New Architectures: Performance Highlights and New Algorithms

Oliver A. McBryan

CU-CS-403-88

University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

New Architectures: Performance Highlights and New Algorithms

Oliver A. McBryan

CU-CS-403-88 June, 1988

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

New Architectures: Performance Highlights and New Algorithms[†]

Oliver A. McBryan [*]

Dept. of Computer Science,
University of Colorado,
Boulder, CO 80309-0430.

ABSTRACT

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 present elliptic PDE solutions currently running at 3.8 gigaflops, and an atmospheric dynamics model running at 1.7 gigaflops, on a 65,536 processor computer.

One intrinsic disadvantage of a parallel machine is the need to perform inter-processor communication. It is important to ensure that such communication time is maintained at a small fraction of computation time. We analyze standard multigrid algorithms in two and three dimensions from this point of view, indicating that performance efficiencies in excess of 95% 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 multigrid efficiency on 65,536 processors. The fundamental difficulty is the inability to keep 65,536 processors busy when operating on very coarse grids.

Most algorithms used for implementing applications on parallel machines have been derived directly from algorithms designed for serial machines. The previously mentioned multigrid example indicates that such "parallelized" algorithms may not always be optimal. Parallel machines open the possibility of finding totally new approaches to solving standard tasks - intrinsically parallel algorithms. In particular, we present a class of superconvergent multiple scale methods that were motivated directly by massively parallel machines. These methods differ from standard multigrid methods in an intrinsic way, and allow all processors to be used at all times, even when processing on the coarsest grid levels. Their serial versions are not sensible algorithms.

† Presented to the Second International Suprenum Conference, Bonn.

* Research supported in part by DOE contract DE-AC02-76ER03077, by NSF grant DMS-8619856 and by NSF Cooperative Agreement DCR-84200944.

1. INTRODUCTION

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 oil reservoir simulation, in combustion studies, in weather forecasting 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 100 Gflops. 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 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 particularly 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[1], as we demonstrate in section 3. Furthermore, the combined efforts of many researchers have demonstrated that parallel computing is feasible.

The one great disadvantage of a parallel computer, is that it is much harder to program than a serial machine. Each processor must be assigned a distinct component 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.

In section 2 we begin by reviewing some of the parallel architectures that are currently available or are under development. For further details on several of these architectures, and especially for examples of applications such as partial differential equation solution on these machines, we refer to our papers[1-6]. Section 3 provides an example of the extraordinary performance attainable with current parallel machines applied to model problems (PDE solution).

In section 4 we present an equally encouraging case study comparing performance of a real application on a massively parallel machine and on a CRAY-XMP. In section 5 we give estimates for the efficiency of multigrid methods on parallel machines. These lead to very satisfactory efficiency levels when applied to *moderately* parallel machines - we provide details for the case of the SUPRENUM parallel computer. In contrast to the successes reported in sections 3 and 4, and indicated in section 5, section 6 reports on the difficulties inherent to multigrid and other hierarchial methods on *massively* parallel machines. Finally in section 7, we describe a new class of multiscale algorithms which have been motivated by massive parallelism, and which prove to converge at much faster rates than standard multigrid algorithms while fully utilizing all processors.

2. SOME REPRESENTATIVE PARALLEL SYSTEMS

2.1. Classification of Parallel Computers

Parallel computers may be broadly categorized in two types - SIMD or MIMD[7]. SIMD and MIMD are acronyms for *Single Instruction stream - Multiple Data stream*, and *Multiple Instruction stream - Multiple Data stream* respectively. In SIMD computers, every processor executes the same instruction at every cycle, whereas in an MIMD machine, each processor executes instructions independently of the others. The vector unit of a CRAY computer is an example of SIMD parallelism - the same operation must be performed on all components of a vector. Most of the interesting new parallel computers are of MIMD type which greatly increases the range of computations in which parallelism may be effectively exploited using these machines. However, this occurs at the expense of programming ease - MIMD computers are much more difficult to program than SIMD machines. Many current designs incorporate both MIMD and SIMD aspects - typically each node of an MIMD system is itself a vector processor.

Another easy categorization is between machines with global or local memories. In local memory machines, communication between processors is entirely handled by a communication network, whereas in global memory machines a single high-speed memory is accessible to all processors. Beyond this, it becomes difficult to categorize parallel machines. There is an enormous variety in the current designs, particularly in the inter-connection networks. For a taxonomy of current designs, see the paper of Schwartz[8].

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 X-MP, CRAY-2 and ETA-10 are therefore omitted from the subsequent discussions.

2.2. A Partial List of Multi-processors

There are at least 50 to 100 parallel computer projects 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 in production (e.g., ICL DAP, Denelcor HEP, Intel iPSC, NCUBE, FPS T-Series, Connection Machine) and more are under development. One should also remember that the latest

CRAY computers, (e.g. CRAY X-MP and CRAY-2) involve multiple processors, and other vector computer manufacturers such as ETA Systems, NEC, Fujitsu and Hitachi have similar strategies.

Table 1 lists a selection of the parallel computers under development. This is just a sample of the projects mentioned above, but covers a wide range of different architectures chosen more or less at random.

Table 1: Some Parallel Computer Projects

ICL DAP	Caltech Hyper-Cube
Intel iPSC hypercube	NCUBE hypercube
Denelcor HEP-1	NYU/IBM Ultra-computer/RP3
Connection Machine CM-2	FPS T-Series
CRAY X-MP and CRAY-2	ETA-10
IBM 3096	Multiflow
Goodyear MPP	MIT Data-flow Machines
BBN Butterfly	Wisconsin Database Machine
SUPRENUM-1	IBM GF-11 and TF1
Paralex Pegasus	Ametek 2010
Myrias 4000	Cedar Project
Flex	Alliant FX-8
Sequent Balance	Encore Multimax
CCI Navier-Stokes Machine	TERA

2.3. Machine Characteristics

In this section we will look briefly at the characteristics of a number of these machines. The machines currently under development have a number of processors ranging between 2 and 65,536. The machines listed above vary greatly in local processing power. The Intel iPSC has 128 processors, each essentially an IBM PC - initially an 80286 processor, and in the latest versions, an Intel 80386. The 512 processor GF-11 has 512 20-Mflop processors for a combined peak rating of 11 Gflops. The 65,536 processor Connection Machine, CM-2, can deliver at a peak rate of 24 Gflops on some 32-bit floating point operations.

HEP and TERA

The Denelcor HEP was the first commercial parallel computer. The HEP featured a shared memory, with special *access* bits to provide for memory locking on every word. The processors were pipelined units, each capable of executing a large number of separate instruction streams simultaneously. Each processor was rated at 10 Mips. The new TERA computer, designed by HEP creator Burton Smith, will support 256 processors, each similar in many respects to the HEP, and will provide up to 256 Gflops of computational power.

Intel iPSC

The Intel iPSC is the first commercial hypercube computer, and has been the most widely available highly parallel computer in recent years. Built from 128 Intel 80286 processors, peak computer power is under 10 Mflops, yet the iPSC was the basis for a large number of useful experiments in parallel computing. The recently announced 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 and 1280 Mflops single precision. Special communication processors allow message circuits to be established between remote processors without intervention from intermediate processors.

