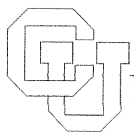
Multigrid Methods on Parallel Computers - a Survey on Recent Developments

Oliver A. McBryan
Paul O. Frederickson
Johannes Linden
Anton Schuller
Karl Solchenbach
Klaus Stuben
Clemens-August Thole
Ulrich Trottenberg

CU-CS-504-90



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Multigrid Methods on Parallel Computers - a Survey on Recent Developments

Oliver A. McBryan, Paul O. Frederickson, Johannes Linden
Anton Schuller, Karl Solchenbach, Klaus Stuben
Clemens-August, Thole Ulrich Trottenberg

CU-CS-504-90 December 1990

Department of Computer Science
University of Colorado at Boulder
Campus Box 430
Boulder, Colorado 80309-0430

(303) 492-7514
(303) 492-2844 Fax
mcbryan@boulder.colorado.edu

Multigrid Methods on Parallel Computers – a Survey on Recent Developments

Oliver A. McBryan[1,5] *Paul O. Frederickson*[2]
Johannes Linden[3] *Anton Schüller*[3] *Karl Solchenbach*[4]
Klaus Stüben[3] *Clemens-August Thole*[4] *Ulrich Trottenberg*[3,4,6]

[1]Center for Applied Parallel Processing, University of Colorado, Boulder, CO 80309-0430
[2]RIACS, NASA Ames Research Center, Mail Stop 230-5, Moffet Field, CA 94035, USA
[3]Gesellschaft für Mathematik und Datenverarbeitung mbH, Postfach 1240, D-5205 Sankt Augustin 1, West Germany
[4]SUPRENUM GmbH, Hohe Str. 73, D-5300 Bonn 1, West Germany
[5]Research supported by Air Force Office of Scientific Research, under grant AFOSR-89-0422
[6]Research funded in part by means of the Federal Ministry of Research and Technology (BMFT) (grant No. ITR8601 9) and the Ministry of Economy and Technology of Nordrhein-Westfalen (MWMT) (project No. 323-8605200).

Abstract

Multigrid methods have been established as being among the most efficient techniques for solving complex elliptic equations. We sketch the multigrid idea emphasizing that multigrid solution is generally obtainable in time directly proportional to the number of unknown variables on serial computers. Despite this, even the most powerful serial computers are not adequate for solving the very large systems generated, for instance, by discretization of fluid flow in three dimensions.

A breakthrough can be achieved here only by highly parallel supercomputers. On the other hand, parallel computers are having a profound impact on computational science. Recently, highly parallel machines have taken the lead as the fastest supercomputers, a trend that is likely to accelerate in the future. We describe some of these new computers, and issues involved in using them.

We describe standard parallel multigrid algorithms and discuss the question of how to implement them efficiently on parallel machines. The natural approach is to use grid partitioning.

One intrinsic feature of a parallel machine is the need to perform interprocessor communication. It is important to ensure that time spent on such communication is maintained at a small fraction of computation time. We analyze standard parallel multigrid algorithms in two and three dimensions from this point of view, indicating that high performance efficiencies are attainable under suitable conditions on moderately parallel machines.

We also demonstrate that such performance is not attainable for multigrid on massively parallel computers, as indicated by an example of poor efficiency on 65,536 processors. The fundamental difficulty is the inability to keep 65,536 processors busy when operating on very coarse grids. This example indicates that the straightforward parallelization of multigrid (and other) algorithms may not always be optimal.

However, parallel machines open the possibility of finding really new approaches to solving standard problems. In particular, we present an intrinsically parallel variant of standard multigrid. This "PSMG" method (parallel superconvergent multigrid) allows all processors to be used at all times, even when processing on the coarsest grid levels. The sequential version of this method is not a sensible algorithm.

Acknowledgements

This paper merges new results and results which have been obtained by the authors and their co-workers in the fields of parallel computing and multigrid techniques in the last few years. Several parts of this paper have been published elsewhere before. The authors have tried to arrange the pieces here in a consistent form.

Our particular thank is given to all co-workers who have - directly or indirectly - contributed to this paper. We want to mention explicitly Ute Gärtel, Rolf Hempel, Max Lemke, Anton Niestegge, Eric van de Velde.

Contents

1 Introduction 5

2 Parallel Supercomputers 8
- 2.1 Classification of Supercomputer Architectures . . . 8
- 2.2 Machine Characteristics of some Multiprocessors . . . 12
  - 2.2.1 Intel iPSC . . . 12
  - 2.2.2 Connection Machine . . . 13
  - 2.2.3 SUPRENUM . . . 14
- 2.3 A Software Concept Based on Message-Passing . . . 15

3 Introduction to Multigrid 18
- 3.1 The Basic Idea and Algorithmical Structure . . . 18
- 3.2 Extensions of the Basic Idea . . . 20
  - 3.2.1 Nonlinear Problems . . . 20
  - 3.2.2 The Full Multigrid Approach . . . 21
- 3.3 Multigrid Components, Performance Analysis; Some General Remarks . 21
- 3.4 Some Special Considerations for Poisson-like Equations, Red-Black-Relaxation . . . 23

4 More Sophisticated Multigrid Techniques 25
- 4.1 Anisotropic Operators . . . 25
- 4.2 First Order Differential Terms . . . 28

5 Standard Parallel Multigrid 30
- 5.1 Some General Remarks . . . 30
- 5.2 Isotropic Equations and Systems . . . 30
- 5.3 Parallel Multigrid for Anisotropic Operators . . . 32

6 Grid Partitioning, General Grid Structures, Implementation Aspects 36
- 6.1 Grid Partitioning for Regular Grids . . . 36
- 6.2 Grid Partitioning for the Anisotropic 3D Operator . . . 38
- 6.3 More General Grid Structures . . . 40
- 6.4 Implementation of Parallel Grid Algorithms . . . 41
- 6.5 Example of a Parallel Grid Program (SUPRENUM) . . . 45
- 6.6 Communications Libraries . . . 47

7 Multiprocessor Efficiency of Multigrid 48
- 7.1 Basic Notations and Measures . . . 48
- 7.2 A System Model for a Homogeneous Architecture . . . 49
- 7.3 Some Results . . . 50
  - 7.3.1 Analysis of 2D Multigrid Efficiency . . . 50
  - 7.3.2 A Concrete 2D Example . . . 52
  - 7.3.3 Analysis of 3D Multigrid Efficiency . . . 53
  - 7.3.4 A Concrete 3D Example . . . 53

7.3.5 Comparison of 2D and 3D Efficiency 54

8 Some Measured Results on Parallel Computers **55**
8.1 Multigrid on Vector Computers 55
8.2 Multigrid on the Caltech Hypercube 56
8.3 Multigrid on the Intel iPSC 57
8.4 Multigrid on SUPRENUM 61
8.5 Multigrid on the Connection Machine 62

9 A Different Parallel Multigrid Approach **64**
9.1 Parallel Superconvergent Multigrid 64
9.2 The Basic Idea 64
9.3 Multiscale Convergence Rates 67
9.4 PSMG: Algorithmic Form 68
9.4.1 Application to Poisson's Equation 68
9.5 PSMG Performance 69
9.6 How Does PSMG Compare with Standard MG? 71

1 Introduction

Parallel Computers. Supercomputers are the key to the simulation of a wide range of important physical problems. Such simulations typically require large numbers of degrees of freedom to provide sufficient resolution, particularly when engineering accuracy, rather than simple qualitative behavior, is required. In many cases one is currently limited by available computer resources, rather than by an understanding of the underlying physics.

As an example, it is very desirable to simulate accurately the flow of air over a plane. Current aircraft design strategy involves the use of wind tunnels. However wind tunnel testing is limited with respect to aircraft size and Mach number, although extrapolations from smaller scale models can overcome some of the limitations. Planned wind tunnel testing for the Boeing 7J7 was greatly reduced thanks to advances in computational aerodynamics, substantially curtailing 7J7 development time and, consequently, costs. But the computational techniques now in use do not simulate the complete physics for the flow past the entire aircraft; they model various aspects of the flow that, when combined, give guidance to the design, but not answers. The major limitation is that as more of the plane is included in the simulation, the numerical grids become larger, requiring more processing power and memory. The same phenomenon is seen in weather forecasting, in oil reservoir simulation, in combustion studies, and wherever quantitative computations in three dimensions are performed.

Major advances in many of these areas are expected as soon as computer power increases to about 1 Tflops ($= 10^{12}$ Flops). This would correspond to an increase of close to an order of magnitude in resolution in each of the coordinate directions compared to current machines. Conventional supercomputers with one or a few processors are limited by various factors, including the need to dissipate energy in a small volume, effects of the finite speed of light, and bottlenecks related to memory access. It is widely believed that parallel computers provide the only near-term hope of reaching this range of computer power. Furthermore, in most applications the cost per megaflop is a relevant issue. Massively parallel computers provide economies of scale not available to conventional computers larger than a PC. Parallel computers may be built from lower cost technologies, because the individual processors need not be extremely powerful.

Because of these factors, parallel computers have been widely studied in recent years. Substantial research has been accomplished related to these machines, including both theoretical advances, involving algorithm design, and computational experiments. Hardware advances have reached the point where the fastest available supercomputers are now highly parallel machines. Furthermore, the combined efforts of many researchers have demonstrated that parallel computing is feasible.

One disadvantage of a parallel computer, is that it is somewhat harder to program than a serial machine. Each processor must be assigned a distinct portion of the work to be performed, and substantial synchronization of the processors is then required in order to ensure that the results from individual processors are merged appropriately. The difficulties of programming parallel machines have spawned a whole range of new research areas for computer science and are a primary reason why this area has been so dynamic in recent years.

Multigrid (MG). For a wide class of problems in scientific computing, in particular for partial differential equations, the multigrid (more general: the multi-level) principle has proved to yield highly efficient numerical methods. However, the principle has to be applied carefully: if the "multigrid components" are not chosen appropriately for the given problem, the efficiency may be far from optimal. This has been demonstrated for many practical problems. Unfortunately, the general theories on multigrid convergence do not give much help in constructing really efficient multigrid algorithms, though some progress has been made in bridging the gap between theory and practice during the last few years. The research in finding highly efficient algorithms for non-model applications therefore is still a sophisticated mixture of theoretical considerations, a transfer of experiences from model to real life problems, and systematical experimental work. The emphasis of practical research activity today lies – among others – in the following fields:

- finding efficient multigrid components for really complex problems, e.g. Navier-Stokes equations in general geometries
- combining the multigrid approach with advanced discretization techniques: using dynamic local multigrid refinements; adding artificial terms (viscosity, pressure, compressibility, etc.) in certain multigrid components; using "double" discretization, τ-extrapolation, defect correction in connection with multigrid to obtain higher accuracy; using coarse-grid continuation techniques etc.
- constructing highly parallel multigrid algorithms

In this paper, we plan to deal only with the last topic.

Parallel Multigrid. Multigrid methods are known to be "optimal", i.e. the number of arithmetic operations that have to be performed to achieve either discretization accuracy or fixed accuracy is proportional to the number of discrete unknowns which are to be calculated [50]. This statement applies directly to standard sequential MG algorithms. With the availability of parallel computers, the question arises as to how well MG methods are suited for parallel computing. Several authors have studied the parallel implementation of multigrid on different parallel architectures [9, 10, 28, 29, 42, 51].

Sometimes one finds the conjecture that MG is – in some sense – an essentially sequential principle, or the opinion that full MG efficiency is obtained only on sequential computers and that there is always a loss of efficiency for MG on parallel architectures. Certainly, standard MG requires a sequential processing of the different grid levels. We do not intend to give a final answer to this question, but we want to contribute to a clarification of the situation.

First, one may distinguish the approaches where standard MG algorithms are utilized in the parallel context from those approaches where essentially new MG algorithms (or better: MG-like algorithms) are designed specifically for parallel computing.

To the class of new MG-like algorithms belong also all those attempts where several multigrid levels are simultaneously processed. Such methods have been considered by

a number of authors [17, 19]. A breakthrough has, however, not yet been achieved; for theoretical reasons, one may also doubt whether these approaches can give a substantial gain.

A new MG idea that provides significant progress for *massively* parallel machines has been introduced recently by the first author together with Paul Frederickson [13, 14, 15, 16]. Here on each level several coarse grid problems are solved simultaneously in order to improve the MG-convergence. We sketch this "parallel superconvergent multigrid method" in Chapter 9.

2 Parallel Supercomputers

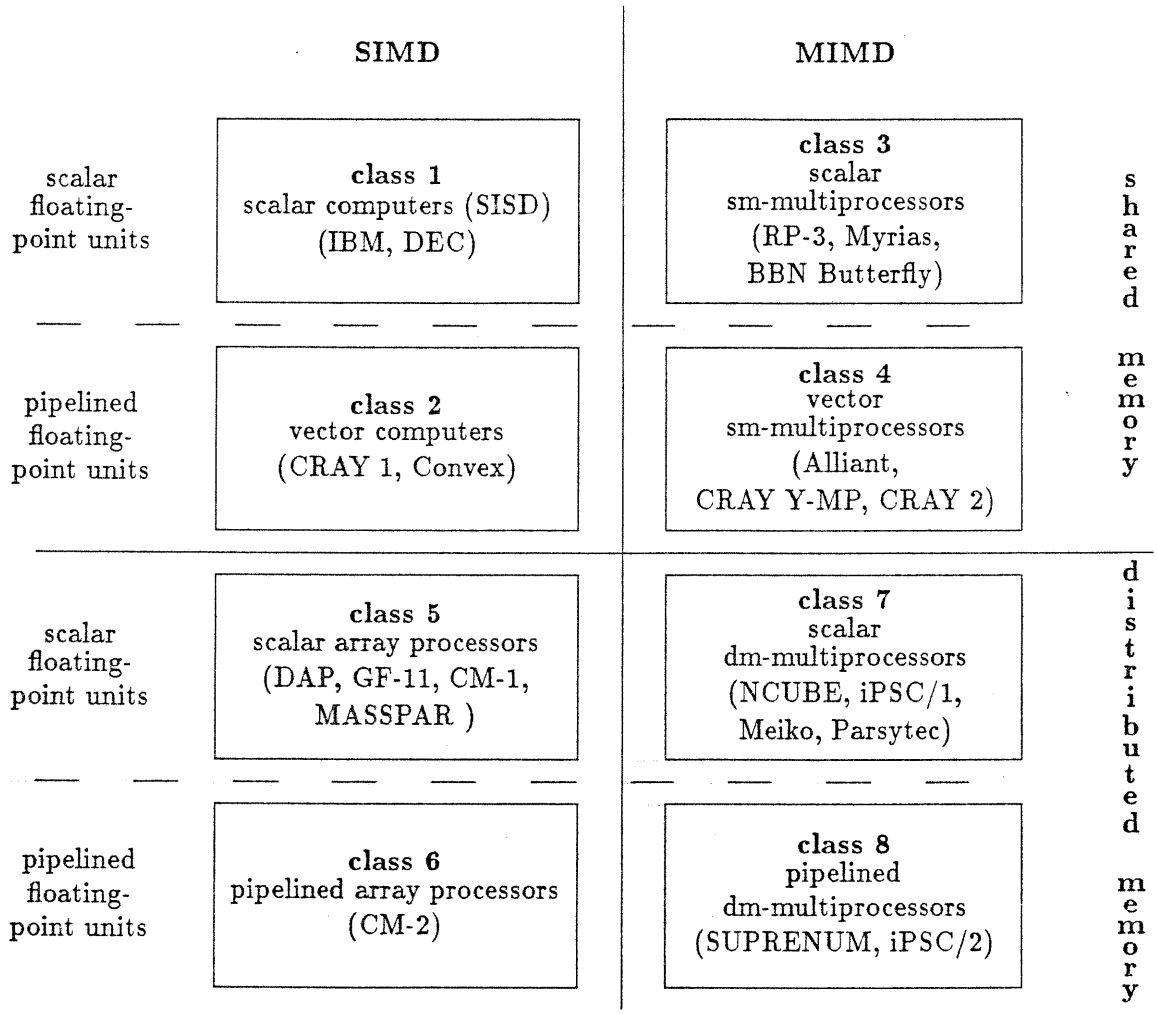


Figure 1: Classification of parallel and supercomputer architectures

2.1 Classification of Supercomputer Architectures

There are many different approaches to the classification of computer architecture, especially with respect to parallel processing. A classification may be based on quantitative aspects (the degree of parallelism or the granularity), the structure of the control-flow (SIMD, MIMD, data-driven, demand-driven), the hardware technology (VLSI, VHSIC, air cooled, liquid cooled) or the topology of the processing elements and the memory units. For a taxonomy of parallel designs see [46].

In this section, we – more pragmatically than systematically – distinguish 8 classes of architectures which play an important role in the supercomputer world. The basic classification categories represented in Figure 1 are:

1. SIMD vs. MIMD (vertical line)
   SIMD operation mode means that parallel functional units execute the same instruction sequence on different data. The two best-known realizations of the

SIMD principle are pipelined floating point units (class 2, 4, 6, 8) and array processors (class 5 and 6).

The MIMD principle (class 3, 4, 7, 8) is the favorite operation mode for multiprocessors based on entire and independent processors. Each processor may execute a different instruction stream within the same application.

2. Shared vs. distributed memory (horizontal line)
   One of the central problems to be solved in the design of multiprocessor systems is memory access. Basically, there are two possibilities for system organization:

   - shared memory:
     each processor has direct access to the total memory.
   - distributed memory:
     each processor has direct access only to its own private memory.

   Sometimes both memory organization types are combined in hierarchical memory systems (e.g. RP3).

   Furthermore, in which category a system is placed, may reflect the user's view of the memory organization rather than its hardware realization. The BBN Butterfly and Myrias SPS-2, for instance, are multiprocessor systems with distributed memory units which are interconnected by a network. Both machines offer a shared memory model to the user and are therefore often regarded as shared memory systems (class 3) although the hardware looks more similar to distributed memory machines (class 7).

3. Scalar vs. pipelined floating point units (dashed horizontal line)
   Scalar floating point units are restricted in their floating point performance. Presently, the most cost effective way to achieve floating point rates of 20 - 50 Mflops or more (per node) is by utilizing vector or scalar-pipelined processing (e.g. Weitek chips or Intel's i860). MIMD multiprocessors which target the top of supercomputer performance have to employ such processors as basic floating point units. Therefore, the most powerful architectures today are often two-level MIMD/SIMD multiprocessor systems (class 4 and 8). The efficient use of these architectures requires parallelism on two levels: the coarse grain parallelism related to the global MIMD structure and the fine grain parallelism (pipelining or vectorization) which is necessary to achieve maximum node performance. Similarly, the Connection Machine CM-2 is an SIMD/SIMD two-level system (class 6), also using fine grain vectorization.

In the following we briefly describe these 8 classes of supercomputer architectures and name typical representatives of them.

Class 1: scalar computers

The "traditional" von Neumann computer architecture (SISD) is the basis for mainframes, minicomputers, and microcomputers. Using current hardware technology, the floating point performance of such architectures seems to be far from supercomputer performance, except on truly scalar algorithms.

Class 2: vector computers

Historically the first machines to be called supercomputers were vector computers. Their hardware architecture is based on very fast arithmetic pipelines which support the rapid execution of vector instructions operating on all components of the vector operands simultaneously. Vectors in that sense consist of components which can be processed independently. Hence, vector processing is a special form of parallel processing based on fine grain parallelism. Application codes have to be vectorized (i.e. operations are defined on vectors and certain data dependencies between operations are excluded) in order to exploit the potential speed of the hardware. The need for vectorization resulted in new vector algorithms and in special compiler tools (vectorizers) for the automatic vectorization of existing codes.

Examples of current vector machines are the CRAY Y-MP, the CYBER 205, the ETA-10, the Fujitsu VP, the NEC-SX, the Hitachi S-810, and the IBM 3090-VF.

Due to the technological progress in VLSI chip development vector computer architectures today can also be realized in standard (microcomputer) technology. These systems are smaller, somewhat slower and considerably cheaper than the classical vector computers and therefore are called minisupercomputers. The vector-minisupercomputers take advantage of the existing software and tools for vector machines – some systems are even CRAY-compatible. Examples are the Convex C2 and the SCS-40.

Class 3: shared memory scalar multiprocessors

Another way to increase computing performance is to combine several single processors into a multiprocessor system and replace sequential processing by parallel processing. The optimal degree of parallelism (fine or coarse granularity) depends on the number and the power of the single processors as well as on the memory organization. The shared memory concept restricts the number of CPUs to about 8 or 16 today (e.g. the Alliant). If the memory is accessed via a network rather than a direct interconnection a larger number of CPUs can be connected at the cost of longer access times. Examples are the IBM RP-3 and the CEDAR project (=clusters of Alliant systems). Further examples in this class are the Sequent, Flexible, Encore, Concurrent Computers, and (at the software level) Myrias machines.

Class 4: shared memory vector multiprocessors

The step from a single processor to a multiprocessor system (class 1 to class 3) is, of course, also possible and obvious for vector computers (class 2). Similarly as for scalar multiprocessors the performance is increased by combining several vector CPUs into multiprocessor systems, with of course the same memory access problems. The shared memory concept limits the number of vector processors (typically today ≤ 8). The parallelism on these systems is often used to increase the throughput of the system (running different jobs on different CPUs) but not the execution speed of an individual job. However, MIMD-parallel as well as SIMD-like processing is also possible (e.g. using macrotasking or microtasking constructs on the CRAY Y-MP). Representatives of this class are the multi-headed versions of the CRAY Y-MP, CRAY 2, and the ETA-10.

Class 5: scalar array processors
The era of parallel computers started with array processors which perform one instruction simultaneously on an array of operands (i.e. in SIMD mode). Recently these systems have been upgraded to massively parallel multiprocessors (with many thousands of processors). Each processor is relatively small and weak but the enormous degree of parallelism results in supercomputer performance. Typically, these systems are used for a restricted class of special applications such as image processing. We mention here the historical Illiac IV, the Goodyear MPP, the ICL DAP, the original Connection Machine CM-1, and the MASSPAR.

Class 6: pipelined array processors
The combination of (SIMD) array and pipelined (vector) processing has been realized in the Connection Machine 2 which presently is the system with the highest floating point performance rate for appropriate applications on regular data structures.

Class 7: scalar distributed memory multiprocessors
Today, multiprocessor systems with a large (and in principal unlimited) number of processors require that the memory units are physically associated with the processors (distributed memory). The basic unit of such a system, consisting of the CPU, the arithmetic coprocessor, the memory, and the communication unit will be called a *node* in the following. The first prototypes of this class were based on hypercube topologies and were built at the California Institute of Technology. Intel's iPSC/1 was the first commercial product, followed by Ncube and Symult. Recently Intel introduced its second generation based on more powerful nodes but the same hypercube structure, and is planing a third generation based on a grid rather than hypercube topology. Multiprocessor systems with transputer nodes have also entered the market (Meiko, Parsytec). While listed earlier under shared memory systems, the Myrias SPS-2 system belongs in this category based on its hardware design.

