Software Engineering Methods for Designing Parallel and Distributed Applications from Sequential Programs in Scientific Computing

Peter Luksch
Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR-TUM)
Institut für Informatik, Technische Universität München (TUM)
D-80290 München
e-mail: luksch@informatik.tu-muenchen.de
WWW: http://wwwbode.informatik.tu-muenchen.de/

Abstract

Although quite a few large scale applications have been ported to multiprocessors in recent years, no framework has yet been set up for the process of designing parallel and distributed applications from sequential programs that takes into account the specific needs of large scale scientific applications where programs have complex, irregular control flow and data structures. This paper proposes such a framework and reports on its application in the portable parallelization of the industrial Computational Fluid Dynamics (CFD) software package TfC (TASCflow for CAD) within the scope of an interdisciplinary research project.

1. Introduction

There is no doubt that parallel processing is necessary to meet the computational requirements of grand challenge applications like CFD. Having been restricted mainly to research laboratories for a long time, parallel processing is now gaining increased attention in software industry as standards have become available that allow to produce portable software for parallel and distributed hardware platforms that range from low-cost networks of workstations (NOWs) to high-performance massively parallel processors (MPPs).

While standards like PVM [2] and MPI [3] have been established for writing portable message passing programs, there is not yet a framework for the software design process in parallel and distributed computing that can be generalized to at least a certain class of applications. Quite a few research projects have been completed in recent years, in which applications from different domains of Scientific Computing have been parallelized, e.g. within the scope of the EC[1] funded EUROPORT initiative. Most projects however did not address software engineering explicitly and did not aim at generalizing their experiences to a wider range of applications.

Lack of software engineering methods for parallel and distributed software design is one important reason why productivity in developing parallel programs is so low compared to the design of sequential software. Setting up a framework for the design process that is applicable to at least a certain class of applications in Scientific Computing is the central objective of SEMPA. SEMPA (Software Engineering Methods for Parallel and distributed Applications in Scientific Computing) is an interdisciplinary research project which is funded by the BMBF[2] and which brings together computer scientists, mechanical engineers, and numerical analysts from industry and academia (see table 1).

Methods that are to be applicable in practice must be developed in a "real world" environment. Therefore, in SEMPA, the parallelization of ASC's CFD code TfC serves as a case study for the definition and evaluation of the methods developed in the project. The software engineering methods as well as the framework in which they are applied are developed in an evolutionary process. Practical experience will motivate modifications of existing methods or the definition of new ones which are subsequently evaluated in practice. Therefore the framework will evolve dynamically throughout the project. Its initial version, which is motivated by LRR-TUM's experience form previous parallelization projects in Scientific Computing and by ASC's software engineering (SWE) guidelines [10]. In section 3, its application for the design and implementation of a parallel version of TfC, ParTfC, is considered in more detail. Portability and efficient execution on a wide range of hardware platforms are central requirements in the design of ParTfC.

Making efficient use of NOWs for parallel production

[1] European Community, now European Union (EU)

[2] German Federal Department of Education, Science, Research, and Technology (*Bundesministerium für Bildung, Wissenschaft, Forschung und Technologie*)

1060-3425/97 $10.00 © 1997 IEEE

partner	research domain, product
LRR-TUM	computer science: parallel and distributed architectures and applications
Advanced Scientific Computing (ASC)	mechanical engineering, product: TASCflow (3d CFD code)
Institut für Computer Anwendungen (ICA III), Universität Stuttgart	numerical analysis, especially adaptive multigrid methods
GENIAS Software GmbH	computer science, distributor of software for parallel and distributed systems: among other things CODINE, a batch queueing system for NOWs

Table 1: Project SEMPA: partners

runs is the third major project objective. A resource management system is being developed that manages the coexistence of interactive users and parallel batch jobs in NOWs. The parallel version of TfC will be used as a test case for the development and evaluation of the resource manager. In this paper we focus on the software engineering aspects of SEMPA.

2. A Framework for the Design of Parallel and Distributed Software in Scientific Computing

In Scientific Computing, parallel and distributed software is rarely developed from scratch. Commonly, projects start from a monoprocessor implementation of an algorithm that represents the experience of (possibly several generations of) applications experts and programmers. Therefore, we address particularly the problem of designing a parallel version of an application based on the sequential implementation. As already mentioned, we focus on complex software packages with irregular control and data structures.

Our experience has shown that for these types of applications, a parallel version cannot be generated directly from the sequential code by means of automatic or interactive parallelization tools, such as FORGE [6]. These tools can, however, support the human analyst in detecting control flow and data dependencies. Data parallel languages like High Performance Fortran (HPF) which support SPMD style parallelism can be used successfully only if a program has regular data structures, which is not the case with most applications that deal with large sparse matrices like TfC.

Parallel program design in the problem domains considered here thus will remain the task of the software engineer. Figure 1 illustrates the framework for this design process as it is proposed and used in SEMPA. Its phases will be discussed in more detail in the subsequent sections.

