placed at each node can be arbitrarily complicated and changes may occur anywhere inside them, it is hard to employ data dependency analysis techniques to determine the order of processing automatically.

The second problem is whether or not the overlapped data have to be updated. In a relaxation, they have to be updated at the end of each iteration, because the originals have changed. In an interpolation, however, this update is not necessary, because the data on the old grid are not altered, but only used to compute the data on the new grid. The endmost argument, `upd_ovl`, indicates, whether an update of the overlaps has to be done at the end of the execution of the skeleton.

This handling of overlap updates is closely related to the BSP notion of *superstep*, where all computations are done first, followed by an overall coordinated communication, in which all necessary data exchanges take place (McColl 1996; Valiant 1990).

Finally, the third skeleton, `mg_mapline_env`, updates all nodes of a line at one time. Similarly to *mapenv*, it gets as argument an environment, which in this case consists of the neighboring lines of the current line, up to a given depth. The lines are passed to the argument function using the data structure `Line`, which contains all nodes of a grid line, in a compact form (see Section 5.2.3). Also similarly to *mapenv*, this skeleton gets the argument function `dep_f`, which describes the local dependencies among line updates. The syntax of the skeleton is:

```
void mg_mapline_env (MGrid <$t> mg, int env_depth, long dep_f (double),
                     Line <$t> map_f (Line <$t> *, double), Bool upd_ovl);
```

`mg_mapline_env` can be used for instance to compute line relaxations, as outlined in Section 5.5.1. With the help of the dependency function, both relaxation methods where no order of update is imposed, such as Line Jacobi, and methods where the order of update is given, such as Line Gauss–Seidel or Line SOR (Modi 1990), can be implemented.

5.4.4 The *fold* Skeleton

The skeleton `mg_fold` performs a *collect* operation on all nodes of the current level. Similarly to `array_fold` (Section 4.3.4), this skeleton performs a preliminary conversion of the data by calling the argument function `conv_f`. Then, the (possibly converted) data at all nodes are combined using the function `fold_f`, which must be associative and commutative, otherwise the result is unpredictable. The syntax of the skeleton `mg_fold` is:

```
$u mg_fold (MGrid <$t> mg, $u conv_f ($t, Coords), $u fold_f ($u, $u)) ;
```

5.4.5 Inter-Grid Operations

The final two operations, `mg_get_finer_corresp` and `mg_get_coarser_corresp`, are needed in the transfer of data between grid levels, i.e., for prolongation and restriction. They return, for a given node, the list of the nearest nodes on the next finer and coarser level, respectively, up to a given depth. It is worth noting, that these operations are *not* parallel, but bound to single grid nodes. Parallelism is achieved by using them inside a map-like skeleton, as outlined in Section 5.5.1.

```
void mg_get_finer_corresp (MGrid <$t> mg, Coords coords, int env_depth,
                           $t *finer_data, Coords *finer_coords) ;

void mg_get_coarser_corresp (MGrid<$t> mg, Coords coords, int env_depth,
                             $t *coarser_data, Coords *coarser_coords) ;
```

The skeletons defined in this section represent the basic operations for multigrid (Hackbusch and Trottenberg 1982), so that practically any multigrid algorithm can be implemented with them. Examples of skeleton-based implementations of two multigrid applications are given in the next section.

5.5 Multigrid Applications

We now turn to applications of multigrid methods and consider two programs, MG00 and MG01 (Stüben 1982; Stüben and Trottenberg 1984), originally developed and implemented sequentially at the German National Research Center for Information Technology (GMD). We show how these applications can be parallelized using the multigrid skeletons presented in the previous section. Moreover, we present several variants of implementations, that use different multigrid primitives. We give comprehensive run-time results, showing that the achieved speedups are remarkably good. Concluding this section, we discuss the results obtained and explain the high scalability of our skeleton-based implementation.

A common characteristic of the considered applications is the use of standard, i.e., nonadaptive, grid refinement and coarsening techniques. The reason for this restriction is that multigrid programs usually tend to be large and complex, especially in the case of parallel implementations, and we did not want to complicate the implementation further by using local grid refinements.

Adaptive grid refinement involves on the one hand enhancing existing skeletons, like for instance the constructor of a grid level, by further functional arguments, and on the other hand, supplying additional dedicated skeletons, for example for load balancing. We have outlined the implementation of a full multigrid algorithm with adaptive grid refinement (Botorog and Kuchen 1995).

5.5.1 MG00

The first application we consider is MG00. After describing the algorithm, we present its parallel implementation with skeletons. We then show, how several relaxation methods—Jacobi, Gauss–Seidel, SOR, Red–Black Jacobi, and Line Jacobi—can be implemented by means of skeletons in this framework. Detailed run-time results and speedups are given throughout this section.

Description of the Algorithm

MG00 is a simple multigrid algorithm, which solves the Poisson equation $-\Delta u = f$ on the unit square with Dirichlet boundary conditions, i.e., conditions on the solution u. It consists of a number of correction cycles used as iterative solver, i.e., the multigrid cycles are repeated, until the norm of the error drops below a given threshold. The procedure is classic multigrid (in contrast, for instance, to full multigrid) and uses the correction scheme CS. It employs the simplest form of correction cycles, namely V-cycles, as well as the simplest restriction operator, namely injection.

Skeleton-Based Implementation with Jacobi Relaxation

We first define the user data structure `Data` that is plugged in at each node, as a record containing the approximation of the solution `u`, the right hand side `f`, and the residual `r`. Moreover, as we employ Jacobi relaxation, the record is extended by an additional component, `u_new`, which is used to store the new values obtained by relaxation, until all nodes of a grid have been processed. The reason is that in Jacobi relaxation, only old values are used to compute the new values, so the field `u` should not be overwritten until the end of an iteration. The user data structure is defined as follows:

```
typedef struct _Data { double u, u_new, f, r; } *Data ;
```

The procedure `MG00`—given in the following—gets as arguments the depth of the multigrid hierarchy to be constructed (`nr_levels`), the initial discretization step (`h0`), the number of pre- and post-smoothing steps (`nu_1` and `nu_2`, respectively), the number of iterations on the coarsest level (`nic`), as well as the residual reduction factor used in the convergence criterion (`eps`).

MG00 starts by parameterizing the pardata `MGrid` by the user-defined data structure `Data`. Then, it creates the finest grid in the unit square (described by the end points 0 and 1 and by the constant functions `zero` and `one`), with an overlap of 1, and distributed vertically onto the processors (default). The argument function `init_elem` initializes the user data at each node as follows:

- The right hand side field `f` is set according to the partial differential equation to be solved ($-\Delta u = f$).
- The approximation of the solution `u` is set to the value given by the Dirichlet boundary conditions on the border of the domain, and to 0 at the interior nodes.
- The residual `r` and the auxiliary field `u_new` are set to 0.

After that, the `nr_levels` -1 correction grids are constructed. They are generated by standard coarsening, i.e., by doubling the mesh size in each step. As the skeleton `mg_create_level` makes the newly created grid level become the current level of the hierarchy (Section 5.4.2), after the hierarchy has been built, we have to go back to the finest level, to start the multigrid cycles. This step is done by the skeleton `mg_set_crt_level`.

The rest of the procedure is a loop, in which a multigrid V-cycle is performed by the procedure `CS` described next. The exit condition of the loop is that the norm of the current residual (`resid`) drops below the norm of the initial residual multiplied by the factor `eps`. The factor `resid` is computed by the *fold* skeleton, which first converts each residual to its absolute value, and then determines the maximum over all these results. Finally, the results are output and the grid hierarchy is deallocated.

```
void MG00 (int nr_levels, double h0, int nu_1, int nu_2, int nic,
           double eps) {
   MGrid <Data> mg ;  double h = h0 ;  double resid ;  int i ;

   mg = mg_create (0.0, 1.0, &zero, &one, h, 1, init_elem,
                   COLUMN_SLICES) ;          /* create finest level */

   for (i = 1 ; i < nr_levels ; i++) {
      h *= 2 ;                          /* create next coarser level */
      mg_create_level (mg, h, init_elem_zero) ;
   }

   mg_set_crt_level (mg, nr_levels) ;     /* go to the finest level */

   do {                                                /* V-cycle loop */
      CS (mg, nr_levels, nu_1, nu_2, nic) ;    /* correction scheme */
                                  /* compute maximum norm of residual */
      mg_mapenv (mg, 1, no_dep_f, compute_residual (1.0/h*h), False) ;
      resid = mg_fold (mg, get_residual_abs, max_val) ;

   }  while (resid < eps * initial residual) ;        /* stop condition */

   output results ;
   mg_destroy (mg, destroy_elem) ;
}
```

The procedure CS implements the multigrid correction cycle. On the coarsest level, the problem is solved iteratively, because this solution is simpler and more efficient to implement in parallel than an exact solution. On all other grids, the correction cycle, consisting of ν_1 relaxations, followed by a coarse grid correction, and then by ν_2 relaxations, is performed. Hereby, we use Jacobi relaxation, however, we show later in this section how it can be replaced by other smoothers.

Each pre- and post-smoothing iteration consists of two steps. First, the new value of the solution is computed at each node and stored into the auxiliary component `u_new`. This step is done by a call of the *mapenv* skeleton, because the differential operator Δ accesses both data at the current node and data at neighboring nodes. However, as we use Jacobi relaxation, no order is necessary in processing the nodes, therefore the null function `no_dep_f` can be passed as dependency function. Second, after the new values have been computed at all nodes, they have to be copied into the appropriate component of the user data structure. This step is achieved by calling the *map* skeleton with the argument function `update_u_values`, that assigns the value of `u_new` to `u`.

