Deriving Efficient Parallel Programs By Systematic Coarsing Specification Parallelism

Sergei Gorlatch *
University of Passau, D-94030 Passau, Germany
e-mail: gorlatch@fmi.uni-passau.de

Abstract

A general design methodology FOCUS is applied to parallel program development. The starting point of the development process is a mathematical description of a numerical algorithm. Two kinds of intermediate architecture-independent representations are used: an applicative one based on stream processing functions, and an imperative one based on an abstract SPMD-model of computation. The target parallel program is obtained as a result of consecutively reducing the parallelism of the original specification aiming at the real efficiency. The use of the approach is demonstrated on a real-life example: a two-dimensional integration on a sparse grid. Experimental results of the hypercube-implementation are presented.

Keywords: formal design methodology, multiprocessor architectures, numerical applications, parallel programming.

1 Motivation and Related Work

The aim of our research is to apply formal methods for parallel program specification and development. We take the approach in which the starting point in the program development is a problem-oriented, often non-procedural, formal specification of an algorithm. The specification describes *what* is to be done but not *how* it is to be done. Procedural aspects enter the development when the implementation is mapped to a language executable on existing processor networks.

In the present paper we continue the study started in [Gor92], trying to achieve the following features of the development process:

- the development starts with a mathematical specification which is familiar and natural for the expert in the actual application area;

*This work was done at the Technical University of Munich under the sponsorship of the Alexander von Humboldt Foundation and the SFB 0342 Project of the DFG (Germany)

- architecture-dependent design decisions are postponed until late in the development process; most decisions are made based on some abstract computational model;
- the abstract model, despite its architecture-independence, should provide the designer with realistic information (quantitative or qualitative) about the expected efficiency of the program under development.

The development of parallel programs from mathematical specifications is a very promising and actively investigated area. There are not only formal development methods (they are surveyed in [KLGG89]) but also supporting systems for them such as Model [TSSP85], Crystal [CCL89] and Suspense [RW89]. However, these methods and tools are usually targeted at some specific types of parallel architecture (systolic, shared-memory etc.), and work on regular problem domains (e.g. full homogeneous grids).

Our goal is to loose these restrictions. Firstly, we attempt to stay independent of the target architecture in the course of program development as long as possible. Secondly, we accept specifications defined on non-regular domains; as an example we consider a recently developed class of algorithms on so-called "sparse" grids.

The idea of our approach is to use some intermediate implementations of the program under development. In [Gor92] one such implementation in the applicative language AL was developed and optimized. In this paper we take the next step and consider an imperative implementation called abstract SPMD. While the former representation is used for extracting and formal analyzing the "maximal" parallelism of the specification, the latter one serves for obtaining the target parallel program which can be then efficiently implemented on real multiprocessor architectures.

Our approach can be outlined, based on a general design methodology for distributed systems, FOCUS ([BDD+92]), as follows. The main feature of the applications we are interested in is that they usually have a precise mathematical specification which we call a *requirement specification*. This specification does not have to be constructive (e.g.: "compute an integral for the function f in a given domain with a given accuracy"). To find corresponding algorithmic concepts and investigate their adequacy is the task of experts in numerical methods. They develop a *design specification* which we consider the starting point of the development. Such a specification is usually still mathematical, e.g.: a system of recursive equations determining relations between the matrix of coefficients, the vector of the right hand side and the solution vector in the Gauss method for solving linear systems. The aim of the development process is to transform such a specification into imperative parallel program. As an intermediate level we develop a so-called *abstract implementation* which is represented in a data-flow language called AL in FOCUS . The relation between the design specification and its abstract implementation is provided by the stream-based semantics of the AL language; AL-programs can be optimized using formal transformation rules. An abstract program should then be transformed into a *concrete program* for the target multiprocessor architecture.

Our work is an attempt to extend the FOCUS methodology into the direction of parallel program development for multiprocessor architectures from specifications of numerical algorithms. Our main concern are the development aspects, and not those of the verifica-

tion. We do not arrive at the concrete imperative program immediately from the FOCUS AL-representation. Instead, we use an additional stage to "coarsen" the maximal data-flow parallelism into realistic one. At this stage another kind of abstract implementation is used, we call it *abstract SPMD-program*. This implementation gives the program designer some information about program efficiency without specifying architectural details of implementation. We restrict ourselves to a qualitative analysis of efficiency here, leaving the more precise quantitative one for the future research.

In the paper, we firstly describe our example application area of sparse grid numerical algorithms [Zen90], present a formal specification for a sample algorithm and its AL-implementation from [Gor92]. Then an abstract SPMD-model is sketched. The development of an imperative parallel program from the AL-representation is described as a sequence of particular "coarsing" steps. We discuss how these steps influence the resulting program efficiency in terms of abstract SPMD-model. Finally, an experimental implementation on the iPSC/2 Intel multiprocessor and its results are briefly described.

2 Specification and Applicative Implementation

A new class of sparse grid algorithms is considered as an example application area for our approach. In this section, we outline the general idea of sparse grid algorithms, present an example design specification and describe briefly the results obtained in [Gor92] for such a specification.