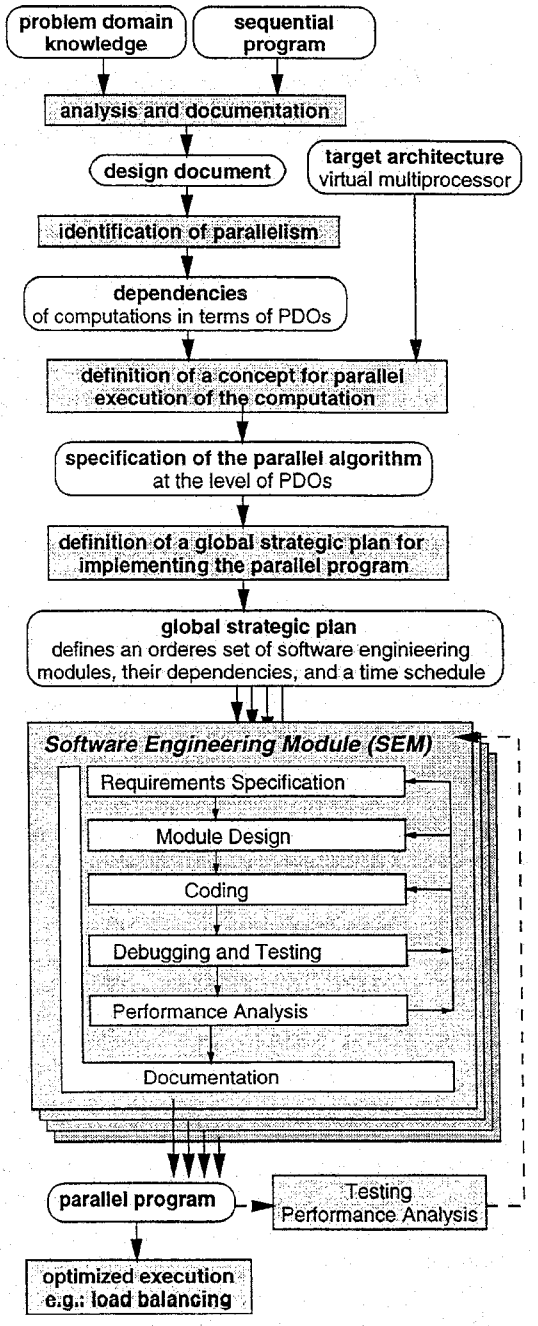


Figure 1: The design process for parallelizing applications in Scientific Computing

2.1. Analyzing the Algorithm and its Sequential Implementation

As the monoprocessor implementation usually has a long history of development and optimization, it is typically written in FORTRAN 77. The amount and quality of the available documentation depends on the individual case. Often, documentation is incomplete, the source code is not well structured, and comments are rare. Therefore, the first step is to acquire and document the know-how that is necessary to define a concept for parallel execution of the computation. The basic outline of the algorithms used in Scientific Computing applications, in most cases, is documented in some publication. Many details, however, have to be deduced from the implementation.

Computer scientists, who are usually not experts in the problem domain, have to acquire a basic intuitive understanding of the problem that is solved by the program they are to port to a multiprocessor. Otherwise it is very difficult to analyze its control and data structures and to set up a promising concept for its parallel implementation.

The result of the analysis phase is documented in a design document that is to serve as a reference throughout the project. Its main purpose is to summarize the key concepts of the computational model in a language that is understood by computer scientists. It has to provide a comprehensive and concise description of the algorithm at the abstraction level used in the problem domain. This task is similar to object oriented analysis. In contrast to business applications, applications in Scientific Computing are based on mathematical models that provide a formalism at an appropriate level of abstraction. The model implies a number of objects and relations between them. For instance, in a CFD simulation the following objects can be identified: the *grid*, which consists of *nodes*, *finite volumes* and *elements*, etc. Specifying the program's behavior at the level of these **problem domain objects (PDOs)** provides a good basis for communication between computer scientists and problem domain experts in interdisciplinary projects.

In SEMPA, the analysis of TfC was organized as a series of joint meetings, where ASC's mechanical engineers explained the concepts behind TfC and their implementation to computer scientists from LRR-TUM. Documentation covering all aspects that were judged to be of relevance for the project was written at LRR-TUM and reviewed by ASC. This procedure proved to be an efficient way of capturing the know-how needed to define a strategy for parallelizing TfC. The result has been a comprehensive design document; its structure is summarized in the appendix.

2.2. The Virtual Multiprocessor

The Virtual Multiprocessor (VMP) is an abstract representation of the target systems considered in our project. It extends the concept of the Virtual Machine (introduced by PVM [2]) by the concept of symmetric multiprocessing. Like in the Virtual Machine, a number of processing nodes (PN) are virtually fully connected by a network that can be a local or a wide area network, or a high-performance interconnection network, which is typically found in MPPs. The VMP can be an MPP or a NOW, which may be heterogeneous, i.e. processing nodes may be workstations from different vendors.