Ametek

The Ametek 2010 is a new commercial machine based on a grid architecture. Individual nodes consist of a Motorola 68020 (25 MHz) with a 68881 co-processor (150 Kflops). An optional upgrade to the 68882 processor (215 Kflops) is possible. A further option provides for a vector processing board based on Weitek chips, with a 20 Mflops performance. Peak performance of a 1024 node system may be as high as 20 Gflops. Memory per node ranges from 2 to 10 Mbytes. Maximum node memory is 8 Mbytes, with a further 10 Mbytes of memory on the vector board. The most interesting feature of the machine is the message routing system which establishes point-to-point communications between remote nodes. Each node has a routing device that can support simultaneous transmission on four links at 20Mbytes each, without interruption of intermediate computational processors. Communication is by "worm-hole" connectivity rather than the usual store-and-forward, resulting in far greater performance for long-range communication. In this communication mode, a connection circuit is first established between remote nodes, incurring a small startup cost, after which the message is transferred in bit-serial fashion in a single operation. Worm-hole communication provides extremely fast long-distance communication, whereas a standard store and forward model would incur large overheads due to the long path-lengths on a grid.

AMT DAP

The DAP was the first massively parallel single-bit computer, and has been widely used for a range of scientific applications. Its current incarnation as the AMT 510 attached processor, provides the capability to attach a 1024 processor DAP array to any VAX or SUN computer. The 510 is a 32×32 array of processors, arranged as a two-dimensional grid and is implemented in VLSI on 16 chips. Additional busses connect all processors on each row and column and are used for broadcasts and other non-local operations. Up to 1 Mbit of memory may be installed per processor, for a combined total of 128 Mbytes. The computer is SIMD, and can execute at up to 60 Mflops, although boolean operations perform at up to 10 Gips.

Paralex Gemini and Pegasus

Paralex Research Inc. is developing a line of highly parallel local memory systems in the supercomputer class. The initial Gemini product supports up to 1000 nodes with peak performance up to about 2 Gips and 500 Mflops. The Gemini uses a hypercube to provide connectivity, and also features a high performance UNIX front end. The second generation Pegasus machine, due in 1989, will support 512 nodes with 8 Gbytes of memory and will provide 25 Gips and 15 Gflops peak rate. This system will be based on the new SPARC technology being licensed by SUN Microsystems. The follow-on Genesis system, planned for 1990, will provide up to 2 Tflops (teraflops) of performance.

Connection Machine

The Connection Machine CM-1 designed by Thinking Machines, Inc., of Cambridge, MA, has 65,536 1-bit processors, though this may be regarded as a prototype for a machine that might have 1,000,000 processors. While designed primarily for artificial intelligence work, this machine has proved to have potential applications to scientific computing applications[1,5]. The recently introduced CM-2 adds 2,000 Weitek floating point processors and 512 Mbytes of memory, to provide a powerful computer for numerical as well as symbolic computing. The CM computers are SIMD machines. 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 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 discretization 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.

Myrias

The Myrias computer, built by Myrias Research Corp. of Edmonton, Alberta, will also have 64K processors with 8 Gbytes of memory and a peak rate of 1600 Mflops. This machine is definitely designed for scientific computing.

RP3 and Ultracomputer

The NYU/IBM Ultracomputer/RP3 projects connect large numbers of processors and an equal number of memory banks through a complex switch. The systems are shared memory computers, although individual local memories are supported too. The design scales up well to at least 64K processors. The initial IBM configuration, the RP3, is a 64 processor machine, with an option to extend to 512 processors, which is being built at IBM, with a peak processing power of about 1 Gip and 500 Mflops (for 512 processors). Smaller prototypes with 8 to 16 processors are running at NYU, and are used for software development for the RP3.

SUPRENUM

The German SUPRENUM project involves coupling up to 256 processor clusters with a network of 200 Mbyte busses. The busses are arranged as a rectangular grid with 16 horizontal and 16 vertical busses. Each cluster consists of 16 processors connected by a fast bus, along with I/O devices for communication to the global bus grid and to disk and host computers. There is a dedicated disk for each cluster. Individual processors can deliver up to 16 Mflops of computing power and support 8 Mbytes of memory. The very high speed of the bus network makes this an 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 cluster containing 16 nodes is already in operation, and a full machine with 16 clusters will be available in 1989.

GF-11 and TF1

The GF-11 is another IBM parallel computer, designed to perform very specific scientific computations at Gflop rates. The GF-11 has 576 processors (including 64 backup processors), coupled through a three stage Benes network which can be reconfigured at every cycle in 1024 different ways by an IBM 3084 control processor. Peak processing power of 11 Gflops will allow previously uncharted computational regimes to be explored. The machine has been designed primarily for solving quantum field theory programs and is not a general purpose computer; in particular, very little software is available. It is an SIMD architecture but with some flexibility in that the settings of local registers may be used to control the behavior of individual processors.

FPS T-Series

The FPS T-Series is a message-based system built from off-the-shelf parts and with a hypercube topology. Each node consists of an INMOS T400 Transputer, a microprocessor distinguished by its built-in communication facilities, which allow it to communicate efficiently with up to 4 other transputers. The FPS T-series machines come in configurations ranging from 16 to 16,386 processors. Each node has a Weitek floating point vector unit and 1 Mbyte of memory. Peak processing speed per node is in the range of 16 Mflops. Since a transputer has only 4 external connections, the multiple connections needed to support a 14-dimensional hypercube and external I/O are obtained by multiplexing the available lines using a 1-to-4 switch. An extremely attractive feature of the transputer is its low startup cost (of order a microsecond) for initiating communication. A host of machines based on the Transputer are being developed in Europe, most using the more advanced T800 Transputer, which supports floating point computation as well as up to six communication links.

CCI Navier-Stokes Machine

The NASA sponsored Navier-Stokes Machine being built at Princeton University involves an experiment with reconfigurable pipelines as well as parallelism. Up to 64 processors are supported with hypercube connections. Each node consists of a CPU, 32 arithmetic processing units and 2 Gbytes of memory. Each of the arithmetic units may be specified to be an adder, multiplier, etc., and connections can then be specified between them in order to represent efficiently a pipeline to evaluate an expression. Reconfiguring the connections takes only 50 nano-seconds. Since each arithmetic unit has a peak processing power of 20 Mflops, the combined processing power per node is 640 Mflops. CCI Corporation plans to market a commercial version of the Navier-Stokes Machine.

Other Approaches

A variety of other important architectures are also under development. These include various dataflow machines (with bus, tree and grid structures), examples include the MIT Tagged Token machine, the NTT Dataflow grid machine and the Manchester Dataflow Machine. Another important class are the tree-structured machines (binary trees, trees with sibling or perfect shuffle connections), examples of which are the Columbia University DADO machine and the CMU Tree Machine. Because of the simplicity of the connections, nearest neighbor machines, such as the MPP, and ring architectures, such as the University of Maryland's ZMOB (256 processors on a ring), are also popular designs.