The transfer of data from a finer to a coarser grid (restriction), is done in two steps also. First, we move to the coarser grid, as this grid is the target of the assignments of user data. Then, we call the *map* skeleton with an argument function that computes the residual on the finer grid and injects it to the current grid, thereby making use of inter-grid operations. The argument function `inj_resid` is presented later in this section.

```
void CS (MGrid <$t> mg, int level_nr, int nu_1, int nu_2, int nic) {
   if (level_nr == 1)                                  /* coarsest grid */
      for (i = 0 ; i < nic ; i++) {              /* iterative solution */
         mg_mapenv (mg, 1, no_dep_f, jacobi, True) ;  /* relaxation */
         mg_map (mg, update_u_values, True) ;      /* update of 'u' */
      }
   else {
      for (i = 0 ; i < nu_1 ; i++) {                  /* pre-smoothing */
         mg_mapenv (mg, 1, no_dep_f, jacobi, True) ;
         mg_map (mg, update_u_values, True) ;
      }

      mg_set_crt_level (mg, -1) ;          /* move to coarser level */
      mg_map (mg, inj_resid (mg), True) ;        /* inject residual */

      CS (mg, level_nr-1, nu_1, nu_2, nic) ;  /* V correction cycle */

      mg_set_crt_level (mg, 1) ;             /* move to finer level */
                                 /* interpolate and make correction */
      mg_map (mg, interp_corr (mg), True) ;

      for (i = 0 ; i < nu_2 ; i++) {                 /* post-smoothing */
         mg_mapenv (mg, 1, no_dep_f, jacobi, True) ;
         mg_map (mg, update_u_values, True) ;
      }
   }
}
```

The prolongation of data implies moving to the finer (target) grid and calling the *map* skeleton with the argument function `interp_corr`. This function interpolates the error from the coarser to the finer grid and then performs the correction, i.e., it adds the error to the current approximation of the solution on the fine grid. Notice that there is only one recursive call of `CS` ($\gamma = 1$), hence the resulting correction cycle is a V-cycle.

As already mentioned, inter-grid data transfers are accomplished by using an inter-grid operation inside the argument function of a map-like skeleton (Section 5.4.5). We illustrate these transfers by presenting the implementation of the function `inj_resid`, which computes the residual on a fine grid and then injects it to the next coarser grid. This function is called by the *map* skeleton for all nodes of the coarse level and uses the operation `mg_get_finer_corresp` to determine, for each coarse-grid node, the nearest corresponding nodes on the fine grid. The coordinates of these nodes are returned in the array `finer_coords`, whereas the user data are stored in the array `finer_data`. Using this data, the function `mk_resid` first computes the residual $r := f - Lu$, and then injects it to the coarse grid (into the record `data`), as follows:

```
Data inj_resid (MGrid <$t> mg, Data data, Coords coords) {
    Data finer_data[...]  ;
    Coords finer_coords[...]  ;

    mg_get_finer_corresp (mg, coords, 1, finer_data, finer_coords) ;
    data.u = 0.0 ;
    data.f = mk_resid (finer_data, finer_coords) ;

    return (data) ;
}
```

Notice that the restricted residual becomes the right hand side `f` on the coarse grid. The reason is that, on the coarse grid, the *residual equation* $Le = r$ is solved, instead of the original equation $Lu = f$, as shown in Section 5.1.1. As the operator L is the same, the fields `u` and `f` are used on coarse grids to store the error and residual, respectively. The initial approximation `u`—actually representing the error `e`— is thereby set to 0.

Data prolongation is achieved similarly to data restriction, however using the operation `mg_get_coarser_corresp`, and transferring the solution `u` from the coarse grid to the error on the fine grid, where it is added to the current solution approximation.

Results

We have run the skeleton-based implementation of the program MG00 on a Parsytec CGel 1024 machine, with a peak performance of 4.4 GFLOPS. Each node of this machine consists of a T805 transputer with 4 MB of memory; the nodes are connected in a two-dimensional mesh topology.

The multigrid program was parameterized as follows: number of grid levels 3, number of pre- and post-smoothing relaxations $\nu_1 = \nu_2 = 2$, number of relaxations on the coarsest grid `nic` $= 3$, convergence factor `eps` $= 0.01$. The initial discretization step h_0 took different values between 1/32 and 1/1024, i.e., the number of grid nodes on the finest level varied from 10^3 to 10^6.

The results are summarized in Table 5.1. Notice that both the upper right corner and the lower left corner of the table are empty. The reason is in the first case memory overflow (the number of grid nodes is too large to fit into the memory of one or few processors), whereas in the second case, the coarsest grid is too thin, i.e., there are fewer grid lines than processors available, some processors thus becoming idle. Of course, the implementation could be enhanced to cope with this problem, by allowing that only some processors work on the coarsest grids, while the others are idle. This idea is described by Ritzdorf and Stüben (1993).

However, the main problem for the evaluation of the results lies in the lack of run times on one processor for larger problem sizes, which does not allow to compute all speedups directly. We have therefore estimated the speedups in these cases, by calculating lower bounds. This estimation was done based on the observation that, for a given network size, the speedups grow in the vast majority of cases with the growing problem size[6]. In terms of Table 5.1, this monotony means that the speedups usually grow within each row from left to right. Thus, we have used as estimate for a missing speedup the value next to it on the left in the same row of the table. However, this operation was done only for the topmost available result in each column. Based on this first approximative speedup, we have estimated the run time on one processor and used it to calculate estimate speedups for the remaining results in that column. This approximation is nearly always lower than or equal to the actual speedup value; thus, the real results are at least as good as, if not better than the estimated ones presented in the following.

p \ h_0	1/32	1/64	1/128	1/256	1/512	1/1024
1	43.03 **1.00**	188.68 **1.00**	1291.18 **1.00**	–	–	–
2	22.58 **1.91**	97.26 **1.94**	652.21 **1.98**	4024.72 ***1.98***	–	–
4	12.33 **3.49**	50.39 **3.74**	333.14 **3.88**	2032.39 ***3.87***	–	–
8	7.12 **6.04**	27.34 **6.90**	172.74 **7.47**	1035.11 ***7.70***	–	–
16	–	15.72 **12.00**	93.24 **13.85**	536.79 ***14.85***	2814.41 ***14.85***	–
32	–	–	53.04 **24.34**	287.85 ***27.68***	1459.06 ***28.64***	–
64	–	–	–	163.85 ***48.64***	780.92 ***53.52***	3491.92 ***53.52***
128	–	–	–	–	443.85 ***94.16***	1874.82 ***99.68***
256	–	–	–	–	–	1065.40 ***175.42***

Table 5.1: Results for MG00 with Jacobi relaxation on the GCel [roman font: absolute run times (in seconds); **bold**: actual speedups; ***bold italics***: estimated speedups]

[6]This phenomenon is known also as the *Amdahl effect*.

For more clarity, we have plotted the speedups versus the number of processors for the different problem sizes and obtained the curves in Figure 5.15. The graphics show a high scalability of the skeleton-based implementation of MG00, the efficiency on 256 processors being roughly 70 percent.

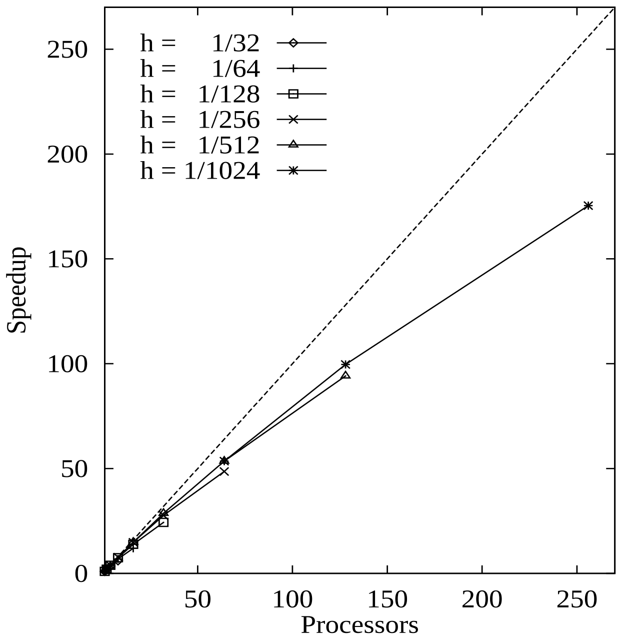


Figure 5.15: Speedups for MG00 with Jacobi relaxation on the GCel

We have implemented the multigrid skeletons—and, based on them, the multigrid applications—on a further parallel machine, the Parsytec GC/PP. This machine has a peak performance of 5.12 GFLOPS and consists of 32 nodes connected by a fat mesh network. Each node contains 2 Power PC 601 processors used for computations and 4 T805 transputers used for communication, as well as 64 MB of memory.

We have run the program MG00 with 4 grid levels, for mesh sizes between 1/64 and 1/1024, using up to 16 nodes of the machine. The rest of the parameters remained unchanged compared to the first implementation. The results are listed in Table 5.2.

We have again plotted the speedups and obtained the curves in Figure 5.16. Notice that they are not as regular as the ones yielded by the CGel implementation (Figure 5.15), having even parts where the speedups are superlinear. This phenomenon is mainly due to cache behavior (Kumar et al. 1994). Still, the obtained speedups are very good: the efficiency on 16 processors is—depending on the problem size—between 71% and 93%.