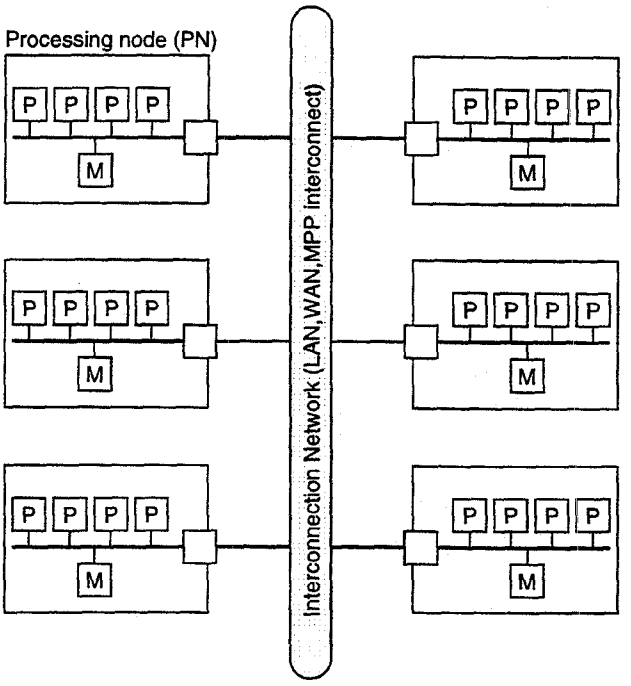


Figure 2: The Virtual Multiprocessor Architecture

The processing nodes themselves can be monoprocessors or symmetric multiprocessors. In a symmetric multiprocessor, a number of processing elements (CPUs) have access to a shared memory. Most workstation vendors already offer high-end systems that implement symmetric multiprocessing. The VMP architecture is illustrated in Figure 2. It offers two levels of parallelism:
inter-PN (distributed memory): Parallelism is implemented explicitly at the process level by message passing.
intra-PN (shared memory): Parallelism within a PN is implementing by multi-threaded program execution.

2.3. Definition of a concept for parallel execution of the computation

Based on the result of the analysis phase, a concept for parallel execution of the computation on the virtual multiprocessor is set up. In Scientific Computing, data distribution (also referred to as SPMD[3] parallelism) is the

[3]Single Program Multiple Data

dominant paradigm for parallel execution, because computational demands for most problems in this application domain scale up proportionally to the size of the problem description, which is fed into the program as input data. Since symmetric multiprocessing is limited with respect to the number of processors, scalability for this type of application can only be achieved with distributed memory systems.

In the SPMD model, the problem description is divided into a number of partitions. Code for communication and synchronization is integrated into the sequential program. The resulting code is executed on each processing node of the virtual multiprocessor. Each of these "replicated worker" processes has one partition for which it computes a solution.

Parallel execution of the computation under consideration is specified in terms of problem domain related objects (PDOs) and their relations to each other. The first step is to find a partitioning of the problem description. Data distribution essentially is a graph partitioning problem. It has to be decided which type of objects form the nodes of the graph to be partitioned. Next, a relation between these objects has to be selected to determine the edges of that graph. Common examples for such a relation are "is connected to" or "directly influences". The decision about the nodes and edges of that graph is specific to the application under consideration and is made based on the knowledge obtained during the analysis phase.

As the problem of graph partitioning is NP complete, an analytic solution is infeasible for problems of relevant size. Since the problem occurs in many application areas, a number of heuristics have been proposed that attempt to determine partitions of almost equal size while minimizing the number of cut edges. Several software packages for graph partitioning are available that can be integrated into parallel applications (for an overview, see [8]). For heterogeneous systems, where processing nodes differ in computational power, the first optimization goal has to be modified. Partition sizes should reflect the relative computational power of the processing nodes on which they are to be processed.

Having determined the partitioning of the PDOs that have been chosen as the basis for data distribution, the data structures associated to them have to be distributed appropriately. Next, synchronization points have to be defined, where information is exchanged across partition boundaries. This task again is specific to the application under consideration. As an example, the concept for parallelizing TfC is described in section 3.

If the processing nodes of the VMP are monoprocessors, the replicated worker processes are implemented as monolithic sequential processes. If, however, processing nodes have a SMP architecture, intra-PN parallelism can be exploited as a second level of parallel processing. Independent subtasks are implemented as threads that execute concurrently in a shared address space. For instance, in TfC a number of equations (e.g. for temperature, enthalpy etc.) are solved independently of each other. Each of them could be solved by one thread. Inter- and intra-PN parallelism can be considered relatively independent of each other. With TfC, inter-PN parallelism is implemented first; for the time being, intra-PN parallelism is considered conceptually only.

The result of the phase of the design process described in this section is a specification of the parallel program in terms of PDOs and relations between them. For TfC this specification has been written in pseudo-code. Alternatively, formal specification languages could be used, for which tool support exists (e.g. for performance prediction or automatic generation of program templates). The main purpose of the specification is to define precisely the interaction between the processes of the parallel program. In SEMPA we use pseudo-code because it is intuitively clear to the engineers in the project and thus provides an appropriate basis for communication in our interdisciplinary project.

The specification should not only describe the parallelization concept as it has been finally agreed on but should also document relevant decisions taken during its definition in order to avoid that a seemingly appealing solution that actually proved to have some drawbacks is reconsidered again later on in the project.

2.4. Definition of a Global Strategic Plan

In the global strategic plan (GSP), the requirements of the project as a whole are defined. Some fundamental design decisions have to be taken when defining a "roadmap" for future activities. The work then is decomposed into pieces of manageable complexity, which are referred to as software engineering modules (SEMs) in SEMPA. A SEM is characterized by its result according to the requirements definition. The result of a SEM represents a milestone in the GSP. Dependencies of SEM's on each other establish an ordered sequence of activities. A coarse grained time axis may be added to the GSP as a means of measuring progress. There is however no point in defining a fine-grained time schedule throughout the whole GSP, since the GSP is very likely to be subject to updates and revisions, especially if the project is research oriented. Time schedules in terms of weeks will be used only within SEMs. Regular meetings of all project members are held in which the GSP is reviewed and updated or revised as needed.