Grids are called "sparse" because of their analogy to sparse matrices. For two-dimensional problems on the unit square with the degree of partition m (i.e. the boundary meshwidth 2^{-m}) the associated sparse grids contain only $O(m \log m)$ grid points instead of $O(m^2)$ for the usual "full" grids (see Figure 1).

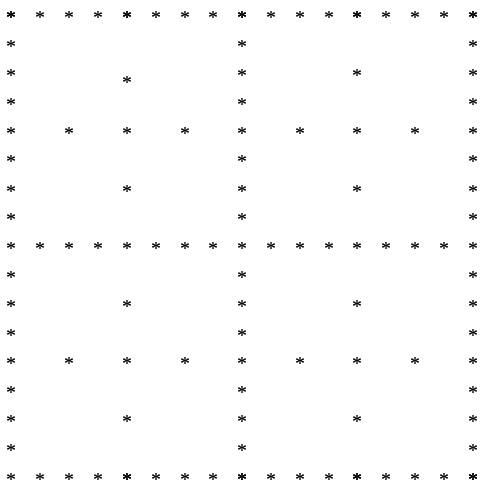

Figure 1: Points in the square sparse grid for m=4

It can be shown (see [Zen90]) that sufficiently smooth functions are represented on sparse

grids with nearly the same accuracy as on full grids. Thus, the main advantage of sparse grids is that the amount of necessary computations is reduced significantly, whereas the accuracy of approximation deteriorates only slightly. The same idea works even better in the multidimensional case and was successfully used for a variety of numerical grid methods ([Gri90],[Zen90]).

As an example of a sparse grid method we consider an algorithm for numerical two-dimensional integration. To simplify the presentation, we restrict ourselves to the non-adaptive version of the algorithm, which uses the meshwidth value m as a parameter.

The idea of the algorithm goes back to Archimedes and is based on domain partition. The value of the integral for a given function f, vanishing on the boundary, in the domain $[a1, b1] \times [a2, b2]$:

$$q = \int_{a1}^{b1} \int_{a2}^{b2} f(x1, x2) dx1 dx2$$

is approximated for the given meshwidth 2^{-m} as $q^{(m)} = A(a1, b1, a2, b2, m)$ where the function A is defined recursively using the auxiliary functions N and HB as follows:

$$\left.\begin{array}{rcl} A(a1,b1,a2,b2,m) & = & \text{if } m = 0 \text{ then } 0 \text{ else } A(a1, \frac{a1+b1}{2}, a2, b2, m-1) + \\ & & A(\frac{a1+b1}{2}, b1, a2, b2, m-1) + N(a1,b1,a2,b2,m) \\ N(a1,b1,a2,b2,m) & = & \text{if } m = 0 \text{ then } 0 \text{ else } N(a1,b1,a2,\frac{a2+b2}{2}, m-1) + \\ & & N(a1,b1,\frac{a2+b2}{2}, b2, m-1)) + HB(a1,b1,a2,b2) \\ HB(a1,b1,a2,b2) & = & Expr\{f(a1,a2), f(a1,b2), f(b1,a2), f(b1,b2), \\ & & f(\frac{a1+b1}{2}, a2), f(\frac{a1+b1}{2}, b2), f(a1, \frac{a2+b2}{2}), f(b1, \frac{a2+b2}{2}), \\ & & f(\frac{a1+b1}{2}, \frac{a2+b2}{2}), a1, b1, a2, b2\} \end{array}\right\} \quad (1)$$

For simplicity we use the informal notation $Expr$ reflecting just the values it depends on, rather than the precise expression for the function HB.

For specifications similar to (1) a precise computational semantics based on fixed-point theory can be constructed as in [PZ81]. This semantics can be used for proving correctness of an implementation with respect to the corresponding specification.

Now we present briefly the construction of an abstract program from the specification (1) (see [Gor92] for details). The first variant of an abstract implementation corresponds to classical data-flow programs [Den85]: the data-flow graph *Q1* (we call it a net) is presented in Figure 2. This net takes $a = (a1, b1, a2, b2, m)$ as an input and produces the value q as an output. It includes agents A, N and HB which correspond directly to the functions in (1); agent $S3$ sums up three values; agents IF implements condition in (1); the rest of agents: *G1, H1*, etc. prepare input tuples of values for other agents in the net.

A data-flow net can be described by a corresponding program in the applicative language AL (see [BDD+92]). Such a program includes all agent declarations and a system of equations describing their interconnections. The language AL has a formally defined semantics and semantically sound transformation rules ([Ded92]).