An interesting aspect is to compare the code sizes of the sequential and the skeleton-based implementation of MG00. The size of the original sequential implementation is a few hundred lines of code, and is comparable to the size of the *Skil* program (not taking into account the skeletons). This confirms one of the fundamental ideas of algorithmic skeletons, namely that a parallel implementation of a problem can be decomposed into two more or less independent components: the actual application program, and the handling of parallelism done inside the skeletons. Consequently, once the skeletons for a given class of problems have been implemented, the overhead of writing a parallel program for a given problem becomes comparable to the effort of writing a sequential program.

h_0 / p	1/64	1/128	1/256	1/512	1/1024
1	–	9.99 **1.00**	58.20 **1.00**	159.72 **1.00**	–
2	1.34 ***1.94***	5.16 **1.94**	29.55 **1.97**	80.57 **1.98**	617.27 ***1.98***
4	0.82 ***3.17***	2.97 **3.36**	16.65 **3.50**	59.45 **2.69**	341.79 ***3.58***
8	0.47 ***5.53***	1.51 **6.62**	5.52 **10.54**	30.95 **5.16**	161.46 ***7.70***
16	–	0.88 **11.35**	4.25 **13.69**	10.93 **14.61**	81.87 ***14.94***

Table 5.2: Results for MG00 with Jacobi relaxation on the GCPP [roman font: absolute run times (in seconds); **bold**: actual speedups; ***bold italics***: estimated speedups]

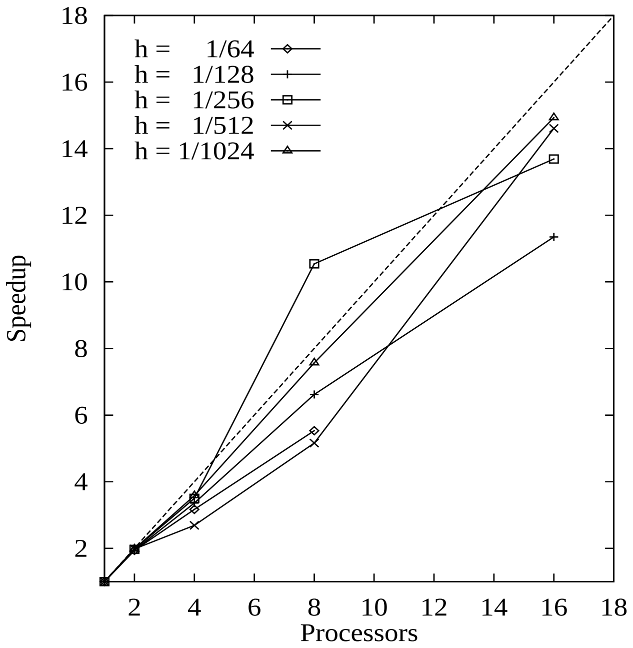


Figure 5.16: Speedups for MG00 with Jacobi relaxation on the GCPP

Implementation with Gauss–Seidel Relaxation

We shall now show, how the Jacobi smoother can be replaced by other relaxation methods in the implementation of MG00. We do not list the entire code of these variants, but highlight only the important aspects and differences to the program given in the beginning of this section. Thereby, all parameters of MG00 have the same values as for the GCel implementation with Jacobi relaxation.

Moreover, as the dependency function supplied as argument to the *mapenv* skeleton plays a central role in our exposition, we employ a special notation for it. Instead of writing the name of the argument function, we denote its return value in a stencil-like fashion, as a matrix of bit flags. The semantics of these bits is as follows: if the central bit is set, then the current node is processed, otherwise not. If any other bit is set, then the *new* data value at this node must be used (i.e., the value computed in the current iteration), otherwise the old value must be used (as described in Section 5.4.3).

We start with the Gauss–Seidel relaxation method. The first difference to the Jacobi-based implementation is that the component `u_new` is no longer needed in the user data structure, because the nodes are processed in a given order. Thus, the new values do not have to be buffered, but can be used as soon as they are available.

The second difference concerns the distribution of the grids onto the processors. For efficiency reasons, each level is divided into diagonal slices, reflecting the wavefront data dependencies occurring in this iterative solver (see Section 5.3.5). However—and this operation is a slight deviation from the standard Gauss–Seidel method—the slices are processed in parallel, i.e., a combination of simultaneous replacements (on the level of the slices) and single steps (inside each slice) results. Because of the diagonal distribution of the grid, the depth of the overlap domains must be 2 for a 9-point stencil, to avoid communication during the relaxation. Thus, the following call of the *create* skeleton is used:

```
mg_create (0.0, 1.0, &zero, &one, h, 2, init_elem, DIAGONAL_SLICES) ;
```

As the new values yielded by relaxation no longer have to be buffered, the approximation of the solution `u` does not have to be updated at the end of an iteration. Therefore, the pre- and post-smoothing steps have the simpler form shown next. Notice hereby that the bit field returned by the dependency function has the central bit set, i.e., each node is processed during an iteration, and that the flags corresponding to the northern and western neighbors are also set, reflecting the data dependencies of the Gauss–Seidel method (Section 5.3.5).

```
for (i = 0 ; i < nu ; i++)                                  /* smoothing */
```
$$\texttt{mg\_mapenv (mg, 1, } \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \texttt{, gauss\_seidel, True) ;}$$

The speedups obtained for the implementation of MG00 with Gauss–Seidel relaxation on the Parsytec GCel machine are given in Figure 5.17, for simplicity only in graphical form. Similarly to the previous results, the scalability is good, the efficiency reached on 128 processors being 78 percent.

Implementation with SOR Relaxation

Successive overrelaxation (SOR) is a variant of Gauss–Seidel relaxation, which improves convergence by using the multiplicative factor ω in the computation of the new values, according to the following formula:

$$u_{i,j}^{(n+1)} := (1-\omega)\, u_{i,j}^{(n)} + \frac{1}{4}\,\omega \left(u_{i-1,j}^{(n+1)} + u_{i,j-1}^{(n+1)} + u_{i+1,j}^{(n)} + u_{i,j+1}^{(n)} - h^2 f_{i,j} \right)$$

It can be shown that SOR converges, if $\omega \in (0,2)$. The use of $\omega > 1$ is called *overrelaxation*, that of $\omega < 1$ *underrelaxation* (Modi 1990). For $\omega = 1$, SOR reduces to the Gauss–Seidel method.

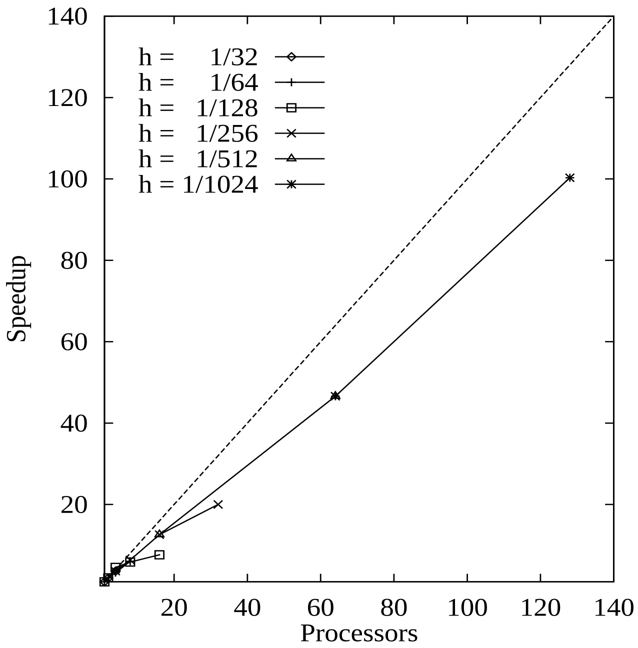


Figure 5.17: Speedups for MG00 with Gauss–Seidel relaxation

The implementation of SOR is similar to that of Gauss–Seidel, with the difference that the additional parameter ω has to be supplied to the argument function of the *mapenv* skeleton that performs the point-wise relaxation. This argument passing is done by *partial application*, as shown in the following example:

$$\texttt{mg\_mapenv (mg, 1, } \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \texttt{, sor (omega), True) ;}$$

The program MG00 was implemented using SOR relaxation with $\omega = 0.8$ on the GCel machine. The resulting speedup curves are shown in Figure 5.18, the efficiency on 64 processors being 92 percent.

Implementation with Red–Black Jacobi Relaxation

In red–black relaxation methods, the nodes of the domain are considered to be marked with two colors, red and black, in checkerboard fashion. One iteration over the entire domain consists of two steps: first, the values at the red nodes are updated, using the values at the black nodes, then the values at the black nodes are updated, using the values at the red nodes (Modi 1990). This method can be combined with several relaxation procedures, yielding Red–Black Jacobi, Red–Black Gauss–Seidel, etc. We have considered here the Red–Black Jacobi method.

In contrast to the classic Jacobi relaxation, buffering the newly computed values until the end of the iteration is not necessary here. The reason is that the red nodes are updated independently from the black nodes, and vice-versa. As we use 5-point stencils and as the 4 direct neighbors of a node have the complementary color, we always use the old values of the 4 neighbors. Hence, the values can be updated directly in place; therefore, neither additional storage, nor an extra update at the end of each iteration is necessary.

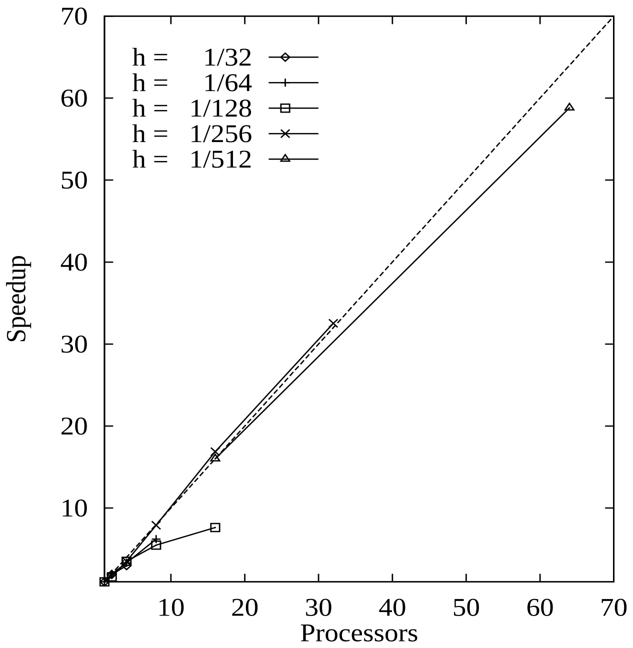


Figure 5.18: Speedups for MG00 with SOR relaxation

Each iteration is implemented by two calls of the *mapenv* skeleton, one updating only the red nodes, the other only the black nodes (we thereby assume the existence of two appropriate test functions). Notice that the nodes to be processed are indicated to the skeleton by a set central bit in the template returned by the dependency function.

$$\texttt{mg\_mapenv (mg, 1, red (coords) ?} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} : \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \texttt{, jacobi, True) ;}$$

$$\texttt{mg\_mapenv (mg, 1, black (coords) ?} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} : \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \texttt{, jacobi, True) ;}$$

We have implemented a variant of MG00 using Red–Black Jacobi relaxation. The speedup curves are given in Figure 5.19, the parallel efficiency reached thereby being 86 percent on 128 processors.