Figure 3 shows the GSP for the parallel implementation of TfC. The objective of this project is to develop a prototype of a portable parallel version of TfC executing on

NOWs as well as MPPs. Besides portability, modularity, extensibility and maintainability are important objectives. Since support for achieving these objectives is limited in FORTRAN 77, the language in which TfC is written, evaluation of newer languages, especially object-oriented features is a natural requirement.

One fundamental decision that can be deduced from Figure 3 is the fact that evaluation of new languages starts independently from parallelization of TfC. The motivation behind this decision is that re-implementing a complex software packages like TfC before starting parallelization was considered infeasible given the time and manpower restrictions imposed by the funding institution. Therefore we decided to use the Algebraic Multigrid (AMG) solver [7] as a case study for evaluating new languages. The solver, which is a module of reasonable complexity but manageable size, is being re-implemented in C++ making use of the language's object-oriented capabilities. A Fortran 90 implementation of AMG is the basis for evaluating the potential of High Performance Fortran (HPF). The third track in the GSP diagram, labeled "resource management and load balancing", is not directly related to the parallel implementation of TfC but is another major part of SEMPA.

2.5. Software Engineering Modules

A Software Engineering Module (SEM) is a package of work that typically takes a couple of weeks or a few months to complete. A SEM iteratively runs through a number of phases, as explained below. Each of these phases is an iterative process for itself as its result is subject to a formal review. Depending on the result of the review, some rework may be necessary which again will be reviewed. The reviewers usually should not have been involved in the implementation of the task whose results are to be reviewed. Careful review of each step in the design is of particular importance in interdisciplinary projects as it helps to identify misconceptions as early as possible.

The **requirements specification** phase defines the function of the software to be developed, i.e. it states *what* the SW is to do, not *how* this is achieved. In addition, user interface and system requirements are stated as well as performance requirements. All requirements have to be quantified and must be measurable; they are assigned priorities according to [4] (essential, conditional, optional).

In the **module design phase**, a solution strategy is developed and the software architecture is defined. Alternative approaches are considered and compared to each other; the one that is being adopted is outlined in an algorithmic specification (written in pseudo-code). The decision process must be carefully documented for future reference. The specification is implemented in the **coding** phase following the coding style guidelines agreed on in the project.

If coding is complete, debugging and testing will introduce some iterations in the design cycle. The search for programming errors is supported by debugging tools. A suite of test cases that ideally has been defined in the requirements definition phase is used to validate the parallel computation by comparing its results to the corresponding output of a run of the sequential program. Errors located in this phase mostly can be fixed by some recoding. More severe errors require a revision of the design or even the requirements specification. This type of iteration of couse should occur extremely rarely in a good design process.

Performance Evaluation is the final step in implementation of a SEM. It can be achieved by estimation techniques that predict performance parameters before running the program or, which is much more common, by profiling program execution using a performance analysis tool. A specific test suite for use in performance evaluation should have been defined in the requirements specifications.

Documentation is generated in each phase of the design cycle. It can be divided into user manuals, design documents, test and performance evaluation reports. Upon completion of all SEMs the parallel program is completed and tested as a whole. Performance data is gathered from program executions on a number of hardware platforms using the test cases as inputs. Having been validated, the program is subject to further optimizations, which may or may not require additional software development within the scope of the application program. Load balancing, for example, aims at improving the efficiency of execution in case that load becomes distributed unevenly either because of the algorithm's behavior or because of the effects of multiuser operation.

2.6. Dynamic load balancing

Even distribution of load is a key prerequisite for parallel efficiency. Although (static) data distribution at program start will attempt to distribute load evenly, load imbalance may occur at run time. In many algorithms the amount of computational load varies dynamically. Examples are adaptive grid algorithms, where refinement in one partition and coarsening in an other may result in load imbalance. Another example is the Algebraic Multigrid Method [7] where the coarsening factor depends on the current coefficient values.

In NOWs, multi-user operation is a second source of load imbalance. The (system) load caused by the activity of other users changes dynamically. As a result, the computational power that is available for the tasks of the parallel program at each processing node changes dynamically. Consequently, processes proceed at different speed. Since there is some form of synchronization in any parallel program, the slowest process determines the overall execution rate. Minimizing the number of synchronization points makes a parallel application somewhat more robust