Class 8: pipelined distributed memory multiprocessors
These systems combine the advantages of the vector and the parallel processing concepts. The multiprocessor architecture is derived from the class 7 machines whereas the node architecture is taken from low-cost pipelined computers (class 2). The basic idea is to combine powerful pipelined nodes, with their attractive cost/performance ratio, into a multiprocessor system. Due to the size and the cost of a single node, their number is in practice limited to several thousands. The computational speed of the nodes in turn imposes strong requirements on the speed of the communication. If the communication problem is solved satisfactorily these machines are among the most powerful supercomputers existing today. Systems currently entering the market are SUPRENUM and the Intel iPSC/860.

The classification of parallel computers in Figure 1 is by no means unique and complete. An important classification category which is not taken into account in Figure 1 is the hardware technology. Systems based on VHSIC hardware (like the CRAY and ETA systems) are much more powerful (and expensive) than systems based on microcomputer technology (like the Alliant) although they belong to the same class.

Furthermore, there is an enormous variety in the current designs, particularly in the inter-connection topologies. While many interesting parallel machines involve only a few processors, we will concentrate in this paper on those machines which have moderate to large numbers of processors. Important classes of machines such as the CRAY Y-MP, CRAY 2 and ETA-10 are therefore omitted from many of the subsequent discussions.

2.2 Machine Characteristics of some Multiprocessors

There are at least 100 parallel computer projects (classes 3 to 8) underway at this time worldwide. While some of these projects are unlikely to lead to practical machines, a substantial number will probably lead to useful prototypes. In addition, several commercial parallel computers are already or have been in production (e.g., ICL DAP, Denelcor HEP, Intel iPSC, NCUBE, FPS T-Series, Connection Machine, MASSPAR, SUPRENUM, Symult 2010, Myrias SPS-2, Evans and Sutherland ES-1, Meiko, Parsytec) and more are under development. One should also remember that the latest CRAY computers, (e.g. CRAY X-MP, CRAY 2 and CRAY Y-MP) involve multiple processors, and other vector computer manufacturers such as ETA Systems, NEC, Fujitsu and Hitachi have similar strategies.

In this section we will look briefly at the characteristics of several parallel machines that were used to measure multigrid performance in later chapters. The machines considered here in more detail are the Intel iPSC, the Connection Machine CM-2 and the SUPRENUM-1.

2.2.1 Intel iPSC

The Intel iPSC was the first commercial hypercube MIMD-computer, and has been the most widely available highly parallel computer in recent years. Built from 128 Intel 80286 processors, peak computer power of the original iPSC/1 is under 10 Mflops, yet the iPSC was the basis for a large number of useful experiments in parallel computing. The iPSC/2 computer is a second generation machine that provides greatly increased processing power and communication throughput. Each node contains an 80386 microprocessor with up to 8 Mbytes of memory (extendible to 16Mbytes with 64 processors). There are three available numeric co-processors: an Intel 80387 co-processor (300 Kflops), a Weitek 1167 scalar processor (900 Kflops) and a VX vector board (6 Mflops double precision, maximum of 64 nodes). Thus the top-rated system has 64 nodes capable of 424 Mflops double precision (64 bit) and 1280 Mflops single precision (32 bit). Special communication processes allow message circuits to be established between remote processors without intervention from intermediate nodes.

In fall 1989 Intel announced an i860-based version of the older iPSC/2 hypercube system. These iPSC/860 systems are basically standard iPSC/2 hypercubes with the node processors replaced by Intel i860 processors. In terms of raw floating point performance the peak rate is thereby increased to 60 Mflops per node (64 bit). In practice it is unlikely that more than 40 Mflops per node can be realized due to the memory model used by the i860. Some simple vector type kernels, hand-coded in assembler, are currently running from 28 to 38 Mflops/node. Well-designed Fortran programs

are currently yielding about 3-4 Mflops/node due to the state of the i860 Fortran compilers. Several major i860 compiler efforts are underway and will undoubtedly improve substantially on the early results. Because the communication facilities of the iPSC/860 are those of the iPSC/2, the system is constrained to a maximum of 128 nodes.

While the iPSC/860 utilizes the slow iPSC/2 communication hardware and software, communication proves to be much faster on the i860 system than on the iPSC/2. This is because most of the message startup communication overhead is software overhead involved in negotiating the communication protocol. Because the i860 is so much faster than the 80386, the software overhead is correspondingly decreased. The effect is to reduce messaging time by about a factor of three.

The iPSC/860 actually supports heterogeneous boards – a mixture of i860 and 80386 boards is allowed. This permits special 80386 nodes to take advantage of the flexible interfaces to graphics, disk and other peripherals available to that processor. For example 780 Mbyte disks may be attached to such nodes via a 4 Mbyte/sec SCSI interface. Frame buffers, VME bus devices and Ethernet also plug into these boards.

Intel has also announced plans to develop a rectangular version of the iPSC/860 – the iPSC/3. This system is very similar in architecture to the earlier Symult 2010 system. There are 8 communication paths per node, allowing 4 bidirectional channels as required for a two-dimensional grid. With the new communication structure, the iPSC will be freed from the constraint of a maximum of 128 nodes. Indeed Intel has announced a 2048 processor version of the iPSC/3, called Touchstone. A prototype to be completed in spring 1991, will have 576 processors.

2.2.2 Connection Machine

The Connection Machine CM-1 designed by Thinking Machines Corp., has 65,536 1-bit processors, though this may be regarded as a prototype for a machine that might have 1 M processors. While designed primarily for artificial intelligence work and image processing, this machine proved to have potential applications to scientific computing and database applications. The later CM-2 computer adds 2048 Weitek floating point processors and 8 Gbytes of memory, to provide a powerful computer for numerical as well as symbolic computing. The CM computers are SIMD machines. All processors receive the same instruction on each cycle. Logic is supported by allowing individual processors to skip the execution of any instruction, based on the setting of a flag in their local memory. The CM machines are based on a hypercube communication network, with a total communication bandwidth of order 3 Gbytes/sec. Communication is by worm-hole type routing. The system supports parallel I/O to disks at up to 320 Mbyte/sec, and to frame buffers at 40 Mbyte/sec.

Connection Machine software consists of parallel versions of Fortran, C and Lisp. In each case it is possible to declare parallel variables, which are automatically allocated on the hypercube. Programs execute on a front end machine, but when instructions are encountered involving parallel variables, they are executed as parallel instructions on the hypercube. The system supports the concept of *virtual processors*. A user can specify that he would like to compute with a million (or more) virtual processors, and such processors are then similar to physical processors in all

respects except speed and memory size. A typical use is to assign one virtual processor per grid point in a discretized application. This provides a very convenient programming model. Parallel global memory reference is supported using both regular multi-dimensional grid notations (NEWS communication) and random access (hypercube) modes.

The Connection Machine is one of the few examples where a program from a serial machine (e.g. work station) or from a CRAY may be moved to a parallel machine and run essentially without change. The CM-2 Fortran is Fortran-77 with the addition of the array extensions of Fortran 90. All array data types and operators are implemented as parallel objects or operators on the CM-2. In the case of Fortran-77 programs, a preliminary vectorizer is available that produces Fortran 90 as output. Because of the SIMD architecture no synchronization instructions are required.

It is extremely rare to approach the 24 Gflops peak rate of the CM-2. In practice one attains about 10% to 20% of that rate. In part this is because in addition to normal hypercube communication (e.g. to nearest neighbors in a grid) there is also extra communication required for every floating point operation. Since the floating point operations are off-chip, each of the 32 bit-serial processors that share a Weitek, must send its data to the Weitek for processing. Furthermore, since the data arrives bit-serially, it needs to be "transposed" so as to be presented as floating point data to the Weitek. These two steps between the processors and the Weitek units account for much of the performance loss. Standard numerical algorithms such as relaxations or conjugate gradient iterations perform at from 2 to 4 Gflops, which is also typical of performance on regular-grid evolution problems. Only in situations where numbers can be deposited in the 64 Weitek registers (shared by 32 processors), and then computed on for a substantial time *without leaving the Weitek*, can the theoretical speed be approached. For example, parallel polynomial evaluation proceeds at up to 20 Gflops – which ensures that transcendental functions are extremely fast on the system. A new Fortran compiler for the CM-2 now supports the programming of the Weitek processors and allows applications to run at 4-6 Gflops.

2.2.3 SUPRENUM

The German SUPRENUM project involves coupling up to 256 processors ("nodes") with a two-level network of fast busses. The concept combines the flexibility of an overall MIMD system with the cost-effectiveness of SIMD-vector processing in each node. A node consists of a standard microprocessor CPU, 8 Mbyte of private memory, a fast floating-point vector unit (10 Mflops peak performance, 20 Mflops with chaining) and dedicated communication hardware.

Up to 16 of these computing nodes are combined into a "cluster" using a bus called the clusterbus (256 Mbytes/sec). Each cluster also contains a local disk, a disk controller node, a monitor node which supports performance measurements , a communication node for the connection to the upper bus level (the SUPRENUMBUS).

As shown in Figure 2, 16 clusters are connected by a 4D hypercube structure ($\equiv 4 \times 4$ torus) of serial high-speed SUPRENUMBUS's.

The SUPRENUM-1 prototype is completed by a front end system which is used for operating and maintaining the high performance kernel as well as for software

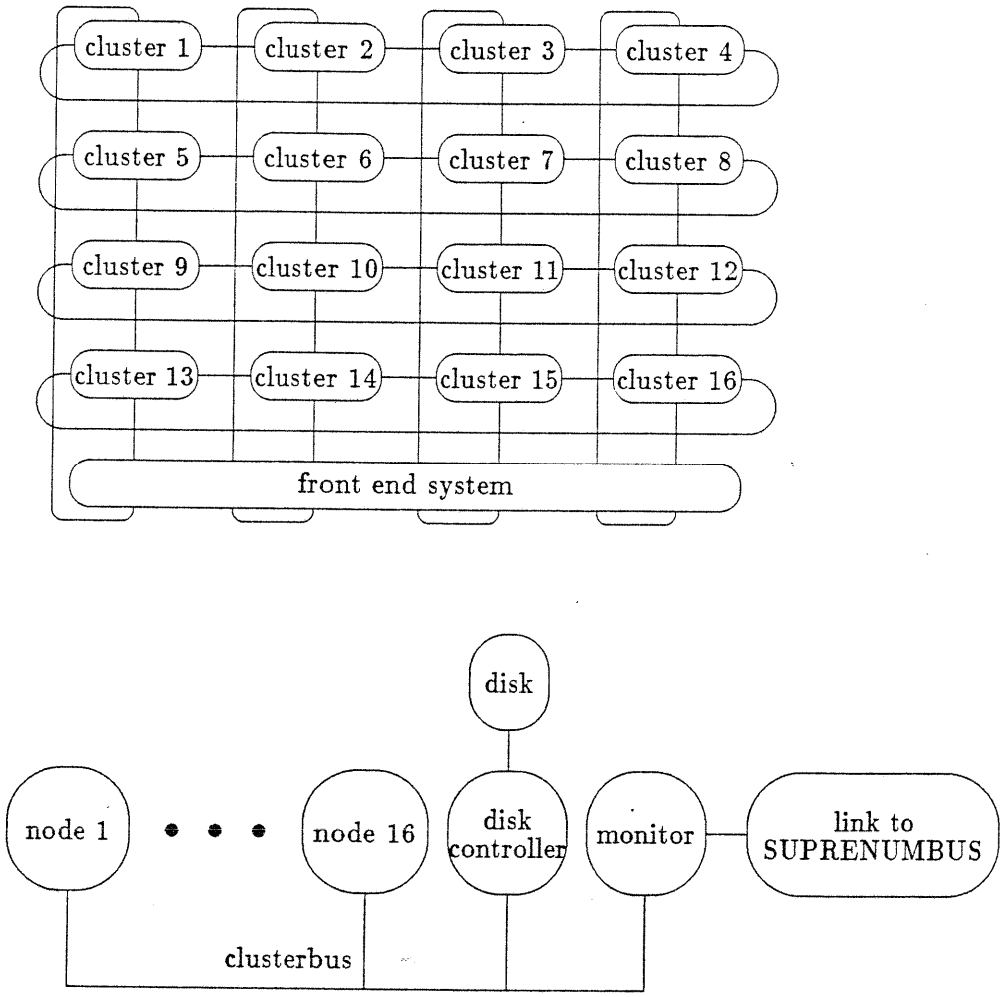


Figure 2: Structure of the SUPRENUM-1 prototype with 256 nodes in 16 clusters

development.

Essential investments in the SUPRENUM development have been made for system and application software. Figure 3 gives an overview of the SUPRENUM system software, showing that the abstract SUPRENUM architecture is the central model here. It is described in Section 2.3.

The very high speed of the bus network makes this a very interesting machine for a wide range of applications, including those requiring long-range communication. No more than three communication steps are ever required between remote nodes. A prototype containing 4 clusters (64 processors) is already in operation, and a full machine with 16 clusters will be available by the end of 1990.

2.3 A Software Concept Based on Message-Passing

For MIMD multiprocessor computers with local memory, a software concept has turned out to be suitable that is based on a *process* system and on a *message-passing communication* handling. The process concept for SUPRENUM (the so-called *abstract*

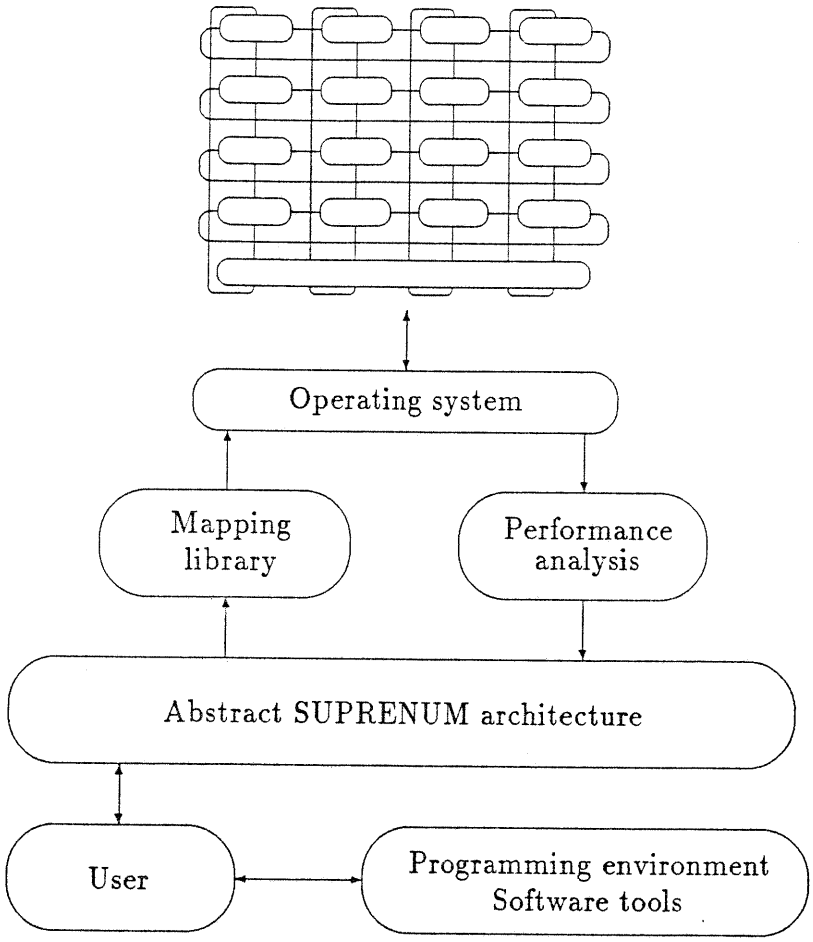


Figure 3: The SUPRENUM system software

SUPRENUM architecture) is a dynamic one which is characterized by the following elements:

- Processes are autonomous program units which run in parallel.
- Processes can terminate themselves, and can create but not terminate other processes.
- Processes communicate only by exchange of messages, and no shared memory is available.
- Applications are started by one initial (or host) process typically running on the front end machine.
- In arithmetic expressions and communication instructions, array constructs are supported.
- The user-defined process system is homogeneous and independent from the actual hardware configuration.

Processes are mapped to the real hardware at run-time. In SUPRENUM a *mapping-library* [31] provides optimized mapping strategies for some standard process systems (like trees, rings, grids) and uses heuristical strategies for irregular process structures.

3 Introduction to Multigrid

In the following, we briefly describe the basic structure of a general multigrid method (abstracted from [38], Section 2). For more details and discussions, we refer to the available introductory publications on multigrid methods. In particular, we recommend the proceedings [24], which, besides a collection of several specialized papers, contain three introductory contributions [5, 22, 50]. Other recommendable publications are [8], which is a more recent one of introductory type, and [23]. A multigrid bibliography up to 1986 can be found in [4].

In this paper we do not consider multigrid methods in a finite element context (see [1, 39, 43] or [4] for further references).

3.1 The Basic Idea and Algorithmical Structure

We first consider the case of a linear, scalar elliptic boundary value problem. The discretization of such a problem on a uniform grid of mesh size h yields a linear $N \times N$-system of equations (N being proportional to $1/h^2$) which we denote by

$$L^h u^h = f^h . \tag{1}$$

Here, one may regard L^h as a matrix and u^h, f^h as vectors. In the context of multigrid methods, however, it is more convenient to consider (1) as a *grid equation*, i.e., L^h as an operator acting on grid functions and u^h, f^h as grid functions. Here and in the following we will write *exact* solutions of grid equations in upper case letters, e.g., the solution of (1) will be denoted by U^h.

Solving (1) by means of classical methods such as elimination or relaxation is rather expensive if the number of unknowns is large. Standard relaxation methods are particularly inefficient. For instance, in the case of second order 2D elliptic equations, solving (1) by means of Gauss-Seidel relaxation (up to a fixed accuracy) needs computational work which is proportional to N^2 or $1/h^4$. Using accelerated methods like SOR, the best one can do is to reduce the work to be proportional to $N^{3/2}$ which is still very unsatisfactory in practice.

On the other hand, relaxation methods of Gauss-Seidel type are highly efficient in terms of *error smoothing*. To demonstrate this, consider point Gauss-Seidel applied to the discretized Poisson equation, which means that we solve the following equations point by point in some order

$$u^h_{ij} = \frac{1}{4}\,(u^h_{i-1,j} + u^h_{i+1,j} + u^h_{i,j-1} + u^h_{i,j+1} - h^2 f^h_{ij}), \tag{2}$$

using the most recent approximations for the values on the right hand side. Thus, in terms of the error $v^h = U^h - u^h$,

$$v^h_{ij} = \frac{1}{4}\,(v^h_{i-1,j} + v^h_{i+1,j} + v^h_{i,j-1} + v^h_{i,j+1}) .$$

This is just an averaging process, which explains heuristically why the error becomes smooth after very few relaxation sweeps. Although the situation here is quite special, relaxation methods with similarly good smoothing properties exist under quite general

circumstances. In fact, there is a close relation between *ellipticity* and the possibility of efficient smoothing (cf. Section 3.3). However, the simple point Gauss-Seidel method will usually not suffice as an effective smoother.

The goal of any multigrid method is to exploit smoothness by the use of coarser grids. Since the errors become smooth by relaxation (rather than the approximations themselves), coarse-grid corrections make sense only in terms of the error. Note that for any approximation u_{old}^h of (1), its error $V^h = U^h - u_{old}^h$ satisfies the *correction equation* $L^h v^h = r^h$ with the residual $r^h = f^h - L^h u_{old}^h$. Now, if V^h is sufficiently smooth, it makes sense to approximate it by solving a *coarse-grid-correction (CGC) equation* of the form

$$L^H v^H = r^H \quad \text{with} \quad r^H = I_h^H r^h , \tag{3}$$

where the (formal) index H marks coarser-grid quantities. Here, L^H denotes a reasonable discretization of the same differential operator on the coarser grid and I_h^H denotes some *restriction* which maps fine-grid functions by local averaging into coarse-grid functions. The coarser grid itself may be constructed by simply doubling the mesh-size h in all coordinate directions: $H = 2h$. This coarsening strategy, which is the one most frequently used, is called *standard coarsening*. However, since other choices will be discussed later, we keep the above formal notation.

By interpolating the solution V^H of (3) back to the finer grid (using some interpolation operator I_H^h), we finally obtain a new approximation to the solution of (1):

$$u_{new}^h = u_{old}^h + I_H^h V^H .$$

Summarizing, we present the formal structure of a general (iterative) *two-grid method*. Starting with u_{old}^h, one iteration step (one *cycle*) of such a method proceeds as follows (cf. Section 2.3 in [50]):

(1) Pre − smoothing : Compute $\overline{u}^h$ by applying $\nu_1 \geq 0$ sweeps of a given relaxation method to (1) with starting guess u_{old}^h:

$$\overline{u}^h = RELAX^{\nu_1}(u_{old}^h ; L^h, f^h)$$

(2) Coarse − grid correction :
- Computation and restriction of residual $r^H = I_h^H(f^h - L^h \overline{u}^h)$
- Computation of the exact solution V^H of $L^H v^H = r^H$
- Interpolation and correction $\hat{u}^h = \overline{u}^h + I_H^h V^H$

(3) Post − smoothing : Compute u_{new}^h by applying $\nu_2 \geq 0$ sweeps of the given relaxation method to (1) with starting guess $\hat{u}_h$:

$$u_{new}^h = RELAX^{\nu_2}(\hat{u}^h ; L^h, f^h)$$

For relaxation methods with good smoothing properties, ν_1 and ν_2 are small numbers, typically 1 or 2 (cf. the discussions in later sections). Such a two-grid method, although reflecting the main principles, is generally not suitable for an efficient use in practice. Solving the CGC equation (3) is still too expensive because the "coarse grid" is still very fine. However, as the method is iterative anyway, there is no need to compute V^H *exactly*. The most natural way to *approximate* V^H is to apply the same idea as described above to the CGC equation itself, using a still coarser grid.

More precisely, we replace V^H by an approximation v^H which is obtained by applying $\gamma \geq 1$ two-grid cycles to (3), starting with the zero grid function as first guess. By extending this idea in a recursive way to a sequence of coarser and coarser grids, we finally obtain a *complete multigrid cycle*. For more details and a flow chart see Sections 4.1, 4.2 in [50]. The parameter γ controls the accuracy to which the CGC equations are approximated. The smallest possible choice, $\gamma = 1$, is usually sufficient. Safer, but really needed only in "less regular" situations, is the choice $\gamma = 2$ (cf. Section 4.2). Corresponding cycles are often referred to as V- and W-cycles, respectively. In practice, for reasonable multigrid methods, larger values of γ are not needed. Apart from V- and W-cycles, we also consider F-cycles in some parts of this paper. The F-cycle needs more work than a V- but less than a W-cycle. It has been described, for instance, in [52].

3.2 Extensions of the Basic Idea

In the following, we introduce two important extensions of the procedure outlined above, namely, the treatment of *nonlinear equations* by means of the FAS-approach and the *full multigrid (FMG) strategy*. For more details, see [6], [50].

3.2.1 Nonlinear Problems

One obvious way to solve nonlinear problems is to linearize the given differential problem *globally* (e.g., by a *Picard-like method*: freezing non-linear expressions to old approximations, or by *Newton's method*, which is generally more robust) and then to solve the resulting sequence of linear problems by a linear multigrid method of the form described above. Another possibility is to suitably generalize the ideas described in the previous section. The resulting approach is called the *full approximation scheme (FAS)* and is – from a multigrid point of view – not only conceptually more straightforward (no global linearizations are needed) but in many cases also more efficient.