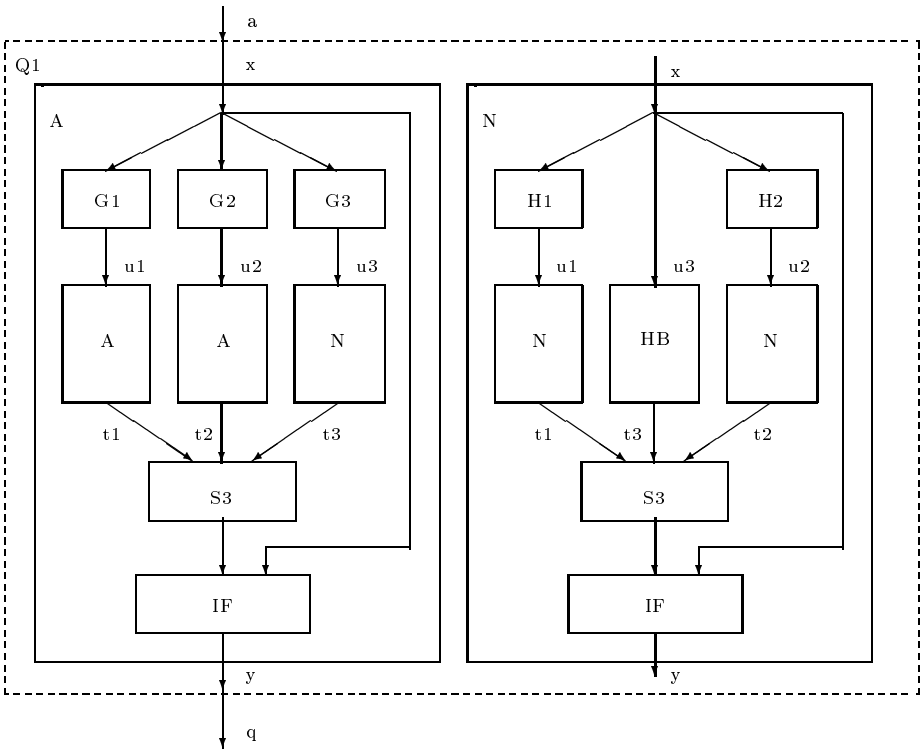

Figure 2: Data-flow net for the specification

Claim. The program Q1 implements the specification (1).

The proof is based on the formal semantics for specifications and AL-programs and on the definition of the implementation relation [Ded92].

The inner structure of some agents shown in Figure 2 is recursive (they use/call themselves). This means that agents can be unfolded leading to a number of different instantiations of the same agent working in parallel. The process of generating new agent instantiations in the data-flow program Q1 finishes when the current value of the recursion level m decreases to zero. Each agent begins to work as soon as the necessary input values are computed (by other agents) and have arrived at its input channels. E.g., agent *IF* can start to work without a value of its second input provided the fifth component of the first input is equal to zero (in this case it produces zero as output). Instantiations finish their work in the opposite order of their generation, because every instantiation depends on the output of the instantiation called by it.

3 Model of parallelism

In this section we present an SPMD-model as an abstraction of parallel machine.

Technological factors are now forcing a convergence towards systems formed by a collection of essentially powerful processors connected by a communication network. This organi-

zation characterizes most recent commercial massively parallel systems like Intel iPSC and Paragon, Thinking Machines CM-5, Transputer-based systems (e.g., GC from Parsytec) etc. We want our model to address these common features of modern systems and to supress machine-specific ones such as network topology and routing algorithm.

We do not restrict ourselves to any fixed programming paradigm. No single programming methodology has become clearly dominant yet: message-passing, data-parallel and shared-memory styles are all popular. The computational model should be applicable regardless of programming style.

The last but not least requirement on the model is its adequacy. One of the most widely used parallel models, the PRAM, assumes that all processors in a system work synchronously and that interprocessor communication is free. By exploiting these features some theoretically very fast algorithms can be developed, but they have often poor performance under more realistic assumptions.

We deliberately restrict ourselves to the well-known SPMD-structure of the program (Single Program Multiple Data). In this model processors perform computations on their local data, while executing asynchronously local copies of the same code. When data from other processors are required, processors perform communication operations. Communication is also used for processor synchronization.

We choose the SPMD-model for the following reasons:

- according to experts in numerical methods, more than a half of all parallel applications can be adequately represented by such a structure;
- the SPMD execution model is the basis for several modern parallel programming languages like *occam* and various parallel dialects of *C* and *Fortran* targeting massively parallel systems;
- the SPMD-structure can be efficiently implemented on most multiprocessor architectures using different parallel programming paradigms (shared memory, message passing etc.);
- this structure works well for our sample application domain.

The generally used definition of the SPMD is quite vague. On the one hand, it is possible to write an SPMD program which behaves step-by-step equally for all processors in the system, i.e., we obtain exactly the case of SIMD-program. On the other hand, the program may consist of a number of subroutines and an outer CASE-like statement which chooses a unique subroutine for each processor - it will then be general MIMD parallelism. What is usually referred to as SPMD, is in fact some "reasonable average" case of the general model. There is normally one node program, the variants of which for particular processors behave differently at run time due to the IF-THEN-ELSE statements where conditions depend on the coordinates of a processor in some communication structure.

We use a model inspired from [Gor83]. As we are interested only in a qualitative analysis, we discuss the main features and characteristics of the model without naming them or giving them particular values. There are several *processors* in the system running in parallel

asynchronously and communicating by *message passing*. We do not impose the explicit use of send-receive primitives in algorithms: e.g., shared-memory organization can be implemented through an implicit exchange of messages, so our model qualifies also for it. We suppose that the processors in the system are *homogeneous*, i.e. they all have equal characteristics. One of these is the *computation cost*, i.e. the average time used by a processor to execute one operation on its local data.

The program may dynamically create new processes. According to our understanding of the SPMD-model, a process creation means that just one more instantiation of the code starts to run. In the model this takes a special amount of time which we call *creation time*.