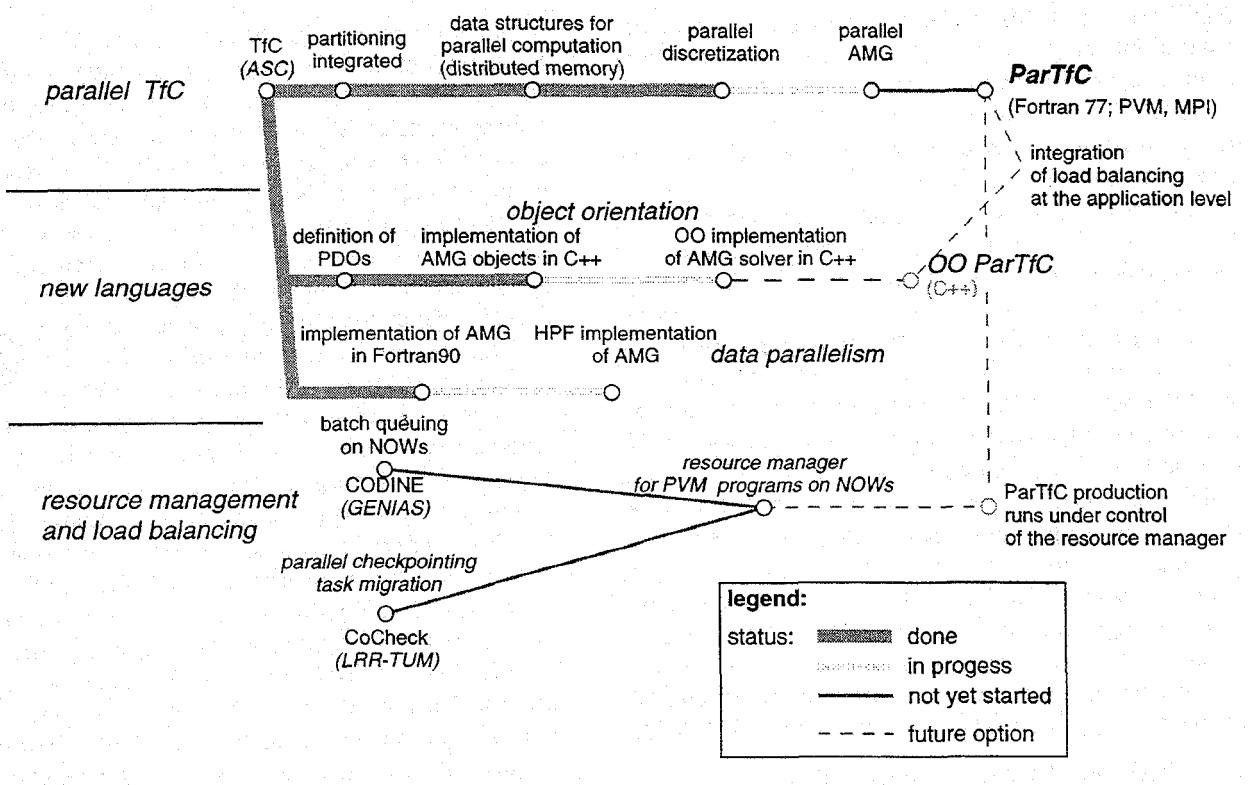


Figure 3: The global strategic plan for the TfC related part in SEMPA

against load imbalance caused by multi-user operation.

Dynamic load balancing can be implemented at the system level by migrating processes from one processing node to another. This approach is considered in SEMPA within the scope of resource management. The alternative approach is to implement load balancing by re-distributing the problem description. In contrast to the previous approach, its implementation is specific to the individual application but is portable across hardware platforms. It is typically integrated in a parallel application that already has been optimized based on the results of detailed performance analysis. The integration of load balancing follows the iterative design and implementation process described in the previous section.

3. Parallel TASCflow for CAD

TfC is a widely applicable CFD package that is used to predict laminar and turbulent viscous flows in complex three dimensional geometries. Flows can either be steady or transient, isothermal or convective, incompressible or compressible (subsonic, transonic and supersonic). TfC is used for the design of pumps, turbomachines, fans, hydraulic turbines, ventilation systems for buildings, as well as for the simulation of pollutant transport, combustors, nuclear reactors, heat exchangers, automotive, aerodynamics, pneumatics, ballistics, projectiles, rocket motors and many more.

In this section, the parallelization of TfC, which serves as a case study for the definition, application and evaluation of software engineering methods in SEMPA, is outlined. First, the essential features of the algorithms that are needed for the following discussion are summarized. Then, the definition of an appropriate basis for data partitioning, which is the most fundamental decision to be taken when setting up a parallelization concept for a Scientific Computing application, is documented in some detail. Subsequently, the parallelization of TfC's main modules, finite volume discretization and the AMG solver are described.

3.1. The algorithms implemented in TfC

The equations solved by TfC have the form of the general transport equation, which states the physical conservation principle for a control volume (illustrated in Figure 4).

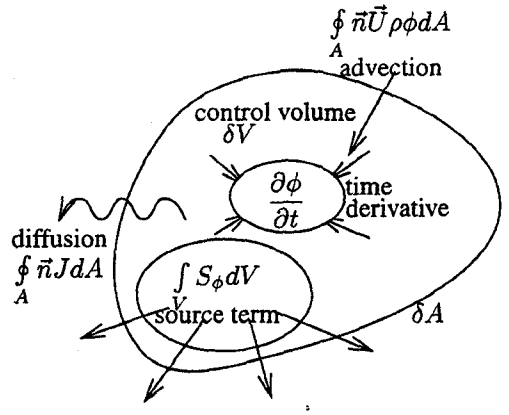


Figure 4: Finite volume

$$\underbrace{\frac{\partial}{\partial t}\int_{\delta V}\rho\phi dV}_{\text{time derivative}} + \underbrace{\oint_{\delta A}\vec{n}\vec{U}\rho\phi dA}_{\text{advection}} = \underbrace{\oint_{\delta A}\vec{n}\vec{J}dA}_{\text{diffusion}} + \underbrace{\int_{\delta V} S_\phi dV}_{\text{source term}}$$

ϕ is the fluid property that is being transported, for instance $\phi = 1$ in the mass equation, $\phi = \vec{U}$ (velocity) in the momementum equation. In TfC, mass and momentum equations are solved as a coupled system of equations (i.e. $\phi = (U_x, U_y, U_z, P)^T$, where U_x, U_y, U_z are the velocity components in each dimension and P denotes pressure), while the scalar equations are solved separately, e.g. $\phi = H$ (enthalpy).