3. A PERFORMANCE HIGHLIGHT: 3.8 GFLOPS PDE SOLUTION

Discretization of elliptic partial differential equations such as the equation

$$\vec{\nabla}\cdot\vec{k}(\vec{r})\vec{\nabla}u = f(\vec{r}) \ ,$$

by finite element or finite difference methods, leads to systems of equations with sparse coefficient matrices. The fill-in of the matrix tends to follow diagonals and the bandwidth is about $dN^{1/2}$ or $dN^{2/3}$, for two or three dimensional space respectively, where N is the dimension of the matrix and d is the degree of the finite elements used for the discretization. Furthermore, typically only $O(1)$ diagonals have nonzero elements. We have developed a parallel preconditioned conjugate gradient algorithm on the Connection Machine to solve systems of equations with such coefficient matrix structures.

The Preconditioned Conjugate Gradient Method[9-13] finds the solution of the system of equations $Ax = f$, to a specified accuracy ε by performing the following iteration on the vector x, which has been appropriately initialized:

```
$r = f - Ax$
$p = Br$
loop
    $s = <r,Br>/<p,Ap>$
    $r = r - s \cdot Ap$
    $x = x + s \cdot p$
    $rbr = <r,Br>$
    $s = <r,Br>/old<r,Br>$
    $p = Br + s \cdot p$
until $converged$
```

Here B is an approximate inverse of A, which is assumed to be positive definite symmetric, and $<x,y>$ denotes the inner product of vectors x and y. The *preconditioning* operator B can be effective in improving substantially the convergence rate of the algorithm[14].

We parallelize the algorithm by exploiting parallelism in every operation of the iteration. All of the vectors in the algorithm are allocated as CM parallel variables (pvars). For our Poisson-like test problem with a 5-point discretization on a rectangle, the operation $x \to Ax$ is easily written using the NEWS grid addressing modes of the CM. For simplicity we have chosen the pre-conditioning operator B to be the diagonal of the operator A. The other communication intensive operations in the conjugate gradient algorithm are the several inner products of vectors which are required. These inner products perform at very high speeds on the CM by taking advantage of the hypercube structure to evaluate the global sum. For full details on the implementation, we refer to our paper[5].

The performance of this algorithm for a two-dimensional PDE discretized with a five-point formula on a 65,536 processor CM-2 is presented in Figure 1, where we have given results for solution of equations on grids up to size 4096×4096. The four curves are for different virtual processor ratios (VPR). For each curve we have configured the CM as a square grid of $65{,}536 \times VPR$ virtual processors. As can be seen, the highest performance is attained with the highest VPR ratio - corresponding to fullest utilization of memory. The top point on each curve is for a grid that fills the array of virtual processors, and thus corresponds to full processor utilization. For grids that are smaller than the specified number of virtual processors, some virtual processors will be inactive, resulting in decreased megaflop performance. This explains the quadratic shape of the curves with increasing grid size (linear in the number of grid points).

4. MASSIVE PARALLELISM VS HIGH-SPEED VECTORIZATION

An Implementation Case-study

As an example of the current capabilities of massively parallel architectures, we describe the implementation of a standard two-dimensional atmospheric model - the Shallow Water Equation - on the Connection Machine. These equations provide a primitive but useful model of the dynamics of the atmosphere. Because the model is simple, yet captures features typical of more complex codes, the model is frequently used in the atmospheric sciences community to benchmark computers[15]. Furthermore, the model has been extensively analyzed mathematically and numerically[16, 17]. We have recently implemented the shallow water equations model on the Connection Machine, and compared the performance there with the CRAY-XMP.*

4.1. The Shallow Water Equations

The shallow water equations, without a Coriolis force term, take the form

$$\frac{\partial u}{\partial t} - \zeta v + \frac{\partial H}{\partial x} = 0\,,$$

$$\frac{\partial v}{\partial t} - \zeta u + \frac{\partial H}{\partial y} = 0\,,$$

$$\frac{\partial P}{\partial t} + \frac{\partial Pu}{\partial x} + \frac{\partial Pv}{\partial y} = 0\,,$$

where u and v are the velocity components in the x and y directions, P is pressure, ζ is the

* Joint work with R. Sato and P. Swarztrauber of the National Center for Atmospheric Research.

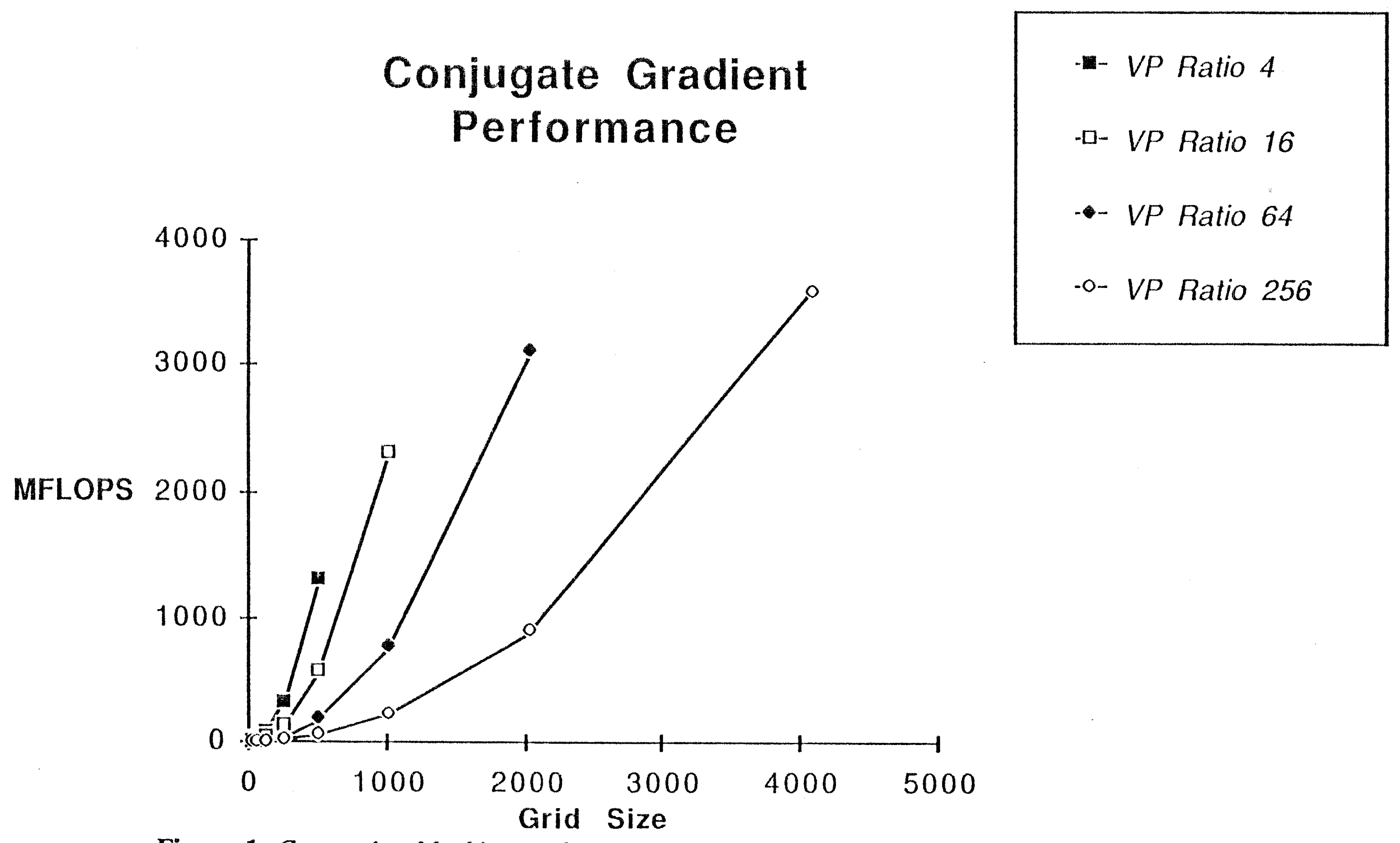

Figure 1: *Connection Machine performance in solution of a 2D elliptic PDE by the pre-conditioned conjugate gradient method on grid sizes ranging up to* 4096×4096, *using 65,536 processors and four different virtual processor ratios (VPR).*