As already stated, we do not take care about the particular organization of the communication network in our idealized system, but we take into account its main characteristics. Firstly, we should consider the *delay*, i.e., the time it takes for the system to transmit a message from its source to its destination. Recent developments in communication make it possible to simplify this characteristic compared with our previous considerations in [Gor83]. E.g., in the Paragon, the delay is almost independent of the physical distance between processors in the network. We assume therefore that the delay is a random value in some interval.

In addition to the delay, we consider such characteristic of communication as *start-stop time*. The matter is, that in modern processors, computations can to some extent overlap with communications: the start-stop time is exactly an amount of time the processor is busy with communication and cannot do any other useful work. When the interconnection network is operating within its capacity, the time to transmit a message includes two start-stop times (at the sender and the receiver) and one delay time. The ability of the network to maintain only a limited number of connections simultaneously is expressed by the *network bandwidth*.

We demonstrate below how the presented model may be used in a qualitative sense to guide the parallel program development process.

4 Parallel Program Development

The data-flow implementation obtained in Section 2 is known to be a maximal parallel one: each action is executed as soon as data for it is ready. Our aim in the course of program development will be to harness this usually fine-grained parallelism and to derive a target program which can be implemented efficiently within our SPMD-model.

4.1 Elementary and Complex Agents

The abstract implementation (net $Q1$) is derived quite straightforwardly from the specification: an agent corresponds to an operation or function call in the original specification. The only choice being made is whether some agent is represented as elementary (without inner structure) or as a complex agent whose structure may include other agents, recursively as well.

In fact, by this choice we make one very important decision – we restrict the granularity of parallelism. Representing, e.g., *HB* as an elementary agent, we assume that this agent will be considered further as an entity not eligible for the parallelization. It means that we will neither try to parallelize the computation of the expression $Expr$ nor try to evaluate function f: to do so we would need their inner structure which is hidden from now on.

The choice described above is the first step in reducing the potential parallelism of the initial specification aiming at systems with powerful processors and therefore coarse-grained parallelism.

The AL–program and the corresponding network contain now all the parallelism of the chosen degree of granularity, inherently contained in the original specification of the algorithm. This parallelism consists of the simultaneous execution of several agent-instantiations. Whereas every instantiation of an elementary agent is executed sequentially, an instantiation of a complex agent has two kinds of parallelism: 1) an agent may have parallel structure implying parallel execution of its parts; 2) if there are recursive calls of agents then new (independent) instantiations are created that once again may have parallel structure with new agent calls.

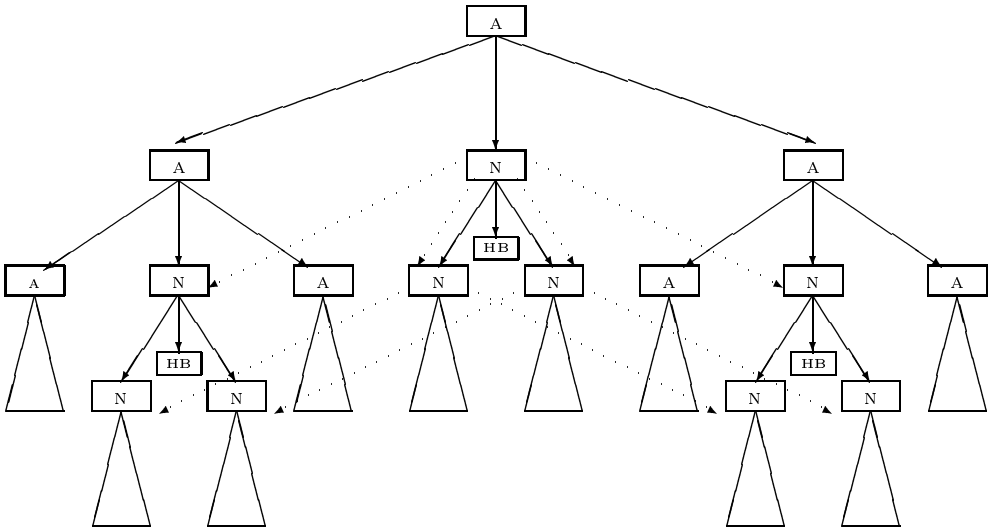

Figure 3: The tree of recursive calls (fragment)

For net Q1 we have a tree of recursive calls, an initial fragment of which is shown in Figure 3. Any node corresponds to one instantiation of agents A, N or *HB*. Instantiations of other agents are omitted to simplify the figure. A solid edge from one node to another means that the latter instantiation is called by the first one. Subtrees are depicted as triangles. Dotted lines in Figure 3 do not directly belong to the tree of agent-calls; they illustrate the following point. In our example all necessary values of the function f are computed in each instantiation of the agent N (more precisely, in the agent *HB* within the corresponding instantiation of N); some of these values are computed repetitively. Repetitive computations can be eliminated by means of introducing additional communications between

N-instantiations (see [Gor92]); these communications are shown in Figure 3 as dotted lines.