The general transport equation is solved in two steps. First it is transformed into a set of linear equations by applying finite volume discretziation followed by equation assembly. In the second phase, the set of linear equations is solved.

Discretization is based on the concept of a grid. The grid defines a number of locations (grid nodes) at which ϕ is to be computed. TfC uses unstructured grids that are obtained by filling space with elements of four topology types (depicted on bottom of Figure 7). Grids are typically generated by means of a CAD system. Generating appropriate grid topologies for the problem to be solved is of crucial importance for the quality of the numerical solution. A discussion of this topic, however, is beyond the scope of this paper.

TfC employs a finite volume discretization scheme that is implemented based on elements. Figure 5 explains its basic ideas for the two dimensional case. The grid is represented by filled black nodes and solid lines. For each connection of two nodes, the central point is determined. These points are connected to the center of gravity of the elements as illustrated by the dashed lines. The dashed lines define a finite volume, which has exactly one grid node in its center; the solid lines define elements, which have grid nodes as their corners. The surface of the finite volume is subdivided into faces. In our example, the surface consists of eight faces (dashed lines), each of which belongs to one element.

The flux balance states that the sum of the ϕ fluxes through the faces of a volume must be equal to the flux generation inside the volume (source term). The flux through a face is approximated as the product of an approximation of the flow through the integration point ϕ_{ip} associated with the face and the face's area. Finite volume discretization is done element-based in TfC. For all integration points ip of an element, ϕ_{ip} is computed. TfC implements several second order discretization schemes. With second order discretization, ϕ_{ip} is computed as a linear combination of nodal values, nodal gradients and element gradients of the element to which ip belongs. For each element, a local coefficient matrix is obtained. Its coefficients l_{ij} express the influence node j has on the flux at node i. Local coefficients are assembled in a global coefficient matrix A.

Since flux at some node i is influenced only by nodes in its neighborhood, A is very sparse. In addition, it is diagonal dominant and structurally symmetric, but – due to the unstructured nature of the grid – does not have any regular structure. The system of linear equations is solved by an Algebraic Multigrid solver (AMG) [7]. The algorithm implements V and W cycles. Its outline is given in Figure 6. While the grid used in discretization (which is the finest level of multigrid) is static, coarser grids are determined adaptively, taking into account not only the geometry but also the current coefficient values. A modified ILU decomposition is used as a smoother in AMG.

3.2. Data Distribution

As mentioned earlier, TfC is being parallelized according to the SPMD paradigm. The outline of the algorithm given in the previous section motivates two alternatives for implementing a data distribution:

node based partitioning: The grid itself is considered as the partitioning graph, i.e. nodes of the grids are nodes of the graph, connections between grid nodes are the graph's edges. This seems to be a natural choice, since the grid is the basic structure of the CFD simulation algorithm.

element based partitioning: Since all computation in TfC is done element-based, it seems reasonable to take the elements as nodes of the partitioning graph and to draw an edge between graph nodes if the corresponding elements share a face. The partitioning graph in this case is the dual graph of the one defined by the grid nodes and their connections.

While node-based partitioning results in a unique map-

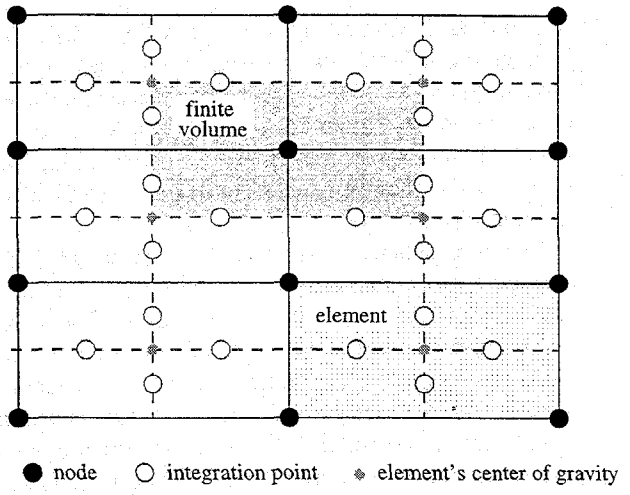
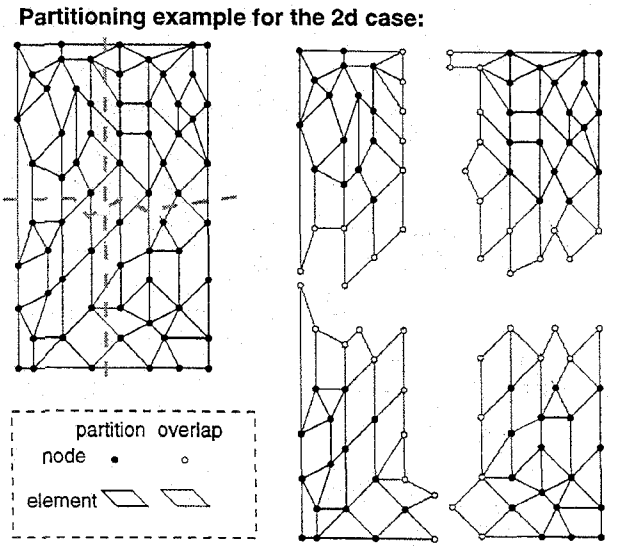