Implementation with Line Jacobi Relaxation

Line relaxations and, more generally, block relaxations update all elements of a line (block) in one step. This update is usually done by solving exactly a sparse, e.g., a tridiagonal system involving the unknowns at the considered nodes (Modi 1990). Depending on the order in which the lines (blocks) are updated, we obtain different iterative methods, such as Line Jacobi, Line SOR, Line Red–Black Gauss–Seidel, etc.

We consider here Line Jacobi relaxation, where all lines are updated at the same time, i.e., using using the old values at the nodes in the neighboring lines. Similarly to point-wise Jacobi, we need here both the auxiliary storage and the update step at the end of each iteration performed by the *map* skeleton.

The line relaxation is implemented by the skeleton `mg_mapline_env`, which is the block-wise counterpart of the (point-wise) `mg_mapenv` skeleton. Thus, the smoother is

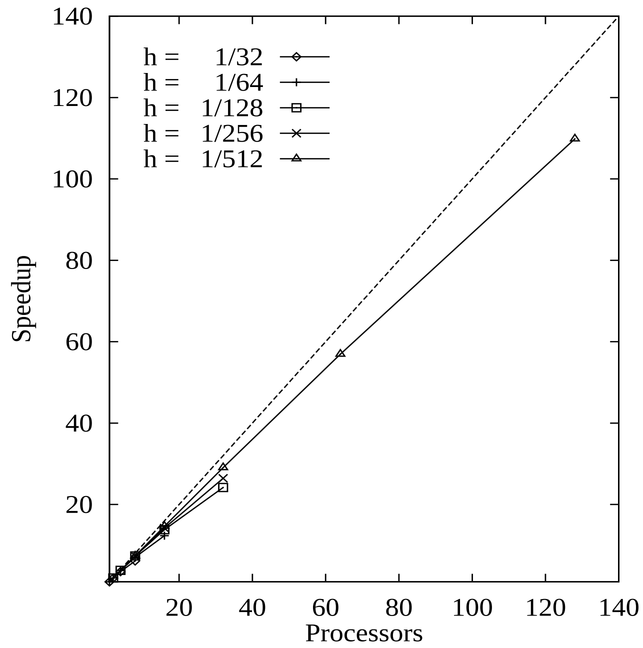


Figure 5.19: Speedups for MG00 with Red–Black Jacobi relaxation

implemented as shown next, whereby the template returned by the dependency function denotes that all lines are processed during an iteration (the central bit is set) and that no dependencies occur between the lines (no other bit is set).

```
for (i = 0 ; i < nu ; i++) {                              /* smoothing */
   mg_mapline_env (mg, 1, [0 1 0], line_jacobi, True) ;
   mg_map (mg, update_u_values, True) ;
}
```

The speedup curves are shown in Figure 5.20. Notice that the scalability of this version is not as good as in the previously presented versions. This behavior is due to the complex structure of the computations, where in each step, a number of linear systems equal to the number of lines has to be solved (Modi 1990). Still, an efficiency of 60 percent is reached on 256 processors.

5.5.2 MG01

The second multigrid application we consider is MG01 (Stüben 1982; Stüben and Trottenberg 1984). In contrast to MG00, we present here a single version of the skeleton-based implementation, however, for three different computational domains. Results are given for each of the three domains.

Description of the Algorithm

MG01 is a considerably more complex multigrid application than MG00. It solves the Helmholtz equation $-\Delta u + c\,(x, y)\; u = f$ on two-dimensional domains bounded by two functions, y_{hi} and y_{low}, with Dirichlet boundary conditions (Stüben 1982). The program

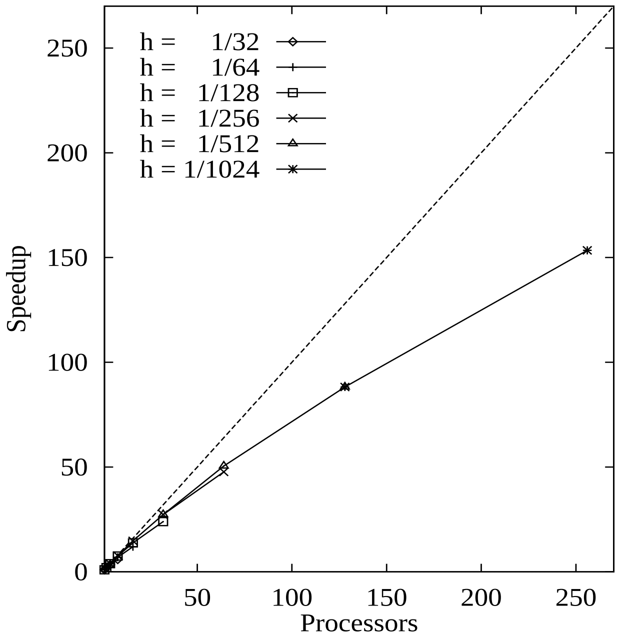


Figure 5.20: Speedups for MG00 with Line Jacobi relaxation

implements a full multigrid algorithm with W correction cycles, employing the FAS correction scheme. The restriction is done by full weighting (Joppich and Mijalković 1993). The size of the original sequential version of the MG01 program is a few thousand lines of code. In contrast, the skeleton-based implementation is merely a few hundred lines long.

Skeleton-Based Implementation

The user-defined data structure `Data` contains, in addition to the approximation of the solution `u`, the right hand side `f`, and the residual `r`, also the auxiliary field `u_new` (needed in the Jacobi relaxation), as well as the error `e` and a further auxiliary variable `w` (both needed in the implementation of the full approximation scheme).

```
typedef struct _Data { double u, u_new, f, e, w, r; } *Data ;
```

The procedure `MG01` gets as arguments the number of levels of the grid hierarchy, the initial discretization step (`h`), the number of pre- and post-smoothing steps (`nu_1` and `nu_2`, respectively), the number of iterations on the coarsest grid (`nic`), and the number of correction cycles (`ncc`).

The grid hierarchy—initially consisting of one level, which will then become the coarsest one—is generated by the *create* skeleton in the domain described by:

$$\begin{aligned} x_{low} &= -1 \qquad x_{hi} = 1 \\ y_{low} &= -1 + \sqrt{1 - x^2} \\ y_{hi} &= 1 + \sqrt{1 - x^2} \end{aligned}$$

and distributed vertically onto the processors. Then, a first approximation of the solution is computed on this grid, by using the *mapenv* skeleton. As Jacobi relaxation is employed, the solution has to be updated after each iteration.

The rest of the procedure `MG01` contains the full multigrid cycle. Each iteration herein starts by refining the current grid, using the standard procedure of halving the mesh size `h`. After that, the solution is prolongated from the former current grid, to the newly created grid. The latter is now both the finest and the current grid. The prolongation of data is done by the *map* skeleton, whose argument function performs a linear interpolation on the values of the coarser grid, storing the results in the nodes of the finer grid. For that, the function `interpolate` makes use of inter-grid operations. Finally, a number of correction cycles are performed using the full approximation scheme (FAS). The procedure `MG01` is defined as follows:

```
void MG01 (int nr_levels, double h, double nu_1, double nu_2,
           double nic, double ncc) {
   MGrid <Data> mg ;  int i, j ;

                                  /* create initial (coarsest) grid */
   mg = mg_create (-1.0, 1.0, &unit_semicirc_low, &unit_semicirc_hi,
                   h, 1, init_elem, COLUMN_SLICES) ;

   for (i = 0 ; i < nic ; i++) {        /* solution on initial grid */
      mg_mapenv (mg, 1, no_dep_f, jacobi, True) ;
      mg_map (mg, update_u_values, True) ;
   }

   for (i = 2 ; i <= nr_levels ; i++) {                /* FMG cycle */
      h /= 2 ;
      mg_create_level (mg, h, init_elem_zero) ;      /* refine grid */
      mg_map (mg, interpolate (mg), True) ;  /* prolongate solution */

      for (j = 0 ; j < ncc ; j++)
         FAS (mg, i, nu_1, nu_2, nic, 2) ;    /* correction scheme */
   }

   output results ;
   mg_destroy (mg, destroy_elem) ;
}
```

The procedure `FAS`—defined next—performs a multigrid correction cycle, comprising ν_1 pre-smoothing steps, a coarse grid correction, and ν_2 post-smoothing steps. The coarse grid correction is based on the full approximation scheme, as described in Section 5.1.5. It consists of three parts: data restriction, recursive application of the procedure `FAS`, and data prolongation.

To perform the restriction of data, the residual is first computed on the fine grid. This operation corresponds to step 1 in the FAS algorithm given in Section 5.1.5 and is implemented by the *mapenv* skeleton. After that, the residual `r` and the current approximation of the solution `u` are restricted to the next coarser grid[7]. This restriction corresponds to step 2 and partly to step 3 in the FAS algorithm, and is done here by full weighting, i.e., by computing the averages of the values at the nearest nodes on the finer

[7]This is where the additional component `w` of the user data structure is necessary.

grid (Joppich and Mijalković 1993). The second part of step 3 of FAS, the computation of the right hand side of the coarse grid equation, $L_H \hat{I}_h^H w_h^{(n)} + r_H^{(n)}$, is implemented using the *mapenv* skeleton, because both operators used in this computation, L_H and $\hat{I}_h^H$, access the neighborhoods of nodes.

The recursive call of the procedure `FAS` leads to the creation of the multigrid correction cycle. As can be seen from the call of `FAS` in the program `MG01`, the factor γ is here 2, therefore the resulting cycle is a W-cycle (see Section 5.1.4).

