A CLASS OF PARALLEL ITERATIVE METHODS IMPLEMENTED ON MULTIPROCESSORS

BY

ANTHONY CHRONOPOULOS

Diploma, National and Capodistrian University of Athens, 1979
M.S., University of Minnesota, 1981

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1987

Urbana, Illinois

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

THE GRADUATE COLLEGE

January 1987

WE HEREBY RECOMMEND THAT THE THESIS BY

ANTHONY CHRONOPOULOS

ENTITLED A CLASS OF PARALLEL ITERATIVE METHODS IMPLEMENTED ON MULTIPROCESSORS

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF DOCTOR OF PHILOSOPHY

[illegible] Director of Thesis Research

[illegible] Head of Department

Committee on Final Examination†

[illegible] Chairperson

[illegible]

Ahmed Sameh

Paul Saylor

[illegible]

† Required for doctor's degree but not for master's.

O-517

A CLASS OF PARALLEL ITERATIVE METHODS IMPLEMENTED ON MULTIPROCESSORS

Anthony Chronopoulos, Ph. D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1987
C. W. Gear, Advisor

Iterative methods for systems of linear equations can be efficiently used to solve large sparse problems. The Conjugate Gradient method is such an iterative method for symmetric and positive definite problems. A multitude of iterative methods based on the Conjugate Gradient has been developed for symmetric and nonsymmetric problems. This dissertation generalizes a class of these iterative methods to s-step (or s-dimensional) methods and discusses their convergence and stability. For example the s-Dimensional Optimum Gradient Method is an s-step steepest descent method analyzed by G. Forsythe [Fors68]. The s-step methods have superior parallel properties (simultaneous execution of $2s$ inner products) and do better memory management (better data locality) than their one-step counterparts. Implemented on shared memory systems with memory hierarchy and message passing architectures the s-step methods are significantly faster than their one-step homologues. This is also supported by numerical experiments. These s-step iterative methods can be used to improve the efficiency of codes for solving stiff systems of ordinary equations on parallel computers.

TO THE MEMORY OF GEORGE E. FORSYTHE

ACKNOWLEDGEMENTS

I am grateful to my advisor professor C. W. Gear for his continuous support and invaluable advice. I thank professors A. Sameh and P. Saylor for their help and advice. I also thank the members of my committee, Professors Reed, and Skeel for their interest in my work.

I wish to thank the Center for Supercomputing Research and Development for allowing me to run the experiments on the ALLIANT FX/8 multiprocessor and the Research Scientist William Jalby for his advice and help.

I also wish to thank my office mates and friends Steve Ashby and Fen–Lien Juang for putting up with me and for the many discussions we held. The members of the Computational Environments Group, Alex, Kyle, Sumantri, Thu, Quasim, Jim, John, Ben, Tom, Tam, Mr. Wang, Sohail as well as my friend Constantine have all contributed towards my education as have Ruth with her technical assistance, and Barb with her help in so many things. I am grateful to my home country for providing the education up to my undergraduate degree and to this country for the outstanding graduate programs that I was given the opportunity to participate in.

I am grateful to my father Theodore Chronopoulos for all his love and care before leaving this world. Finally, a very special thanks to my mother Fotini, my brother Panagiotis and sister Panagiota for a lifetime of love and care.

Νυν η ταπεινωση των Θεων
Νυν η σποδος του Ανθρωπου
Νυν Νυν το μηδεν
και Αιεν
ο κοσμος ο μικρος, ο Μεγας.

ODYSSEA ELYTI (AXION ESTI: p88.)

TABLE OF CONTENTS

CHAPTER 1.

INTRODUCTION

One of the main issues in parallel computer architectures is the design of efficient memory systems. Two main trends of parallel architectures are outstanding: the shared memory multiprocessors and the message passing architectures. In the message passing architecture every processor has its private memory. Examples of message passing machines are the INTEL iPSC and the Connection Machine (both with the Hypercube architecture). Shared memory multiprocessor and pipelined processor supercomputers have been successfully implemented. Systems like the CRAY X-MP, CDC CYBER 205, FUJITSU VP-200 have a highly interleaved shared memory.

More recently hierarchical shared memory systems have been designed. They consist of two levels of memory, a small fast first level (local memory or cache) connected to the functional units and to a larger slower second level (global memory). Some implementations of these multilevel memory systems are the CONVEX C1 the ALLIANT FX/8 which provide a hardware managed cache and CRAY-2 with a programmable local memory for each processor.

Accurate numerical solution of mathematical problems derived from modeling physical phenomena often requires a capacity of computer storage and a sustained processing rate that exceed the ones offered by the existing supercomputers. Such problems arise from oil reservoir simulation, electronic circuits, chemical quantum dynamics and atmospheric simulation to mention just a few.