vorticity: $\zeta = \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}$ and H, related to the height field, is given by: $H = P + (u^2 + v^2)/2$. It is required to solve these equations in a rectangle $a \le x \le b$, $c \le y \le d$. Periodic boundary conditions are imposed on u, v, and P, each of which satisfies $f(x+b,y) = f(x+a,y)$, $f(x,y+d) = f(x,y+c)$.

A scaling of the equations results in a slightly simpler format. Introduce mass fluxes $U=Pu$ and $V=Pv$ and the potential velocity $Z=\zeta/P$, in terms of which the equations reduce to:

$$\frac{\partial u}{\partial t} - ZV + \frac{\partial H}{\partial x} = 0\,,$$
$$\frac{\partial v}{\partial t} + ZU + \frac{\partial H}{\partial y} = 0\,,$$
$$\frac{\partial P}{\partial t} + \frac{\partial U}{\partial x} + \frac{\partial V}{\partial y} = 0\,.$$

4.2. Discretization

We have discretized the above equations on a rectangular staggered grid with periodic boundary conditions. The variables P and H have integer subscripts, Z has half-integer subscripts, U has integer and half-integer subscripts, and V has half-integer and integer subscripts respectively.

Initial conditions are chosen to satisfy $\vec{\nabla}\cdot\vec{v} = 0$ at all times. We time difference using the Leap-frog method. We then apply a time filter to avoid weak instabilities inherent in the leap-frog scheme:

$$F^{(n)} = f^{(n)} + \alpha\,(f^{(n+1)} - 2f^{(n)} + f^{(n-1)})\,,$$

where α is a filtering parameter. The filtered values of the variables at the previous time-step are used in computing new values at the next time-step. For a complete description of the discretization we refer to[15].

4.3. CRAY Fortran Implementation

The Fortran code implementing the above algorithm involves a 2D rectangular grid with variables: $u(i,j)$, $v(i,j)$, $p(i,j)$, $z(i,j)$, $psi(i,j)$, $h(i,j)$. There are three main loops, two corresponding to the leap-frog time propagation of various quantities, and one for the filtering step. A typical code sequence, used in the updating of the U, V and P variables, is:

```
    do 200 j=1,n
    do 200 i=1,m
      unew(i+1,j) = uold(i+1,j)+
                tdts8*(z(i+1,j+1)+z(i+1,j))*(cv(i+1,j+1)+cv(i,j+1)+cv(i,j)
                +cv(i+1,j))-tdtsdx*(h(i+1,j)-h(i,j))
      vnew(i,j+1) = vold(i,j+1)-tdts8*(z(i+1,j+1)+z(i,j+1))
                *(cu(i+1,j+1)+cu(i,j+1)+cu(i,j)+cu(i+1,j))
                -tdtsdy*(h(i,j+1)-h(i,j))
      pnew(i,j) = pold(i,j)-tdtsdx*(cu(i+1,j)-cu(i,j))
                -tdtsdy*(cv(i,j+1)-cv(i,j))
200 continue
```

Each such loop is followed by code to implement the periodic boundary conditions. In the above case, the corresponding boundary code takes the form:

```
    do 210 j=1,n
      unew(1,j) = unew(m+1,j)
      vnew(m+1,j+1) = vnew(1,j+1)
      pnew(m+1,j) = pnew(1,j)
210 continue

    do 215 i=1,m
      unew(i+1,n+1) = unew(i+1,1)
      vnew(i,1) = vnew(i,n+1)
      pnew(i,n+1) = pnew(i,1)
215 continue

    unew(1,n+1) = unew(m+1,1)
    vnew(m+1,1) = vnew(1,n+1)
    pnew(m+1,n+1) = pnew(1,1)
```

Note that there are loops for both the horizontal and vertical boundaries, and in addition some corner values are copied as single items.

4.4. Connection Machine Implementation

For the Connection Machine implementation, we specify that the the machine is to be organized as a 2D rectangular grid of virtual processors, with one virtual processor (vp) per grid point i,j. The grid variables u, v, p, z, h are then allocated as *Pvars* - parallel variables. The connection machine software automatically stores them according to the specified grid format.

The Connection Machine code corresponding to the main double loop given previously in Fortran, is actually simpler than on the CRAY. To begin with, the loops actually disappear from the code. This is because all global do loops are replaced by parallel operations. A second simplification is that relative, rather than absolute, addressing is provided for. Because of the local nature of the discretization equations, such relative addressing is far more convenient. The form of the CM code is then:

```
unew = uold + tdts8*(east(z) + z)*(east(cv) + south(east(cv)) + south(cv) + cv) - tdtsdx*(h - south(h))

vnew = vold - tdts8*(north(z) + z)*(north(cu) + cu + west(cu) + north(west(cu))) - tdtsdy*(h - west(h))

pnew = pold - tdtsdx*(north(cu) - cu) - tdtsdy*(east(cv) - cv)
```

Here *north*, *south*, *east* and *west* are specific relative addressing modes understood by the CM software when dealing with rectangular grids. Note that no explicit communication routines are evident, such as one would usually see in corresponding code for other hypercube processors. We have implemented the shallow water equations in two CM languages - C^* and **LISP*. The C^* code is essentially exactly as above, whereas the **LISP* version differs in employing reverse polish notation in writing the expressions.

All boundary loops are replaced by parallel operations with processor *selection*. Basically we create boolean grid variables which record which processors lie on each of the four boundaries of the rectangle. The boundary loops then take the form:

```
*when top_boundary
   unew(1,col) = unew(m+1,col)
   vnew(m+1,col) = vnew(1,col)
   pnew(m+1,colj) = pnew(1,col)
```

The **when* causes the following instructions to be executed only in processors where the boolean parallel variable *top_boundary* is true. Note that while the copying operations are executed only on the selected boundary processors, these operations take as long as if they were executed on on *all* processors. This is a consequence of the SIMD nature of the CM. These operations will in fact be particularly slow, since they are effectively gather-scatter operations - data must be fetched from points at the opposite side of the grid, and are consequently remotely and irregularly located within the hypercube.