Assuming (1) to be nonlinear in u^h, we point out that smoothing can be performed as in the linear case merely by replacing relaxation by corresponding non-linear variants. Due to the lack of a superposition principle, however, we can no longer compute corrections on the coarser grids as in the linear case: instead of computing "correction quantities", we have to solve for "solution quantities". We still have to interpolate *smooth* quantities back to the finer grids to update old approximations. This leads to the following two-grid process for nonlinear problems. Its generalization to a complete FAS multigrid cycle is done in the same straightforward way as in the linear case.

(1) Pre – smoothing : Compute $\overline{u}^h$ by applying $\nu_1 \geq 0$ sweeps of a given (non-linear) relaxation method to (1) with starting guess u^h_{old}:

$$\overline{u}^h = RELAX^{\nu_1}(u^h_{old}; L^h, f^h)$$

(2) Coarse – grid correction :
- Computation and restriction of residual $r^H = I^H_h(f^h - L^h\overline{u}^h)$
- Restriction of current approximation $\overline{u}^H = I^H_h\,\overline{u}^h$
- Computation of the exact solution U^H of $L^H u^H = r^H + L^H\overline{u}^H$

- Computation of correction $V^H = U^H - \overline{u}^H$
- Interpolation and correction $\hat{u}^h = \overline{u}^h + I_H^h V^H$

(3) Post − smoothing : Compute u_{new}^h by applying $\nu_2 \geq 0$ sweeps of the given relaxation method to (1) with starting guess $\hat{u}^h$:

$$u_{new}^h = RELAX^{\nu_2}(\hat{u}^h; L^h, f^h)$$

Compared to the linear case, only the computation of the coarse-grid correction V^H has changed. The fine-to-coarse transfer of the temporary approximation $\overline{u}^h$ is done by means of some local averaging $I\!I_h^H$ which may be different from I_h^H. One possibility is to define $\overline{u}^H$ at each coarse-grid point to have just the value of $\overline{u}^h$ at the corresponding fine-grid point (*straight injection*). This simplest definition is the standard one and can safely be used unless the solution of (1) is expected to be strongly varying with respect to the scale of the grid.

Note that, for linear problems, the new procedure is exactly equivalent to the former one (independent of the concrete choice of $I\!I_h^H$). Note also that, assuming straight injection for $I\!I_h^H$, convergence of the above cycle implies that the solution U^H of the *coarse-grid* equation approaches the *fine-grid* solution U^h (injected to the coarser grid) rather than the solution of $L^H u^H = f^H$.

3.2.2 The Full Multigrid Approach

Once a multigrid cycle with all its necessary components is specified, it can be used to solve a given problem (1) iteratively. As with any other iterative method, there is the significant problem of finding a good first approximation. A convenient approach to finding a reasonable first guess is the combination of an iterative solver with the idea of *nested iteration*. This essentially means that, before starting to iterate on the given grid, one computes approximations on coarser grids which are interpolated to finer grids until one finally reaches the actual computational grid.

The combination of this idea with multigrid iterations (in contrast to its combination with SOR, say) is called the *full multigrid (FMG) method*. Because of the h-independent convergence of proper multigrid iterations, a *fixed* number of iterations on each level is sufficient. Moreover, performing *just one* cycle on each level (including the finest one), will generally suffice to yield a final approximation of (1) which is numerically correct *up to h-truncation error* without any further iterations on the finest level (cf. Section 6 in [50]).

3.3 Multigrid Components, Performance Analysis; Some General Remarks

The above methods provide only formal schemes. For a practical application the various *components* (e.g., discretizations on the finest and the coarser grids, relaxation for smoothing, coarsening strategy, inter-grid transfer operators, etc.) must be specified. Generally, this specification is by no means obvious. This is because a proper choice of the different components depends on certain characteristics of the differential and discrete problem (ellipticity, relative size of coefficients, singular perturbations, discontinuities, etc.) and often also on the solution itself (singularities, solution–dependent

coefficients, non–uniqueness, etc.). A detailed discussion would go beyond the scope of this paper. We want to give, however, a brief review of some basic but important techniques which have turned out to be quite useful in practice.

What is needed is some kind of realistic *a priori analysis* which, for a concrete problem, allows one to estimate the interactions between the possible multigrid components and to predict their influence on the final multigrid performance. The most natural tool is Fourier analysis, because in terms of Fourier components one can most easily distinguish low and high frequency components of any grid function and explicitly investigate their correct treatment by the CGC step and the relaxation separately. However, the application of Fourier analysis is feasible in practice only under certain simplified assumptions:

One possibility is to restrict attention to certain fundamental classes of model problems such as linear, symmetric, constant coefficient problems with Dirichlet, Neumann or periodic boundary conditions on rectangular domains. For such problems, optimal multigrid strategies have been developed which turned out to be more efficient than classicial solvers by orders of magnitude [50]. The major quantities which can explicitly be computed by means of Fourier analysis and which have been used for a systematic comparison of different multigrid strategies, are the *smoothing factor* μ_ν and the *two-grid convergence factor* ρ_ν. The latter quantity is just the factor by which the error is reduced by one two-grid cycle using ν relaxation sweeps for smoothing, while μ_ν is an approximation to ρ_ν assuming "optimal" CGC behavior. From a different point of view, the smoothing factor can be interpreted as the factor by which *high* frequency error components are damped by ν relaxation sweeps.

Since we will present some concrete results on the above quantities ρ_ν, μ_ν in the following chapter, we want to point out their significance: μ_ν allows one to distinguish different relaxations with regard to their smoothing qualities (roughly, a value of μ_1 around 0.5 indicates quite good smoothing, while a value close to 1 is absolutely unacceptable). The interplay of smoothing *and* coarse-grid correction is reflected in the more precise two-grid factor ρ_ν. Thus, significant deviations between these two quantities indicate that the coarse-grid components or the inter-grid transfers (or both) may have been chosen improperly. Assuming that the "nature" of all intermediate grid problems is comparable to the finest grid problem, ρ_ν can be expected to yield a quite precise prediction for the convergence rate of a corresponding complete cycle, at least of the W-cycle. In practice, provided the coarse grid components are suitable, μ_ν gives a sufficient prediction of the behavior of the final multigrid cycle, and is much easier to compute than ρ_ν.

In more complex situations, the above analysis can no longer be applied rigorously. In such cases, however, *local* (or *interior*) *Fourier analysis* which has been introduced and extensively used by A. Brandt [6], is still applicable.

The reader who is interested in more details is referred to [6] which contains important discussions on this issue and, in particular, general guidelines for designing relaxation schemes for complex situations. Concerning details on the rigorous "model" analysis mentioned before, see [50].

3.4 Some Special Considerations for Poisson-like Equations, Red-Black-Relaxation

By *Poisson-like equations*, we mean equations of the form

$$a(x,y)u_{xx} + b(x,y)u_{yy} = f(x,y) \tag{4}$$

with positive functions a, b and $a \approx b$ everywhere in a given domain (similarly in 3D). In the sense mentioned in the previous section, the performance of a multigrid method does not depend sensitively on the concrete values of the coefficients or on the shape of the domain (cf. Section 10 in [50]) or on the particular boundary conditions. In particular, the incorporation of specific boundary conditions requires only proper algorithmical adaptations at the boundary itself.

Thus, in order to devise a reasonable multigrid scheme for nearly isotropic problems, discretized by means of standard symmetric differences on a *uniform* grid of mesh size h, we may study the corresponding difference equations of the most simple isotropic model equation, Poisson's equation:

$$\Delta u = f(x,y)\,,$$

disregarding boundary conditions. Note that if the discretization step sizes are different in each direction, we have a *discrete* anisotropy, and the discussion of the next chapter applies. For the isotropic case, plain point Gauss-Seidel relaxation has quite good smoothing properties. Thus, for any discrete isotropic problem (4), it appears to be reasonable to use Gauss-Seidel relaxation (with "lexicographic" ordering of grid points, say) for error smoothing in conjunction with *standard coarsening.*

More precisely, one may use a sequence of uniform grids of mesh size h, $2h$, $4h$,..., up to a coarsest grid which contains as many points as necessary for technical reasons, along with the corresponding symmetric discretizations of the same differential operator (4) and Gauss-Seidel as smoother on each level. The inter-grid transfer operators are usually not critical. One can safely use *linear interpolation* for the corrections and some local averaging in the residual transfers, e.g., the so-called *full weighting.* This is defined (in matrix terminology) as the transpose of linear interpolation scaled such that a constant grid function is mapped into the same grid function on the next coarser grid. In stencil notation the full weighting operator can be written as

$$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}.$$

In order to understand the above method more quantitatively, one has to perform some Fourier analysis as pointed out before. Here, in Table 1, we give only some results for Poisson's equation in terms of μ_ν and ρ_ν for different values of ν. In particular, we see that $\mu_2 = 0.250$ and $\rho_2 = 0.193$. Due to the discussions on the significance of these quantities in the previous section, we can conclude that we have excellent smoothing and a proper coarse-grid correction. Also, there are no negative influences from much coarser grids, as the predicted convergence is precisely obtained in practice when using a complete multigrid cycle. We could have also used $\nu = 3$ or

problem: $\Delta u = f(x,y)$				
	lexicogr.		red-black	
ν	μ_ν	ρ_ν	μ_ν	ρ_ν
1	0.500	0.400	0.250	0.250
2	0.250	0.193	0.063	0.074
3	0.125	0.119	0.034	0.053

Table 1: Results obtained by Fourier analysis

even $\nu = 1$ smoothing steps for smoothing instead of $\nu = 2$; a careful comparison of the numerical work per cycle versus convergence speed, however, shows that $\nu = 2$ is the most efficient choice.

The above multigrid strategy is just one of a number of possibilities. In Table 1, we give one more example which is only slightly different from the above strategy. The only difference is that the order of points in the Gauss-Seidel relaxation is changed to "red-black", i.e. the points are divided into "red" and "black" points in a checkerboard fashion, first all red points are relaxed and then all black points. The resulting method is seen to be essentially more efficient. Using $\nu = 2$ smoothing steps in the multigrid process will give considerably more than one digit of error improvement per multigrid iteration. In particular, the latter algorithm is among the most efficient known on single processor computers: it is more efficient than the one above by about 50%. And, as we will see in Chapter 5, "red-black" relaxation is well suited for vectorization and parallelization.

4 More Sophisticated Multigrid Techniques

In this chapter (abstracted from [38], Section 3) we would like to discuss certain situations where the basic multigrid approaches as sketched in the previous chapter have to be modified and extended. Such situations are

- anisotropic operators, particularly in 3D
- influences of first order differential terms

With respect to systems of equations, we refer to [6].

4.1 Anisotropic Operators

We call equation (4) *anisotropic* if the coefficients are very different from each other in significant parts of the computational domain. This is a simple example which we can use to demonstrate that a careless choice of the necessary multigrid components may lead to quite inefficient solvers. If we apply the algorithm of the previous section without change, it will seriously deteriorate. In fact, the stronger the anisotropy, the slower the convergence. In our discussion, we will distinguish the 2D- and the 3D-case.

(a) The 2D – case:
Let us consider the anisotropic model equation

$$\varepsilon\, u_{xx} + u_{yy} = f(x,y) \tag{5}$$

with some constant $0 < \varepsilon \ll 1$, discretized on a uniform grid as above. Using pointwise Gauss-Seidel relaxation, the error version of (2) changes to

$$v_{ij}^h = \frac{1}{2(1+\varepsilon)} \left(\varepsilon\, v_{i-1,j}^h + \varepsilon\, v_{i+1,j}^h + v_{i,j-1}^h + v_{i,j+1}^h\right) \tag{6}$$

which is essentially an averaging process *in the y-direction.* Consequently, after a few relaxation sweeps, errors will become smooth in the y-direction, not, however, in the x-direction. If, on the contrary, $\varepsilon \gg 1$, it is just the other way around. Obviously, such errors can no longer be efficiently reduced by means of a coarser grid which is obtained by doubling the mesh size in *both* directions. Point relaxation and standard coarsening is not a reasonable combination for anisotropic problems! There are two possible remedies.

Line – relaxation: The first possibility is to keep standard coarsening but to change the relaxation procedure such that errors become smooth in *both* coordinate directions. This can be achieved by solving *simultaneously* for those unknowns which are *strongly connected.* That is, use Gauss-Seidel *line-relaxation* with the lines parallel to the y-axis *(y-line relaxation)* if $\varepsilon \ll 1$, and use *x-line relaxation* if $\varepsilon \gg 1$.

Semi – coarsening: Alternatively, one may keep point relaxation if one changes the coarsening strategy according to the one-dimensional smoothness of errors. Define the coarser grid by doubling the mesh size only in those directions in which errors are smooth. That is, double the mesh size only in the y-direction *(y-semi coarsening)* if $\varepsilon \ll 1$ and in the x-direction if $\varepsilon \gg 1$ *(x-semi coarsening).*

Both approaches can be directly applied to anisotropic variable coefficient problems (4) with the proviso: $\varepsilon \ll 1$ ($\varepsilon \gg 1$) represents the case that in some parts of the computational domain $a \ll b$ ($a \gg b$) while in the remaining parts $a \approx b$.

In practice, usually the first approach is used because it leads to quite simple algorithms even if we do not assume *anything* about the relative size of the coefficients a and b, i.e., if we allow that both $a \ll b$ *and* $a \gg b$ hold in different parts of the computational domain. In such situations, line-relaxation can still be used in connection with standard coarsening if it is applied *alternatingly*, i.e., one smoothing step is just one x-line sweep followed by one y-line sweep. The resulting multigrid method will, in particular, converge rapidly also in case of isotropic problems; it is, however, somewhat more expensive than the simpler scheme using point relaxation.

In contrast to this, the second approach requires, in the case of arbitrary coefficients, semi-coarsening of different directions in different parts of the domain depending on the local sizes of the coefficients. The corresponding algorithms are rather complicated, because the control of a proper coarsening has to be done in an automatic and adaptive way. Note that, even in cases of only "one-sided" anisotropies, the recursive coarsening process will not be as straightforward as in the first approach because the coarser grids are no longer uniform. For instance, in the case of (5) and $\varepsilon \ll 1$, the first coarser grid has the mesh sizes h and $2h$ in the x- and y-direction, respectively. The symmetric discretization of $\varepsilon\, u_{xx} + u_{yy}$ on this grid is given by

$$\frac{1}{4h^2}\left(4\varepsilon\, u^H_{i-1,j} + 4\varepsilon\, u^H_{i+1,j} + u^H_{i,j-1} + u^H_{i,j+1} - 2(1+4\varepsilon)u^H_{ij}\right).$$

Relaxing the CGC equation by pointwise Gauss-Seidel means, in terms of the error,

$$v^H_{ij} = \frac{1}{2(1+4\varepsilon)}\left(4\varepsilon v^H_{i-1,j} + 4\varepsilon v^H_{i+1,j} + v^H_{i,j-1} + v^H_{i,j+1}\right).$$

Note that, compared to (6), the anisotropy has *decreased.* Thus, in constructing the next coarser grid, one has to distinguish two cases: If $4\varepsilon \ll 1$ we have to continue by y-coarsening, but if $4\varepsilon \approx 1$ one should continue by standard coarsening.

In Table 2, we summarize some concrete results for different strategies to solve (5) with various ranges of ε. Here, we only consider standard coarsening. Concerning point relaxation, the order of points is assumed to be red-black. Correspondingly, in line-relaxation, the lines are scanned "zebra-wise", i.e., one first relaxes every other line, and afterwards the remaining ones. Grid transfers are assumed to be as in the previous section, i.e., linear interpolation for corrections and full weighting for residual transfers.

We note that, as before, μ_ν and, in particular, ρ_ν very precisely predict the convergence of corresponding complete multigrid cycles. The results clearly show that the use of point relaxation is limited to values of ε which are not too different from 1 (not much larger than 2 or much smaller than 0.5, say). On the other hand, y-line relaxation is mainly suitable for $\varepsilon \leq 1$, while alternating line relaxation gives an efficient method for any value of ε. All results directly carry over to variable coefficient problems by looking at the worst frozen coefficient cases.

problem: $\varepsilon u_{xx} + u_{yy} = f(x,y)$							
relaxation	ν	ε	μ_ν	ρ_ν	ε	μ_ν	ρ_ν
point	3	0.5	0.088	0.088	2	0.088	0.088
y-line	2		0.053	0.028		0.198	0.198
alt-line	2		0.020	0.013		0.020	0.013
point	3	0.1	0.564	0.564	10	0.564	0.564
y-line	2		0.053	0.047		0.683	0.683
alt-line	2		0.041	0.038		0.041	0.038
point	3	0.01	0.942	0.942	100	0.942	0.942
y-line	2		0.053	0.052		0.961	0.961
alt-line	2		0.051	0.051		0.051	0.051

Table 2: Results obtained by Fourier analysis

(b) The 3D – case:

The 3D-case is considerably more involved than the 2D-case. We do not want to discuss all possible 3D-approaches, but rather point out some important differences to the 2D-situation. For a detailed investigation we refer to [52] which gives a complete survey on the various 3D-strategies.

We have stated above that, in 2D-problems, one can always use standard coarsening, if it is combined with *alternating line relaxation* for smoothing. This may make one believe that one obtains a similarly "robust" multigrid method in 3D by a straightforward generalization, namely, by using line relaxation now alternating with respect to all 3 coordinate directions. This is, however, by no means true! To outline some aspects, let us consider the 3D-model equation

$$a\,u_{xx} + b\,u_{yy} + c\,u_{zz} = f(x,y,z), \tag{7}$$

with at least one coefficient being significantly different from the others (otherwise we can use point relaxation for smoothing). The general rule which carries over from the 2D-case is that we obtain good smoothing of errors in all coordinate directions if we relax *all* strongly coupled unknowns *simultaneously*. For instance, if $a \approx b \ll c$, we may safely use z-line relaxation. Consequently, (triple) alternating line relaxation will provide a perfect smoother independent of which coefficient is large *as long as the remaining two are approximately the same.*

If, however, $a \ll b \approx c$, the situation is different as (according to the above rule) we now have to use (y,z)-*plane relaxation.* In contrast to line relaxation (which leads to simple tri-diagonal systems of equations), such a relaxation cannot be efficiently performed by standard solvers for banded matrices. In fact, the only way to perform such a relaxation efficiently is by using a (2D-) multigrid method for each plane. (Some recommendations on how to proceed are contained in [52].) An alternative to the use of plane relaxation is, similar to the 2D-case, to use point relaxation instead but combined with (y,z)-*semi coarsening* (i.e., doubling the mesh sizes with respect to y and z only).

Finally, if $a \ll b \ll c$, the situation is similar. As before, proper smoothing is guaranteed by *(y,z)-plane relaxation* combined with standard coarsening. However, if (y,z)-plane relaxation is performed by multigrid, one has to observe that there is an anisotropy in each plane which implies that one should use z-line relaxation for smoothing inside the plane-multigrid method. For the same reason, it can easily be seen that *(y,z)-semi coarsening* is possible but has to be combined with z-line relaxation rather than point relaxation.

In Table 3, we summarize appropriate strategies to solve (7) for some extreme constant coefficient cases. As before, these results are of direct significance for general variable coefficient problems. Again, the two-grid convergence factors shown, will also be obtained by complete multigrid cycles. We want to explicitly point out, however, that in order to judge which strategy is really the most efficient, it is not sufficient to merely look at convergence (see, e.g., the very last method in the table, which converges extremely fast), but one has to take the numerical work per cycle into account. We are not going to discuss this any further here.

Summarizing, we see that the development of a 3D-multigrid algorithm needs a careful investigation of the problem at hand. A smoothing process which, in connection with standard coarsening, certainly takes care of all possible anisotropies in variable coefficient cases is *alternating plane relaxation.* If, however, one allows for some restriction on the size of the coefficients (e.g., that one particular coefficient is always the smallest one) one should seriously consider using semi-coarsening, this way avoiding plane-relaxation.

We will come back to the anisotropic 3D case in some detail in Section 5.3.

problem: $a\,u_{xx} + b\,u_{yy} + c\,u_{zz} = f(x,y,z)$						
a	b	c	relaxation	coarsening	ν	ρ_ν
1	1	1	point	standard	2	0.198
1	1	100	z-line	standard	2	0.074
			point	z-semi	2	0.017
1	100	100	(y,z)-plane	standard	2	0.052
			point	(y,z)-semi	2	0.074
1	100	10000	(y,z)-plane	standard	2	0.052
			z-line	(y,z)-semi	2	0.052
			point	z-semi	2	0.009

Table 3: Results obtained by Fourier analysis

4.2 First Order Differential Terms

Additional first order derivatives in (4) do not introduce any new problem if their coefficients are "small enough". That is, using symmetric differences, the discrete equations can be solved efficiently with the same techniques as explained in the pre-

vious two sections. In particular, the choice of a proper relaxation technique for smoothing has to be oriented only to the principle part of the differential equation.

However, the situation changes if such lower order terms become dominant on a grid with meshsize h. As an example, consider the linear convection-diffusion equation

$$\varepsilon \Delta u + a u_x + b u_y = f(x,y).$$

As long as $\varepsilon \geq \frac{h}{2} \max(|a|, |b|)$, central differencing leads to diagonally dominant difference schemes – a property, which assures *numerical stability*. If this condition is (considerably) violated, the difference scheme becomes unstable: The discrete solution may become highly oscillating and will have nothing to do with the differential solution. The most common remedy for numerical instabilities is the use of either some kind of artificial viscosity (i.e., enlarging ε depending on h) or one-sided (upstream) differencing for the first order derivatives.

In the multigrid context, however, it is not enough to have a stable difference scheme for the fine–grid mesh size h only, because on coarser grids with much larger mesh sizes the same scheme (e.g., central differencing) may still become unstable even for first order terms of moderate size. Even if this happens on only a few of the grids used in the multigrid cycle, the overall multigrid convergence may be totally spoiled. The multigrid iteration may even diverge.

Note that the straightforward recursive definition of a *fixed* multigrid cycle, in particular, the V-cycle, requires implicitly that all "intermediate" two-grid methods exhibit similar convergence properties. In particular, the smoothing effects of the relaxation should be similar on all grids. In terms of error components, a loss of numerical stability on any grid would mean that there are some high frequency components showing no or at least only small residuals. Relaxations which usually compute the error changes only in dependence upon the (local) residuals, will not affect these high frequency components essentially and, consequently, they will loose their smoothing properties. Thus, in designing a multigrid algorithm, one has to assure that *all* the difference operators used on the various grids are actually stable discretizations (may be of lower order) of the same differential operator.

The problem of deciding whether a given difference scheme for complex problems is sufficiently stable, generally needs some *quantitative* insight into what stability really means. Such a quantification of stability is given by the so-called *h-ellipticity measures* introduced by A. Brandt (in terms of Fourier components). These are easy-to-compute measures for the stability of difference schemes for a fixed meshsize h and actually quite useful in developing and analyzing discretizations even for complex problems. In this framework, also the strong interdependency of stability and smoothing already mentioned above can be stated more precisely. Brandt points out, that *discrete ellipticity* (i.e. h-ellipticity measures sufficiently bounded away from zero) is, generally, a necessary and sufficient condition for the existence of smoothing relaxations. The interested reader is referred to [6].