The first stage in the prolongation of data is the computation of the error on the coarse grid. This computation corresponds to step 4 in the FAS algorithm. After that, the error is interpolated from the coarser to the finer grid, and the correction is performed on the solution (steps 5 and 6 in the FAS algorithm, respectively).

```
void FAS (MGrid <$t> mg, int level_nr, int nu_1, int nu_2, int nic,
          int gamma) {
    int j ;

    if (level_nr == 1)
        compute approximate solution on coarsest grid ;
    else {
        ν1 pre-smoothing steps ;
                                               /* compute the residual */
        mg_mapenv (mg, 1, no_dep_f, make_resid, True) ;
        mg_set_crt_level (mg, -1) ;           /* move to coarser level */
        mg_map (mg, full_weighting (mg), True) ;       /* restriction */
        mg_mapenv (mg, 1, no_dep_f, make_rhs, True) ;

        for (j = 0 ; j < gamma ; j++)                       /* W cycle */
            FAS (mg, level_nr-1, nu_1, nu_2, nic, gamma) ;

        mg_map (mg, make_err, True) ;                 /* compute error */
        mg_set_crt_level (mg, 1) ;              /* move to finer level */
        mg_map (mg, interp_corr (mg), True) ;          /* prolongation */

        ν2 post-smoothing steps ;
    }
}
```

Results

We have run the skeleton-based implementation of the program MG01 on a Parsytec CGel 1024 machine. The multigrid program was parameterized as follows: number of grid levels 3, number of pre- and post-smoothing relaxations $\nu_1 = \nu_2 = 2$, number of relaxations on the coarsest grid `nic` $= 3$, number of correction cycles per level `ncc` $= 2$. The initial discretization step h_0 (for the coarsest grid) took different values between 1/8 and 1/256.

MG01 was run for three different domains. Domain A is the unit square used also in the implementation of MG00. Domain B is bounded by two semicircles, as described in the beginning of this section[8]. Domain C is L-shaped and included in the unit square. The three domains are outlined in Figure 5.21.

[8] The call of the skeleton `mg_create` in the procedure `MG01`, is given for domain B.

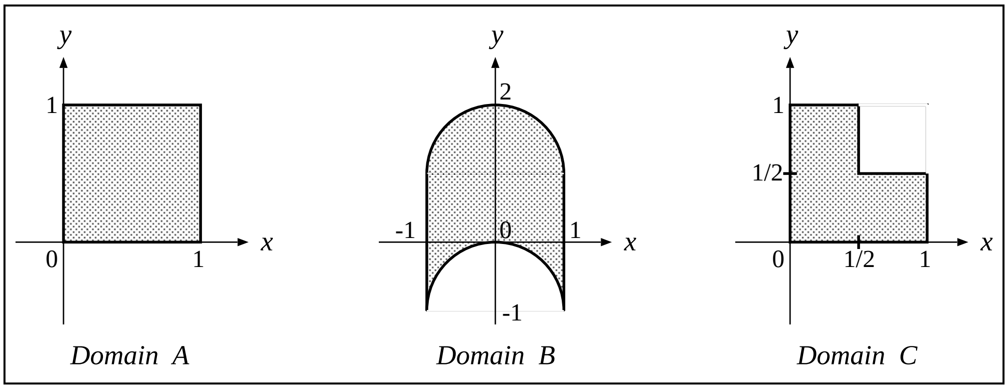


Figure 5.21: Domains for MG01

The speedup curves obtained for the skeleton-based implementation of MG01 on domains A and B are given in Figure 5.22. Although the results are not as good as the ones for MG00, the scalability is still satisfactory. The efficiency reached 56 percent on 256 processors for domain A, and 68 percent on 128 processors for domain B.

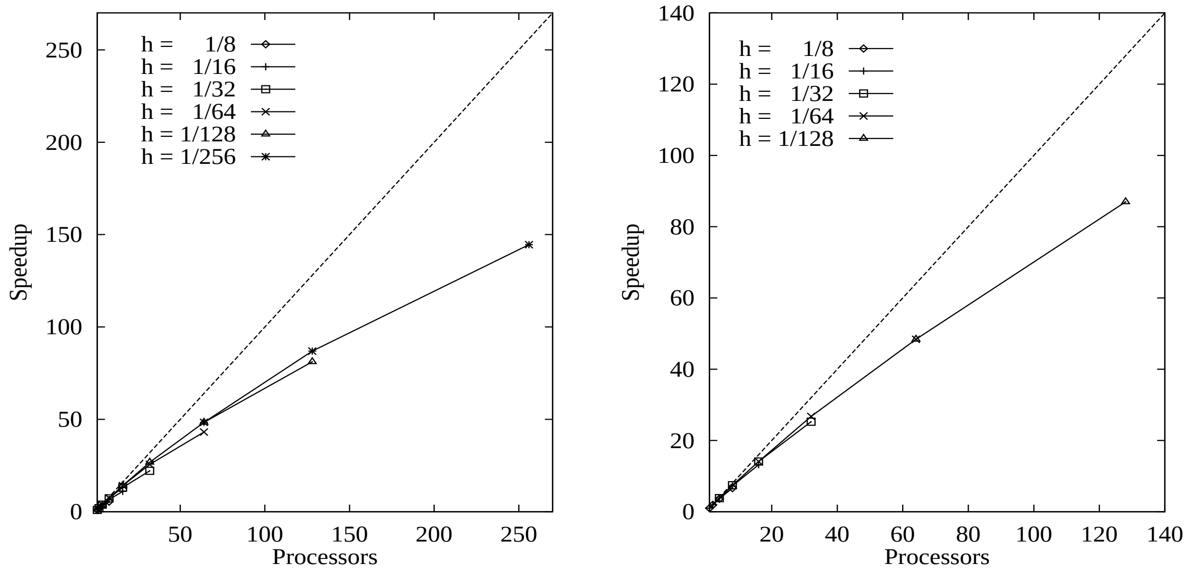


Figure 5.22: Speedups for MG01 for domain A (left) and for domain B (right)

The speedup curves for domain C are shown in Figure 5.23. The curves are similar to those obtained for the first two domains, the efficiency on 64 processors being 85 percent.

5.5.3 Discussion of the Results

Concluding this chapter, we give a short overview of our results, explaining the high performance reached by our implementation.

The skeleton-based implementations of the multigrid applications MG00 and MG01 are characterized by a high scalability, the efficiency being 85% to 90% on 64 processors, 70% to 85% on 128 processors, and 60% to 70% on 256 processors. As these results are very good, we present the main reasons for their quality.

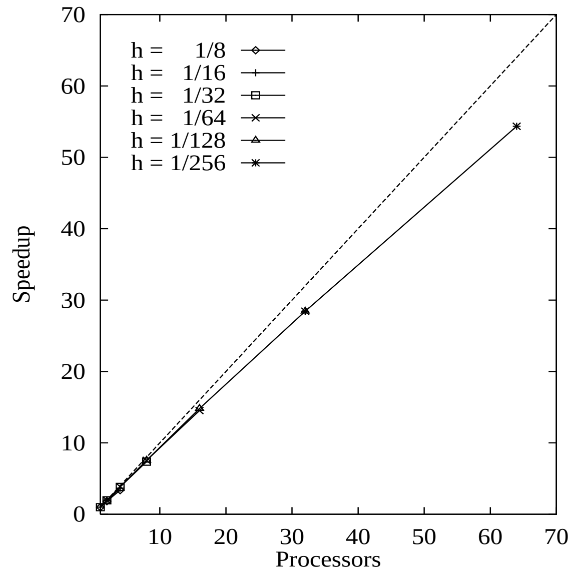


Figure 5.23: Speedups for MG01 for domain C

First, we employ only *structured, rectangular grids*. On the one hand, this restriction simplifies the internal representation of the grid hierarchy, as no explicit links are necessary between neighboring nodes, or between corresponding nodes on neighboring levels. On the other hand, all operations on grid nodes are highly efficient, because the nodes can be addressed directly, rather than via links.

Second, we employ *no adaptive grid refinement*. As a consequence, there is no need for dynamic load balancing, all parallel operations imposing a comparable load on all processors. Moreover, the inter-grid locality condition, meaning the placement of corresponding nodes from neighboring levels on the same or on near processors, is automatically fulfilled. Therefore, inter-grid data transfers can be performed with little or no communication overhead.

Third, we make intensive use of *overlap domains*. This optimization eliminates any chaotic, demand-driven communication during the execution of a skeleton, replacing it by a coordinated and compacted communication step at the end of the skeleton execution, in a BSP-like style (McColl 1996; Valiant 1990).

Finally, we use *appropriate grid distributions*, like for instance the diagonal distribution, which minimizes the necessary communication during Gauss–Seidel and SOR relaxations.

On the whole, the first two items can be regarded as restrictions placed on the discretization of the problem and on the grid generation, whereas the latter two are implementation optimizations. The attempt to lift the restrictions shows, how hard the parallel implementation of multigrid algorithms is in the general case. As an example, Bastian presents in his work a direct parallel implementation of multigrid methods with adaptive refinement and unstructured grids (Bastian 1994). The efficiency reported by him is 38 percent on 64 processors on the Intel Paragon—the absolute run time on 72 processors being higher than that on 64 processors—and merely 15 percent on 64 transputers. According to Bastian (1994), at least 25 percent of the efficiency losses are due to the local grid refinements.

This comparison should, however, not lead to the conclusion that skeleton-based implementations are more efficient or have a higher scalability than their direct counterparts. Our goal here was merely to explain our good results, by emphasizing the influence of unstructured grids and adaptive refinement on the performance of a parallel implementation, rather than to compare two totally different approaches.

Of course, a skeleton-based implementation of an algorithm could never beat an equally well optimized direct implementation. The goal of a skeletal implementation is rather to come as close as possible to the performance of an efficient direct implementation, however, paying a considerably smaller price in terms of programming effort.

Chapter 6

Conclusions

What is required before parallel computation can succeed, is the adoption of a unifying bridging model for parallel computation.

Leslie Valiant

In recent years, both the need for and the availability of parallel computers has been constantly growing. Unfortunately, the rapid evolution on the hardware side has not been matched by corresponding progress in the software field. Thus, the difficulty of parallel programming—comprising writing, testing, debugging, porting, and evaluating the performance of parallel programs—is widely considered as one of the main drags in establishing parallel processing as a key technology.

Having identified the root of the evil in the low level of currently used programming paradigms, the parallel software community concentrated its efforts toward finding appropriate higher abstraction levels. A wealth of approaches to high-level parallel programming, especially on the language level, emerged herefrom, ranging from standard communication libraries that ensure only program portability, to fully automatically parallelized sequential languages.

Among these approaches, algorithmic skeletons have gained a steadily increasing importance. Unlike other high-level parallel paradigms, skeletons have both theoretical foundations, such as underlying formalisms and cost models, and practical relevance, due to their expressive power, flexibility, and—as we have shown—the possibility to be implemented efficiently.