There is an essential simplification that occurs in the case that the grid dimensions are both powers of two. On a hypercube, power-of-two grids *are* periodic. Thus in such cases the code for boundary copying may simply be omitted, and in fact we modified the translated program to detect such cases automatically. This is in fact the *only* change we made in translating the original Fortran program to *LISP, apart from the introduction of parallel variables and operators, and the coding of loops using processor selection.

4.5. Performance Results: CRAY-XMP/48 vs. CM-2

The CRAY-XMP4/8 performed at 560 Mflops with 4 processors on a 512×512 grid. The corresponding performance on a single processor was 148 Mflops.

The CM-2 performed at 1,714 Mflops with 65,536 processors on a 2048×2048 power of two grid. Measurements were actually made for a 512×1024 grid on an 8192 processor CM-2 and were scaled linearly to a full-size machine with 65,536 processors. We feel it makes most sense to give performance figures for standard configurations of a machine, and clearly the largest configuration is the most appropriate for supercomputer comparisons. From previous experience, such scaling is entirely appropriate for the CM-2 provided that only nearest neighbor communication is involved. Non-local communication (as would occur on a grid whose dimensions are not powers of two) would not scale linearly, since longer range communication is required on the larger machine.

In addition to being over three times faster than the four-processor CRAY, the CM-2 can clearly handle much larger problems in memory than can the CRAY. Solving a 2048×2048 grid problem on the CRAY would necessitate recoding the whole problem to use a solid state disk (SSD), and would result in much lower performance.

However the CM-2 has a severe disadvantage when processing a grid that is not a power of two in each direction. For such grids, CM performance dropped by a factor of 2.4 due to boundary effects. Some of the effect is simply due to SIMD inefficiency - having to keep all non-boundary processors idle while processing the boundary points - but in fact the non-local nature of the periodic boundary copying was the most serious factor. Particularly unpleasant is the fact that copying of the single corner points (see the end of section 4.3) takes as long as the whole edges, due to the SIMD architecture, and is therefore extremely inefficient. Performance for a Dirichlet or Neumann boundary condition would not have degraded to this extent because of the local nature of such boundary conditions.

5. COMPUTATIONAL EFFICIENCY OF MULTIGRID

In many situations the most efficient algorithms for the numerical solution of large sparse elliptic problems are the various multigrid algorithms[18-20]. Recently several efficient parallel implementations of multigrid algorithms have been reported on both SIMD and MIMD parallel computers[3-5, 21-26]. We present in this section an analysis of the computational efficiency of moderately parallel computers when performing the multigrid solution of partial differential equations. In order to be concrete, we apply the analysis to the 256 processor SUPRENUM architecture. The analyses given here need to be extended in various ways. For a much more complete analysis of MG algorithms on parallel machines we refer to[27]. For further references on multigrid behavior on various parallel architectures, and for new parallel multigrid approaches, we refer to our papers[1, 3-6, 23, 28-36].

5.1. 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 $n = 16\,m$, 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 γ 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:

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

while the corresponding time spent in communication is:

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

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

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

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

$$E \approx 1\ /\ [1 + 6\ (\nu{+}2)/(\nu R{+}P{+}I) \cdot \beta/\gamma \cdot m^{-1}\ ]\ .$$

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.

5.1.1. 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 400 μsecs, and a per-word transfer cost β of about 1 μsec. We will assume a computation rate of 8 Mflops, so that $\gamma = 1/8$ μsecs, 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 3600 so that the problem is not "large" as defined above. Inserting the above numbers into the expression for r_{comm} we obtain:

$$E \approx 1 \,/\, [\ 1 + 3\cdot(9\cdot 400 + 2\cdot 512\cdot 1)\cdot 8/9 \,/\, 512^2\ ] = .955 \quad ,$$

which indicates a very efficient solution. Since the term $l\alpha$ is more than three times as large as $2m\beta$ 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 μsec, 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. 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.

5.2. 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 $m = 2^l$,

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

while the corresponding time spent in communication is:

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

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 bandwidth. 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 $r_{comm} \equiv T_{comm}/T_{comp}$, the ratio of communication to computation, satisfies:

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

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

$$E \approx 1 / [\ 1 + 7\ (\nu+2)/(\nu R + P + I) \cdot \beta/\gamma \cdot m^{-1}\ ] .$$

5.2.1. A Concrete 3D Example

As an interesting test, we consider the three-dimensional case for the current SUPRENUM machine which has 8 Mbytes per node, a startup cost α for communication of 400 μsecs, and a per-word transfer cost β of about 1 μsec. We will assume a computation rate of 8 Mflops so that $\gamma = 1/8$ μseconds. 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 $N = 128 \cdot 10^6$ 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 2400 so that the problem is *not* "large" as defined above. The efficiency is found from the expression for r_{comm} to be:

$$E \approx 1 / [\ 1 + 21/4 \cdot (6 \cdot 400 + 4/3 \cdot 80^2 \cdot 1) \cdot 8/13 / 80^3\ ] = .9355 .$$

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.

5.2.2. 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 $m = 1/16\, N^{1/2}$ in two-dimensions, but by $m = 1/6.35\, N^{1/3}$ 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 dramatically 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, but not 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 less than a quarter of the total communication cost, whereas it constitutes over three quarters 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 great 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 97.64% as against 96.23% 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.

6. MULTIGRID ON THE CONNECTION MACHINE

The two previous sections have indicated substantial successes for the implementation of PDE solutions on the Connection Machine. We will now see that for certain hierarchial 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[5] and we summarize the main ideas here. The CM-2 implementation follows exactly the same strategy.

As a test problem we solve the five-point discretized Poisson equation for a rectangular grid, using modified Jacobi relaxation 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. See Figure 2 for a clearer description of the grid relationships. 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 1×1 grid, the efficiency can be at most 1/65536.

We present performance curves measured for multigrid on the CM-2 in Figure 3. The bar chart shows the megaflops generated in solution as a function of the number of grid levels utilized. As the number of levels increases, Mflops drop dramatically as expected - most processors are sitting idle most of the time. 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 Thus the multigrid algorithm cannot be said to be very efficient on the CM. Despite this, multigrid is still a substantial benefit as indicated by the curve of solution time in the same figure - the solution time drops steadily with increasing numbers of grid levels, 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