5 Standard Parallel Multigrid

5.1 Some General Remarks

In this chapter we will consider the parallelization of standard MG methods. On each grid level we perform each of the basic grid operations (the MG components: relaxation, computation of defects, interpolation, and restriction) using as much parallelism as possible. It has long been known that certain relaxation methods are parallel in a natural way, namely

- *Jacobi*-type relaxations

and

- *Gauss-Seidel*-type relaxations with *multi-color* (red-black, four color etc.) ordering of the grid points.

Clearly, computation of defects, interpolation and restriction can be also performed in parallel.

The first systematical papers on parallel MG were those of Grosch [20, 21] and Brandt [7]. In [7] most of the essential phenomena with parallel MG are already discussed or at least mentioned. In particular, it is stated that the time complexity $T^*(N)$[1] of a suitable standard parallel full multigrid (FMG) solver for the 2D-Poisson model equation is $T^*(N) = O\left(log_2 N\right)^2$, where N = number of grid points.

We will consider three cases: Poisson's equation, Stokes equations and the anisotropic 3D operator. Each of these cases represents an essentially larger class of equations (Poisson-like, Stokes-like, etc.) to which the considerations of this chapter carry over immediately. Parallelization is obvious and straightforward for the Poisson and Stokes cases (Section 5.2). Parallel smoothing for the anisotropic case (Section 5.3) has to be adapted to the problem parameters as in the sequential case (Section 4.1). Common grid transfer operators such as bilinear (in 2D) or trilinear (in 3D) interpolation, full weighting, half weighting, and injection are parallel by nature. Thus our main emphasis in this chapter will be the parallelization of smoothing schemes.

5.2 Isotropic Equations and Systems

1. We consider a parallel MG-solver for the *3D-Poisson equation* on the unit cube $(0,1)^3$ with periodic boundary conditions. A V-cycle of this algorithm is characterized by the following components.

Discrete operator:	ordinary second order 7-point approximation Δ_h on a regular cubic grid with meshsize h and $N = h^{-3}$ unknowns.
Relaxation:	3D-red-black pointwise, all red (black) grid points are simultaneoulsy treated in the first (second) relaxation half step; $\nu_1 = 1 = \nu_2$ relaxation steps.
Coarsening:	standard coarsening: $h \longrightarrow 2h$ ordinary 7-point operator Δ_{2h} on the coarse grid.

[1]The time complexity $T^*(N)$ measures the sequential (= non-parallel) arithmetic overhead of an algorithm. It may be defined as the number of parallel arithmetic operations which have to be executed sequentially, i.e. one after the other.

Grid transfers:	$h \longrightarrow 2h$: 3D full-weighting;
	$2h \longrightarrow h$: trilinear interpolation.
Cycle type:	V-cycle, correction scheme

The time complexity for a V-cycle of this algorithm is $T^*(N) = O(log_2 N)$.

2. As a simple example of a system of elliptic equations, we consider the *2D-Stokes equations*

$$\begin{aligned} -\nabla^2 u + \frac{\partial p}{\partial x} &= f^1 \\ -\nabla^2 v + \frac{\partial p}{\partial y} &= f^2 \\ \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} &= f^3 \end{aligned}$$

defined on $\Omega = (0,1)^2$ with boundary conditions

$$\begin{aligned} u &= g^1 \\ v &= g^2 \end{aligned}$$

on $\partial\Omega$. In order to guarantee a unique solution the usual compatibility condition is additionally required.

The Stokes equations are discretized in the usual way on a staggered quadratic grid (meshsize h): p is defined at cell centers, whereas u and v are defined at the centers of the cell faces.

A V-cycle of a parallel MG-solver for this discrete problem is characterized by the following components:

Relaxation:	One relaxation step consists of two parts: Firstly, the momentum equations are relaxed for u and v simultaneously using fixed values of p. Then a so-called distributive relaxation sweep [6] is performed which updates the unknowns u, v and p in order to fulfill the continuity equation. Both parts of the relaxation are performed using a red-black ordering. Altogether, $\nu_1 = 1$, $\nu_2 = 2$ of these relaxation steps are carried out on each level.
Coarsening:	standard coarsening $h \longrightarrow 2h$ on staggered grids (the coarse grid is not a subset of the fine grid); ordinary 7-point operator Δ_{2h} used on coarse grid.
Grid transfers:	$h \longrightarrow 2h$: 2D half-weighting on staggered grids
	$2h \longrightarrow h$: bilinear interpolation
Cycle type:	V-cycle, correction scheme

The time complexity of this algorithm (V-cycle) is also $T^*(N) = O(log_2 N)$.

In the two algorithms above only *pointwise* relaxation is needed for smoothing, since the corresponding equations are isotropic. For the anisotropic case in the following section, we would need parallel versions of line and plane-relaxations. This means that things become somewhat more complicated.

5.3 Parallel Multigrid for Anisotropic Operators

In this section (abstracted from [53]) we describe parallel MG algorithms for the anisotropic 3D model operator

$$Lu = au_{xx} + bu_{yy} + cu_{zz} \tag{8}$$

of Section 4.1.

This operator can be regarded as representative of a large class of elliptic 3D problems. With respect to its anisotropy, it reflects certain general phenomena and typical 3D difficulties. On the other hand, the operator is simple enough to allow systematic investigations.

We consider only standard MG algorithms which have proven to be highly efficient for sequential treatment. We show that these algorithms are also highly parallel in a natural way. This is clear for the isotropic case

$$0 < a = b = c$$

which has been considered in Section 5.2. The red-black relaxation is known to give very good smoothing and to be highly parallel. On the other hand, as long as standard coarsening is maintained, in the cases

$$0 < a, b \ll c \qquad (a, b \text{ same magnitude})$$

and

$$0 < a \ll b, c \qquad (b, c \text{ arbitrary}),$$

the need for *line-relaxation* and *plane-relaxation* respectively for good smoothing, was mentioned in Section 4.2. (We consider here only standard coarsening; see, however, the remark at the end of this section.)

We consider the elliptic equation

$$Lu = f^{\Omega}(\Omega) \qquad \text{with periodic boundary conditions}$$

in the unit cube $\Omega = (0,1)^3$, where L is the model operator (8) with constant coefficients $a, b, c > 0$. We assume that the problem is discretized on an Ω-matching grid Ω_h of mesh size

$$h = \tfrac{1}{n} \qquad (n = 2^l;\ l = 0, 1, 2, ...)$$

by use of the ordinary 7-point approximation L_h of order h^2.

In the sequel, we describe parallel MG algorithms for this discrete problem. For this purpose, it is useful to distinguish the four cases

$$\begin{array}{lllll} \text{(3D-1)}: & a & \sim & b & \sim & c \\ \text{(3D-2)}: & a & \sim & b & \ll & c \\ \text{(3D-3)}: & a & \ll & b & \sim & c \\ \text{(3D-4)}: & a & \ll & b & \ll & c \end{array}$$

In addition to the 3D case, we also consider the corresponding 2D operator $au_{xx} + bu_{yy}$

$$(2D\text{-}1): \quad a \sim b$$
$$(2D\text{-}2): \quad a \gg b$$

as well as the 1D case (operator au_{xx}, discretized by second order central differences). In particular, the 2D cases and the 1D case occur in an auxiliary way in connection with the 3D algorithms described below (parallel solution of 2D and 1D problems in the plane- and line-relaxation, respectively).

Using the terminology of [50], we describe for each of the cases one step of a parallel standard MG iteration, listing the corresponding MG components.

(1D): The corresponding tridiagonal linear system is solved by 1D-*cyclic reduction*. This is a well-known parallel linear solver. (It can be regarded as a parallel 1D-MG method, namely a V-cycle degenerating to a direct linear solver. Its components are $\nu_1 = 0$, $\nu_2 = 1$ relaxation steps, only fine grid points not belonging to the coarse grid are treated in the relaxation step, standard coarsening $h \longrightarrow 2h$, L_{2h} is the ordinary 3-point operator, 1D full weighting, interpolation only to those fine grid points which belong also to the coarse grid.)

(2D-1) 2D-Poisson-like equation:

Relaxation:	2D-red-black pointwise; $\nu_1 = 1$, $\nu_2 = 0$ relaxation steps; all red (black) grid points are simultaneously treated in the first (second) relaxation half step.
Coarsening:	standard coarsening $h \longrightarrow 2h$; L_{2h} is the ordinary 5-point operator.
Grid-transfers:	$h \longrightarrow 2h$: 2D full weighting; $2h \longrightarrow h$: bilinear interpolation; all grid transfer operations are simultaneously performed for the respective grid points.
Cycle-type:	V-cycle (or F,W-cycle); correction scheme.

(2D-2) 2D-anisotropic equation:

Relaxation:	y-linewise in a red-black-zebra order of lines; $\nu_1 = 1$, $\nu_2 = 0$ relaxation steps; all red (black) zebra lines are simultaneously treated in the first (second) relaxation half step; for each line, the parallel 1D-cyclic reduction algorithm is applied.

All other components are chosen as in the (2D-1) case.

(3D-1) 3D-Poisson-like equation:

Relaxation:	3D-red-black pointwise; $\nu_1 = 1$, $\nu_2 = 1$ relaxation steps; all red (black) grid points are simultaneously treated in the first (second) relaxation half step.

Coarsening: standard coarsening $h \longrightarrow 2h$;
L_{2h} is the ordinary 7-point operator.

Grid-transfers: $h \longrightarrow 2h$: 3D full weighting;
$2h \longrightarrow h$: trilinear interpolation;
all grid transfer operations are simultaneously performed for the respective grid points.

Cycle-type: V-cycle (or F,W-cycle);
correction scheme.

(3D-2) 3D anisotropic equation, one dominant direction:

Relaxation: z-linewise in a 2D-red-black order of lines;
$\nu_1 = \nu_2 = 1$ relaxation steps;
all red (black) lines are simultaneously treated in the first (second) relaxation half step;
for each line, the parallel 1D-cyclic reduction algorithm is applied.

All other components are chosen as in the (3D-1) case.

(3D-3) 3D anisotropic equation, two dominant directions:

Relaxation: (y,z)-plane relaxation in a red-black-zebra order of planes;
$\nu_1 = \nu_2 = 1$ relaxation steps;
all red (black) planes are treated simultaneously in the first (second) relaxation half step;
for each plane, one V-cycle of the parallel (2D-1)-MG algorithm is applied.

All other components are chosen as in the (3D-1) case.

(3D-4) 3D anisotropic equation, each coefficient of different size:

Relaxation: (y,z)-plane relaxation in a red-black-zebra order of planes;
$\nu_1 = \nu_2 = 1$ relaxation steps;
all red (black) planes are treated simultaneously in the first (second) relaxation half step;
for each plane, one V-cycle of the parallel (2D-2)-MG-algorithm (with z-line relaxation!) is applied.

All other components are chosen as in the (3D-1)-case.

Result 5.1 *Table 4 contains the time complexity $T(N)$ for one (V-, F-, W-) cycle of each of the algorithms described above.*

As a simplifying measure of the time complexity, we use "dimensional-weighted stencil units". This means that we count:

3 for the parallel application of a 3D stencil to a 3D grid function (or for the parallel calculation of a 3D defect);

2 for the parallel application of a 2D stencil to a 2D grid function (or for the parallel calculation of a 2D defect);

1 for the parallel application of a 1D stencil to a 1D grid function.

	V-Cycle	*F-Cycle*	*W-Cycle*	ρ^*
(3D-1)	$15l \ +3$	$\frac{15}{2}l^2 + \frac{21}{2}l \ +3$	$18n-15$	0.2
(3D-2)	$2l^2 + 19l \ +3$	$\frac{2}{3}l^3 + \frac{21}{2}l^2 + \frac{77}{6}l \ +3$	$O(n)$	0.07
(3D-3)	$12l^2 + 35l \ +3$	$4l^3 + \frac{47}{2}l^2 + \frac{45}{2}l \ +3$	$O(n)$	0.05
(3D-4)	$\frac{4}{3}l^3 \ +18l^2 + \frac{119}{3}l \ +3$	$\frac{1}{3}l^4 \ +\frac{20}{3}l^3 + \frac{175}{6}l^2 + \frac{155}{6}l \ +3$	$O(n)$	0.05
(2D-1)	$6l \ +2$	$3l^2 + 5l \ +2$	$8n-6$	0.25
(2D-2)	$l^2 + 8l \ +2$	$\frac{1}{3}l^3 + \frac{9}{2}l^2 + \frac{37}{6}l \ +2$	$O(n)$	0.125
(1D-1)	$2l \ +1$	—	—	0.

Table 4: Time complexity $T(N)$ and convergence factors ρ^*

Here we have used the notations

$N = n^d = 2^{dl}$ *(d = dimension of the problem, l = number of levels, n = number of grid intervals in each coordinate direction.)*

$\rho^* = \sup \rho(M_h^{2h})$ *(cf.[50]).*

From Table 4 we recognize that the time complexity $T(N)$ is polynomial in $l = O(\log N)$ for V- and F-cycles (and linear in n for W-cycles). The polynomial degree depends on the anisotropy of the operator considered. This is due to the fact that we use auxiliarily 2D-MG cycles or 1D-cyclic reductions (with time complexity of order $\geq O(\log N)$) in the plane- and the line-relaxations, respectively. In the sequential case, such auxiliary lower dimensional MG cycles do not change the asymptotic optimality of the 3D-MG algorithms.

In Section 4.2 we have mentioned also certain *semi-coarsening* strategies in connection with *point relaxation for the cases* (3D-2), (3D-3), (3D-4), and which lead to efficient 3D-MG methods. The grid structure of these algorithms is somewhat more complicated than the grid structure of standard coarsening algorithms. However, *one can avoid the increase of the degree of l in $T(N)$ by using semi-coarsening strategies.*

On the other hand, with respect to the practical implementation of parallel 3D-MG algorithms on real MIMD systems, the polynomial increase of $T(N)$ in $\log N$ does not seem to be significant (see also Section 6.2).

Finally, we remark that the time complexity measures of all parallel algorithms above have to be multiplied by a *factor of* $O(\log N)$ if we *consider full* multigrid versions instead of single cycles.

6 Grid Partitioning, General Grid Structures, Implementation Aspects

6.1 Grid Partitioning for Regular Grids

In the previous chapter, we have considered standard highly parallel MG algorithms implemented on hypothetical systems which have as many processors as desired. However, when they are implemented on real vector or parallel computers, it is not usually possible to fully exploit their inherent parallelism.

For example the pipelined processors in vector computers allow only a low degree of parallelism to be achieved. Even highly parallel multiprocessor computers always have a certain limited number of simultaneously working processing elements (say P). Nevertheless, *the high degree of parallelism in the algorithms* is useful or even necessary for several reasons. Firstly, it is preferable to construct algorithms which can be used on any parallel machine independently of P. Secondly, the full performance of vector units can usually be achieved only by using long vectors, and so require an algorithm with a high degree of parallelism. Finally, the recently designed high performance MIMD multiprocessors (such as SUPRENUM) combine the – global – MIMD structure with – local – SIMD pipeline processing (e.g. by vector floating point units) in each node. For such MIMD/SIMD systems, the MIMD and the SIMD degrees of parallelism are effectively *multiplied* provided that both are supported by the implemented algorithm.

We would like to emphasize that the communication problem in MIMD multiprocessors with distributed memory has essential algorithmical implications. Since for such systems one has to make a decision about the interconnection structure of the nodes, this structure defines a "neighborhood" and by that, a topology of the nodes in a natural way. In the design of the algorithms this topology has to be taken into account. In addition to the parallelism that has to be provided by the algorithms, a second important property, "locality" of the algorithms with respect to the given topology, is required. This means that the amount of data which has to be communicated, the number of communication packages, and the distances which have to be spanned in the architecture become of essential significance.

If grid applications are to be implemented on MIMD multiprocessor computers, a straightforward approach is to use *grid partitioning*. For all methods, whether single grid or MG, this means that the original domain is split into P parts (subdomains) in such a way that, with respect to the finest grid, each subdomain consists of (roughly) the same number of grid points (see Figure 4). Each subdomain is then assigned to one of the P processes of the parallel program. The partitioning generates certain artificial boundaries within the original domain.

If we consider a typical component of a parallel grid algorithm, e.g. a parallel relaxation step, we see that on each subdomain this relaxation step can be carried out independently, provided all necessary data are available. Because there are only local interdependencies of the grid points, each process needs foreign data only from boundary areas of neighboring subdomains. After the step is performed, again data have to be communicated (exchanged) across the artificial boundaries (see Figure 5).

The extension of the single grid case to parallel MG is obvious: On the finest grid

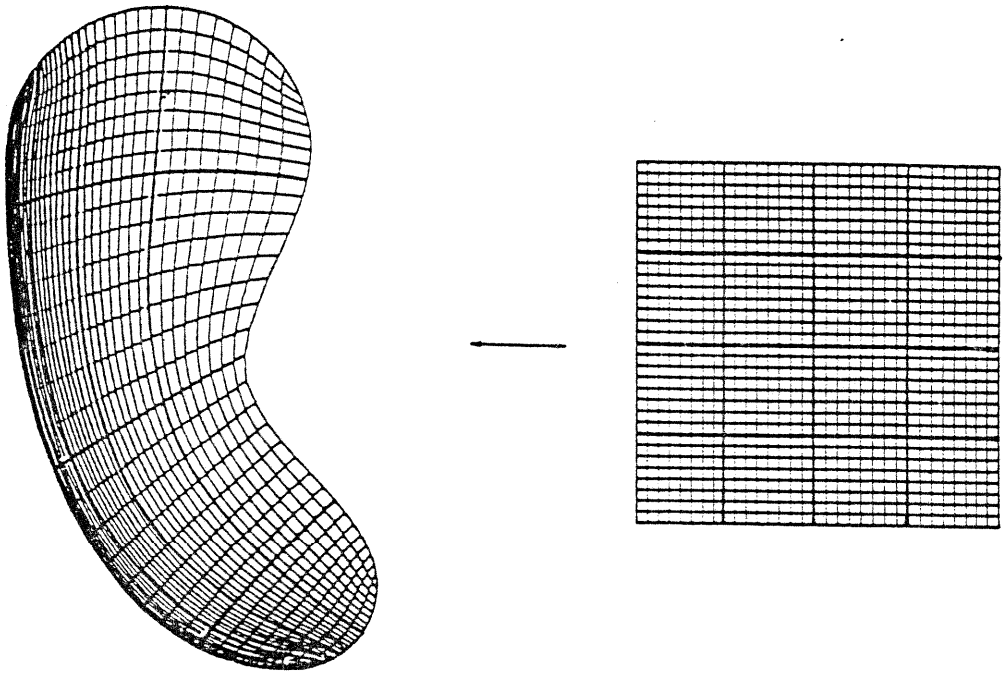


Figure 4: 2D-grid partitioned into 16 logically rectangular subdomains.

level, all communication is strictly local. Similarly, on the coarser grids necessary communication is "local" relatively to the corresponding grid level (i.e. neighborhood is defined with respect to the grid level).

One should distinguish the grid partitioning approach as sketched above from the domain decomposition or substructuring methods which are often considered in connection with finite element or finite difference discretizations on parallel computers. The decomposition and substructuring methods lead to algorithms which are numerically different from the undecomposed or sequential version. In contrast to that, parallel algorithms based on grid partitioning are algorithmically equivalent to their non-partitioned versions (running on sequential computers) in many cases.

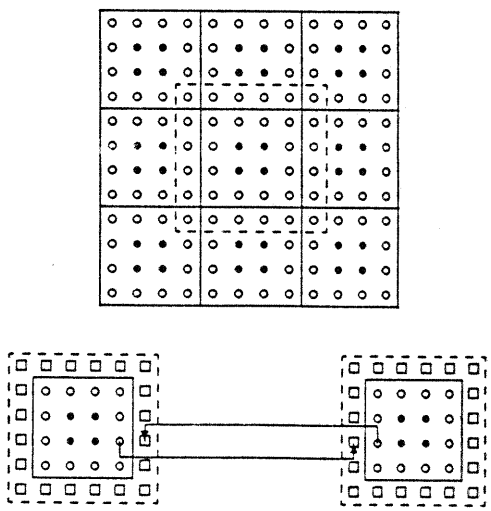


Figure 5: Overlap areas and their exchange.

6.2 Grid Partitioning for the Anisotropic 3D Operator

As a very simple example of the grid partitioning approach, we consider here a problem in the *unit cube*, namely the anisotropic 3D problem as discussed in Section 5.3. Since the corresponding algorithms may become quite complicated (with the recursive structure of plane and line-relaxation), we confine ourselves to the discussion of *algorithmic* aspects which arise when these algorithms are implemented on MIMD computers. The *communication aspects* are *not* explicitly discussed here as we are mainly interested in results which are hardware independent and optimal in the sense that communication overhead is neglected. (This material is abstracted from [53].)

The communication overhead has been investigated very systematically in many papers of Mierendorff and Kolp (for simpler MG test algorithms). They have considered all standard topologies (such as arrays, trees, hypercubes, etc.), but also more sophisticated ones like the SUPRENUM topology (see for example [30]). A somewhat simplified SUPRENUM-like topology will be studied in chapter 7.

Clearly, there are many reasonable ways to partition the given grid (domain) into subgrids (subdomains) and to map them onto the multiprocessor architecture.

For simplicity, we consider here only the most straightforward partitioning and mapping.

We first assume that there are $P = p^d \equiv 2^{dq}$ parallel working processor nodes available in the architecture (d: dimension of the problem). We consider the discrete 3D problem described in section 5.3 with $N = n^d \equiv 2^{dl}$ grid points (n: number of grid points in each coordinate direction). Each of the 3 coordinates of the unit cube is now divided into 2^q equidistant intervals so that Ω consists of 2^{dq} subcubes Ω_{ijk} $(i,j,k = 1,...,2^q)$.

Distinguishing the cases (3D-1), (3D-2), (3D-3), (3D-4), we then apply the respective 3D-MG algorithms. Using grid partitioning, these algorithms are decomposed and applied to the subgrids in a natural way. Communication (exchange of data) is required only along the "interior" boundaries of the subcubes Ω_{ijk}. The degree of parallelism of these decomposed algorithms is determined by P, the number of processing nodes. We assume that the problem size is essentially larger than the "system size", i.e. that $N > P$.

Result 6.1 *Under the above assumptions, the time complexity $T(N,P)$ for the decomposed 3D-MG algorithms (3D-1), (3D-2), (3D-3), (3D-4) is given by Table 5. Additional results refer to the corresponding 2D-cases.*

The $T(N,P)$ expressions are valid for arbitrary N and P as long as $l > q$. However, in practice one will usually regard P as fixed and N as variable ($N \longrightarrow \infty$).

The leading terms which have the form $const * N/P$ clearly coincide for $P = 1$ with the leading terms of the complexity for the corresponding sequential algorithms.

We have not explicitly specified the minor terms $O(...)$ here since they are very lengthy. In order to give an impression of the size of the above expressions and, in particular, of the relation between leading and minor terms, we consider a concrete example in Table 6.