Figure 5: nodes, finite volumes, and elements



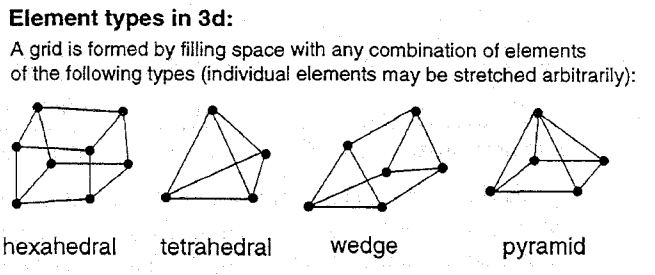


Figure 7: Data distribution in the parallel version of TfC

AMG(*level,MaxLevels,icycle*)
level : current multigrid level
MaxLevels : maximum number of multigrid levels
icyle : 1: V cycle, 2: W cycle
int $k, count$;
if $(i == 1)$
  then $count = 1$ finest grid, "visited" once
  else $count = icycle$ other grid levels, visited *icycle* times
fi
if coarsest grid
  then call direct solver
  else form coarse grid blocks
    for $(k = 1;\ k <= count;\ k++)$ **do**
      restriction
      run a fixed number of iterations of the smoother
      AMG$(level + 1, MaxLevels, icycle)$
      prolongation
    od

Figure 6: Outline of the AMG solver implemented in TfC

ping of unknowns for the linear equation solver, it does not provide a unique assignment of elements whose nodes belong to different partitions. With element-based partitioning the situation is just the other way round: nodes belonging to elements in different partitions are not assigned uniquely to a partition.

For efficiency reasons, the number of points in the algorithm where data is exchanged, should be minimized. Especially on NOWs, parallel processes should be coupled as loosely as possible to minimize the effects of high network latency and multi-user operation.

With element-based partitioning, the contributions to the coefficient of a node that belongs to elements in different partitions are computed by different processes. They have to be gathered in order to set up the linear equations properly. This, however, requires communication between the discretization phase and the solver.

With node-based partitioning, this communication can be avoided. As the coefficients of a node are determined by the evaluation of all its adjacent elements, the complete set of coefficients is obtained correctly if we evaluate all elements that have at least one node in our partition. This approach is illustrated in Figure 7. The elements that are cut by the partitioning of nodes are replicated in the adjacent partitions to form an overlap region. Thus, one communication point in the code can be saved at the cost of some redundant computation, which however is negligible for larger problems.

This approach has been adopted for implementation in the parallel version of TfC. Graph partitioning has been implemented by integrating MeTiS [5], a public domain graph partitioning package, into the CFD program.

The main control loop of the parallel algorithm is dis-

played in Figure 8. Each processor needs to receive nodal values (and gradients) from nodes in its overlap region (see Figure 7). At program initialization, each process P_i (processing partition i) builds lists $R_j^{(i)}$ of nodes whose values it needs to receive from process P_j that computes partition j. List $R_j^{(i)}$ is sent to P_j to tell this process which nodal values it is supposed to send.

```
ParTfC MainControlLoop
Time = StartTime
while Time < EndTime do                          time step loop
    Converged = false
    while ¬Converged do                          coefficient loop
        compute nodal gradients
        communicate nodal gradients
        discretization and equation assembly
        Algebraic Multigrid solver
        communicate nodal values
    od
od
```

Figure 8: The main control loop of ParTfC

3.3. Parallel Finite Volume Discretization

The parallelization of the discretization phase, which takes about 60-75 percent of the run-time of the sequential program, now is relatively straight forward. Each "worker" process evaluates all elements assigned to it in very much the same way as in the sequential program. The element evaluation routines themselves are left unchanged, so that new element types could easily be integrated into the parallel program in the future. This approach emphasizes modularity of the parallel code at the cost of little redundant computation: When evaluating a cut element, local coefficients are computed for all nodes. However, if node i is in the core partition while node j is in the overlap region, only local coefficient l_{ij} is assembled into the global coefficient while l_{ji} is not needed. Since for problems of practical interest the overlap region is very small compared to the core partition, this redundant computation can be accepted for the sake of modularity.

3.4. Parallelization of the Algebraic Multigrid Solver

The AMG solver is more difficult to parallelize than discretization and equation assembly. ILU decomposition is an inherently sequential process. Parallelism can be exploited only at a level of granularity that is too fine grained for efficient implementation on NOWs. However, the task of the smoother is mainly to reduce high frequency error components. It was therefore decided to run the smoother locally only, i.e. to apply ILU decomposition only to the set of equations that has been assembled locally on the partition:

$$\sum_{nb} A_{i,nb}\phi_{nb} = b_i \quad \text{for nodes } i \text{ in the core partition}$$

$$\phi_j = \phi'_j \text{ for nodes } j \text{ in the overlap region}$$

A is the coefficient matrix for the local partition; ϕ'_j is the last update of the nodal value that has been received. In a first demonstration prototype presented at ASC's TASCflow users' conference [9], nodal values were only communicated once in each iteration of the coefficient loop. As expected, this Block-Jacobi type iteration severely slowed down convergence. Of course, the complete prototype will exchange nodal values after each sweep of the smoother.