Relation of Multigrid Levels

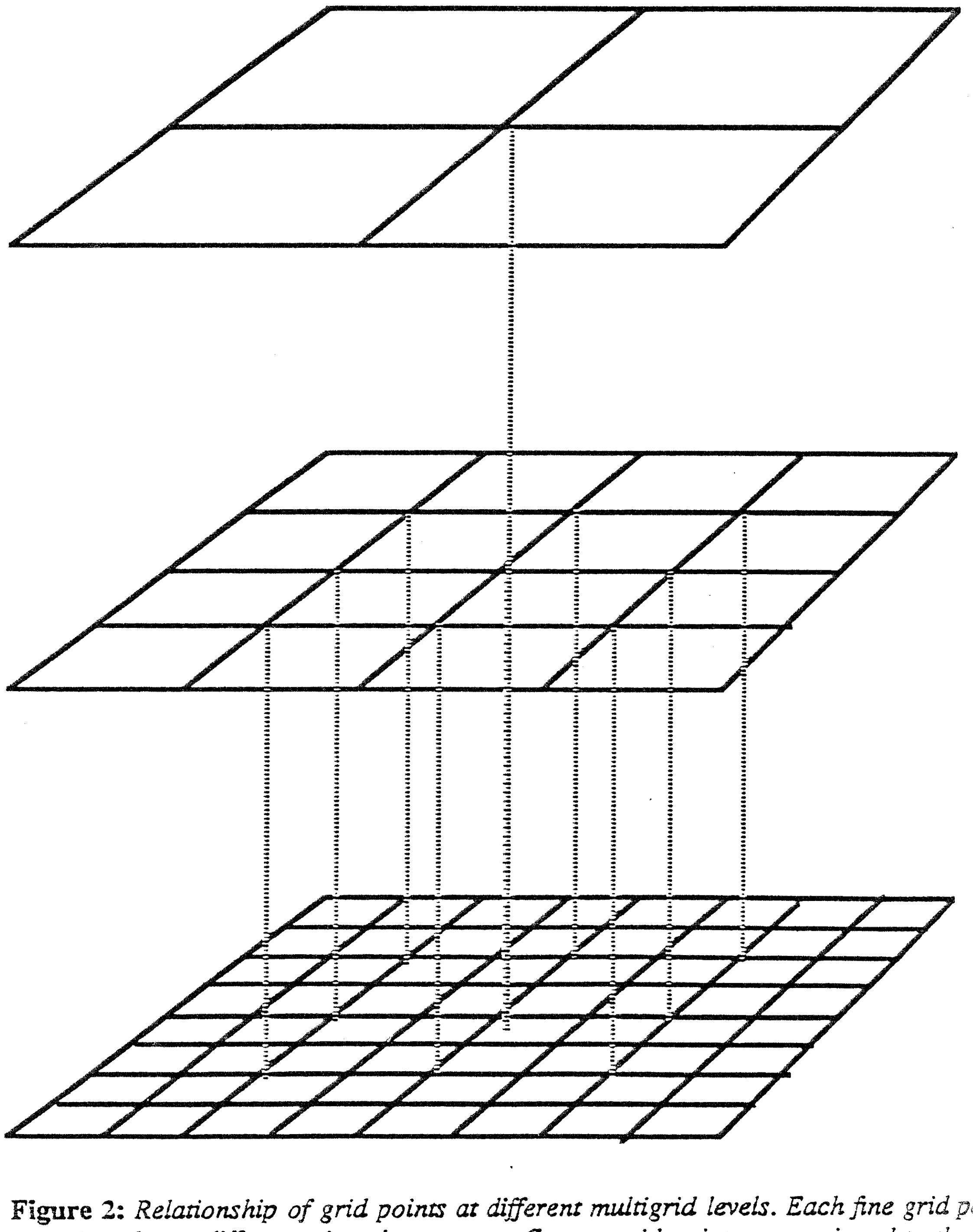

Figure 2: *Relationship of grid points at different multigrid levels. Each fine grid point is assigned to a different virtual processor. Course grid points are assigned to the same processor as the corresponding fine grid point, as shown by dotted lines.*

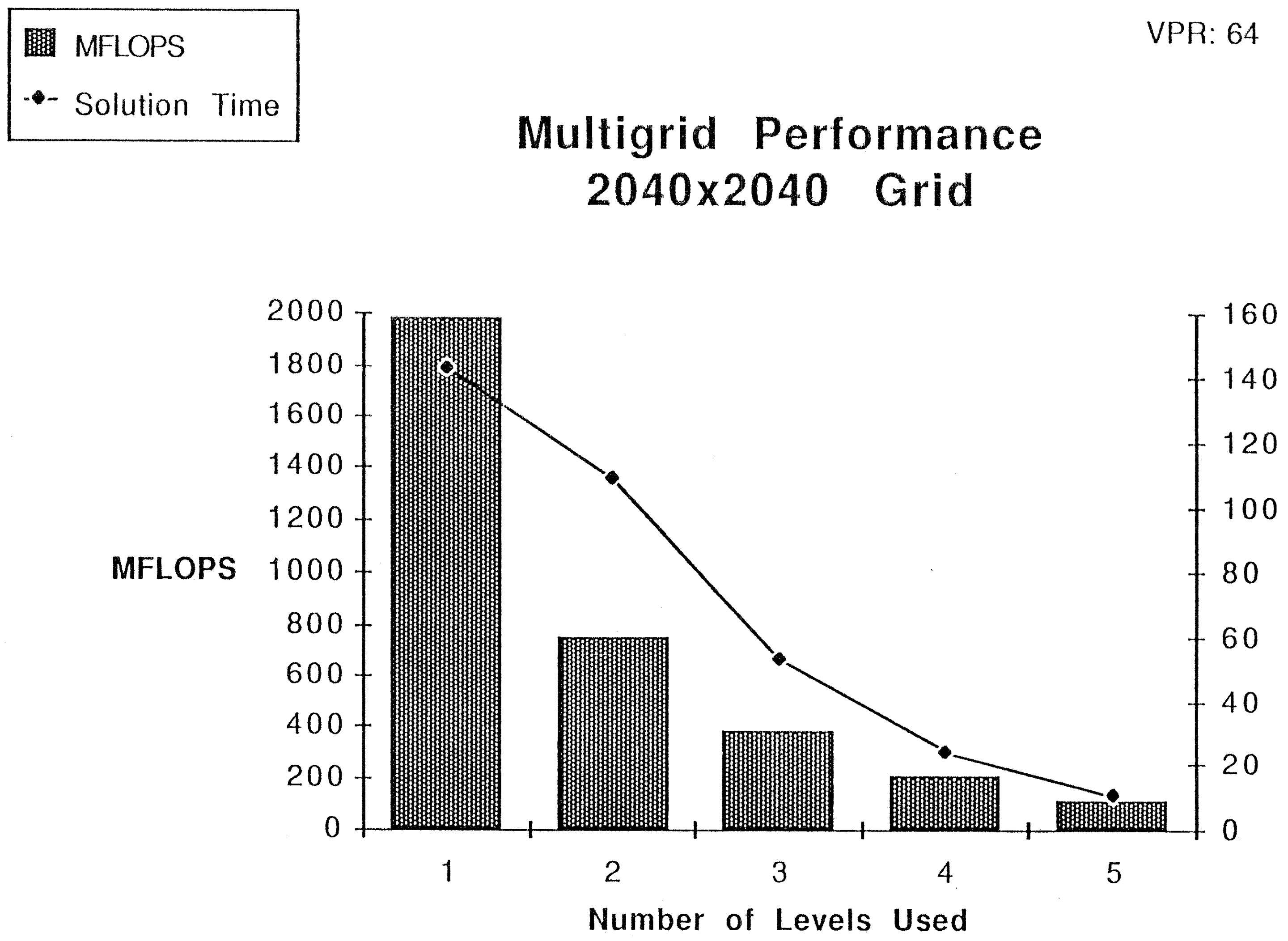

Figure 3: *Connection Machine performance on standard multigrid with (bar chart) varying numbers of grid levels. Megaflop performance with many levels is poor. The curve shows the corresponding solution time for the 2D Poisson equation.*

relaxation, and the solution time curve drops far more steeply as we will see.