	V-Cycle	F-Cycle	W-Cycle
(3D-1)	$\frac{60}{7}\frac{N}{P} + \frac{105q-39}{7}$	$\frac{480}{49}\frac{N}{P} + O(q^2, lq)$	$10\frac{N}{P} + 18n - 25\frac{n}{p}$
(3D-2)	$\frac{76}{7}\frac{N}{P} + O(q^2, q\frac{n^2}{p^2})$	$\frac{6084}{49}\frac{N}{P} + O(q^3, q^2l, q\frac{n^2}{p^2})$	$\frac{38}{3}\frac{N}{P} + O(n, q\frac{n^2}{p^2})$
(3D-3)	$\frac{124}{7}\frac{N}{P} + O(q^2, q\frac{n}{p})$	$\frac{992}{49}\frac{N}{P} + O(q^3, q^2l, q\frac{n}{p})$	$\frac{62}{3}\frac{N}{P} + O(n, q^2\frac{n}{p}, ql\frac{n}{p})$
(3D-4)	$\frac{436}{21}\frac{N}{P} + O(q^3, q\frac{n^2}{p^2}, q^2\frac{n}{p})$	$\frac{3488}{147}\frac{N}{P} + O(q^4, q^3l, q^2\frac{n}{p}, q\frac{n^2}{p^2})$	$\frac{218}{9}\frac{N}{P} + O(n, q^3\frac{n}{p}, q^2l\frac{n}{p}, q\frac{n^2}{p})$
(2D-1)	$4\frac{N}{P} + 6q - 2$	$\frac{16}{3}\frac{N}{P} + O(q^2, ql)$	$6\frac{N}{P} + 8n - 12\frac{n}{p}$
(2D-2)	$\frac{16}{3}\frac{N}{P} + O(q^2, q\frac{n}{p})$	$\frac{64}{9}\frac{N}{P} + O(q^3, q^2l, q\frac{n}{p})$	$8\frac{N}{P} + O(n, ql\frac{n}{p}, q^2\frac{n}{p})$

Table 5: Time complexity $T(N,P)$, neglecting communication overhead

Here, we compare the exact $T(N,P)$ values (bold numbers) with $T(N,1)/P$ where $T(N,1)$ gives the sequential time complexity. The ratios

$$E(N,P) = \frac{T(N,1)/P}{T(N,P)} = \frac{T(N,1)}{P*T(N,P)}$$

are the *efficiency* values of the decomposed algorithms (*neglecting communication!*). See also section 7.1.

	V-Cycle			F-Cycle			W-Cycle		
	T(N,P)	T(N,1)/P	E(N,P)	**T(N,P)**	T(N,1)/P	E(N,P)	**T(N,P)**	T(N,1)/P	E(N,P)
(3D-1)	**35148**	35108	.999	**40374**	40123	.994	**42864**	40958	.965
(3D-2)	**49298**	44556	.904	**57506**	50937	.886	**61392**	52007	.847
(3D-3)	**73716**	72555	.984	**85750**	82919	.967	**93552**	84639	.905
(3D-4)	**95800**	85206	.889	**113086**	97406	.861	**124208**	99444	.801
(2D-1)	**16406**	16383	.999	**22042**	21845	.991	**32000**	24560	.768
(2D-2)	**23024**	21853	.949	**31668**	29143	.920	**47808**	32784	.686

Table 6: Evaluated time complexities T(N,P) for
$P = 512 \quad N = 2^{21} \approx 2$ million in the (3D) case
$P = 256 \quad N = 2^{20} \approx 1$ million in the (2D) case

From the values in Table 6 we recognize that the efficiency is high or at least satisfactory in all cases. The influence of the very coarse grids is not an essential problem, even in the cases of F- and W-cycles. This is true even though we have used a very simple and general partitioning rather than partitionings that are optimized for the specific cases. Nevertheless, there are many interesting observations that can be made in connection with Table 5 and Table 6. We will conclude with one such observation.

We recognize that in those V-cycle cases where line-relaxation is used the efficiency of the respective 3D-algorithms is essentially worse than in the cases of point- or "pure"

plane-relaxation. The reason for this is the influence of the very coarse grids and the partitioning we have used. Compare, for example, the cases (3D-2) and (3D-3). For (3D-2), in each relaxation step n^2 1D-problems have to be solved and therefore $O(n^2)$ "very coarse grid visits" have to be made. On the other hand, in each plane-relaxation step in (3D-3), "only" n 2D-problems are treated and therefore $O(n)$ very coarse grid visits are necessary.

6.3 More General Grid Structures

Although the typical applications in scientific supercomputing belong to a wide range of different scientific and technological fields, many of them are characterized by remarkably similar mathematical models and, as a consequence of that, by very similar data structures. Grids and grid-like data structures are encountered most frequently. (The material here is abstracted from: [49].)

Apart from the very simple grid structures considered so far, there are several more involved structures encountered in various applications. We here distinguish three types of grids:

- **Regular grids**
  Regular grids normally arise from discretization of PDEs on simple domains, i.e. domains which can be mapped to rectangles and cuboids by special transformations.

  Regular grids are characterized by direct grid point addressing and a rectangular (2D) resp. cuboid (3D) address space area. A geometric neighborhood of grid points in this respect also means a logical neighborhood in the address space of the data structure.

- **Block structured grids**
  In many applications, in particular in CFD, it is either not practiced or not appropriate to transform the domain to only *one* rectangle (cuboid). The domain must rather be partitioned into several parts ("blocks") each of which in turn can be transformed to a rectangle or cuboid (see Figure 4 for such a transformation and Figure 6(c) for a simple grid consisting of 2 blocks). This mapping technique results in *block structured grids*, which consist of several regular grids. The relations within each block are of the same type as in the case of regular grids. It can become difficult to maintain the locality at the block boundaries if the block structure itself is not a regular one. A concept seems to be accepted for CFD simulations where each single block shows a regular internal grid structure, but the block structure itself is admitted to be irregular [36, 37].

- **Irregular grids**
  If there is no fixed grid structure, we speak of irregular grids (no fixed number of neighbor points, no regularity of subgrids). The grid points cannot be addressed geometrically but must be addressed indirectly (using pointers). The typical example for irregular grids in this respect are certain finite element nets (see Figure 6(a)). Local relations, which are natural also on irregular grids, can no longer be identified as local relations in the memory.

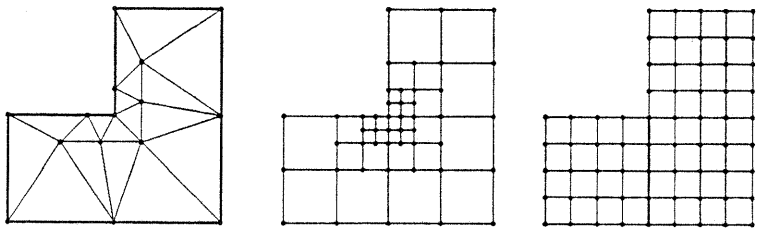


Figure 6: Irregular (a), locally refined (b), and block-structured (c) grid

In this paper, we will only consider grids the structure of which is *statically* defined, i.e. the grid structure is already known a priori and will not be changed in the course of the calculation. For such static grids the grid partitioning approach is very natural.

6.4 Implementation of Parallel Grid Algorithms

The general parallelization approach for grid applications on distributed memory-multiprocessor (dm-mp) architectures is the grid partitioning method as introduced above. With respect to regular grids, we summarize its characteristics here once more:

- A set of equal size subgrids is created and each subgrid is itself regular.
- All subgrids are processed independently and in parallel during each MG step.
- The numerical algorithm is not changed, i.e. its sequential and its parallel version give exactly the same results.
- After each computational step the boundaries of the subgrids have to be updated using communication.

We now formulate some basic guidelines which should be considered if the grid partitioning method (as well as any other parallelization method) is implemented on a dm-mp system. (The material here is in part abstracted from [47].)

1. The implementation should be independent of the topology of the architecture. The structure of a parallel grid program should look identical on hypercubes, trees, or hierarchical architectures like SUPRENUM.

2. The implementation should also be independent of the number of processors. It should be possible to run the same program on different numbers of nodes without recompilation – at least as long as the local memory capacity is sufficient.

3. For reasons of debugging, maintenance, and program aesthetics, a clear program structure is highly recommended, separating the calculation and the communication parts of the code.

4. Finally, the programmer should strive for portability within the class of dm-mp systems. This requires, however, common (language or run-time) constructs describing the parallelism and the communication.

Guideline 1 is satisfied if each subgrid is associated with a *process* (as described in Section 2.3) instead of directly with a node. The system of processes has a user defined topology. For the grid partitioning method a grid or torus is obviously an appropriate process topology since it preserves locality, i.e. neighboring subgrids are associated with neighboring processes. During global collection and distribution steps it may be useful to configurate the processes additionally as a binary or as a spanning tree. The mapping of the process structure to the actual hardware configuration should be done automatically by the system software. Process grids are mapped to hypercubes using gray-codes. On SUPRENUM a special *mapping library* performs optimized process-node mappings for a number of standard process topologies including grids and trees.

Typically, a parallel program for a regular grid application (single grid or multigrid) on a dm-multiprocessor has the following structure:

- The host process creates the set of processes and sends them the necessary control data (identification of their "neighbors", index range of the subdomain, certain global parameters of the algorithm).
- The host process sends each process the initial data belonging to its part of the domain.
- The node processes receive the initial information.
- The computation part in the node processes is separated from the communication part and is very similar to a sequential grid program.
- After each computational step the points near the interior boundaries (which are stored in *overlap areas*) are updated by mutual exchange of data.
- During the computation certain globally dependent results (such as norms of residuals) are assembled treewise.
- After the computation, the results are sent to the host process, where the solution for the entire domain is assembled, or are written directly to an external file system.

Nearly all existing dm-mp systems (Intel iPSC, SUPRENUM etc.) provide a more or less convenient process model. The grid partitioning method for regular grids requires only the static features of the process model, i.e. all processes are created and started at the beginning of the application. Programming of *adaptive* (i.e. solution dependent) grid structures may require, however, dynamic processes which can be created during run-time. Dynamic process concepts are provided by some of the existing dm-mp machines although the fundamental mapping problems are not solved yet. Some work on the mapping problems has been done in the SUPRENUM project [31].

Guideline 2 requires that several processes can run on one node at the same time (multi-processing). Although – due to the process switches – the performance will decrease, this option is necessary in order to provide independence from the actual underlying hardware configuration. In the future, dm-mp systems will not be single-user systems. They will be large supercomputers with the flexibility to allow the

user to have the whole machine (for production runs) or only a small part of it (for development and debug purposes).

Multi-processing is not crucial for regular grids since the user can very easily adapt the number of processes to the number of nodes actually available. In fact, this is much more efficient than having a fixed large number of subgrids and processes and relying on multi-processing. Advanced fault tolerance concepts, however, need the multi-processing feature since they allow the continuation of an application even if a total failure occurs on a node.

Message-passing is the basic communication system on dm-mp architectures. The most convenient – but unfortunately also most expensive – communication protocol is *asynchronous* message-passing. This means that the sending process sends its message into the mailbox of the receiver and immediately continues execution without waiting for completion of the message transfer. The receiving process looks for a message in its mailbox and eventually waits until the desired message arrives. The asynchronous message passing allows an arbitrary sequence of SEND and RECEIVE statements and leads to a clear program structure (guideline 3).

A simpler protocol is *synchronous* message passing where the sender has to wait until the destination process has received the message. This requires a special structure of SEND and RECEIVE statements in order to avoid deadlock situations. The data exchange programmed in the node program in Section 6.5, for instance, will result in a deadlock for synchronous message passing because all processes are sending at the same time. (Of course, a simple rearrangement of the SENDs and RECEIVEs will remove the deadlock). For grid partitioning on regular grids synchronous message-passing is completely sufficient, whereas block-structured grids can be programmed more comfortably using asynchronous message passing.

In both concepts, the RECEIVE statement is blocking, i.e. the node program execution is stopped until the specified message has arrived in the mailbox. This implicitely solves the synchronization problem since communicating processes synchronize themselves. In asynchronous systems it is possible to check the availability of pending messages, prior to issueing a RECEIVE.

Portability (guideline 4) is a very important issue. Parallelism for the most relevant programming languages such as Fortran and C is not yet standardized. Some of the dm-mp systems provide calls to the run-time library for process creation and communication. Data have to be transfered via the procedure interface which requires additional copy steps. SUPRENUM has included most of those constructs into Fortran. Here, the SEND and RECEIVE constructs are similar to WRITE and READ and they support the Fortran-8X array notation. Transputer systems mainly use the OCCAM language which already contains all necessary process handling and communication constructs.

In view of this variety one approach towards portability is the use of special communications libraries which then have to be implemented on all systems. Such libraries are clearly restricted to certain classes of applications and data structures. One effort in this direction is the SUPRENUM communications library for grid applications (see Section 6.6).

There are several other approaches to portability, a very promising one being the definition, implementation and use of certain system independent communica-

tion macros. Such macros cover the whole area of process creation, communication, synchronization etc. If parallel programs are written in terms of these macros, they then run on every parallel machine on which the macros are available. Moreover, the macros can be implemented on dm-mp systems as well as on shared memory machines. In one such concept [3], macros have been implemented on the following systems:

- Encore Multimax
- Sequent Balance
- Alliant FX/8
- Network of Sun workstations
- Intel iPSC/2 hypercube
- Network of IBM RT workstations
- SUPRENUM

Figure 7 summarizes the different mapping stages which are involved in the mapping of grid applications to dm-mp systems. The first step is the user-supplied mapping from the physical boundary fitted grid to a logical rectangular structure (by grid generation programs), followed by mapping of the rectangular grid structure to a process system (using the grid partitioning method). The final mapping step maps the process system to the nodes.

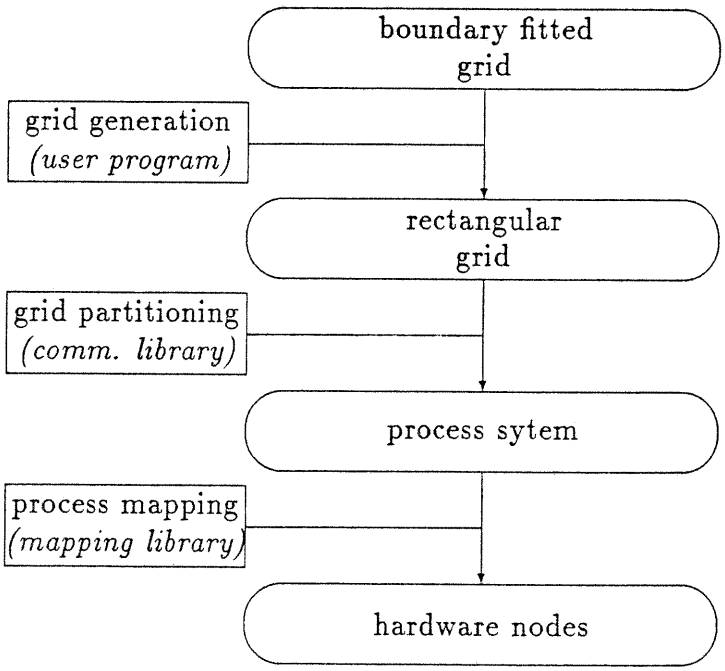


Figure 7: Mapping steps

6.5 Example of a Parallel Grid Program (SUPRENUM)

The concrete program example given here consists of a simple host and node program for a 2D relaxation (e.g. for the iterative solution of Poisson's equation). The process creation and the message passing instructions are taken from SUPRENUM Fortran [2].

The host program runs on a front-end system which can perform user I/O whereas the node program is loaded into the nodes. For regular grid algorithms usually only one node program has to be written which operates on the different subgrids. Thus the operation mode is rather SIMD-like than MIMD-like for these applications. More complex applications and/or data structures require different node programs on the different nodes.

The example programs listed below should be regarded as functional kernels but not as a complete, safe, and efficient implementation of a grid algorithm. For instance, the initial information sent to the node processes usually contains a lot more control data (e.g. for the numerical algorithm). The collection of local solutions and local residual norms as programmed here is neither safe nor efficient. For reasons of safety, for instance, the node processes first should ask the host process whether it is ready to accept the solution data. The local residual norms can be collected much more efficiently via a process tree.

Host program:

```
C declarations
      REAL, ARRAY (0:10000, 0:10000) ::  U,F
      TASKID, ARRAY, ALLOCATABLE (:,:)  ::  PID
      TASKID SOUTH, NORTH, WEST, EAST
      INTEGER, ARRAY, ALLOCATABLE (:,:,:)  ::  IX, IY
      ...
C initialize tags
      INTEGER TIN, TST, TSO, TRE
      DATA TIN/10/, TST/11/, TSO/12/, TRE/13/
C user input:
C maximal number of relaxations MAXIT
C process configuration NPX x NPY, grid size NX x NY
C initial solution U, right hand side F
      READ(...)  MAXIT, NPX, NPY, NX, NY, U, F
      NP=NPX * NPY
C allocate dynamic arrays
      ALLOCATE PID(NPX, NPY), IX(NPX, NPY, 2), IY(NPX, NPY, 2)
C compute size of subdomains
      IPSX=NX/NPX
      IPSY=NY/NPY
      DO 10 IPX = 1, NPX
        DO 10 IPY = 1, NPY
C create node processes
          PID(IPX,IPY) = NEWTASK('PROG_NODE', (IPY-1)*NPX+IPX)
C store index boundaries of subgrids
          IX(IPX,IPY,1) = (IPX-1) * IPSX + 1
          IX(IPX,IPY,2) = IPX * IPSX
          IY(IPX,IPY,1) = (IPY-1) * IPSY + 1
          IY(IPX,IPY,2) = IPY * IPSY
10     CONTINUE
```

```
C send initial information
      DO 20 IPX = 1, NPX
        DO 20 IPY = 1, NPY
          SOUTH = .NOTASKID.
          IF(IPY.NE.1) SOUTH=PID(IPX,IPY-1)
          NORTH = .NOTASKID.
          IF(IPY.NE.NPY) NORTH=PID(IPX,IPY+1)
          WEST = .NOTASKID.
          IF(IPX.NE.1) WEST=PID(IPX-1,IPY)
          EAST = .NOTASKID.
          IF(IPY.NE.NPX) EAST=PID(IPX+1,IPY)
          SEND (TASKID=PID(IPX,IPY), TAG=TIN)
     &      MAXIT, IPX, IPY, IX(IPX, IPY, 1:2), IY(IPX, IPY, 1:2),
     &      SOUTH, NORTH, WEST, EAST
20    CONTINUE

C send initial solution and right hand side
      DO 30 IPX = 1, NPX
        DO 30 IPY = 1, NPY
          SEND (TASKID=PID(IPX,IPY), TAG=TST)
     &      U(IX(IPX,IPY,1):IX(IPX,IPY,2), IY(IPX,IPY,1):IY(IPX,IPY,2)),
     &      F(IX(IPX,IPY,1):IX(IPX,IPY,2), IY(IPX,IPY,1):IY(IPX,IPY,2))
30    CONTINUE

C receive solution in arbitrary order
      DO 40 IP=1,NP
        RECEIVE (TAG=TSO) IPX, IPY,
     &    U(IX(IPX,IPY,1):IX(IPX,IPY,2), IY(IPX,IPY,1):IY(IPX,IPY,2))
40    CONTINUE

C receive residual norms in arbitrary order
      RES = 0.D0
      DO 50 IP=1,NP
        RECEIVE (TAG=TRE) RESLOC
        RES = MAX (RES, RESLOC)
50    CONTINUE

C postprocessing
...

C end of host program
      DEALLOCATE PID, IX, IY
      STOP
      END
```

Node program:

```
C declarations
      REAL,ARRAY,ALLOCATABLE(:,:)  ::  U,F
      TASKID SOUTH, NORTH, WEST, EAST
      INTEGER, ARRAY (2) ::  IX, IY

C tags
      INTEGER TIN, TST, TSO, TRE
      DATA TIN/10/, TST/11/, TSO/12/, TRE/13/

C receive initial information
      RECEIVE(TAG=TIN) MAXIT, IPX, IPY, IX, IY, SOUTH, NORTH, WEST, EAST

C allocate dynamic arrays
      ALLOCATE U(IX(1)-1:IX(2)+1, IY(1)-1:IY(2)+1), F(IX(1):IX(2), IY(1):IY(2))

C receive initial solution and right hand side
      RECEIVE(TAG=TST) U(IX(1):IX(2), IY(1):IY(2)), F(IX(1):IX(2), IY(1):IY(2))

C iterative loop
      DO 10 IT = 1, MAXIT

C subroutine RELAX contains the usual sequential program
```

```
      CALL RELAX(U, F, ...)
C data exchange by message passing:
C First send the current approximation at points which
C belong to the overlap of the western (eastern, southern,
C northern) neighbor process to this process.
C Then receive the current approximation at points which
C belong to the own overlap. These values have been sent
C by the western, eastern, southern, and northern
C neighbors in their SEND statements.
      TEX = 100 + IT
      SEND(TASKID=WEST,TAG=TEX) U(IX(1),IY(1):IY(2))
      SEND(TASKID=EAST,TAG=TEX) U(IX(2),IY(1):IY(2))
      SEND(TASKID=SOUTH,TAG=TEX) U(IX(1):IX(2),IY(1))
      SEND(TASKID=NORTH,TAG=TEX) U(IX(1):IX(2),IY(2))
      RECEIVE(TASKID=WEST,TAG=TEX) U(IX(1)-1,IY(1):IY(2))
      RECEIVE(TASKID=EAST,TAG=TEX) U(IX(2)+1,IY(1):IY(2))
      RECEIVE(TASKID=SOUTH,TAG=TEX) U(IX(1):IX(2),IY(1)-1)
      RECEIVE(TASKID=NORTH,TAG=TEX) U(IX(1):IX(2),IY(2)+1)
C end of loop
10    CONTINUE
C send local solution to host
    SEND (TASKID=MASTER(), TAG=TSO) IPX, IPY, U(IX(1):IX(2), IY(1):IY(2))
C send local residual norms to host
    CALL RESID(U, F, RES, ...)
    SEND (TASKID=MASTER(), TAG=TRE) RES
C end of node program
...
    DEALLOCATE U, F
    STOP
    END
```

6.6 Communications Libraries

In the future, parallelizers like SUPERB [55] will generate host and node programs such as those given in Section 6.5 (semi)-automatically. Until then the development of parallel programs can be simplified by certain library functions which provide typical high-level communication patterns in a convenient way for the user. The SUPRENUM communications library [25, 27, 28], for instance, contains subroutines for the creation of a process system with a grid topology, for the sending of initial information, for the boundary exchange, and for the final collection step. It is advantageous to use such a library because it ensures

- clean and error-free programming,
- easy development of parallel codes,
- portability within the class of dm-mp systems (see guideline 4 in Section 6.4). Programs can be ported to any dm-mp machine as soon as the communication library has been implemented. As an example, programs are routinely ported between the Intel iPSC and SUPRENUM.

A corresponding library for block-structured applications (6.3) is also available. Most of the application software which has been written in the SUPRENUM project is based on these routines.

7 Multiprocessor Efficiency of Multigrid

7.1 Basic Notations and Measures

The usual quantities of interest in evaluating the performance of parallel algorithms are:

- Time $T(N,P)$: time to solve a problem of size N on a multiprocessor system using P nodes,
- mp-speed-up $S(N,P) := T(N,1)\,/\,T(N,P)$,
- mp-efficiency $E(N,P) := S(N,P)\,/\,P$.

Note that on MIMD/SIMD architectures, such as dm-mp systems with vector nodes, the utilization of the hardware resources is the product of the mp-efficiency defined above and the efficiency related to the vector processing unit.