4.2 Global and Local Agents

The first straightforward variant of a procedural implementation of network $Q1$ can be obtained if exactly one process for each agent-instantiation is created. Direct implementation of such variant leads to the following two-stages structure of the program. At the first stage all complex agents are being "executed". In fact this execution means only the creation of processes for every sub-agent, some of which may again be complex and should be unfolded too. As a result of the first stage we get a set of processes; at the second stage they should then be mapped to the processors in the system and synchronized in accordance with their interdependencies. There are two difficulties here: 1) the number of processes grows exponentially; 2) the processes are non-homogeneous concerning their amount of computations and thus cannot be easily synchronized. From the viewpoint of efficiency, the creation of new processes is a pure control-overhead. The useful work is done in this case only at the second stage of the program which needs additional efforts in synchronization (these efforts are of overhead-nature too).

Therefore, although theoretically we have preserved all the previously existed parallelism, it is easy to see that the real efficiency of the parallel program will be poor. Such naive implementations often yield parallel programs that may work slower than sequential ones.

One possible solution in this case is to further reduce the number of parallel processes created, i.e., to use coarser granularity.

We will distinguish two types of agents: local agents and global agents. The difference is in their implementation. Every instance of a global agent means creation of a new parallel process whereas local agent instances are executed as conventional procedure calls, i.e., sequentially within the process where they belong. If we decide, e.g., that the agent A is global and all others (N, $S3$, HB) are local ones then we obtain the situation shown in Figure 4: agent A includes now not only fine-grained agents $G1$, $S3$ etc., but a coarse-grained recursive agent N. It means that the whole execution of N including all its recursive calls is now within one agent, i.e., N is organized as one process.

4.3 Sequentializing

The next step in the parallel program development considers the control flow within one process. We can see that both agents A and N are potentially parallel programs: they have "forks", i.e., parts which can be executed in parallel.

Such parallel execution can be organized on many contemporary multiprocessor systems as multithreading in one physical processor. In practice however, this technique is limited by the available communication bandwidth and by the overhead involved in context switching. Furthermore, in our example, we may decrease the efficiency using multiple threads. The reason is that three potential threads have different weights (see Figure 4). Two of them include global agent A, i.e., the creation of a new process, and the third one contains only local agents. This means that sharing time equally between these three threads will lead to

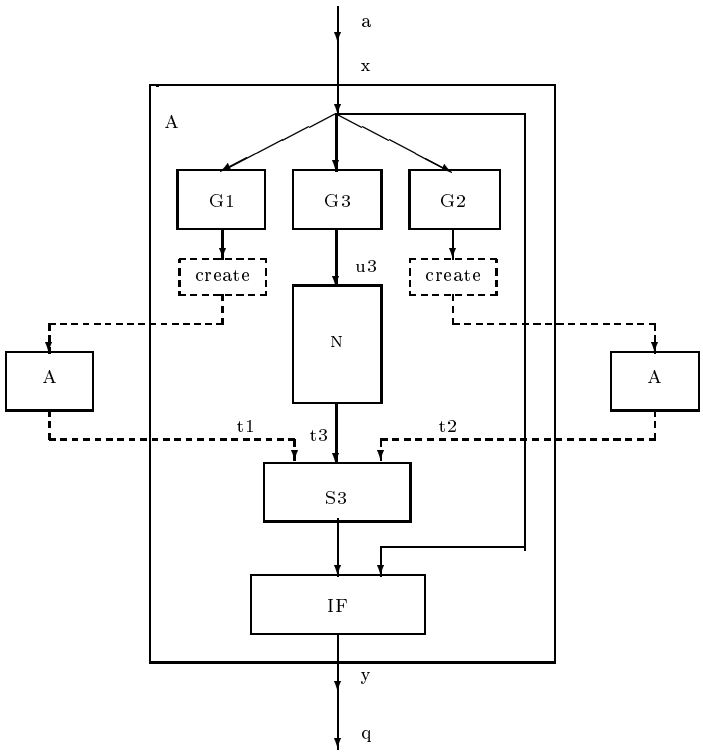

Figure 4: Global and local agents

a delay in creating new parallel processes ready for execution whereas some processors in the system may be idle.

This motivates our next transformation which consists of the "sequentializing" the potentially parallel processes of an agent. We do not use the "fork" of Figure 4, but the structure shown in Figure 5. It consists in calling firstly the global agents (thus creating two new parallel processes) and then doing the inner work, i.e., computing N sequentially.

4.4 Load-balancing

Our program looks now much more "modest" with respect to the use of resources, but nevertheless it still needs a potentially unrestricted number of processors for executing all created processes A. In practice we have always some restricted set of processors and therefore we have to decide how to map the whole set of created parallel processes onto them.

There are two typical approaches to this problem: the universal and the problem-oriented one; they may be either statical or dynamical. Here we briefly analyze them with respect to our case study.