There is an enormous amount of data that must be manipulated to solve these problems with a reasonable accuracy. These data are stored (for the shared memory systems) either in a large global memory (e.g 2-Gbyte for CRAY-2) or in slow secondary storage devices; for the message passing machines they are stored in the private memory of each processor.

Memory contention on shared memory machines constitutes a severe bottleneck for achieving their maximum performance. The same is true for communication cost on a message passing system. For example computations which require the synchronization of all the processors constitute a severe bottleneck for message passing systems. This is because synchronization needs global communication of the system.

It would be desirable to have numerical methods for solving the above mentioned problems which have low communication costs compared to the computation costs. This is interpreted as a small number of global memory accesses for the shared memory systems; and a small number of global communications for the message passing systems.

Therefore both the distributed private memory and hierarchical memory models require careful design of numerical algorithms in order to obtain the maximum efficiency of the system. The algorithm should not only lend itself to vectorization and parallelization but it must provide good data locality. That is the organization of the algorithm should be such that the data can be kept longer in fast registers or local memories and have many arithmetic operations performed on them. A good first measure of the data locality is the size of the

$$Ratio = (Memory\ References)/(Floating\ Point\ Operations).$$

Let us now consider the area of numerical algorithms for large sparse problems. The main vector operations needed are function evaluations or matrix vector multiplications, inner products and linear combinations.

To compute an inner product one or two vectors must be transferred to the functional units to perform $2N-1$ flops. This gives a critical ratio of 1 or 1/2. An inner product is a fully vectorizable operation and causes no communication delays on shared memory supercomputers with few processors. However, on a private memory parallel system the final stage of the computation involves a "fan-in" of the processors to sum up the partial results and a "fan-out" to transmit the result to each processor. This can be a severe bottleneck when the computational rate of each processor is very high as compared with the communication speed between them.

It would desirable to have algorithms which involve more than one inner products at a time. Then the critical ratio decreases (e.g. it is 1/3 if two vectors are referenced and 3 inner products are formed). This can speed up the execution of inner products on shared memory systems with memory hierarchy. Also, for message passing systems the communication cost could be reduced by pipelining the part of the inner products which requires the synchronization of all the processors.

Vector operations like the vector updates

$$v \leftarrow v + c\ u$$

can be pipelined and parallelized. However the critical ratio is about 3/2 and this can cause a processor to be idle (for N cycles, N=vector length, if it has only one port to the memory like CRAY-1 or CRAY-2) during their execution. This ratio can be reduced to

$$(k+2)\ /\ 2k, \quad k \geq 2$$

if vector updates are replaced by linear combinations of the form

$$v + \sum_{i=1}^{k} c_i u_i, \quad k \geq 2.$$

The function evaluations for (general) functions with sparse Jacobian and the (general) matrix vector multiplication are more difficult to analyze. However if the matrix A (or the Jacobian) is structured (banded or bordered) then we could for example benefit from doing Av and A^2v together in some fashion.

We now turn our attention to iterative methods for solving linear systems of equations $Ax=f$, where the matrix A is nonsingular. Iterative methods can be used efficiently to obtain numerically a good approximation to the solution, when the matrix A is large and sparse. The Conjugate Gradient (CG) method [HeSt52] is a widely used iterative method for solving such systems when the matrix A is symmetric and positive definite. Generalizations of CG exist for nonsymmetric problems.

An $s-$*step* generalization of an iterative method can be loosely defined as an iterative method which performs s consecutive steps of the method simultaneously without introducing a significant number of additional computations. This means for example that the inner products (needed for s steps of the one-step method) can be performed simultaneously.

In this work we introduce a class of $s-$*step* iterative methods for symmetric and nonsymmetric problems and analyze them theoretically. We then implement them on parallel systems. Experiments were carried out on a shared memory multiprocessor system. They show both the stability of the new methods and their superior performance over the one-step methods. Finally we show how these methods can be used efficiently in conjunction with codes for solving systems of Ordinary Differential Equations.

CHAPTER 2.

ITERATIVE METHODS FOR LINEAR SYSTEMS

2.1. Introduction

Let us consider the system of linear equations

$$Ax = f$$

where A is a large and sparse matrix of order N. Direct methods are inefficient for solving this problem because of the large amount of work and storage involved. Iterative methods can be used to obtain an approximate solution. Assume that A is a Symmetric Positive Definite (SPD). Then the Conjugate Gradient Method applies. In this chapter we review the basic properties of the Conjugate Gradient method and prepare the ground for generalizing this method to an equivalent method with superior vectorization and parallelization properties and thus faster on certain parallel and vector machines.