The sequential reference time $T(N,1)$ refers to the parallel program running on one node. $T(N,1)$ may be a hypothetic time because for large values of N the memory of one node might be too small.

Another very common definition of the speed-up is

$$S^*(N,P) = \frac{T(N,1,A_{opt})}{T(N,P,A_{par})}$$

which assumes that the best sequential algorithm available A_{opt} is compared to the (possibly different) parallel algorithm A_{par}. S^* depends on the efficiency of the parallel implementation (mp-efficiency) as well as on the *numerical* efficiency of the algorithms. Both types of efficiency are important but they should be clearly separated. In this paper we are solely interested in the mp-efficiency as defined above. We want to compare, for example, parallel single grid methods to sequential single grid methods and not to the much faster sequential MG methods.

The basic reasons for mp-efficiency being significantly smaller than 1 are

- not enough parallelism in the algorithm (i.e. the number of processes which can execute in parallel is smaller than the number of nodes available),
- unbalanced load,
- communication (including synchronization[2]).

The mp-efficiency can be expressed as

$$E(N,P) = \frac{1}{1+r}.$$

The quantity r is a measure for the "parallel overhead". It is used here because it allows a clear separation of the different overhead sources (see Section 7.2). The aim of an implementation of parallel algorithms is, of course, the minimization of the total overhead (not necessarily of each of its components). In order to quantify this goal, we formulate the following conditions:

[2]Synchronization will be neglected in this chapter.

(A) $E \to 1$ for $N \to \infty$ and P fixed. For sufficiently large problems on a given machine the efficiency should be close to 1. This condition gives no information about the efficiency for realistic values of N or about the asymptotic behaviour as $N/P \to \infty$.

(B) $E \longrightarrow 1$ for $N/P \longrightarrow \infty$. This is a stronger condition than (A) requiring that the efficiency depends only on the process size $s = N/P$. Condition (B) ensures the scalability of the parallel application which means that $T(N,P) = T(2N,2P)$, i.e. a twice as large problem can be solved in the same time using twice as many nodes. This is important if the parallel algorithm runs on massively parallel dm-mp systems with thousands of nodes.

(C) $E \geq 0.5$ for "realistic" process sizes $s = N/P$. For certain applications N is given and condition (C) should determine a reasonable value for P. We can also derive from this condition the minimum size of the node memory. If condition (B) is fulfilled we can define $s_{1/2}$ as the process size for which E=0.5. (This definition is independent of the size of the machine.)

7.2 A System Model for a Homogeneous Architecture

In the rest of the paper we always assume that each node of the dm-mp system is associated to exactly one process, i.e. we do not distinguish between nodes and processes. As mentioned in Section 6.4 this assumption is reasonable for regular grids.

The basic parameters which describe the performance of a homogeneous dm-mp system are

- P number of processes (=nodes). The number of processes in each direction of the process grid is assumed to be a power of 2
- α the start-up time for sending and receiving a message
- β the transfer time per word of a message
- δ the time for one floating point operation.

The start-up and transfer times are often given in nondimensionalized form $\alpha' = \alpha/\delta$, $\beta' = \beta/\delta$. α' is the number of flops that can be performed during one message start-up, β' is number of flops during the transfer of one word. For our simulation, δ is assumed to be constant. On vector nodes it depends, of course, heavily on the vector length.

The total time for a parallel program which is based on message-passing consists of a calculation component T_{comp} and a communication component T_{comm}.

Communicating a message of length L items costs $\alpha' + \beta' L$ time units (=flops). So T_{comm} is a combination of the *start-up* time T_{st} (depending on α') and the *transfer* time T_{tr} (depending on β'). T_{tr} is an expression for the lumped transfer times turning up in different system components.

For existing dm-mp systems with vector nodes (SUPRENUM, iPSC-VX, iPSC/860) the communication parameters α' and β' lie in the range of 1000–10000 and 10–100, respectively. For dm-mp systems with less powerful scalar arithmetic co-processors these values are much smaller.

If calculation and communication cannot be overlapped at all, the total time is

$$T_{tot} = T_{comp} + T_{st} + T_{tr}. \tag{9}$$

On many dm-mp systems special communication hardware is available and the transfer of the data to the communication channel or bus may be done in parallel to the computational work. If perfect overlap of communication and calculation can be achieved the total time reduces to

$$T_{tot} = max(T_{comp}, T_{st}, T_{tr}). \tag{10}$$

We expect the real time to lie somewhere between the worst case (9) and best case (10) estimates – depending on the actual hardware and the particular application. In the subsequent simulations we always use the worst case estimate (9).

The total parallel overhead r is the sum of its components

$$r_{comp} = \frac{PT_{comp}(N,P)}{T_{comp}(N,1)} - 1,$$

$$r_{st} = \frac{PT_{st}(N,P)}{T_{comp}(N,1)},$$

$$r_{tr} = \frac{PT_{tr}(N,P)}{T_{comp}(N,1)}.$$

$$r = r_{comp} + r_{st} + r_{tr}. \tag{11}$$

The simple linear one-stage communication model is strictly valid only for homogeneous parallel systems on which logical process neighbors can be mapped to physical neighbors (as e.g. on hypercubes) or on which the distance of communication is unimportant. On systems with hierarchical architectures (like SUPRENUM) only the communication on one level is represented in the model. So the simulations for SUPRENUM refer to one cluster and not to the whole two-level architecture. If the "communication bottleneck" occurs in the node or on the clusterbus level, the simulation might be quite realistic. If, however, the communication times are determined by the upper bus network, either a more sophisticated simulation should be used (as in [30]) or a (modified) one-level simulation has to be applied to the upper level of the architecture.

7.3 Some Results

7.3.1 Analysis of 2D Multigrid Efficiency

Consider a two-dimensional multigrid algorithm which requires performing relaxations, projections and interpolations. We will distribute the problem over a set of processors by subdividing the grids into rectangular subgrids, with one assigned to each processor. To be more specific, we will assume that we are on an $N = n \times n$ grid, with $n = 16m$, where m is a power of 2: $m = 2^l$, and that the data are distributed in square blocks of size $m \times m$ to each of 256 processors. Assume further that each

relaxation, interpolation or projection operator involves R, I or P arithmetic operations per grid point and 1 communication operation per boundary point. Depending on the exact multigrid strategy used the amount of communication involved in projection is often less than in relaxation, but we ignore this point. Finally we assume that the time $T(w)$ (in seconds) required to send w words of data to a "neighboring" processor is represented by a linear relationship: $T(w) = \alpha + \beta w$, where α represents the start-up cost for communicating an arbitrarily short message, while β represents the incremental cost per word for sending longer messages. We will denote by δ the time (in seconds) required to execute a typical elementary arithmetic operation, such as an add or a multiply.

In order to minimize the overhead from communication startup, we will buffer all of the boundary data from a side of a square, and then communicate it in one operation. Thus only 4 communication operations are required on each grid level for a relaxation, projection or interpolation. The complete computational cost of a multigrid V-cycle, with ν iterations performed per grid level is then:

$$\begin{aligned} T_{comp} &= (\nu R + P + I)\delta(1 + 1/4 + 1/16...)m^2 \\ &\approx 4/3(\nu R + P + I)\delta m^2, \end{aligned}$$

while the corresponding time spent in communication is:

$$\begin{aligned} T_{comm} &= (\nu + 2)(4T(m) + 4T(m/2) + 4T(m/4) + ...) \\ &= 4(\nu + 2)(l\alpha + (1 + 1/2 + 1/4 + ...)m\beta) \\ &\approx 4(\nu + 2)(l\alpha + 2m\beta). \end{aligned}$$

The factor 2 in the final coefficient of β above should actually be $2 - 2^{1-l}$, which is very close to 2 as long as the number of multigrid levels l is more than say 3. Similarly the coefficient 4/3 in T_{comp} should actually be $4/3(1 - 4^{-l})$, which is again very close to 4/3 for moderately large l.

We have assumed that the computational time per grid level is proportional to the number of grid points - which will not be true when there are fewer grid points than processors. The above formula for computation is therefore a good approximation only for machines with moderate parallelism, or for multigrid cycles where the coarsest grids are not too coarse. Note that vector nodes effectively increase the degree of inherent parallelism in the machine, requiring increased processing time per grid point even when there are several grid points per processor. We have also assumed above that communications in different directions cannot be overlapped and that communication is not limited by the global band-width. If communication in each of the four directions can be overlapped, then T_{comm} becomes 4 times smaller. It is likely that for some machines the communication startup cannot be overlapped, whereas the remainder of the communication can be. In that case the coefficient of β above would be 4 times smaller. However we do *not* make this assumption in the following discussions.

With the above assumptions, the resulting computational efficiency is then given by:

$$E \equiv T_{comp}/(T_{comp} + T_{comm}) = 1/(1 + r_{comm}),$$

where $r_{comm} = T_{comm}/T_{comp}$, the ratio of communication time to computation time, satisfies:

$$r_{comm} \approx \frac{3(\nu+2)(l\alpha+2m\beta)}{(\nu R+P+I)\delta m^2}$$

For large problems, defined as those where $m \gg l\alpha/\beta$, this implies:

$$E \approx \frac{1}{1+\frac{6(\nu+2)}{(\nu R+P+I)}\frac{\beta}{\delta m}} \quad .$$

Thus the efficiency for large problems can be arbitrarily close to 1. We note that our definition of large problem depends on the number of multigrid levels l, as well as on the message startup cost α. The reason is simply that even though coarse grids involve only a few points, they still incur the same message startup cost as on a fine grid. Thus as the number of levels increases, communication inefficiency also increases unless the startup cost is negligible.

7.3.2 A Concrete 2D Example

As an interesting test, we consider the above case for the current SUPRENUM machine which has 8 Mbytes of memory per node, a startup cost α for communication of 2000 μ sec, and a per-word transfer cost β of about 1 μ sec. We will assume a computation rate of 8 Mflops, so that $\delta = 1/8\mu$ secs, and 8-byte floating point words. For relaxation of the simplest variable coefficient 5-point PDE discretization we would have approximately 9 floating point operations per point, and we assume that interpolation and projection are similar, so that $R = P = I = 9$. The largest problem that will fit comfortably on 256 nodes would have $N = 64 \cdot 10^6$ grid points (two words required per point), so that $m = 512$. It follows that the number of levels l would be 9. The ratio $l\alpha/\beta$ is then about 18000 so that the problem is not "large" as defined above. Inserting the above numbers into the expression for r_{comm} we obtain:

$$E \approx \frac{1}{1+\frac{3(\nu+2)}{9\nu+18}\frac{8(9\cdot 2000+2\cdot 512\cdot 1)}{512^2}} = .84 \quad ,$$

which indicates an efficient solution. Since the term $l\alpha$ is much larger than $2m\beta$ we see that even for this large problem, communication is still dominated by the startup costs. Thus if overlapping of the data transmission were allowed on different channels (without overlapping of the startup cost) there would be only a small improvement in efficiency. Similarly a substantially slower data transfer rate than 1 word per μ sec, or equivalently some saturation of communication bandwidth, could be tolerated with little decrease in efficiency. Clearly decreasing the communication startup cost α and/or using fewer multigrid levels l will be the best ways to improve efficiency for this problem. The latter approach may result in an increased number of iterations however. One possibility is to switch to a different solution strategy at a certain level - for example to transfer data to a single processor and use a direct solver there [28]. Note that these estimates have also ignored the difficulty of using all processors, or of attaining full efficiency from vector nodes,when processing on coarse grids.

7.3.3 Analysis of 3D Multigrid Efficiency

Practical problems of interest are more likely to be three-dimensional than two-dimensional, which qualitatively changes the above estimates. In the three dimensional case we obtain for a distribution of a cubic grid of $N = n \times n \times n$ points into cubic blocks each of size $m \times m \times m$, with $m = 2^l$,

$$\begin{aligned} T_{comp} &= (\nu R + P + I)(1 + 1/8 + 1/64 + ...)\delta m^3 \\ &\approx 8/7(\nu R + P + I)\delta m^3, \end{aligned}$$

while the corresponding time spent in communication is:

$$\begin{aligned} T_{comm} &= (\nu + 2)(6T(m^2) + 6T(m^2/4) + 6T(m^2/16) + ...) \\ &= 6(\nu + 2)(l\alpha + (1 + 1/4 + 1/16 + ...)m^2\beta) \\ &\approx 6(\nu + 2)(l\alpha + 4/3m^2\beta). \end{aligned}$$

We have again assumed that the computational time per grid level is proportional to the number of grid points - a reasonable approximation only for moderately parallel machines or for grids that do not become too coarse. We have also assumed again that communications in different directions cannot be overlapped and that communication is not limited by the band-width. If communication in each of the six directions can be overlapped, then T_{comm} becomes 6 times smaller, while if communication transmission alone can be overlapped, then the coefficient of β becomes 6 times smaller. While the latter is a possibility for SUPRENUM, we do not assume that in the following analysis.

The resulting computational efficiency is then given by:

$$E \equiv T_{comp}/(T_{comp} + T_{comm}) = 1/(1 + r_{comm}),$$

where $r_{comm} = T_{comm}/T_{comp}$, the ratio of communication to computation, satisfies:

$$r_{comm} \equiv \frac{21(\nu + 2)(l\alpha + 4/3m^2\beta)}{4(\nu R + P + I)\delta m^3}.$$

For large problems, defined now as those where $m \gg \sqrt{l\alpha/\beta}$, this implies:

$$E \approx \frac{1}{1 + \frac{7(\nu+2)}{\nu R+P+I}\frac{\beta}{\delta m}} \quad .$$

7.3.4 A Concrete 3D Example

As an interesting test, we consider the three-dimensional case for the current SUPRENUM machine (parameters as in Section 7.3.2). For relaxation of the simplest variable coefficient 7-point PDE discretization we would have approximately 13 floating point operations per point, and we assume that interpolation and projection are similar, so that $R = P = I = 13$. The largest problem that will fit on 256 nodes would have $N = 128 \cdot 106$ grid points (two words required per point), so that $m = 80$ at most. It follows that the number of levels would be around 6. The ratio $l\alpha/\beta$ is then about

12000 so that the problem is not "large" as defined above. The efficiency is found from the expression for r_{comm} to be:

$$E \equiv \frac{1}{1 + \frac{21(\nu+2)}{4(13\nu+26)} \frac{8(6 \cdot 2000 + 4/3 \cdot 80^2 \cdot 1)}{80^3}} = .89 \quad .$$

Note that these estimates have ignored the difficulty of using all processors, or of attaining full efficiency from vector nodes, when processing on coarse grids.

7.3.5 Comparison of 2D and 3D Efficiency

Note that while the behavior of the efficiency E as a function of m for the "large" three-dimensional case above is similar to that for the "large" two-dimensional case, the asymptotic efficiency in three dimensions is actually much worse for the same number of grid-points since m is related to the number of grid points N by $m = 1/16N^{1/2}$ in two-dimensions, but by $m = 1/6.35N^{1/3}$ in three dimensions. Since the maximum number of points N is hardware limited by the available memory, it appears to be much harder to achieve high efficiency for the three-dimensional case.

However this conclusion is *not* applicable to the current SUPRENUM machine, primarily because the largest problem that can be solved is not "large" as defined above for either two or three dimensional problems. This fact alters the efficiency of the two-dimensional problems, with less effect on the three-dimensional case, resulting in more or less comparable efficiencies for the two cases for the largest problems that will fit on SUPRENUM. This is in turn traced to the fact that communication startup costs dominate the communication costs in two-dimensions much more than in three dimensions. The reason is that in three dimensions so much data is transferred en masse per processor that the startup cost is now approximately half the total communication cost, whereas it constitutes about 95 % of the total communication cost in two-dimensions.

It follows that for three dimensional problems there is less advantage to reducing the number of grid levels or the communication startup cost, while there is a greater advantage to overlapping the data *transmission* part of communication in different directions, even if communication startup is not overlapped. In fact, if communication transmission is overlapped (reducing the effective size of β correspondingly), then the three-dimensional efficiency rises to .92 as against .84 for the two-dimensional case.

Note that we have discussed above the case of the simplest discretizations of variable coefficient problems. Efficiencies for the constant coefficient Poisson equation discretized on a rectangular grid would be somewhat worse, because there is then relatively less computation per communication. However the vast majority of real applications involve local numerical computations that are substantially more complex than those involved above. Such computations can be expected to perform at higher efficiencies than those we have estimated. As an example, the solutions of hyperbolic equations encountered in many fluid flow problems require very large amounts of numerical computation to be performed before a communication is required.

8 Some Measured Results on Parallel Computers

8.1 Multigrid on Vector Computers

A standard multigrid code for the solution of the 2D Helmholtz equation with Dirichlet boundary conditions on rectangular domains has been optimized for different vector computers: Cray X-MP, Fujitsu VP 200, and CDC Cyber 205. (This section is abstracted from [32].) The multigrid components: 2D red-black pointwise relaxation, bilinear interpolation, half weighting restriction and a higher order full multigrid interpolation, can be vectorized in a straightforward manner over one coordinate direction.

We summarize the results in Table 7. The best results were achieved for the Cray X-MP. The major reason is the shorter start-up-time compared with the other machines, since in multigrid methods large parts of the problem are solved on coarser grids where the vector length is automatically short. For very fine discretizations, the performance of the Fujitsu VP 200 was better. The reason is a much higher discrepancy between short and long vector performance on this machine and the very high asymptotic performance compared to the Cray X-MP.

While memory access with stride 2 did not cause much degradation of the performance on the Cray X-MP, a data access stride of 64 bits was essential for good performance on the Fujitsu VP 200. This can be achieved by using either stride 2, which is natural from the numerical algorithm point of view, and 32 bit arithmetic, or by using red-black instead of lexicographic ordering. In the latter case a consequence will be shorter vectors in the grid transfers or expensive reordering operations. At the time the tests were done, only consecutive data access was possible on the CDC Cyber 205.

In order to achieve longer vectors, field-wise (plain) vectorization was introduced in the most time consuming part of the algorithm, i.e. the relaxation step. In field-wise vectorization the 2D arrays are treated as 1D vectors, thus giving quadratic vector length compared to the case where vectorization is performed with 1D subarrays of the 2D arrays (vectorization over one coordinate direction). On the coarse grids much higher performance was achieved because of quadratic vector length. On fine grids one dimensional vectorization was cheaper, because extra or masked operations had to be introduced on the Dirichlet boundary in field-wise vectorization. Generally, the above strategy is not applicable efficiently to the other multigrid components, because constant stride over the whole 2D grid, which is essential for vectorization, can only be achieved by executing twice as many operations as necessary. Only on the CDC Cyber 205 was this strategy the most efficient one, because of the high vector start-up time.

Results with a more complex multigrid method, for a second order elliptic partial differential equation with variable anisotropic coefficients, show that the above strategy cannot be applied if alternating zebra line relaxation is used instead of point relaxation. The average performance of this code was similar to that of the first model problem.

Summing up, the vectorization speed-up which was achieved with the multigrid code was very satisfactory compared to other methods such as the conjugate gradient method, which is inherently more efficiently vectorizable. For very large problems,

		Multigrid V - cycle		
	Grid points	33^2	129^2	513^2
Cray X-MP 1	scalar	7	9	9
Cray X-MP 1	standard vect.	21	58	83
Cray X-MP 1	field-wise vect.	34	78	95
Cray X-MP 2	microtasking vect.	27	92	146
Fujitsu VP 200	stand. vect. (32 bit)	16	87	193
Fujitsu VP 200	field-w. vect.(32 bit)	26	124	194
Fujitsu VP 200	field-w. vect.(64 bit)	22	70	90
CDC Cyb. 205-2	field-wise vect.	44	102	-

Table 7: Performance of standard multigrid on vector computers in Mflops

the major applications area of multigrid methods, vectorization speed-up factors of almost the same size were achieved. The multigrid method is then to preferable due to its superior numerical efficiency.

The multigrid code was also tested on the Cray X-MP 2, a shared memory vector computer with two processors. Because of the low granularity of the tasks between synchronization the results with macro- and microtasking have been more or less disappointing. When combining parallelization over the outer loop and vectorization over the innermost loop the synchronization overhead was so large that reasonable efficiencies have only been achievable for very large problems. It should be mentioned here that the above model problem is a worst case study for this class of computers, because of the small number of operations that has to be performed at each grid point. For more complex problems better efficiencies should be expected.

In contrast to the Cray X-MP 2 the results on the Alliant FX 8 with four processors are very satisfactory. The major reason is the much lower synchronization overhead compared to the arithmetic computation speed.

A very detailed description of all tests, discussions of the results and conclusions can be found in [32].

8.2 Multigrid on the Caltech Hypercube

A 3D-Poisson MG-solver was implemented on the CalTech Mark II hypercube [48]. Multiprocessor efficiency rates are given in Table 8. Obviously, the problem with $N = 8^3$ grid points is too small for a system with $P = 32$ nodes. However, even medium-size problems with $N = 32^3$ grid points achieve an MP-efficiency of more

than 0.5.

N	time	$S(N,32)$	$E(N,32)$
8^3	0.306	6.8	0.21
16^3	0.847	12.4	0.39
32^3	3.370	18.6	0.58

Table 8: Computing time, MP-speed-up and MP-efficiency for a multigrid method for the 3D-Poisson equation, with periodic boundary conditions using V-cycles, with $\nu_1 = 2, \nu_2 = 1$ relaxations, on the CalTech Mark II hypercube.

Table 9 gives a comparison of the MG-solver with a standard relaxation solver and with a FFT solver. Although the MP-efficiency of the relaxation and the FFT solver is considerably better than for the MG-solver, the absolute computing time is significantly worse. So both, MP- *and* numerical efficiency are important in designing good algorithms for multiprocessor systems.

method	time	$S(32^3,32)$	$E(32^3,32)$
relaxation	381.3	25.9	0.81
MG	3.4	18.6	0.58
FFT	22.0	29.8	0.93

Table 9: Computing time, MP-speed-up and MP-efficiency for different solvers applied to the 3D-Poisson equation with periodic boundary conditions and $N = 32^3$ grid points, on the CalTech Mark II hypercube.

8.3 Multigrid on the Intel iPSC

Example 1: Poisson's equation represents one of the hardest problems among partial differential equations for parallel computers. The reason for this is that the ratio of arithmetic and communication work is very bad in this case.