The first approach suggests the use of some universal load-balancing mechanisms implemented in the operating system. It means in our case that we create all possible parallel processes (remember that they are already quite coarse-grained) but we do not take care about their mapping onto processors - the operating system does it. Such an approach looks

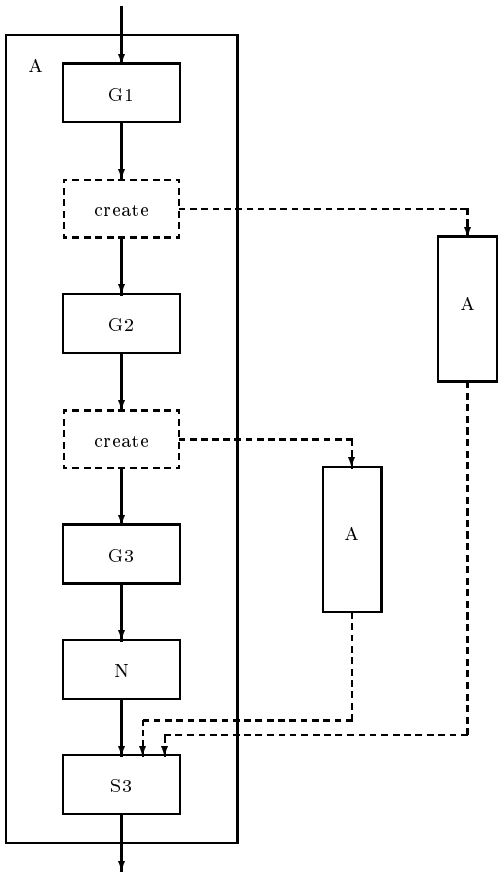

Figure 5: Sequentializing

very attractive for the user who is liberated now from serious technical problems. However, it does not guarantee the efficiency: an algorithm used by the operating system must be general by nature and cannot take into account all the peculiarities of a particular algorithm.

An extreme alternative approach consists in creating exactly as many processes, as there are available processors (static load-balancing). The following simple rule can be used here: the two global calls in each process are implemented in two other processes as long as free processors are available. The processors running the programs on the leaves of the recursive tree (Figure 3) implement both global calls and the local one themselves. So all remaining levels of recursion are implemented in these processors. The parallel program in this case behaves exactly in accordance with the SPMD-concept: the programs in the processors are identical, and the particular mode of execution depends on the position of the processor in the tree.

However, the efficiency of the program may be poor because of this too simple balancing: the computation load is distributed non-homogeneously, and some processors waste a lot of time without work.

In this case we can use more complicated load-balancing. The idea is, that if the process

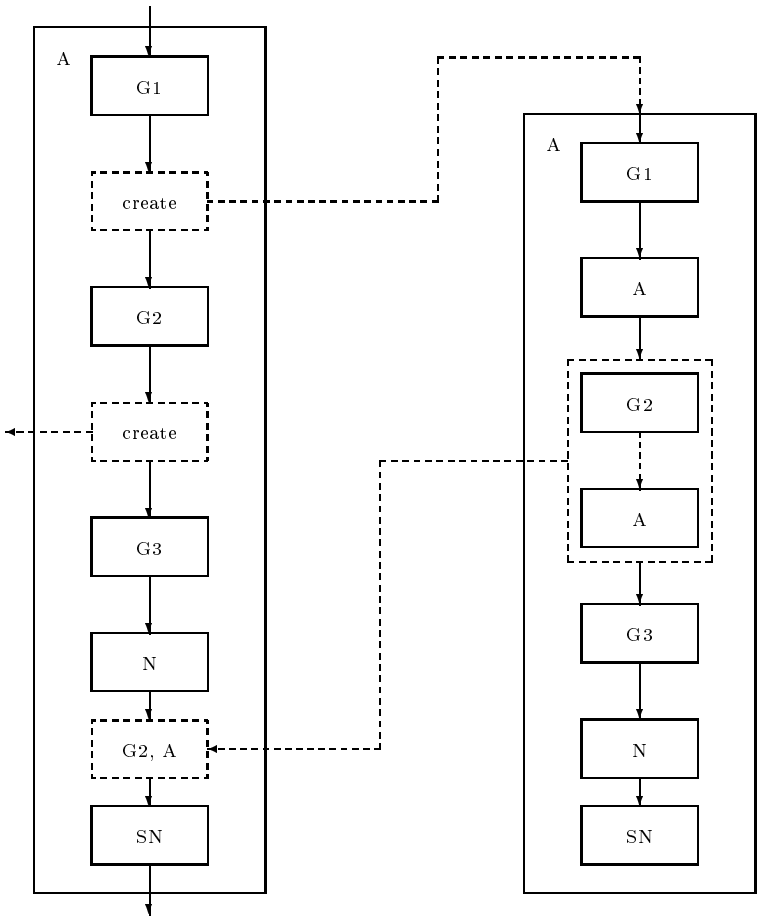

Figure 6: Load-balancing - example

has no "child" whom it could send a global call for execution, then the process postpones this global call (and eventually also the second one) and begins to execute the local call. The postponed calls are then redistributed between processors which become free. An example of such an organization is shown in Figure 6 where the implementation of the agents $G2$ and A is redistributed to the "left" agent after it has finished its N-instantiation. The synchronization needed can be implemented, e.g., by a semaphore. Instead of agent $S3$ we use here agent SN that sums up a variable number of values.

Another decision on load-balancing may be to reconsider the "locality" of agents. We could declare the agent N to be conditionally-local, i.e., it is considered to be local or global depending on the value of actual parameters. The most natural way to do it for our example is to make the decision about locality dependent on the depth of recursion: the instantiation $N(a, b, c, d, m)$ is global if $m \geq M$ and local otherwise, where M is some fixed value depending on the processor number in the system.