Firstly we describe the finite difference discretization of a Partial Differential Equation which gives rise to an SPD linear problem. This will be our model problem. We then present the iterative method of Conjugate Gradients (CG) for SPD problems and discuss its convergence properties. Subsequently we present the Conjugate Residual (CR) method which a useful variant of CG. In Section 2.5 we show how these method applied to the normal equations can be used to solve nonsymmetric problems. Some stability analysis for CG is carried out in Section 2.6.

In Sections 2.7–9 we present the Preconditioned Conjugate Gradient method (PCG). We discuss two outstanding choices of preconditioning for CG the Polynomial (PPCG) and the Incomplete Cholesky Preconditioning (ICCG). ICCG is a sequential preconditioning by its definition. We present a modification of ICCG (due to Van der Vorst [VDVo82]) which is block-vectorizable and block-parallelizable (VICCG). Finally we introduce a fully vectorizable and parallelizable Incomplete Cholesky CG (PICCG) that seems to be efficient for the model problem. This is supported by the error analysis carried out in Section 2.9 and the experiments shown in Chapter 5 .

2.2. A Model Problem

Large, sparse and structured linear systems arise frequently in the Numerical Integration of Partial Differential Equations (PDEs). Thus we borrow our model problems from this area. Let us consider the second order elliptic PDE in two dimensions in a rectangular domain Ω in R^2 with homogeneous Dirichlet boundary conditions:

$$-(au_x)_x - (bu_y)_y + (eu)_x + (hu)_y + cu = g \tag{2.1}$$

where $u = H$ on $\partial\Omega$, and $a(x,y)$, $b(x,y)$, $c(x,y)$, $e(x,y)$, $f(x,y)$ and $g(x,y)$ are sufficiently smooth functions defined on Ω, and $a, b > 0$, $c \geq 0$ on Ω. If we discretize (2.1) using the five-point centered difference scheme on a uniform $n \times n$ grid with

$h = 1/(n+1)$, we obtain a linear system of equations

$$Ax = f$$

of order $N = n^2$. If $e(x,y) \equiv h(x,y) \equiv 0$, then (2.1) is self-adjoint and A is symmetric and weakly diagonally dominant [Varg62]. If we use the natural ordering of the grid points we get a block tridiagonal matrix of the form

$$A = [C_{k-1}, T_k, C_k], \quad 1 \leq k \leq n,$$

where T_k, C_k are matrices of order n; and $C_0 = C_n = 0$. The blocks have the form

$$C_k = \text{diag} \; [\, c_1{}^k, \ldots, c_n{}^k \,]$$

$$T_k = [b_{i-1}^k, a_i^k, b_i^k], \qquad 1 \leq i \leq n,$$

with $b_i{}^k < 0$, $c_i{}^k < 0$, $b_0^k = b_n^k = 0$, and $a_i^k > 0$.

Suppose the three dimensional problem were considered with 7-point line discretization, natural ordering of the planes, and point red-black ordering of the discretization points in each plane. The matrix A would then be symmetric, weakly diagonally dominant, block tridiagonal of order n^3 and it has the form:

$$A = [D_k, \overline{T}_k, D_k], \quad 1 \leq k \leq n,$$

where, $D_k = \text{diag} \; [\, d_1{}^k, \ldots, d_{n^2}{}^k \,]$ with $d_j^k < 0$ and the blocks $\overline{T}_k$ have the form of the matrix for the 2-D case.

2.3. The Conjugate Gradient Method (CG)

The conjugate gradient method (CG) for solving the SPD linear system $Ax = f$ is an iterative scheme which computes approximate solutions x_i starting with an initial guess x_0. In infinite precision arithmetic the exact solution $h = A^{-1} f$ is reached in at most N iterations. Since the error $e_i = h - x_i$ is reduced at every iteration an accurate approximation to the solution is reached after fewer than N steps and CG is a useful iterative method. We will present the method along with some theory describing its convergence and stability.

Algorithm 2.1 : The conjugate gradient method (CG).

```
Initial guess $x_0$
Compute $p_0 = r_0 = f - Ax_0$
Compute $(r_0, r_0)$
For $i = 0$ Until Convergence Do
    $a_i = \frac{(r_i, r_i)}{(p_i, Ap_i)}$
    $x_{i+1} = x_i + a_i p_i$
    $r_{i+1} = r_i - a_i Ap_i$
    $b_i = \frac{(r_{i+1}, r_{i+1})}{(r_i, r_i)}$
    $p_{i+1} = r_{i+1} + b_i p_i$
EndFor.
```

The work per iteration is one matrix vector product, 5N multiplications and 5N additions. The nonzero entries of the matrix A and the vectors x, r, p, Ap need to be stored.