In this thesis, we have presented a new approach to skeleton-based parallel programming. Our main aim was to show that skeletons can be used successfully in practice, not only from the point of view of the classes of problems that can be handled with them, but also regarding the efficiency and scalability of the overall implementation.

The first problem we have considered was the host language. We have designed *Skil* as an imperative language enhanced with a series of functional features and with a polymorphic type system. Thus, *Skil* enables a high programming level and at the same time can be efficiently implemented.

Unlike other related approaches, skeletons are here not internal language constructs, but they are written in *Skil*. *Skil* thus provides the framework in which both skeletons and applications are implemented. Thereby, parallelism is encapsulated in the skeletons —which should have a high degree of flexibility and reusability, whereas the application is written as a sequential program. As skeletons are the only parallel constructs available, the programming style is more restricted than, for instance, in languages with explicit message passing. However, the gains with respect to the structure and clarity of programs, as well as in easiness of programming and testing, and not least in portability, outweigh this limitation by far.

We have then described a compile-time technique for the implementation of the functional features. This technique is crucial for the efficiency of our implementation, because it eliminates the overhead caused by classic run-time techniques based on closures. As our transformation is static, we have imposed certain limitations in case of recursive HOFs and developed a static criterion to check such HOFs for possibly nonterminating instantiations. We have demonstrated that our transformation terminates and is correct. Concluding Chapter 3, we have presented the main optimization technique employed by the *Skil* compiler, namely inlining, showing that this technique can bring important improvements to the performance of programs.

The efficiency of *Skil* programs was addressed in Chapter 4. Here, we have described a series of skeletons for the work with distributed arrays and showed how four matrix applications can be implemented using these skeletons. Run-time results have pointed out that our implementation was faster than a similar one in a functional language with skeletons, approaching the performance of hand-written C code based on message passing. The importance of this result lies in showing that the skeletal approach, which allows writing high-level parallel programs, can also lead to efficient implementations and thus becomes interesting for real applications.

Finally, in Chapter 5, we have considered the use of skeletons to solve real-world problems in parallel, choosing as application area multigrid algorithms. We have studied the data access and communication patterns occurring in these methods and derived from them a set of skeletons, which can be used as the building blocks for most multigrid algorithms. We have then illustrated the skeleton-based implementation of multigrid by means of two applications and presented several variants of implementations, using different multigrid primitives. Run-time results have shown that our programs achieve excellent speedups, even for large processor numbers.

Apart from demonstrating the high scalability of our implementation, the multigrid applications we have presented highlight a crucial aspect of the skeletal approach, namely, that the user no longer has to write global procedures for a whole distributed data structure and explicitly handle problems such as communication or data distribution, but merely has to supply local functions, which parameterize the skeletons appropriately. Thus, a decoupling of the actual application programming—in our case, of the numerical method—and of the handling of parallelism is achieved.

⋆ ⋆ ⋆

The field of algorithmic skeletons is presently in a state of growth and structuring, therefore, several directions can be envisaged for further development. We start by describing new perspectives opened by our work.

First, the distributed data structure *array* and its skeletons could be enhanced to handle not only block-wise distributions of arrays onto the processors, but also cyclic ones, block-cyclic ones, etc. This extension would show that skeletons can implement all data distributions offered, for instance, by HPF.

Second, the multigrid skeletons could be enhanced to cope with adaptive grid refinement. This extension would imply on the one hand enhancing existing skeletons, like for instance the constructor of a grid level, by further functional arguments, and on the other hand, supplying additional dedicated skeletons, e.g., for load balancing, as outlined by Botorog and Kuchen (1995).

Third, the multigrid data structure and skeletons could be extended to handle unstructured, irregular grids, such as triangulations. This extension would allow to cope with problems discretized by the finite element method. At the same time, it would imply the use of dynamic data structures, as well as the explicit storage and use of the links between the grid nodes, neither of which has been considered yet in skeleton-based parallel implementations.

Last, the existing multigrid skeletons could be generalized to skeletons for hierarchically structured problems. Thus, further applications, such as N-body problems (Wu et al. 1997), could be implemented in the resulting unified framework. Technically, this generalization would mainly amount to enhancing the skeletons with the capability of transferring data not only between pairs of neighboring levels, but among several levels at a time.

Beyond the frame of our work, other research areas could be considered for parallel implementation with skeletons. Examples include computational geometry (Zippel 1990) and computer algebra (Dehne et al. 1996), which seem to fit naturally into the skeletal approach, as in each of these areas, many algorithms are built out of a relatively small set of primitive operations and variations thereof.

⋆ ⋆ ⋆

The future of parallel programming lies undoubtedly in the high-level programming paradigms. However, the main problem nowadays is perhaps the extreme diversity of high-level approaches, whereby none of these approaches—including algorithmic skeletons—achieves a consensus throughout the parallel processing community. What is required at this stage, is a confluence step, joining the most promising ideas into a single approach. Although skeletons might not be *the* answer, they could surely make an important contribution to this overall solution.

Appendix A

The Language *Skil*

A.1 Overview of Differences between *Skil* and C

Enhancements: To support the integration of skeletons, five kinds of features were added to the standard C language (Kernighan and Ritchie 1988):

- Higher-order functions
- Function currying
- Type variables
- Conversion of operators to functions
- Features supporting data parallelism (`pardata`, `$$dyn`, `$$par`)

Restrictions: To simplify the grammar of the language, the following C features have been eliminated or restricted:

- No enumeration type (`enum`).
- Structures and unions must always have a name.
- The simple types are: `void`, `char`, `short`, `int`, `long`, `float`, `double`, `signed` and `unsigned`. The types are used one at a time, i.e., no combination, such as `unsigned char` or `long float` is allowed. The types `short`, `long`, `signed` and `unsigned` denote by default `short int`, ... , `unsigned int`, respectively.
- The type qualifiers `const` and `volatile` are ignored by the compiler.
- No wide chars or wide strings are supported.

- Only new style function definitions, in which the types of the arguments are given in the argument list, are supported (Kernighan and Ritchie (1988), page 202).
- Only pedantic initializations of arrays and structures are allowed (Kernighan and Ritchie (1988), page 220).

Obsolete features: These features of C are still available in *Skil*, however they have become obsolete due to the integration of the functional features. They have been kept in the language for compatibility reasons, but *Skil* users are encouraged to use the appropriate *Skil* constructs instead. The obsolete features are:

- Pointers to functions as arguments or result of functions. This feature is accounted for in *Skil* by higher-order functions.
- `void *` is no longer needed as universal pointer, `$t *` should be used instead.
- Casts are only needed to perform conversions that are handled in C by the compiler, such as casting a char to an int. The reason is that, to perform correct instantiations, the type checker of the *Skil* compiler is more strict then that of the C compiler, therefore such conversions have to be specified explicitly.

A.2 The *Skil* Grammar

The grammar of *Skil* is given in EBNF notation. Each production has the form:

$$nonterm \rightarrow alt_1 \mid \ldots \mid alt_n$$

Moreover, the following notations are employed:

[*exp*]	–	optional expression
{ *exp* }	–	repetition 0 or more times
(*exp*)	–	grouping
$exp_1 \mid exp_2$	–	alternatives
`int`	–	terminal symbols of the language grammar, which are also terminal symbols in the microsyntax, are written in typewriter font
identifier	–	terminal symbols of the language grammar, which are not terminal symbols in the microsyntax, but stand for classes of terminal symbols, are written in sans-serif font

For simplicity, we present only the grammar of the language, but not its microsyntax, which is similar to that of C, the only difference consisting in the syntax of names of type variables, which must be legal identifiers preceded by a `$`, e.g., `$t`, but not `$1` or `$$`.

transl_unit → *ext_decl* { *ext_decl* }

ext_decl → *fundef*
  | *decl*

fundef → *decl_spec* *declarator* *comp_stmt*

decl → *decl_spec* [*init_declarator* { , *init_declarator* }] ;
  | `typedef` *decl_spec* *declarator* [*typeargs*] ;
  | `pardata` identifier [*typeargs*] [*type_name*] ;

decl_spec → [*storage_class*] *type_spec*

storage_class → `auto` | `register` | `static` | `extern`

type_spec → `void` | `char` | `short` | `int` | `long` | `float`
  | `double` | `signed` | `unsigned`
  | typedef_name [*typeargs*]
  | typevar_name
  | *struct_union_spec*

struct_union_spec → *struct_union* identifier [*typeargs*]
  [`{` *struct_decl* { *struct_decl* } `}`]

struct_union → `struct` | `union`

typeargs → `<` [*type_name* { , *type_name* }] `>`

init_declarator → *declarator* [`=` *init*]

struct_decl → *type_spec* *struct_declarator* { , *struct_declarator* } ;

struct_declarator → *declarator*
  | *declarator* : *const_expr*

declarator → [*pointer*] *direct_declarator*

direct_declarator → identifier
  | (*declarator*)
  | *direct_declarator* `[` [*const_expr*] `]`
  | *direct_declarator* ([*params_ell*])

pointer → `*` [*pointer*]

params_ell → *param_decl* { , *param_decl* } [, ...]