5 Implementation and Experimental Results

The ultimate goal of the development process is to obtaine an efficient parallel program for a target multiprocessor. For our experiments we used Intel Hypercube iPSC/2 under the MMK operating system [TATS90]. The MMK offers a transparent multitasking process model, which means that the programmer can define multiple parallel processes. The communication between tasks is realized with mailboxes supporting a broad spectrum of varying communication semantics. The MMK programming model is claimed to be object-oriented: the programmer has active objects (tasks), communication objects (mailboxes), synchronization objects (semaphores) and storage objects (memory). All objects can be dynamically created and deleted. We present here for simplicity the version of implementation with the

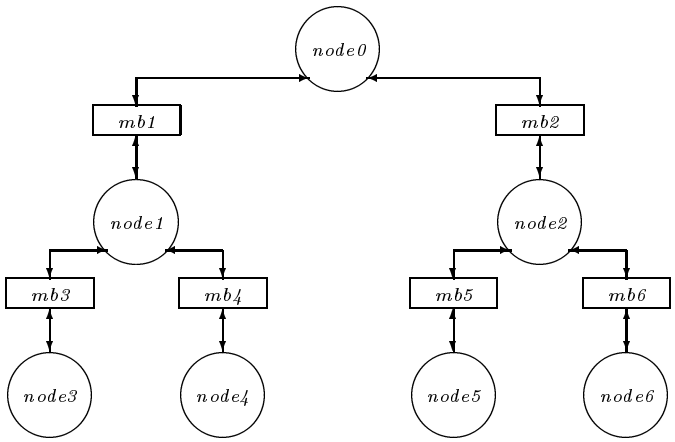

Figure 7: The structure of parallel program

static load-balancing and synchronous communication. Following the SPMD-model, we have a number of identical node-programs (see Figure 7). In fact, we do not use the costly dynamic process creation, we rather load the processes (programs) into all processors at the initial phase. The root-program (in terms of the recursive tree) starts immediately, others wait for the message from the father-node with the values of actual parameters. After receiving them, the node-program tries to call both agents A in its body by sending corresponding parameters to its sons if any, and then executes agent N. If there are no sons, i.e. all processors in the system are already busy, the node-program executes its body sequentially. After finishing its body, the node-program receives the intermediate results from its sons, sums them up with its own result and sends the sum to its father. The sum in the root-program is the final result of the whole program. Programs communicate with each other synchronously.

The structure of the node-program is presented here using simplified C-syntax. The program consists of the main function and several other functions: *A, N, HB* and *S3*, that correspond directly to the agents in program $Q1$. The main program uses an input mailbox and an output one, e.g., mailbox $mb5$ is an input for *node* 5 and the first output of *node* 2 (see Figure 7).

```
main (mbin, mbout1, mbout2)
 {
   RECV (mbin, par);
   if (2*i+1 < p) { SEND (mbout1, G1(par)); c1 = 1 };
   if (2*i+2 < p) { SEND (mbout2, G2(par)); c2 = 1 };
   t3 = N(G3(par));
   if (c1==1) RECV (mbout1, t1) else t1 = A(G1(par));
   if (c2==1) RECV (mbout2, t2) else t2 = A(G2(par));
   y = S3(t1, t2, t3);
   SEND (mbin, t);
  }
A (par) { . . . };
N (par) { . . . };
HB (par) { . . . };
S3 (par) { . . . };
```

The communication statements `SEND` and `RECV` have two arguments: a name of a mailbox and a name of the variable whose value is to be sent or received. Variable *par* represents channel x from program $Q1$, i is a number of the node and p is the number of processors.