The sequential and parallel complexity of CG for the 2–D and 3–D model problem is tabulated (n_d = the number of nonzero diagonals of A). For uniprocessors or multiprocessors with a small number of processors (e.g. 4 or 8 processors) the matrix vector products dominate the algorithm whereas on parallel systems the inner products dominate because of the communication costs. Performing an inner product on a parallel system can be thought of as a binary tree height reduction with the nodes of the tree being the processors of the system. The parallel system is assumed to have $O(N^{1/k})$ processors where $k \geq 1$.

The next theorem describes some of the relations between the residual and the direction vectors in the conjugate gradient method. The convergence of the method follows this theorem.

Vector Operation	Sequential	Parallel
Vector Updates	$3N$	$O(1)$
Inner Products	$2N$	$O(\log_2 N)$
Matrix Vector Products	$n_d N$	$O(\log_2 n_d)$

Table 2.1 Serial and Parallel complexity of CG parts.

Theorem 2.1 : The residual vectors $r_0, r_1, \cdots$ and the direction vectors $p_0, p_1, \cdots$ generated by the CG process satisfy the following relations:

(i) $p_j = \|r_j\|^2 \sum_{l=0}^{j} \frac{r_l}{\|r_l\|^2}$

(ii) $(r_i, r_j) = 0, \text{ for } i \neq j$

(iii) $(p_i, Ap_j) = 0, \text{ for } i \neq j$

(iv) $(p_i, r_j) = 0, \text{ for } i < j$

$(p_i, r_j) = \|r_i\|^2, \text{ for } i \geq j$

(v) $(r_i, Ap_i) = (p_i, Ap_i)$

(vi) $(r_i, A^k r_j) = (r_i, A^k p_j) = 0, \text{ for } i > j+k$

(vii) $(p_i, A^k p_j) = 0, \text{ for } i > j+k+1$

Proof: [HeSt52] Induction and the defining identities of Algorithm 2.1 can be used to prove (i)–(v). (vi) holds for k=1 because $(r_i, Ar_j) = (r_i, Ap_j) - b_{j-1}(r_i, Ap_{j-1}) = 0$, for $i > j+1$. Let us assume that it holds for $k > 1$, then

$$(r_i, A^{k+1} r_j) = (r_i, A^{k+1} p_j) - b_{j-1}(r_i, A^{k+1} p_{j-1}) =$$

$$(r_i, \frac{1}{a_j} A^k (r_{j+1} - r_j)) - \frac{b_{j-1}}{a_{j-1}} (r_i, A^k (r_j - r_{j-1})) = 0$$

The second relation follows now from the first, if we express p_i as in (i). ■

One implication of this theorem is that the CG process finishes in at most N iterations. This holds because the residual r_N is orthogonal to all the direction vectors by (iv) and the direction vectors span R^N because they are linearly independent by (iii). Relations (vi) and (vii) motivate the orthogonality relations holding in the s-step CG which will be introduced in the next chapter.

Theorem 2.2 : The scalars a_i, b_i can be computed by the formulas:

$$a_i = \frac{(p_i, r_i)}{(p_i, Ap_i)} = \frac{(p_i, r_0)}{(p_i, Ap_i)} = \frac{(r_i, r_i)}{(p_i, Ap_i)}$$

$$b_i = -\frac{(r_{i+1}, Ap_i)}{(p_i, Ap_i)} = -\frac{(r_{i+1}, Ar_i)}{(p_i, Ap_i)} = \frac{(r_{i+1}, r_{i+1})}{(r_i, r_i)}$$

The steplength a_i satisfies the relations

$$\frac{1}{a_0} = \mu(r_0), \quad \mu(p_i) < \frac{1}{a_i} < \mu(r_i), \text{ for } i > 0$$

where $\mu(p) = \dfrac{(p, Ap)}{\|p\|^2}$ is the Rayleigh quotient of the vector p.

Proof: [HeSt52] The formulas for the parameters follow from the algorithm and Theorem 2.1 . If we use the relations:

$$\|p_i\|^2 = \|r_i\|^2 + b_{i-1}^2 \|p_{i-1}\|^2$$

$$(r_i, Ar_i) = (p_i, Ap_i) + b_{i-1}^2 (p_{i-1}, Ap_{i-1})$$

we get that $\|r_i\| < \|p_i\|$ and $(r_i, Ar_i) > (p_i, Ap_i)$. Thus, $\mu(p_i) < \mu(r_i)$. ∎

The last inequality above states that the steplength a_i is bounded by the reciprocals of the smallest and largest eigenvalues of A.