7. NEW MULTISCALE ALGORITHMS*

In the previous section, however, 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 $O(log\ N)$, rather than an $O(1)$ solution method.

We describe now an algorithm which we will call PSMG (Parallel Superconvergent Multigrid)[6], that takes a step towards solving this problem. The new algorithm still requires $O(\log(N))$ 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 suprenum 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. We will deal for simplicity with periodic boundary data. For a complete exposition, including proofs and numerical results, we refer to our papers[6, 34].

7.1. 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[18, 20].

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.

* Joint work with Paul Frederickson, Los Alamos National Laboratory.

A typical fine grid.

The standard multigrid coarse grid - the even points.

An 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 the number of grid points is 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 $A^{(l)}$, $l = 0,1, \cdots ,L$, 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

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

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:

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

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

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

and a new residual

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

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 defining 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:*

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

Note that since $A^{(L-1)}$ is by fiat 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:

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

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

$$u'' = u + e'' \quad .$$

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

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

with a corresponding iteration operator $\mathrm{S}^{(L)} = I - S^{(L)} A^{(L)}$. By suitably choosing $A^{(L)}$, $Q^{(L)}$ and $S^{(L)}$, the above procedure should lead to convergent solutions. In particular our strategy will involve choosing pairs $Q^{(L)}$, $S^{(L)}$ 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:

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

with the decrease in residual given by:

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

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

$$T^{(L)} = S^{(L)} + (I - S^{(L)} A^{(L)})\, Q^{(L)} A^{(L-1)^{-1}} \quad .$$

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

$$\tau = \sup_{L} \, \| \mathrm{T}^{(L)} \| \quad .$$

Clearly τ provides an upper bound on the 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:

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

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

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

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

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

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

$$\mu = \sup_{l,L} \, \| \mathrm{M}^{(l)} \| \quad .$$

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

7.2. 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:

$$\mathrm{M}^{(l)} = \mathrm{T}^{(l)} - (\mathrm{S}^{(l)} - \mathrm{T}^{(l)})\, \mathrm{M}^{(l-1)} \quad , \quad \mathrm{M}^{(0)} = 0 \quad .$$

In the case that all operators are translation invariant, each operator may be represented as multiplication by a function $\mathrm{M}^{(l)}(k)$, $\mathrm{T}^{(l)}(k)$ or $\mathrm{S}^{(l)}(k)$ in frequency space, and the above recurrence then applies to these functions for each wave-number k. We conclude that $\| \mathrm{M}^{(l)} \| \le \mu^*$, where

$$\mu^* \equiv \sup_{k} \, [\max_{l} | \mathrm{T}^{(l)}(k) | \; / \; [1 - \max_{l} | \mathrm{S}^{(l)}(k) - \mathrm{T}^{(l)}(k) |]] \quad .$$

While this bound is the basis for rigorous proofs of convergence, we also use it to create a numerical method to optimize the convergence rate. The bound μ^* is a function of the operators $S^{(l)}$ and $Q^{(l)}$. By performing a numerical non-linear optimization procedure we attempt to chose the best possible S and Q. We give some examples in the following section, referring to[6] for complete details.

7.3. Application to Poisson's Equation

In order to complete the description of the algorithm it is essential to define the operators $Q^{(l)}$ and $S^{(l)}$ used for interpolation and smoothing. In this section, we describe how to choose $Q^{(l)}$ and $S^{(l)}$ 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 $N = n \times n$ points, $n=2^L$, which we label with the index $i = (i_1, i_2)$, $0 \le i_1, i_2 < n$. 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

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

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 $S^{(l)}$ to be defined by simple symmetric three parameter nine-point star operators (with appropriate scale length):

$$Q^{(l)} = \begin{bmatrix} q_{11} & q_1 & q_{11} \\ q_1 & q_0 & q_1 \\ q_{11} & q_1 & q_{11} \end{bmatrix} , \quad S^{(l)} = h_l^2 \begin{bmatrix} s_{11} & s_1 & s_{11} \\ s_1 & s_0 & s_1 \\ s_{11} & s_1 & s_{11} \end{bmatrix} .$$

For simplicity, we take the parameters q_i and s_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:

$$Q = \begin{bmatrix} q_{22} & q_{12} & q_2 & q_{12} & q_{22} \\ q_{12} & q_{11} & q_1 & q_{11} & q_{12} \\ q_2 & q_1 & q_0 & q_1 & q_2 \\ q_{12} & q_{11} & q_1 & q_{11} & q_{12} \\ q_{22} & q_{12} & q_2 & q_{12} & q_{22} \end{bmatrix} .$$

7.3.1. Five-point Convergence Rates

For the case of the five-point discretization with nine-point star S and Q, optimization of the above bound μ^* leads to a rigorous multigrid convergence rate bound of approximately .2115 (with a corresponding two-grid rate of .1486) for the case of V-cycles with one smoothing per iteration level. The optimal S matrix in this case is defined by the parameters

$$q_0=.25, \quad q_1=.125, \quad q_{11}=.0625 \ .$$

$$s_0=.311393, \quad s_1=.0761886, \quad s_{11}=.0249449 \ .$$

Improved results are obtained with a 25-point star Q, where we obtain a multigrid convergence rate of .0831 (with two-grid rate of .0632) for the parameter choice:

$$q_0=.391397, \quad q_1=.111803, \quad q_2=-.0413862,$$

$$q_{11}=.0625, \quad q_{12}=.00659854, \quad q_{22}=.00603699,$$

$$s_0=.322645, \quad s_1=.0857152, \quad s_{11}=.0308174 \ .$$

7.3.2. Mehrstellen Convergence Rates

Convergence rates for the Mehrstellen discretization are dramatically sharper. With nine-point star Q and S operators, we obtain a multigrid convergence rate bound μ^* of .02754 (with two-grid bound of .02609) for single-smoother V-cycles. This bound is obtained with Q as in the five-point case and the choice:

$$s_0=.3059, \quad s_1=.0464891, \quad s_2=.0156655 \ .$$

By combining a 3×3 S operator, a 5×5 Q operator and the Mehrstellen operator, we have constructed a PSMG scheme for the Poisson equation which has a two grid convergence rate τ of .00434 and has .00446 as an upper bound on its multigrid convergence rate μ^*. The corresponding optimal parameters are:

$$q_0=.341997, \quad q_1=.0972999, \quad q_2=-.0175355,$$

$$q_{11}=.0625, \quad q_{12}=.0138501, \quad q_{22}=-.00546389,$$

$$s_0=.337042, \quad s_1=.0629468, \quad s_{11}=.0245344 \ .$$

Faster convergence rates may be obtained by using more than one smoothing operation per level. For example, by using two smoothing steps per level, we obtain a multigrid convergence rate bound μ^* of .0013, for which the optimal parameter choice is:

$$q_0=.339308, \quad q_1=.0976648, \quad q_2=-.0168118,$$

$$q_{11}=.0625, \quad q_{12}=.0136676, \quad q_{22}=-.00551516,$$

$$s_0=.351804, \quad s_1=.0739205, \quad s_{11}=.0322007 \ .$$

References

1. O. McBryan, "Solving PDE at 3.8 Gigaflops," University of Colorado CS Dept Preprint, Sept 1987.

2. O. McBryan and E. Van de Velde, "Parallel Algorithms for Elliptic Equations," *Commun. Pure and Appl. Math.*, vol. 38, pp. 769-795, 1985.