Since coarse grid blocks are determined at run time, the administrative work that is done for the finest grid at program initialization, i.e. building and distributing lists $R_j^{(i)}$ (see section 3.2.), has to be redone at runtime each time a coarse grid is formed.

4. Conclusion and Future Work

The methodology proposed in this paper addresses the specific requirements of parallelizing large-scale software systems in Scientific Computing that are characterized by complex control flow and data structures which prevent successful application of automatic or interactive parallelization tools. The framework for the design cycle proposed in section 2 is general enough to be applied to most applications in Scientific Computing. Data distribution based on graph partitioning is applicable to all problems where the underlying mathematical model implies a discretization of space (such as the concept of a grid in CFD) that can be taken as a basis for partitioning the problem description.

However, as our case study, ParTfC, is not yet completed, the methodology is likely to evolve further. Therefore, the scope of its applicability cannot yet be finally assessed. For the same reason, evaluation of the methodology concentrates on the early stages in the design process. The first step, analysis of the sequential program turned out to be much more time consuming than originally expected although the code is well structured and a certain extend of documentation has been available from the problem domain experts. Our method of know-how acquisition, program analysis, and design documentation has proved to be very effective especially within the scope of an interdisciplinary cooperation. The resulting document already proved to help new project members to get started

quickly in their work. As the ParTfC development progresses, detailed evaluation of the method proposed for the later phases in the design process will become available.

For the type of application considered here, most phases in the design process require experts' intuition and creativity. Therefore we believe the potential for automating the design process (or individual phases thereof) to be limited. As a consequence, we concentrate on developing a framework that supports the software engineer in his work by providing guidelines for systematic design, implementation and documentation. The framework is to provide guidelines how to make efficient use of available tools for individual tasks in the design process. Based on the experience gained in ParTfC development, requirements will be derived that may indicate new directions for future research and development in the domain of tools for analyzing and designing parallel programs.

5. Acknowledgements

The work presented in this paper is being funded by the German Federal Department of Education, Science, Research and Technology, **BMBF**. Design and implementation of ParTfC are carried out in a close interdisciplinary cooperation of of F. Menter and F. Unger from ASC, P. Luksch, U. Maier, S. Rathmayer, and M. Weidmann from LRR-TUM, and P. Bastian from ICA. The resource management system for NOWs is being developed in cooperation of LRR-TUM and GENIAS.

References

[1] "IEEE Standards Collection Software Engineering". IEEE (1994).

[2] *Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam.* "PVM: Parallel Virtual Machine – A Users' Guide and Tutorial for Networked Parallel Computing". MIT Press (1994). http://www.netlib.org/pvm3/book/pvm-book.html.

[3] *W. Gropp, E. Lusk, and A. Skjellum.* "Using MPI - portable parallel programming with the Message-Passing Interface". Scientific and Engineering Computation Series. The MIT Press, Cambridge, MA, 1 edition (1994).

[4] IEEE Recommended Practice for Software Requirements Specifications. in [1] (1993).

[5] "METIS: Unstructured Graph Partitioning and Sparse Matrix Ordering System". George Karypis and Vipin Kumar, University of Minnesota (1995).

[6] *Sabine Rathmayer.* Automatische und interaktive Parallelisierungswerkzeuge. In "Workshop Software Engineering im Scientific Computing" (June 1995). http://wwwbode.informatik.tu-muenchen.de/archiv/artikel/sesc95/Rathmayer.ps.gz.

[7] *Michael Raw.* A Coupled Algebraic Multigrid Method for the 3D Navier Stokes Equations (1994).

[8] *Ralf Diekmann und Robert Preis.* "Software Engineering im Scientific Computing", chapter Statische und dynamische Lastverteilung für parallele numerische Algorithmen, pages 128–134. Vieweg Verlag, Wiesbaden (1996).

[9] *Friedemann Unger.* SEMPA Software Engineering Methods for Parallel Applications, Project Status. In *Georg Scheuerer*, editor, "4th TASCflow User Conference". ASC (May 1996).

[10] *Friedmann Unger and Florian Menter.* Software Engineering Guidelines. SEMPA-Report SEMPA-ASC-96-01, ASC GmbH, Holzkirchen (April 1996).

Appendix: Structure of a Design Document for Use in Interdisciplinary Projects

1. **Introduction**
   (a) **Introduction to the Application Domain.** Provides the necessary background to project members form discipines other than the problem domain.
   (b) **Problem Specification in terms of PDOs.** Purpose: illustrate the structure of the problem, which should be reflected in the program structure.
   (c) **Structure of the Software Package.** Overview of directory and file structure of the source code, hierarchy of modules.
2. **PDOs and their representation in the software.** Relates PDOs to data structures and operations on them to subroutines.
3. **Documentation of the individual modules according to the module hierarchy.**
   (a) **Theoretical background** that is specific to the module under consideratioion (if appropriate).
   (b) **Requirements specification.** Define the *function* of the module, i.e. the *what* (not the *how*).
   (c) **Algorithmic specification.** Specifies *how* the task specified in the previous section is accomplished by defining the algorithm that is used to solve the problem.
   (d) **Abstract implementation.** This section describes the program at the level of the implementation language.
   (e) **Source code documentation.** Usually found in the source files.