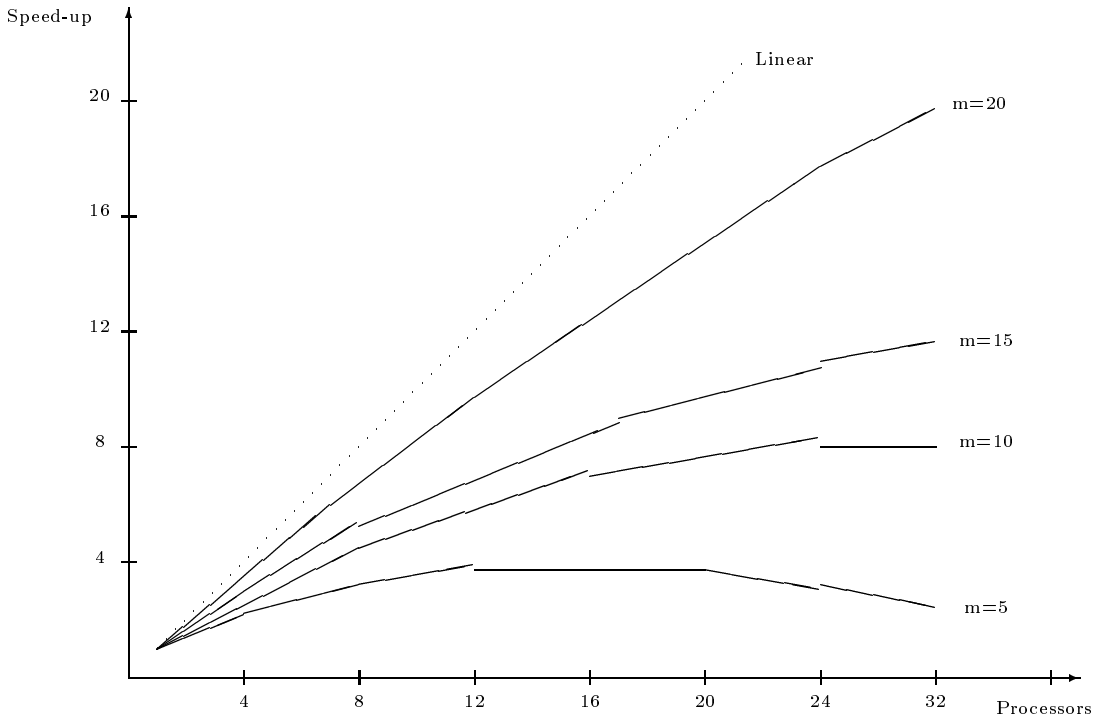

Figure 8: Experimental results

The experiments on iPSC/2 aimed at measuring the speed-up of the parallel implementation in comparison with the sequential one. As a sequential variant the original SPMD-

program for one processor (without any communications) was used. The dependence of the speed-up on the number of processors for different values of recursion depth m is shown in Figure 8. Dotted line corresponds to the so-called linear speed-up which is equal to the system's processors number p.

6 Conclusion and Future Work

In the paper we have shown how a systematic methodology can be used for parallel program development aiming at a broad class of multiprocessor architectures in From the initial mathematical specification we have arrived in a stepwise manner at an efficient parallel implementation.

The gains of our development approach are as follows:

- the design specification is formulated at a level of abstraction which is very close to a mathematical description; specification development and its validation with respect to requirements are left to the expert user;
- correctness and performance issues are addressed without specifying architectural details of an implementation;
- the original parallelism of the specification is transformed into an efficient imperative parallel program;
- the development process consists of a set of particular implementation steps (choice of elementary agents, determining agents locality, sequentializing, load-balancing) wich influence the target program efficiency.

Experiments on a hypercube show that efficient parallel implementation must pay attention to both computational aspects (the total amount of work done and load-balancing across processors) and communication aspects (proper synchronization). The development strategy should therefore coordinate work assignment with data placement, provide a balanced communication schedule, and overlap communication with processing.

We are working now at generalizing the methodology for a broad class of specifications and also at providing our abstract SPMD-model with some cost-calculus for quantitative estimating the efficiency of the target parallel program.

7 Acknowledgements

I am grateful to the Alexander von Humboldt Foundation whose grant made possible my work in Munich. I am very obliged to Prof. Manfred Broy and Prof. Christoph Zenger who initiated and friendly encouraged this research. Many thanks to Max Fuchs, Ketil Stoelen, Thomas Bonk and other colleagues from the SFB 0342 project for their help.

References

[BDD+92] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. F. Gritzner, and R.Weber. The design of distributed systems - an introduction to FOCUS. Technical Report SFB-Nr. 342/2/92, Techn. Univ. Muenchen, January 1992.

[CCL89] M. Chen, Y. Choo, and J. Li. Theory and pragmatics of compiling efficient parallel code. Technical Report TR-760, Yale University, 1989.

[Ded92] F. Dederichs. Transformation verteilter systeme: Von applikativen zu prozeduralen darstellungen. SFB-Nr. 342/17/92 A, Techn. Univ. Muenchen, January 1992.

[Den85] J. Dennis. Data flow computation. In M. Broy, editor, *Control Flow and Data Flow*, volume 14 of *NATO ASI Series F: Computer and System Sciences*, pages 346–397. Springer, 1985.

[Gor83] S. Gorlatch. Macroconveyor asynchronous computation of structural functions. *Cybernetics*, (5):41–46, 1983.

[Gor92] S. Gorlatch. Parallel program development for a recursive numerical algorithm: a case study. SFB-Bericht Nr. 342/7/92a, Technische Universitaet Muenchen, March 1992.

[Gri90] M. Griebel. A parallelizable and vectorizable multi-level algorithm on sparse grids. Technical report, Techn. Univ. Muenchen, October 1990.

[KLGG89] Yu. Kapitonova, A. Letichevsky, S. Gorlatch, and G. Gorlatch. The use of the equations over data structures for program specification and synthesis. *Cybernetics*, (1):22–35, 1989.

[PZ81] A. Pnueli and R. Zarhi. Realizing an equational specification. *Lect.Notes Comp.Sci.*, 115:459–478, 1981.

[RW89] Th. Ruppelt and G. Wirtz. Automatic transformation of high-level specifications into parallel programs. *Parallel Computing*, 10:15–28, 1989.

[TATS90] T.Bemmerl, A.Bode, T.Ludwig, and S.Tritscher. Mmk - multiprocessor multitasking kernel. Technical report, Techn. Univ. Munich, December 1990.

[TSSP85] J. Tseng, B. Szymanski, Y. Shi, and N. Prywes. Real-time software cycle with the Model system. *IEEE Trans. Software Engin.*, 11:1136–1140, 1985.

[Zen90] Chr. Zenger. Sparse grids. Technical Report SFB-Nr. 342/18/90 A, Techn. Univ. Muenchen, October 1990.