3. O. McBryan and E. Van de Velde, "The Multigrid Method on Parallel Computers," in *Proceedings of 2nd European Multigrid Conference, Cologne, Oct. 1985*, ed. J. Linden, GMD Studie Nr. 110, GMD, July 1986.

4. O. McBryan and E. Van de Velde, *Hypercube Algorithms and Implementations*, SIAM J. Sci. Stat. Comput., 8, pp. 227-287, 1987.

5. O. McBryan, "The Connection Machine: PDE Solution on 65536 Processors," *Los Alamos National Laboratory Technical Report, 1987, and Parallel Computing*, to appear.

6. P. O. Frederickson and O. McBryan, "Parallel Superconvergent Multigrid," in *Multigrid Methods: Theory, Applications and Supercomputing*, ed. S. McCormick, Marcel-Dekker, April 1988.

7. M. J. Flynn, "Very high-speed computing," *Proc. IEEE*, vol. 54, pp. 1901-1909, 1966.

8. J. Schwartz, "A Taxonomic Table of Parallel Computers, Based on 55 Designs," Ultracomputer Note #69, Courant Institute, New York, 1983.

9. C. Lanczos, "An Iteration Method for the Solution of the Eigenvalue Problem of Linear Differential and Integral Operators," *J. Res. Nat. Bur. Standards*, vol. 45, pp. 255-282, 1950.

10. M. R. Hestenes and E. Stiefel, "Methods of conjugate gradients for solving linear systems," *J. Res. Nat. Bur. Standards*, vol. 49, pp. 409-436, 1952.

11. J. K. Reid, "On the method of Conjugate Gradients for the Solution of Large Sparse Systems of Linear Equations," in *Large Sparse Sets of Linear Equations*, ed. J. K. Reid, pp. 231-54, Academic Press, New York, 1971.

12. P. Concus, G. H. Golub, and D. P. O'Leary, "A Generalized Conjugate Gradient Method for the Numerical Solution of of Elliptic Partial Differential Equations," in *Sparse Matrix Computations*, ed. D. J. Rose, Academic Press, New York, 1976.

13. G. H. Golub and C. F. Van Loan, *Matrix Computations*, John Hopkins Press, Baltimore, 1984.

14. M. Engeli, Th. Ginsburg, H. Rutishauser, and E. Stiefel, *Refined Iterative Methods for Computation of the Solution and the Eigenvalues of Self-Adjoint Boundary Value Problems*, Birkhauser Verlag, Basel/Stuttgart, 1959.

15. G.-R. Hoffman, P.N. Swarztrauber, and R.A. Sweet, "Aspects of using multiprocessors for meteorological modeling," in *Multiprocessing in Meteorological Models*, ed. D. Snelling, pp. 126-195, Springer-Verlag, Berlin, 1988.

16. R. Sadourny, "The dynamics of finite difference models of the shallow water equations," *JAS*, vol. 32, pp. 680-689, 1975.

17. G.L. Browning and H.-O. Kreiss, "Reduced systems for the shallow water equations," *JAS*, to appear.

18. A. Brandt, "Multi-level adaptive solutions to boundary-value problems," *Math. Comp.*, vol. 31, pp. 333-390, 1977.

19. W. Hackbusch, "Convergence of multi-grid iterations applied to difference equations," *Math. Comp.*, vol. 34, pp. 425-440, 1980.

20. K. Stuben and U. Trottenberg, "On the construction of fast solvers for elliptic equations," *Computational Fluid Dynamics*, Rhode-Saint-Genese, 1982.

21. A. Brandt, "Multi-Grid Solvers on Parallel Computers," ICASE Technical Report 80-23, NASA Langley Research Center, Hampton Va., 1980.

22. D. Gannon and J. van Rosendale, "Highly Parallel Multi-Grid Solvers for Elliptic PDEs: An Experimental Analysis," ICASE Technical Report 82-36, NASA Langley Research Center, Hampton Va., 1982.

23. O. McBryan and E. Van de Velde, "Parallel Algorithms for Elliptic Equation Solution on the HEP Computer," *Proceedings of the Conference on Parallel Processing using the Heterogeneous Element Processor, March 1985*, University of Oklahoma, March 1985.

24. C. Thole, "Experiments with Multigrid Methods on the Caltech Hypercube," GMD-Studien Nr. 103, Gesellschaft fur Mathematik und Datenvararbeitung, St. Augustin, West Germany, 1986.

25. T. F. Chan and Y. Saad, "Multigrid algorithms on the hypercube multiprocessor," Technical Report YALEU/DCS/RR--368, Dept. of Computer Science, Yale University, 1985.

26. T. F. Chan, Y. Saad, and M. H. Schultz, "Solving elliptic partial differential equations on hypercubes," *Hypercube Multiprocessors 1986*, SIAM, Philadelphia, 1986.

27. O. Kolp and H. Mierendorff, "Efficient Multigrid Algorithms for Locally Constrained Parallel Systems," Proceedings of the Second Copper Mountain Multigrid Conference, 1985.

28. O. McBryan and E. Van de Velde, "Parallel Algorithms for Elliptic Equations," in *New Computing Environments: Parallel, Vector and Systolic*, ed. A. Wouk, SIAM, 1986.

29. O. McBryan and E. Van de Velde, "Elliptic and Hyperbolic Equation Solution on Hypercube Multiprocessors," Courant Institute Preprint, Aug. 1985.

30. O. McBryan and E. Van de Velde, "Multiple Grid Algorithms on Parallel Processors," *Presentation to Second International Multigrid Conference*, Copper Mountain, Colorado, March 1985.

31. O. McBryan and E. Van de Velde, "Elliptic Equation Algorithms on Parallel Computers," *Commun. in Applied Numerical Methods*, vol. 2, pp. 311-316, 1986.

32. O. McBryan and E. Van de Velde, "Hypercube Algorithms for Computational Fluid Dynamics," in *Hypercube Multiprocessors 1986*, ed. M. T. Heath, pp. 221-243, SIAM, Philadelphia, 1986.

33. O. McBryan, "Some Comments on the FPS T-Series Computers," Los Alamos National Laboratory Preprint, Sept 1986.

34. P. O. Frederickson and O. McBryan, "Superconvergent Multigrid Methods," Cornell Theory Center Preprint, May 1987.

35. O. McBryan and E. Van de Velde, "Matrix and Vector Operations on Hypercube Parallel Processors," *Parallel Computing*, vol. 5, pp. 117-125, Elsevier, 1987.

36. O. A. McBryan, "Numerical Computation on Massively Parallel Hypercubes," in *Hypercube Multiprocessors 1987*, ed. M. T. Heath, pp. 706-719, SIAM, Philadelphia, PA, 1987.