param_decl → *decl_spec* [*declarator* | *abstr_declarator*]

init → *assign_expr*
  | `{` *init* { , *init* } [,] `}`

type_name → *type_spec* [*abstr_declarator*]

abstr_declarator → *pointer*
  | [*pointer*] *direct_abstr_declarator*

direct_abstr_declarator → `(` *abstr_declarator* `)`
  | [*direct_abstr_declarator*] `[` [*const_expr*] `]`
  | [*direct_abstr_declarator*] `(` [*params_ell*] `)`

stmt → *lab_stmt*
  | *expr_stmt*
  | *comp_stmt*
  | *sel_stmt*
  | *iter_stmt*
  | *jmp_stmt*

lab_stmt → (identifier | `case` *const_expr* | `default`) `:` *stmt*

expr_stmt → [*expr*] `;`

comp_stmt → `{` { *decl* } { *stmt* } `}`

sel_stmt → `if` `(` *expr* `)` *stmt* [`else` *stmt*]
  | `switch` `(` *expr* `)` *stmt*

iter_stmt → `while` `(` *expr* `)` *stmt*
  | `do` *stmt* `while` `(` *expr* `)` `;`
  | `for` `(` [*expr*] `;` [*expr*] `;` [*expr*] `)` *stmt*

jmp_stmt → `goto` identifier `;`
  | `continue` `;`
  | `break` `;`
  | `return` [*expr*] `;`

expr → [*expr* `,`] *assign_expr*

assign_expr → *cond_expr*
  | *unary_expr* *ass_op* *assign_expr*

ass_op → `=` | `*=` | `/=` | `%=` | `+=` | `-=` | `<<=` | `>>=` | `&=`
  | `^=` | `|=`

cond_expr → *logic_or_expr* [`?` *expr* `:` *cond_expr*]

const_expr → *cond_expr*

logic_or_expr → [*logic_or_expr* `||`] *logic_and_expr*

logic_and_expr → [*logic_and_expr* `&&`] *incl_or_expr*

incl_or_expr → [*incl_or_expr* `|`] *excl_or_expr*

excl_or_expr → [*excl_or_expr* `^`] *and_expr*

and_expr → [*and_expr* `&`] *equal_expr*

equal_expr → [*equal_expr* (`==` | `!=`)] *rel_expr*

rel_expr → [*rel_expr* (`<` | `>` | `<=` | `>=`)] *shift_expr* *shift_expr* → [*shift_expr* (`<<` | `>>`)] *add_expr*

add_expr → [*add_expr* (`+` | `-`)] *mult_expr*

mult_expr → [*mult_expr* (`*` | `/` | `%`)] *cast_expr*

cast_expr → *unary_expr*
| `(` *type_name* `)` *cast_expr*

unary_expr → *postfix_expr*
| (`++` | `--`) *unary_expr*
| *unary_op* *cast_expr*
| `sizeof` *unary_expr*
| `sizeof` `(` *type_name* `)`

unary_op → `&` | `*` | `+` | `-` | `~` | `!`

postfix_expr → *prim_expr*
| *postfix_expr* `[` *expr* `]`
| *postfix_expr* `(` [*arg_exprs*] `)`
| *postfix_expr* (`.` | `->`) identifier
| *postfix_expr* (`++` | `--`)

prim_expr → identifier
| *constant*
| (`$$dyn` | `$$par`) `(` *type* `)`
| `(` *expr* `)`
| `(` *funop* `)`

funop → `||` | `&&` | `|` | `^` | `&` | `==` | `!=` | `<` | `>` | `<=` | `>=`
| `<<` | `>>` | `+` | `-` | `*` | `/` | `%` | `~` | `!`

arg_exprs → [*arg_exprs* `,`] *assign_expr*

constant → integer | character | string | real

Bibliography

Adams, J. C., W. S. Brainerd, J. T. Martin, B. T. Smith, and J. L. Wagener (1992). *Fortran 90 Handbook.* New York: McGraw-Hill.

Aho, A. V., J. E. Hopcroft, and J. D. Ullman (1974). *The Design and Analysis of Computer Algorithms.* Reading, MA: Addison-Wesley.

Anderson, T., D. Culler, D. Patterson, and the NOW Team (1995). A case for NOW (networks of workstations). *IEEE Micro 15*(1), 54–64.

Appel, A. (1992). *Compiling with Continuations.* Cambridge, England: Cambridge University Press.

Bacci, B., M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi (1995). P^3L: A structured high level programming language and its structured support. *Concurrency: Practice and Experience 7*, 225–255.

Banâtre, J. P. and D. Le Métayer (1991). Introduction to Gamma. In *Research Directions in High-Level Programming Languages*, Volume 574 of *Lecture Notes in Computer Science*, pp. 197–230. Berlin: Springer-Verlag.

Banerjee, U., R. Eigenmann, A. Nicolau, and D. A. Padua (1993). Automatic program parallelization. *Proceedings of IEEE 81*(2), 211–243.

Barnes, J. and P. Hut (1986). A hierarchical $O(n \log n)$ force calculation algorithm. *Nature 324*, 446–449.

Bastian, P. (1994). *Parallele adaptive Mehrgitterverfahren.* Ph. D. thesis, University of Heidelberg, Germany. (in German).

Bell, J., F. Bellegarde, and J. Hook (1997). Type-driven defunctionalization. In *Proceedings of ICFP '97*, pp. 25–37. New York: ACM Press.

Berrendorf, R., M. Gerndt, W. E. Nagel, and J. Prümmer (1993). SVM Fortran. Technical Report KFA-ZAM-IB-9322, Central Institute for Applied Mathematics, Research Center Jülich, Germany.

Bird, R. (1989). Lectures on constructive functional programming. In *Cognitive Methods in Computer Science*, Volume F55 of *NATO ASI Series*, pp. 151–216. Berlin: Springer-Verlag.

Blelloch, G. (1995). NESL: A nested data-parallel language (3.1). Technical Report CMU-CS-95-170, Carnegie Mellon University, Pittsburgh, PA.

Botorog, G. H. (1997). Deriving skeletons for multigrid algorithms. In *Proceedings of PDCN '97*, pp. 180–183. Anaheim, CA: IASTED/Acta Press.

Botorog, G. H. and H. Kuchen (1995). Algorithmic skeletons for adaptive multigrid methods. In *Proceedings of Irregular '95*, Volume 980 of *Lecture Notes in Computer Science*, pp. 27–41. Berlin: Springer-Verlag.

Botorog, G. H. and H. Kuchen (1996a). Efficient parallel programming with algorithmic skeletons. In *Proceedings of EuroPar '96*, Volume 1123 of *Lecture Notes in Computer Science*, pp. 718–731. Berlin: Springer-Verlag.

Botorog, G. H. and H. Kuchen (1996b). *Skil*: An imperative language with algorithmic skeletons for efficient distributed programming. In *Proceedings of HPDC-5 '96*, pp. 243–252. Los Alamitos, CA: IEEE Computer Society Press.

Botorog, G. H. and H. Kuchen (1996c). Using algorithmic skeletons with dynamic data structures. In *Proceedings of Irregular '96*, Volume 1117 of *Lecture Notes in Computer Science*, pp. 263–276. Berlin: Springer-Verlag.

Botorog, G. H. and H. Kuchen (1998). Efficient high-level parallel programming. *Theoretical Computer Science*. (to appear).

Bougé, L., P. Fraigniaud, A. Mignotte, and Y.Robert (Eds.) (1996). *Euro-Par'96 Parallel Processing (Proceedings)*, Volume 1123–1124 of *Lecture Notes in Computer Science*. Berlin: Springer-Verlag.

Brandt, A. (1977). Multi-level adaptive solutions to boundary-value problems. *Mathematics of Computation 31*(138), 333–390.

Brandt, A. (1982). Guide to multigrid development. In *Multigrid Methods*, Volume 960 of *Lecture Notes in Mathematics*, pp. 220–312. Berlin: Springer-Verlag.

Bratvold, T. A. (1994). Parallelizing a functional program using a list-homomorphism skeleton. In *Proceedings of PASCO '94*, Volume 5 of *Lecture Notes Series in Computing*, pp. 44–53. Singapore: World Scientific.

Briggs, W. L. (1987). *A Multigrid Tutorial.* Philadelphia, PA: SIAM.

Carriero, N. J., D. Gelernter, T. G. Mattson, and A. H. Sherman (1994). The Linda alternative to message-passing systems. *Parallel Computing 20*, 633–655.

Chang, P. P., S. A. Mahlke, W. Y. Chen, and W. W. Hwu (1992). Profile-guided automatic inline expansion for C programs. *Software – Practice and Experience 22*(5), 349–369.

Chapman, B. M., P. Mehrotra, and H. P. Zima (1992). Programming in Vienna Fortran. *Scientific Programming 1*(1), 31–50.

Chin, W. N. and J. Darlington (1996). A higher-order removal method. *Lisp and Symbolic Computation 9*, 287–322.

Cole, M., S. Gorlatch, C. Lengauer, and D. Skillicorn (Eds.) (1997). *Theory and Practice of Higher-Order Parallel Programming.* Dagstuhl-Seminar-Report 169. Dagstuhl: IBFI.

Cole, M. I. (1989). *Algorithmic Skeletons: Structured Management of Parallel Computation.* London: Pitman/MIT Press.

Darlington, J., A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, and Q. Wu (1993). Parallel programming using skeleton functions. In *Proceedings of PARLE '93*, Volume 694 of *Lecture Notes in Computer Science*, pp. 146–160. Berlin: Springer-Verlag.

Darlington, J., Y. Guo, H. W. To, and J. Yang (1995). Functional skeletons for parallel coordination. In *Proceedings of EuroPar '95*, Volume 966 of *Lecture Notes in Computer Science*, pp. 55–66. Berlin: Springer-Verlag.

Davidson, J. W. and A. M. Holler (1988). A study of a C function inliner. *Software – Practice and Experience 18*(8), 775–790.

De Keyser, J. and D. Roose (1992). Partitioning and mapping adaptive multigrid hierarchies on distributed memory computers. Technical Report TW 166, University of Leuven, Belgium.

Dehne, F., A. Fabri, and A. Rau-Chaplin (1996). Scalable parallel computational geometry for coarse grained multicomputers. *International Journal on Computational Geometry 6*(3), 379–400.

Deldarie, H., J. R. Davy, and P. M. Dew (1995). The performance of parallel algorithmic skeletons. In *Proceedings of Zeus '95*, pp. 65–74. Oxford, England: IOS Press.

Fang, N. (1996). Engineering parallel algorithms. In *Proceedings of HPDC-5 '96*, pp. 233–242. Los Alamitos, CA: IEEE Computer Society Press.

Feo, J. T. (Ed.) (1992). *A Comparative Study of Parallel Programming Languages: The Salishan Problems.* Amsterdam: North-Holland.

Field, A. J. and P. G. Harrison (1988). *Functional Programming.* Wokingham, England: Addison-Wesley.

Flynn, M. J. (1972). Some computer organizations and their effectiveness. *IEEE Transactions on Computers 21*(9), 948–960.

Foster, I., R. Olson, and S. Tuecke (1992). Productive parallel programming: The PCN approach. *Scientific Programming 1*, 51–66.