The parallel multigrid code MGDEMO (for details see [28]), which solves the 2D-Poisson equation, was implemented on the Intel iPSC/2. Table 10 shows some results. The following parameters were used in the algorithm: V-cycles, half injection as restriction, red-black relaxation, linear interpolation of corrections, two relaxation steps before, one after the coarse grid correction. On very coarse grids (grids with only two points in each direction per process) the number of active processes was reduced to one by agglomeration of the whole distributed application. MGDEMO is portable between different types of parallel machines because all communication between processes is performed by routines of the SUPRENUM communications library.

processors	points	mode	time/cycle (sec)	Mflops	efficiency
$16 = 4 \times 4$	17×17	scalar	0.05	0.15	0.11
$16 = 4 \times 4$	33×33	scalar	0.08	0.35	0.14
$16 = 4 \times 4$	65×65	scalar	0.13	0.86	0.26
$16 = 4 \times 4$	129×129	scalar	0.26	1.72	0.45
$16 = 4 \times 4$	257×257	scalar	0.63	2.77	0.68
$16 = 4 \times 4$	513×513	scalar	1.97	3.54	0.86
$16 = 4 \times 4$	1025×1025	scalar	7.07	3.92	0.95
$16 = 4 \times 4$	17×17	vector	0.07	0.11	-
$16 = 4 \times 4$	33×33	vector	0.11	0.27	-
$16 = 4 \times 4$	65×65	vector	0.17	0.68	-
$16 = 4 \times 4$	129×129	vector	0.28	1.61	-
$16 = 4 \times 4$	257×257	vector	0.48	3.64	-
$16 = 4 \times 4$	513×513	vector	0.94	7.38	-
$16 = 4 \times 4$	769×769	vector	1.52	10.25	-
$1 = 1 \times 1$	257×257	scalar	6.64	0.26	1.00
$4 = 2 \times 2$	257×257	scalar	1.84	0.94	0.93
$16 = 4 \times 4$	257×257	scalar	0.63	2.77	0.68
$1 = 1 \times 1$	129×129	vector	0.68	0.65	-
$4 = 2 \times 2$	129×129	vector	0.38	1.17	-
$16 = 4 \times 4$	129×129	vector	0.28	1.61	-
$1 = 1 \times 1$	193×193	vector	1.17	0.82	-
$4 = 2 \times 2$	193×193	vector	0.57	1.71	-
$16 = 4 \times 4$	193×193	vector	0.35	2.64	-

Table 10: MGDEMO benchmarks on the Intel iPSC/2

The tests show efficiencies near 0.5 in scalar mode (compiler vectorization feature switched off) even for medium problem sizes such as 1000 points per process.

In the case of vector mode there are no efficiency rates listed in the table. For fair results, the whole problem would have had to be run on one single processor (differing vector lengths) and this was not possible because the iPSC/2 vector boards available at the GMD, where these tests were carried out, only have 1 Mbyte of vector memory. Though all arithmetic parts of the code were vectorized, the scalar code shows a slightly better performance for problems up to a size of about 32×32 points per processor. This is due to a quite high vector start-up time. For larger problems the vector mode proves to be faster than the scalar one. For the largest problem fitting on the vector boards the machine performed at more than 10 Mflops.

Example 2: Another example is the generalized biharmonic boundary value problem

$$\triangle^2 u + a(x,y)\triangle u + b(x,y)u = f(x,y)$$

with boundary conditions given for u and the normal derivatives u_n. For multigrid algorithms it is useful to rewrite the differential equation as a system

$$\begin{aligned} \triangle v + a(x,y)v + b(x,y)u &= f(x,y) \\ \triangle u - v &= 0. \end{aligned}$$

Information on the numerical treatment of such biharmonic type systems can be found in [34, 35, 45].

processors	points	mode	time/cycle (sec)	Mflops
$1 = 1 \times 1$	17×17	scalar	0.58	0.13
$1 = 1 \times 1$	33×33	scalar	1.43	0.16
$1 = 1 \times 1$	65×65	scalar	3.76	0.18
$1 = 1 \times 1$	129×129	scalar	11.24	0.20
$4 = 2 \times 2$	17×17	scalar	0.45	0.19
$4 = 2 \times 2$	33×33	scalar	0.87	0.28
$4 = 2 \times 2$	65×65	scalar	1.72	0.43
$4 = 2 \times 2$	129×129	scalar	3.96	0.59
$4 = 2 \times 2$	257×257	scalar	11.11	0.73
$16 = 4 \times 4$	17×17	scalar	0.43	0.21
$16 = 4 \times 4$	33×33	scalar	0.75	0.34
$16 = 4 \times 4$	65×65	scalar	1.24	0.65
$16 = 4 \times 4$	129×129	scalar	2.16	1.19
$16 = 4 \times 4$	257×257	scalar	4.46	1.96
$16 = 4 \times 4$	513×513	scalar	11.71	2.65

Table 11: Timing of a parallel multigrid algorithm for the generalized biharmonic equation

The results in Table 11 were obtained by a multigrid algorithm which consists of F-cycles, Full Weighting, linear interpolation of corrections and red-black relaxation. Additional relaxations were performed along the boundaries with the u_n boundary conditions. One relaxation step was executed before and one after the coarse grid correction. Seven relaxation steps were applied to solve the problem on the coarsest grid, which consisted of 5×5 points in all test cases of Table 11.

For example, the results of the 129×129 problem on 1 and 16 processors show that the efficiency is

$$E = \frac{11.24}{2.16 \cdot 16} \approx 0.33 \quad .$$

It is not surprising that this efficiency is worse than that of MGDEMO for the corresponding problem size in Table 10 because F-cycles, with more communication on coarse grids, were applied for the biharmonic problem. On the other hand, the times for the 129×129 problem on 1 processor and for the, approximately 16 times larger, 513×513 problem on 16 processors are approximately the same, indicating very good efficiency of the parallel application.

Example 3: A parallel multigrid solver for 3D anisotropic elliptic problems (see Section 4.1) was developed using the communications library.

relaxation	processors	points	efficiency
3D-point	$8 = 2 \times 2 \times 2$	$17 \times 17 \times 17$	0.80
	$8 = 2 \times 2 \times 2$	$33 \times 33 \times 33$	0.92
	$16 = 4 \times 2 \times 2$	$17 \times 17 \times 17$	0.54
	$16 = 4 \times 2 \times 2$	$33 \times 33 \times 33$	0.76
	$16 = 4 \times 2 \times 2$	$48 \times 48 \times 48$	0.85
	$32 = 4 \times 4 \times 2$	$17 \times 17 \times 17$	0.37
	$32 = 4 \times 4 \times 2$	$33 \times 33 \times 33$	0.60
	$32 = 4 \times 4 \times 2$	$48 \times 48 \times 48$	0.73
3D-(x,y)-plane/2D-point	$8 = 2 \times 2 \times 2$	$17 \times 17 \times 17$	0.64
	$8 = 2 \times 2 \times 2$	$33 \times 33 \times 33$	0.82
	$16 = 2 \times 2 \times 4$	$17 \times 17 \times 17$	0.37
	$16 = 2 \times 2 \times 4$	$33 \times 33 \times 33$	0.60
	$16 = 2 \times 2 \times 4$	$48 \times 48 \times 48$	0.77
	$32 = 2 \times 4 \times 4$	$33 \times 33 \times 33$	0.37
	$32 = 2 \times 4 \times 4$	$48 \times 48 \times 48$	0.56
3D-alternating-plane/ 2D-alternating-line	$8 = 2 \times 2 \times 2$	$17 \times 17 \times 17$	0.59
	$8 = 2 \times 2 \times 2$	$33 \times 33 \times 33$	0.75

Table 12: Efficiency of parallel 3D multigrid algorithms on the Intel iPSC/2

Depending on the type of the anisotropy different smoothing schemes have to be applied for different problem classes. In the parallel multigrid solver a plane relaxation step is performed by one V-cycle of a 2D multigrid algorithm. This 2D solver uses point or line relaxation for smoothing, depending again on the anisotropy. The optimization of the communication within this 2D solver is of considerable importance for the efficiency of the parallel code. The results of Table 12 are from [18] (which also includes further details and more results).

In all tests on the Intel iPSC/2 (Table 12) the efficiencies are quite high, even for relatively small problems. If more complex smoothing schemes like line or plane relaxation are used, the efficiencies become somewhat smaller because of the growing communication overhead. Line and plane relaxations themselves require communication between the processes. Thus, for a $17 \times 17 \times 17$ problem on $2 \times 2 \times 2$ processors, the efficiency is 0.8 if point relaxation is used. It decreases to 0.64 for plane relaxation if the 2D multigrid solver uses point relaxation, and to 0.59, for alternating plane relaxation with alternating line relaxation.

Example 4: The 2D-Stokes MG solver (see Section 5.2) was implemented on

the Intel iPSC. Figure 8 shows the MP-efficiency rates in relation to the number of processors and the problem size. For a 64-hypercube, a problem size of $N = 256^2$ grid points is necessary in order to achieve an efficiency of more than 0.5.

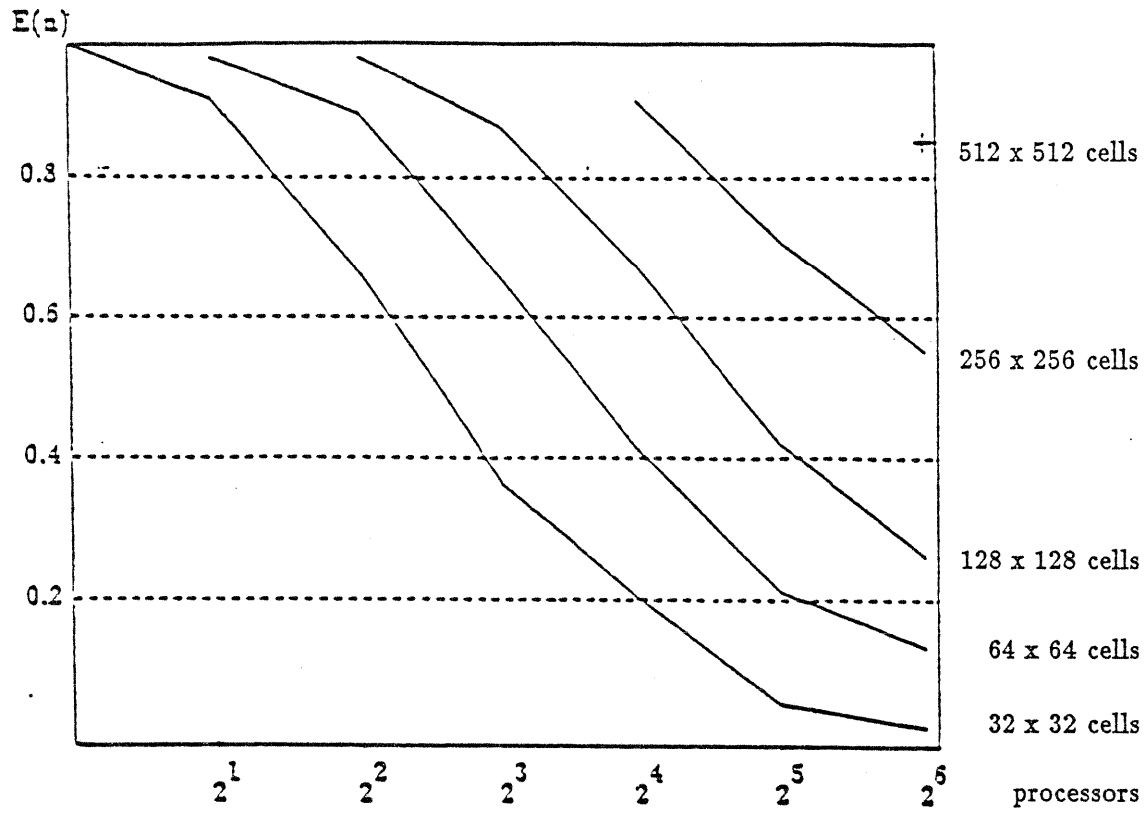


Figure 8: Parallel MG-code for the 2D-Stokes problem. Each curve shows the MP-efficiency for constant problem size N and increasing P on the Intel iPSC.

8.4 Multigrid on SUPRENUM

As a first test, the performance of the parallel multigrid code MGDEMO, which solves the 2D-Poisson equation, was measured on SUPRENUM. Table 10 shows a selection from the results. For more information see [26, 33].

Parameters used in the algorithm were: V-cycles, half injection, red-black relaxation, linear interpolation of corrections, two relaxation steps before, one after the coarse grid correction. On very coarse grids no agglomeration was performed. Instead, an appropriate number of relaxations was carried out on the coarsest grid. The program was fully vectorized.

Though multigrid for the Poisson equation is a very hard test for obtaining good parallel efficiency, the Mflops rates indicate that the performance increases nearly linearly with the number of processors.

processors	points	Mflops
1	513×257	4.0
2	513×513	6.4
4	1025×513	11.7
8	1025×1025	23.7
16	2049×1025	47.5
64	4097×2049	169.0

Table 13: MGDEMO benchmarks on SUPRENUM

8.5 Multigrid on the Connection Machine (CM)

We will now see that for certain hierarchical algorithms there are fundamental obstacles to using *massive* parallelism. The case in point is the implementation of a standard multigrid algorithm on the CM-2. The implementation for the CM-1 is described in detail in [40] and we summarize the main ideas here. The CM-2 implementation follows exactly the same strategy.

As a test problem we solved the five-point discretized Poisson equation for a rectangular grid, using modified Jacobi relaxation (Jacobi relaxation with relaxation parameter ω) on each grid level. Points of the finest grid are assigned to distinct virtual processors. Coarse grid points are allocated to the same processor as their corresponding fine grid point. This simplifies the interactions between grid levels, while somewhat increasing the cost of coarse grid iterations, since coarse grid- points are physically far apart. However, much more serious is the fact that on coarse grids it is impossible to keep all processors active. In the extreme case of a 1 x 1 grid, the efficiency can be at most 1/65536.

We present performance curves measured for multigrid on the CM-2 in Figure 9. The bar chart shows the Mflops generated in solution as a function of the number of grid levels utilized. The case of one grid level is simply solution by relaxation, and gives a very high Mflops rate since all processors are used at all times. As the number of levels increases, Mflops drop dramatically as expected - most processors are sitting idle most of the time. But *the solution time drops steadily with increasing numbers of grid levels* (indicated by the curve in the same figure). Thus, multigrid is still a substantial benefit on the CM, despite the poor overall efficiency.

In the following section we present a more highly parallelizable class of multiscale methods which avoid the difficulties encountered with standard multigrid. For these methods the bar chart in the figure stays essentially horizontal and at the height corresponding to relaxation, and the solution time curve drops far more steeply as we will see.

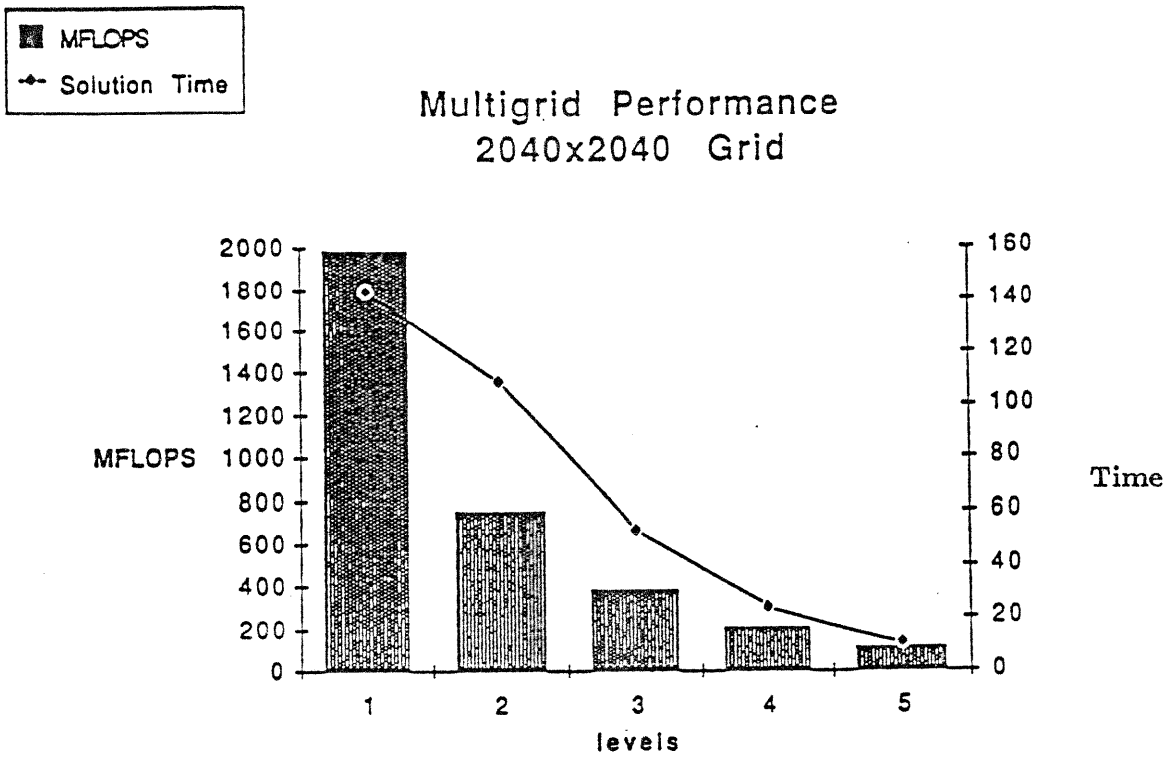


Figure 9: Performance curves measured for multigrid on the CM-2 Connection Machine

9 A Different Parallel Multigrid Approach

9.1 Parallel Superconvergent Multigrid

In the previous section, we have seen the difficulty with multigrid on massively parallel machines. In the extreme case of the coarsest grid, only a single processor is actually doing anything useful. As a result the observed computational time is substantially longer than one might have expected from the equivalent serial algorithm. Algorithmically, parallel multigrid is an $0(\log N)$, rather than an $0(1)$ solution method.

We describe now an algorithm which we will call PSMG (Parallel Superconvergent Multigrid) that takes a step towards solving this problem. The new algorithm still requires $0(\log(N))$ parallel operations for solution, but the constant multiplying the $\log(N)$ is much smaller than before because of more rapid convergence of the solution which therefore requires less iterations to reach a desired level of accuracy. This is accomplished by solving many coarse grid problems simultaneously, combining their results to provide an optimal finer grid approximation. No extra computation time is involved (if N processors are available) since the extra coarse grid problems are solved on processors which would otherwise have been idle.

We state a rigorous convergence criterion for PSMG, which gives a remarkably sharp estimate of the rate of convergence for the case of constant coefficient operators. For example, in some cases an upper bound for the multigrid convergence rate is within a few percent of the supremum of the two-grid convergence rate taken over all grid sizes, even for V-cycles with only one smoothing operation performed per grid level. In some situations PSMG reduces to an exact (direct) solver. Numerical examples involving elliptic operators on rectangular grids are also presented. For simplicity, we will deal with periodic boundary data. For a complete exposition, including proofs and numerical results, we refer to our papers [13, 14, 15, 16, 41].

9.2 The Basic Idea

Consider a simple discretization problem on a 1- dimensional grid. Standard multigrid techniques work with a series of coarser grids, each typically obtained by eliminating every other point of the previous grid. The error equation for the fine grid is then projected to the coarse grid at every second point, the coarse grid equation is solved approximately, and the error is interpolated back to the fine grid and added to the solution there. Finally a smoothing operation is performed on the fine grid. Recursive application of this procedure defines the complete multigrid procedure.

The basic idea behind PSMG is the observation that for each fine grid there are two natural coarse grids - the even and odd points of the fine grid. (For simplicity we assume that periodic boundary conditions are enforced). Either of these coarse grids could be used at any point to construct the coarse grid solution, and both would presumably provide approximately equivalent quality solutions. Multigrid traditionally uses the even points at each grid level.

A typical fine grid.

The standard multigrid coarse grid - the even points.

The alternative multigrid coarse grid - the odd points.

Why not try to combine both of these coarse grid solutions to provide a fine grid correction that is better than either separately? This should be possible since in projecting from the fine grid, the odd and even points receive slightly different data in general, and thus each represents slightly complementary views of the fine grid problem to be solved. Thus it ought to be possible to find a combination of the two solutions that is significantly better than either separately. It would follow immediately that such a scheme would converge faster (fewer iterations) than the corresponding standard multigrid scheme. As a concrete example, if the combination of coarse grid solutions is simply the arithmetic average of the two standard coarse grid interpolation operators, then the algorithm would converge at least as well as the usual multigrid algorithm since the convex combination of two (iteration) operators has norm bounded by the larger of the norms of the two operators.

Note that on a massively parallel machine the two coarse grid solutions may be solved simultaneously, in the same time as one of them would take - we assume here that the number of processors is comparable to the number of fine grid points. As will be seen below, both coarse grid problems are solved using the same set of machine instructions. Consequently the algorithm is well suited to SIMD parallel computers, as well as to MIMD machines. On machines with more modest numbers of processors it may still make sense to switch from standard MG to PSMG at grid levels such that the number of grid points is comparable or less than the number of processors.

The idea outlined above extends naturally to multi-dimensional problems. In d dimensions, 2^d coarse grids are obtained from a fine grid by selecting either the even or the odd points in each of the d coordinate directions. The fine grid solution is then defined by performing a suitable linear interpolation of all 2^d coarse grid points. This procedure is repeated at every grid level.

Suppose we are required to solve a discrete algebraic equation $A^{(L)}\bar{u} = f$ on a rectangular grid $G^{(L)}$ with grid spacing or scale $h_L = 2^{-L}h$. We assume that the operator $A^{(L)}$ has natural scale h_L as would be true for a difference operator on $G^{(L)}$. We introduce a spectrum of operators $A^{(l)}, l = 0, 1, ..., L$, each defined on all of $G^{(L)}$ and of scale $h_l = 2^{-l}h$. Starting from an initial guess u on $G^{(L)}$, we construct the residual

$$r \equiv f - A^{(L)}u = A^{(L)}e, \quad e \equiv \bar{u} - u,$$

where $\bar{u}$ is the exact solution and e is the error. We will use the residual to construct an improved solution u' of the form:

$$u' = u + F^{(L)}r,$$

where $F^{(L)}$ is a linear operator on $G^{(L)}$. This results in a new error

$$e' = \bar{u} - u' = (I - F^{(L)}A^{(L)})e,$$

and a new residual

$$r' = A^{(L)}e' = (I - A^{(L)}F^{(L)})r.$$

Convergence of the above procedure will be guaranteed provided that

$$\|I - A^{(L)}F^{(L)}\| \leq \varepsilon < 1.$$

The PSMG algorithm will be defined by the iteration operator $F^{(L)}$ (denoted $M^{(L)}$ below) in terms of the multiscale operators $A^{(l)}$.

As is usual in multigrid approaches we arrive at the recursive PSMG algorithm by first introducing a two-grid algorithm. The solution of the error equation $A^{(L)}e = r$ is equivalent to the solution of the original equation $A^{(L)}u = f$. In the two-grid PSMG algorithm, we approximate the error e by the exact solution e' of the *coarse scale equation*:

$$A^{(L-1)}e' = r.$$

Note that since $A^{(L-1)}$ is defined on all of $G^{(L)}$, it follows that the error equation is being solved on the *fine* grid, which may be regarded as the union of a set of coarse grids. For example, in the 1-dimensional case the above equation is solved on both the even and odd subgrids. It is for this reason that we prefer the name *multiscale* rather than *multigrid* as a description of the algorithm. Having said this, we will lapse frequently in the sequel into the more familiar use of the word *coarse grid* rather than *coarse scale*! In such cases the term *coarse grid* will be understood to mean the grid $G^{(L)}$ viewed as a union of coarse grids.

Next we will combine the multiple coarse grid solutions defined by e' into a fine grid correction e'' by applying a linear combining transformation (interpolation) of the form:

$$e'' = Q^{(L)}e',$$

where the operator $Q^{(L)}$ remains to be specified. This leads to an improved fine grid solution:

$$u'' = u + e''.$$

The final step involves a smoothing operation on the fine grid:

$$\begin{aligned} u''' &= SM^{(L)}(u'', f), \\ &= (I - Z^{(L)}A^{(L)})u'' + Z^{L}f. \end{aligned}$$

with a corresponding iteration operator $S^{(L)} = I - Z^{(L)}A^{(L)}$. By suitably choosing $A^{(L)}, Q^{(L)}$ and $Z^{(L)}$, the above procedure should lead to convergent solutions. In particular our strategy will involve choosing pairs $Q^{(L)}, Z^{(L)}$ which optimize the convergence rate of the algorithm for given $A^{(L)}$.

We note that the two-grid PSMG algorithm may be described in the form:

$$e^{(T)} \equiv e''' = \mathcal{T}^{(L)} e = (I - T^{(L)} A^{(L)}) e,$$

with the decrease in residual given by:

$$r^{(T)} \equiv r''' = (I - A^{(L)} T^{(L)}) r,$$

where the two-grid iteration operator $\mathcal{T}^{(L)} \equiv I - T^{(L)} A^{(L)}$ is determined by:

$$T^{(L)} = Z^{(L)} + (I - Z^{(L)} A^{(L)}) Q^{(L)} (A^{(L-1)})^{-1}.$$

We define the *two-grid convergence rate* τ of this iteration procedure as the quantity:

$$\tau = \sup_{L} \rho(\mathcal{T}^{(L)}),$$

where $\rho(A)$ denotes the spectral radius of an operator A. Clearly τ provides an upper bound on the asymptotic convergence rate per iteration of the two-grid method on any grid.

We obtain the full PSMG algorithm by recursive application of the two-grid algorithm described above. The corresponding error correction then takes the form:

$$e^{(M)} = \mathcal{M}^{(l)} e = (I - M^{(l)} A^{(l)}) e,$$

where the multi-grid iteration operator $\mathcal{M}^{(l)} = I - M^{(l)} A^{(l)}$ is determined by:

$$M^{(l)} = Z^{(l)} + (I - Z^{(l)} A^{(l)}) Q^{(l)} M^{(l-1)}, \quad l = L, ..., 1,$$

with $M^{(0)} \equiv (A^{(0)})^{-1}$. The corresponding residual reduction operator is given by:

$$I - A^{(l)} M^{(l)} = (I - A^{(l)} Z^{(l)})(I - A^{(l)} Q^{(l)} M^{(l-1)}), \quad l = L, ..., 1.$$

We define the *multigrid convergence rate* of this procedure as the quantity:

$$\mu = \sup_{l,L} \rho(\mathcal{M}^{(l)}).$$

Clearly μ provides a bound on the asymptotic convergence rate of PSMG on any grid. Further bounds on the convergence rate μ will be derived that are extremely sharp.

9.3 Multiscale Convergence Rates

In this section we present an upper bound on the convergence rate of the PSMG algorithm, valid for the special but important case of translation invariant grid operators $A^{(l)}$. To motivate the bound, we rewrite the above recurrence relation for $M^{(l)}$ in the form:

$$\mathcal{M}^{(l)} = \mathcal{T}^{(l)} + (\mathcal{S}^{(l)} - \mathcal{T}^{(l)}) \mathcal{M}^{(l-1)}; \quad \mathcal{M}^{(0)} = 0.$$

In the case that all operators are translation invariant, each operator may be represented as multiplication by a function $M^{(l)}(k)$, $T^{(l)}(k)$ or $S^{(l)}(k)$, in frequency space,

and the above recurrence then applies to these functions for each wave-number k. We conclude from the recurrence formula that $\|\mathcal{M}\| \leq \mu^*$, where

$$\mu^* \equiv \sup_L \max_{k \in G^{(L)}} \max_{l \leq L} \left\{ \mid T^{(l)}(k) \mid /(1 - \mid S^{(l)}(k) - T^{(l)}(k) \mid) \right\}$$

While this bound is the basis for rigorous proofs of convergence [16], we also have used it to create a numerical method to optimize the convergence rate. The bound μ^* is a function of the operators $Z^{(l)}$ and $Q^{(l)}$. By performing a numerical non-linear optimization procedure we attempt to choose the best possible Z and Q. We give some examples in the following sections, referring to [13, 17] for complete details.

9.4 PSMG: Algorithmic Form

The PSMG algorithm works with a single grid of points $G^{(L)}$ of size 2^L in each dimension (called the level L grid, or the *fine* grid), but utilizes operators with different scales $l \leq L$ on that grid. Thus the algorithm is strictly speaking multiscale rather than multigrid. There are three basic operators: a finite difference operator A, an interpolation operator Q and a smoothing operator Z. All operators are periodic on the grid in each coordinate direction. The PSMG algorithm extends naturally to both Neumann and Dirichlet boundary conditions, with no increase in convergence rate. The simplest approach to implementing Neumann or Dirichlet boundary conditions is to use reflection or anti-reflection boundary conditions and an extended grid. However we will discuss only the periodic case here for simplicity.

The operators at scale level l, denoted $A^{(l)}$, $Q^{(l)}$, and $Z^{(l)}$, couple points at a distance $d_l \equiv 2^{L-l}$. Each level l operator is defined at all points of the grid $G^{(L)}$. The basic steps involved at level l, $0 < l \leq L$, for the solution of $A^{(L)}U = f$, starting with an initial guess u, are described by:

Algorithm PSMG(l,u,f):

1. Compute residual: $r = f - A^{(l)}u$.
2. Project residual to coarse grid: $r = r$ (trivial injection).
3. Solve coarse grid residual equation using PSMG: $e = PSMG(l-1, 0, r)$.
4. Interpolate to fine grid: $e' = Q^{(l)}e$.
5. Apply a relaxation: $e'' = (I - Z^{(l)}A^{(l)})e' + Z^{(l)}r$.
6. Compute and return the new solution: $u'' = u + e''$.

An exact solver is utilized on the coarsest grid. The PSMG strategy is to choose $Q^{(l)}$ and $Z^{(l)}$ as functions of $A^{(l)}$ in such a way as to optimize the convergence rate of the above algorithm. Explicit choices for $Q^{(l)}$ and $Z^{(l)}$ were given in [13] for the cases where $A^{(l)}$ represents either the standard 5-point or Mehrstellen discretizations of the Laplacian. In each case we provided upper bounds on the convergence rate for the procedure which are uniform in l.

9.4.1 Application to Poisson's Equation

In order to complete the description of the algorithm it is essential to define the operators $Q^{(l)}$ and $Z^{(l)}$ used for interpolation and smoothing. In this section, we

describe how to choose $Q^{(l)}$ and $Z^{(l)}$ in an optimal way for the special case of an operator which has translation invariant coefficients. We will illustrate the ideas for the Poisson equation discretized on a periodic rectangular grid $G^{(L)}$ of $N = n * n$ points, $n = 2^L$, which we label with the index $i = (i_1, i_2), 0 \leq i_1, i_2 < n$. We will use two discretizations of the negative Laplacian $-\Delta$ in our analysis. The first of these is the standard five-point discretization defined by

$$A_5^{(l)} u = h_l^{-2}(4u_i - u_{i-e_1^l} - u_{i+e_1^l} - u_{i-e_2^l} - u_{i+e_2^l}),$$

where $e_i^{(l)}$ are integer vectors of length $d_{(l)} \equiv 2^{L-l}$ in the coordinate directions in index space, or alternatively by the familiar five- point star notation:

$$A_5^{(l)} = h_l^{-2} \begin{bmatrix} & -1 & \\ -1 & 4 & -1 \\ & -1 & \end{bmatrix}.$$

The second discretization we will study is the more accurate Mehrstellen discretization represented by the nine-point star

$$A_9^{(l)} = (6h_l^2)^{-1} \begin{bmatrix} -1 & -4 & -1 \\ -4 & 20 & -4 \\ -1 & -4 & -1 \end{bmatrix}.$$

Similarly, we will choose the operators $Q^{(l)}$ and $Z^{(l)}$ to be defined by simple symmetric three parameter nine-point star operators (with appropriate scale length):

$$Q_9^{(l)} = \begin{bmatrix} q_{11} & q_1 & q_{11} \\ q_1 & q_0 & q_1 \\ q_{11} & q_1 & q_{11} \end{bmatrix}, \quad Z^{(l)} = h_l^2 \begin{bmatrix} z_{11} & z_1 & z_{11} \\ z_1 & z_0 & z_1 \\ z_{11} & z_1 & z_{11} \end{bmatrix}.$$

For simplicity, we take the parameters q_i and z_i to be independent of the scale parameter l.

Since all of these operators are translation invariant, they are diagonalized by the discrete Fourier transform. The analysis of the PSMG algorithm then becomes particularly convenient. To get an improved convergence rate we have also used a 25-point star operator to define Q:

$$Q_{25}^{(l)} = \begin{bmatrix} q_{22} & q_{12} & q_2 & q_{12} & q_{22} \\ q_{12} & q_{11} & q_1 & q_{11} & q_{12} \\ q_2 & q_1 & q_0 & q_1 & q_2 \\ q_{12} & q_{11} & q_1 & q_{11} & q_{12} \\ q_{22} & q_{12} & q_2 & q_{12} & q_{22} \end{bmatrix}.$$

9.5 PSMG Performance

We have analyzed both 5-point and 9-point discretizations using a simple model of massively parallel computation. The model includes the assumption of 1 processor per fine grid point, nearest neighbor communication to 4 neighbors, sufficient parallel

bandwidth for PSMG communications to precede without collisions, but does not allow for overlap of communication with computation. The model assumes an SIMD architecture, although of course an MIMD architecture would provide results that are at least as good.

In the paper [13] we used the bound μ^* introduced above as a basis for optimizing the convergence rate: to be specific, we optimized the bound as a function of the coefficients of Q and Z, resulting in choices for Q and Z that yielded convergence rates at least as good as μ^*. Our earlier results from [13] have been improved substantially recently in the paper [15]. In [15] we optimize the actual convergence rates $\mu^{(L)}$, for a suitably fine grid L (e.g. $L = 11$). Because all operators are self-adjoint, the spectral radius of the multigrid iteration is just the maximum of the frequency space kernel of the operator. Our new procedure involves explicit evaluation of the kernel $M^{(l)}(k)$ for all frequence pairs k in $G^{(L)}$, which we show can be accomplished in only $O(N)$ operations (on a serial machine). We then optimize the maximum of this kernel as a function of Q and Z, resulting in better parameters and convergence rates than were obtained from μ^*.

For a true measure of efficiency of an iterative method it is necessary to consider the work involved in an iteration as well as the convergence rate obtained. If the asymptotic convergence rate of a method is ρ and the method requires w operations per iteration, the normalized operation count is defined as $w/\log_{10}\rho$, and measures the parallel work required per grid level to reduce the error by a factor of 10. In a parallel method, it is necessary to track both arithmetic and communication work.

For several PSMG methods we obtained asymptotic convergence rates, the number of parallel arithmetic and communication operations required on each grid per iteration, and also the normalized operation count for arithmetic and communication [15]. We summarize the results for some cases in Table 14. In the table we use the notation PSMG9-25, for example, to denote the PSMG algorithm with a 9-point A and a 25-point Q. The Z operator is always taken to be a 9-point stencil.

	convergence	steps per level		normalized steps	
method	rate	comp.	comm.	comp.	comm.
PSMG5-9	.08867	14	12	13.31	11.40
PSMG5-25	.02504	22	16	13.74	9.99
PSMG9-9	.02165	16	12	9.61	7.21
PSMG9-25	.00165	24	16	8.62	5.75

Table 14: PSMG convergence rates

The corresponding coefficients for the interpolation operator Q and the smoothing operator Z are (in the notation of [13, 15]):

PSMG5-9:	$q_0 = .25$	$q_1 = .125$	$q_{11} = .0625$
	$z_0 = .278079$	$z_1 = .0534577$	$z_{11} = .0125615$
PSMG5-25:	$q_0 = .361017$	$q_1 = .11458$	$q_{11} = .0625$
	$q_2 = -.0309162$	$q_{12} = .00521024$	$q_{22} = .00316188$
	$z_0 = .361452$	$z_1 = .0891718$	$z_{11} = .0293793$
PSMG9-9:	$q_0 = .25$	$q_1 = .125$	$q_{11} = .0625$
	$z_0 = .300589$	$z_1 = .0432465$	$z_{11} = .0139994$
PSMG9-25:	$q_0 = .34152$	$q_1 = .0995677$	$q_{11} = .0625$
	$q_2 = -.0199225$	$q_{12} = .0127161$	$q_{22} = -.00295755$
	$z_0 = .283286$	$z_1 = .0323815$	$z_{11} = .00835795$

9.6 How Does PSMG Compare with Standard MG?

Normalized convergence rates provide a basis for comparison of PSMG with the fast red-black Poisson solvers of standard multigrid. The comparison is of course meaningful only for the massively parallel design range of PSMG – the case where there are about as many (or more) processors as fine grid points. Recent papers by N. Decker [11, 12] describe a very efficient implementation of a parallel version of a variant (RNTRB) of the standard red-black multigrid algorithm. In [15] we show that the PSMG9-25 method requires less than one half of the arithmetic and one fifth of the communication required by RBTRB. RBTRB, as implemented in [12], requires 13 arithmetic and 21 communication operations for a convergence rate of .19, yielding normalized values of 18.02 parallel arithmetic and 29.12 parallel communication operations per digit of error reduction, as compared to 8.62 arithmetic and 5.75 communication operations required by PSMG. We have studied a wide range of standard RB methods in [41], and there conclude that PSMG is close to four times more efficient than the best of them.

We do not address in this paper the question of whether other standard multigrid cycles may give better parallel performance than RB methods.

References

[1] Bank, R.E.; Dupont, T.: *An optimal process for solving finite element equations.* Math. Comp. 36, 35-51, 1981.

[2] Bolduc, R.: *SUPRENUM Fortran, syntax specifactions.* SUPRENUM Report 7, SUPRENUM GmbH, Bonn, 1987.

[3] Bomans, L.; Hempel, R.: *The Argonne/GMD macros in FORTRAN for portable parallel programming and their implementation on the Intel iPSC/2.* Arbeitspapiere der GMD Nr. 406, GMD, St. Augustin, 1989.

[4] Brand, K.; Lemke, M.; Linden, J.: *Multigrid bibliography.* Arbeitspapiere der GMD Nr. 206, GMD, St. Augustin, 1986.

[5] Brandt, A.: *Guide to multigrid development.* In [24].

[6] Brandt, A.: *Multigrid techniques: 1984 guide with applications to fluid dynamics.* GMD-Studie Nr. 85, 1984.

[7] Brandt, A.: *Multigrid solvers on parallel computers.* In "Elliptic Problem solvers (M. Schultz, ed.)", Academic Press, New York, 1981.

[8] Briggs, W.: *A Multigrid Tutorial.* SIAM, Philadelphia, PA, 1987.

[9] Chan, T.F.; Saad, Y.: *Multigrid algorithms on the hypercube multiprocessor.* IEEE Trans. Comput. 35, 969-977, 1986.

[10] Chan, T.F.; Schreiber, R.: *Parallel networks for multigrid algorithms: architecture and complexity.* SIAM J. Sci. Comput. 6, 698-711, 1985.

[11] Decker, N.: *On the parallel efficiency of the Frederickson-McBryan multigrid.* ICASE Report No. 90-17, February 1990.

[12] Decker, N.: *A note on the parallel efficiency of the Frederickson-McBryan multigrid algorithm.* SIAM Journal on Scientific and Statistical Computing, to appear.

[13] Frederickson, P.O.; McBryan, O.A.: *Parallel superconvergent multigrid.* in Multigrid Methods: Theory, Applications and Supercomputing, S. McCormick, ed., Marcel Dekker, New York 1988

[14] Frederickson, P.O.; McBryan, O.A.: *Superconvergent multigrid methods.* Cornell Theory Center Preprint, May 1987.

[15] Frederickson, P.O.; McBryan, O.A.: *Normalized convergence rates for the PSMG Method.* SIAM Journal on Scientific and Statistical Computing, January 1991, to appear.

[16] Frederickson, P.O.; McBryan, O.A.: *Recent developments for parallel multigrid.* Proceedings of the Third European Conference on Multigrid Methods, October 1990, ed. U. Trottenberg and W. Hackbusch, to appear.

[17] Gannon, D.; Van Rosendale, J.: *On the structure of parallelism in a highly concurrent PDE solver.* J. Parallel and Distributed Comput. 3, 106-135, 1986.

[18] Gärtel, U.: *Parallel multigrid solver for 3D anisotropic elliptic problems.* Arbeitspapiere der GMD, Nr. 390, St. Augustin, 1989.

[19] Greenbaum, A.: *A multigrid method for multiprocessors.* Appl. Math. Comput. 19, 75-88, 1986.

[20] Grosch, C.E.: *Performance analysis of Poisson solvers on array computers.* Report TR 79-3, Old Domion University, Norfork, VA, 1979.

[21] Grosch, C.E.: *Poisson solvers on large array computer.* Proceedings 1978 LANL Workshop on vector and parallel processors (B.L. Buzbee and J.F. Morrison, eds.), 1978.

[22] Hackbusch, W.: *Multigrid convergence theory.* In [24].

[23] Hackbusch, W.: *Multigrid methods and applications.* Springer, Berlin, 1985.

[24] Hackbusch, W.; Trottenberg, U. (eds.): *Multigrid methods. Proceedings of the Conference held at Köln-Porz, November 23-27, 1981.* Lecture Notes in Mathematics Vol. 960, Springer, Berlin, 1982.

[25] Hempel, R.: *The SUPRENUM communications subroutine library for grid-oriented problems.* Argonne National Laboratory Technical Report ANL-87-23; Argonne, 1987.

[26] Hempel, R.; Lemke, M.; Schüller, A.: *First performance results for grid-oriented applications on SUPRENUM.* Arbeitspapiere der GMD, to appear.

[27] Hempel, R.; Schüller, A.: *Vereinheitlichung und Portabilität paralleler Anwendersoftware durch Verwendung einer Kommunikationsbibliothek.* Arbeitspapiere der GMD, Nr. 234, GMD, St. Augustin, 1986.

[28] Hempel, R.; Schüller, A.: *Experiments with parallel multigrid using the SUPRENUM communications library.* GMD-Studie Nr. 141, 1988.

[29] Herbin, R.; Gerbi, S.; Sonnad, V.: *Parallel implementation of a multigrid method on the experimental ICAP supercomputer.* Appl. Math. Comput. 27, 281-312, 1988.

[30] Kolp, O., Mierendorff, H.: *Performance estimations for SUPRENUM systems.* In [54].

[31] Krämer, O.: *SUPRENUM – Mapping Library, User Manual.* Report, GMD, St. Augustin, 1987.

[32] Lemke, M.: *Erfahrungen mit Mehrgitterverfahren für Helmholtz- ähnliche Probleme auf Vektorrechnern und Multiprozessor- Vektorrechnern.* Arbeitspapiere der GMD, Nr. 278, GMD, St. Augustin, 1987.

[33] Lemke, M.; Schüller, A.; Solchenbach, K.; Trottenberg, U.: *Parallel processing on distributed memory multiprocessors*, GI-20. Jahrestagung, A. Reuter, ed., Informatik-Fachberichte 257, Springer, 1990.

[34] Linden, J.: *A multigrid method for solving the biharmonic equation on rectangular domains.* Arbeitspapiere der GMD Nr. 143, GMD, St. Augustin, 1985.

[35] Linden, J.: *Mehrgitterverfahren für das erste Randwertproblem der biharmonischen Gleichung und Anwendung auf ein inkompressibles Strömungsproblem.* GMD-Bericht Nr. 164, Oldenbourg Verlag, München, 1985.

[36] Linden, J.; Lonsdale, G.; Schüller, A.: *Parallel and vector aspects of a multigrid Navier-Stokes solver.* to appear.

[37] Linden, J.; Steckel, B.; Stüben, K.: *Parallel multigrid solution of the Navier-Stokes equations on general 2D domains.* Parallel Computing 7, 461-475, North Holland, 1988.

[38] Linden, J.; Stüben, K.: *Multigrid methods: An overview with emphasis on grid generation processes.* Arbeitspapiere der GMD Nr. 207, GMD, St. Augustin, 1986.

[39] Maitre, J.F.; Musy, F.: *Multigrid methods: convergence theory in a variational framework.* SIAM J. Numer. Anal., 21, 657-671, 1984.

[40] McBryan, O.A.: *Numerical computation on massively parallel hypercubes.* In Hypercube Multiprocessors 1987, ed. M. T. Heath, 706-719, SIAM, Philadelphia, PA, 1987.

[41] McBryan, O.A.: *Sequential and parallel efficiency of multigrid fast solvers.* University of Colorado CS Dept. Tech Report, September 1990.

[42] McBryan, O.A.; Van de Velde, E.F.: *Hypercube algorithms and implementations.* SIAM J. Sci. Comput. 8, s227-s287, 1987.

[43] McCormick, S.F.; Ruge, J.: *Multigrid methods for variational problems.* SIAM J. Numer. Anal., 19, 924-929, 1982.

[44] Ortega, J.M.; Voigt, R.G.: *Solution of partial differential equations on vector and parallel computers.* SIAM Rev. 27, 149-240, 1985.

[45] Schüller, A.: *Mehrgitterverfahren für Schalenprobleme.* GMD-Bericht Nr. 171, Oldenbourg Verlag, München, 1988.

[46] Schwartz, J.: *A taxonomic table of parallel computers, based on 55 designs.* Ultracomputer Note #69, Courant Institute, New York, 1983.

[47] Solchenbach, K.: *Grid applications on distributed memory architectures: Implementation and evaluation.* In [54].

[48] Solchenbach, K.; Thole, C.A.; Trottenberg, U.: *Parallel multigrid methods: Implementation on SUPRENUM-like architectures and applications.* In Supercomputing. Proceedings of the 1st International Conference on Supercompting, June 8-12, 1987 in Athens. Lecture Notes in Computer Science 297, Springer Verlag, New York.

[49] Solchenbach, K., Trottenberg, U.: *SUPRENUM – system essentials and grid applications.* In [54].

[50] Stüben, K.; Trottenberg, U.: *Multigrid methods: Fundamental algorithms, model problem analysis and applications.* In [24].

[51] Thole, C.A.: *Experiments with multigrid methods on the CalTech-hypercube.* GMD-Studie Nr. 103, 1985.

[52] Thole, C.-A.; Trottenberg, U.: *Basic smoothing procedures for the multigrid treatment of elliptic 3D-operators.* Advances in Multigrid Methods. Proceedings of the Conference Held in Oberwolfach, December 8-13, 1984 (D. Braess, W. Hackbusch, U. Trottenberg, eds.). Notes on Numerical Fluid Mechanics, Volume 11, 102-111. Vieweg, Braunschweig, 1985.

[53] Thole, C.-A.; Trottenberg, U.: *A short note on standard parallel multigrid algorithms for 3D problems.* AMC, 27, 101-115, 1988.

[54] Trottenberg, U. (ed.): *Proceedings of the 2nd International SUPRENUM Colloqium "Supercomputing based on parallel computer architectures".* in Parallel Computing 7, North Holland, 1988.

[55] Zima, H.P.; Bast, H.-J.; Gerndt, H.M.: *SUPERB: A tool for semi-automatic MIMD/SIMD parallelization.* Parallel Comput. 6, 1-18, North Holland, Amsterdam, 1988.