Foster, I. and S. Taylor (1990). *Strand: New Concepts in Parallel Programming.* Englewood Cliffs, NJ: Prentice-Hall.

Freiberg, G. (1998). Skelett-basierte Implementierung der "Salishan Problems". Master's thesis (in German). Aachen University of Technology, Germany.

Geist, A., A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam (1994). *PVM: Parallel Virtual Machine – A Users' Guide and Tutorial for Networked Parallel Computing.* Cambridge, MA: MIT Press.

Gorlatch, S. (1996). Systematic extraction and implementation of divide-and-conquer parallelism. In *Proceedings of PLILP '96*, Volume 1140 of *Lecture Notes in Computer Science*, pp. 274–288. Berlin: Springer-Verlag.

Gottlieb, A., R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir (1983). The NYU Ultracomputer—designing a MIMD, shared-memory parallel computer. *IEEE Transactions on Computers 32*(2), 175–189.

Hackbusch, W. and U. Trottenberg (Eds.) (1982). *Multigrid Methods*, Volume 960 of *Lecture Notes in Mathematics.* Berlin: Springer-Verlag.

Herrmann, C. A. and C. Lengauer (1997). Transformation of divide & conquer to nested parallel loops. In *Proceedings of PLILP '97*, Volume 1292 of *Lecture Notes in Computer Science*, pp. 95–110. Berlin: Springer-Verlag.

High Performance Fortran Forum (1993). High Performance Fortran language specification. *Scientific Programming 2*, 1–170.

Hiranandani, S., K. Kennedy, C. Koelbel, U. Kremer, and C. W. Tseng (1992). An overview of the Fortran D programming system. In *Proceedings of LCPC '92*, Volume 589 of *Lecture Notes in Computer Science*, pp. 18–34. Berlin: Springer-Verlag.

Hogen, G., A. Kindler, and R. Loogen (1992). Automatic parallelization of lazy functional programs. In *Proceedings of ESOP '92*, Volume 582 of *Lecture Notes in Computer Science*, pp. 254–268. Berlin: Springer-Verlag.

Hudak, P., Peyton Jones, S.L., and P. Wadler (Eds.) (1992). Report on the programming language Haskell, a non-strict purely functional language. *ACM SIGPLAN Notices 27*(5).

Jagannathan, S. and A. Wright (1996). Flow-directed inlining. In *Proceedings of PLDI '96*, pp. 193–205. New York: ACM Press.

Johnson, C. (1987). *Numerical Solutions of Partial Differential Equations by the Finite Element Method.* Cambridge, England: Cambridge University Press.

Johnsson, T. (1985). Lambda lifting: Transforming programs to recursive equations. In *Proceedings of FPCA '85*, Volume 201 of *Lecture Notes in Computer Science*, pp. 190–203. Berlin: Springer-Verlag.

Joppich, W. and S. Mijalković (1993). *Multigrid Methods for Process Simulation.* New York: Springer-Verlag.

Kernighan, B. and D. Ritchie (1988). *The C Programming Language.* Englewood Cliffs, NJ: Prentice-Hall.

Kuchen, H. (1995). Datenparallele Programmierung von MIMD-Rechnern mit verteiltem Speicher. Habilitation thesis (in German). Aachen University of Technology, Germany.

Kuchen, H., R. Plasmeijer, and H. Stoltze (1994). Efficient distributed memory implementation of a data parallel functional language. In *Proceedings of PARLE '94*, Volume 817 of *Lecture Notes in Computer Science*, pp. 464–477. Berlin: Springer-Verlag.

Kuchen, H., H. Stoltze, I. Dimov, and A. Karaivanova (1995). Distributed memory implementation of elliptic partial differential equations in a data parallel functional language. In *Proceedings of MPPM '95*, pp. 142–150. Los Alamitos, CA: IEEE Computer Society Press.

Kumar, V., A. Grama, A. Gupta, and G. Karypis (1994). *Introduction to Parallel Computing: Design and Analysis of Algorithms.* Redwood City, CA: Benjamin/Cummings.

Leighton, F. T. (1992). *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* San Mateo, CA: Morgan Kaufmann.

Lengauer, C., M. Griebl, and S. Gorlatch (Eds.) (1997). *Euro-Par'97 Parallel Processing (Proceedings)*, Volume 1300 of *Lecture Notes in Computer Science.* Berlin: Springer-Verlag.

Leroy, X. (1993). Polymorphism by name for references and continuations. In *Proceedings of POPL '93*, pp. 220–231. New York: ACM Press.

Levine, J. R., T. Mason, and D. Brown (1992). *Lex & Yacc.* Sebastopol, CA: O'Reilly.

Li, X., P. Lu, J. Schaeffer, J. Shillington, P. S. Wong, and H. Shi (1993). On the versatility of parallel sorting by regular sampling. *Parallel Computing 19*, 1079–1103.

Linden, J., G. Lonsdale, H. Ritzdorf, and A. Schüller (1992). Block-structured multigrid for the Navier-Stokes equations: Experiences and scalability questions. In *Conference on Parallel Computational Fluid Dynamics.* Amsterdam: Elsevier.

Loyens, L. D. J. C. and J. R. Moonen (1994). ILIAS, a sequential language for parallel matrix computations. In *Proceedings of PARLE '94*, Volume 817 of *Lecture Notes in Computer Science*, pp. 250–261. Berlin: Springer-Verlag.

McColl, W. F. (1996). Scalability, portability and predictability: The BSP approach to parallel programming. *Future Generation Computer Systems 12*(4), 265–272.

Milner, R. (1978). Theory of type polymorphism in programming. *Journal of Computer and System Sciences 17*, 348–375.

Modi, J. J. (1990). *Parallel Algorithms and Matrix Computation.* Oxford, England: Oxford University Press.

Parsytec Computer GmbH (1993). *Parix1.2, Software Documentation.* Aachen, Germany: Parsytec Computer GmbH.

Patterson, D. A. and C. H. Sequin (1982). A VLSI RISC. *Computer 15*(9), 8–21.

Pepper, P. and M. Südholt (1996). Formulation and development of parallel numerical algorithms with data distribution algebras. In W. Mackens and S. Rump (Eds.), *Software Engineering im Scientific Computing*, pp. 164–169. Braunschweig, Germany: Vieweg.

Quinn, M. J. (1994). *Parallel Computing: Theory and Practice.* New York: McGraw-Hill.

Ritzdorf, H. and K. Stüben (1993). Adaptive multigrid on distributed memory computers. Technical Report 781, GMD, St. Augustin, Germany.

Robinson, J. A. (1965). A machine-oriented logic based on the resolution principle. *Journal of the ACM 12*(1), 23–41.

Röttger, M., U. P. Schroeder, and J. Simon (1994). Virtual topology library for Parix. Technical Report 148, University of Paderborn, Germany.

Schwarz, J. and F. A. Rabhi (1996). A skeleton-based implementation of iterative transformation algorithms using functional languages. In *Abstract Machine Models for Parallel and Distributed Computing*, pp. 119–133. Oxford, England: IOS Press.

Serrano, M. (1997). Inline expansion: *When* and *how*? In *Proceedings of PLILP '97*, Volume 1292 of *Lecture Notes in Computer Science*, pp. 143–157. Berlin: Springer-Verlag.

Serrano, M. and P. Weis (1995). Bigloo: a portable and optimizing compiler for strict functional languages. In *Proceedings of SAS '95*, Volume 983 of *Lecture Notes in Computer Science*, pp. 366–381. Berlin: Springer-Verlag.

Skillicorn, D. (1994). *Foundations of Parallel Programming.* Cambridge, England: Cambridge University Press.

Snir, M., S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra (1996). *MPI: The Complete Reference.* Cambridge, MA: MIT Press.

Stoltze, H. (1994). *Implementierung einer parallelen funktionalen Sprache mit algorithmischen Skeletten zur Lösung mathematisch-technischer Probleme.* Ph. D. thesis, Aachen University of Technology, Germany. (in German).

Stüben, K. (1982). MG01: A multigrid program to solve $\Delta U - c(x,y)U = f(x,y)$ (on Ω), $U = g(x,y)$ (on $\partial\Omega$) on non-rectangular bounded domains Ω. Technical Report 82.02.02, GMD, St. Augustin, Germany.

Stüben, K. and U. Trottenberg (1984). Multigrid methods: Fundamental algorithms, model problem analysis and applications. Technical Report 96, GMD, St. Augustin, Germany.

Trew, A. and G. Wilson (Eds.) (1991). *Past, Present, Parallel: A Survey of Available Parallel Computer Systems.* London: Springer-Verlag.

Valiant, L. G. (1990). A bridging model for parallel computation. *Communications of the ACM 33*(8), 103–111.

Wadler, P. (1990). Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science 73*, 231–248.

Wu, Q., A. J. Field, and P. H. J. Kelly (1997). M-Tree: A parallel abstract data type for block-irregular adaptive applications. In *Proceedings of Euro-Par '97*, Volume 1300 of *Lecture Notes in Computer Science*, pp. 638–649. Berlin: Springer-Verlag.

Zippel, R. E. (Ed.) (1990). *Computer Algebra and Parallelism (Proceedings)*, Volume 584 of *Lecture Notes in Computer Science*. Berlin: Springer-Verlag.

Lebenslauf

Name:	George Horaţiu Botorog
Geburtsdatum und -ort:	25. Februar 1967, Bukarest (Rumänien)
Familienstand:	verheiratet, 1 Kind
Staatsangehörigkeit:	rumänisch

Sept. 1973 – Juni 1985:	Grundschule und Gymnasium: Lyzeum Nr. 34, Bukarest
Juni 1985:	Ablegung der Abiturprüfung
Sept. 1986 – Sept. 1990:	Studium an dem Polytechnischen Institut Bukarest, Fakultät für Automatik und Computer
Okt. 1990 – Juli 1994:	Informatik-Studium an der RWTH Aachen, Abschluß Diplom
Okt. 1994 – Sept. 1997:	Promotion im Rahmen des Graduiertenkollegs *"Informatik und Technik"* an der RWTH Aachen
Okt. 1997 – Dez. 1997:	Wissenschaftlicher Mitarbeiter am Lehrstuhl für Informatik II der RWTH Aachen