Theorem 2.3 : The approximate solution x_i minimizes the error functional

$$E(x) = (h - x, A(h - x))$$

on the line $y = x_{i-1} + t\ p_{i-1}$. It also minimizes $E(x)$ on the i–dimensional plane

$$P = \{x_0 + \sum_{j=0}^{i-1} a_j p_j\}$$

This plane contains all the points $x_0, \ldots, x_{i-1}$.

Proof: [HeSt52] The error functional can be written in expanded form:

$$E(x) = E(x_0) - \sum_{j=0}^{i-1} \{2a_j(p_j, r_j) - a_j^2(p_j, Ap_j)\}$$

because of the conjugacy of the direction vectors. Now the proof follows. ■

The direction and residual vectors can be expressed as $p_i = \tilde{P}_i(A)r_0$, $r_i = \tilde{R}_i(A)r_0$, for $i = 0, 1, \cdots$. where $\tilde{P}_i$ and $\tilde{R}_i$ are polynomials formed recursively as follows:

$$\tilde{R}_0 = \tilde{P}_0 = 1$$
$$\tilde{R}_{i+1} = \tilde{R}_i - \lambda a_i \tilde{P}_i$$
$$\tilde{P}_{i+1} = \tilde{R}_{i+1} + b_i \tilde{P}_i$$

Thus, the theorem above states that the CG sequence minimizes the functional $E(x)$ over the i-dimensional translated Krylov subspace

$$x_0 + \{r_0, \ldots, A^{i-1}r_0\} = x_0 + \{p_0, \ldots, p_{i-1}\}.$$

Similarly, the polynomials $\tilde{P}_i(\lambda)$, are optimal in the sense that in the i-dimensional space of all polynomials of degree $i - 1$ it is the unique polynomial which minimizes E(x). Since $E(x_i) = \tilde{R}_i(A)\, E(x_0)$ we can use the normalized Chebyshev polynomials

$$T_i\left(\frac{\lambda_n + \lambda_1 - 2\lambda}{\lambda_n - \lambda_1}\right)$$

where $\lambda_1 \leq \cdots \leq \lambda_n$ are the eigenvalues of A, to get a bound on the error at the i–th step.

Theorem 2.4: If $\sigma = \dfrac{\lambda_n + \lambda_1}{\lambda_n - \lambda_1}$ and $\rho = \dfrac{\lambda_n}{\lambda_1}$ (the condition number of A). Then

$$E(x_i) \le \frac{1}{T_i(\sigma)^2} E(x_0) \le 4 \left(\frac{1 - \sqrt{\rho}}{1 + \sqrt{\rho}} \right)^{2i} E(x_0).$$

Proof: [Fors68]

This theorem shows that the CG rate of convergence is at least linear although in practice it seems much faster.

2.4. The Conjugate Residual Method (CR)

The conjugate residual method is a variant of the conjugate gradient method in which the Euclidean norm of the residual $(Ax - f)$ is minimized at every iteration.

Algorithm 2.2: The conjugate residual method.

```
Initial guess $x_0$
Compute $p_0 = r_0 = f - Ax_0$
Compute $(r_0, r_0)$
For $i = 0$ Until Convergence Do
        $a_i = \frac{(r_i, Ar_i)}{(Ap_i, Ap_i)}$
        $x_{i+1} = x_i + a_i p_i$
        $r_{i+1} = r_i - a_i Ap_i$
        $b_i = \frac{(r_{i+1}, Ar_{i+1})}{(r_i, Ar_i)}$
```

```
    $p_{i+1} = r_{i+1} + b_i p_i$
    $Ap_{i+1} = Ar_{i+1} + b_i Ap_i$
EndFor.
```

The work per iteration is one matrix vector product, 6N multiplications and 6N additions. The nonzero entries of A and the vectors x, r, p, Ap, Ar need to be stored. Theorems similar to the ones proved for CG can be proved for CR. For example $(r_i, Ar_j) = 0$ and $(Ap_i, Ap_j) = 0$, for $i \neq j$. At the i-th step the residual error $E(x) = \|Ax - f\|_2^2$ is minimized over the translated i-dimensional Krylov subspace $x_0 + \{r_0, \ldots, A^{i-1} r_0\}$.

2.5. Normal Equations

Let us consider the system of linear equations $Ax = f$, where A is nonsingular, nonsymmetric matrix of order N. This system can be solved by either of the two normal equations systems:

$$A^T A x = A^T f \tag{3.1}$$

$$AA^T y = f, \quad x = A^T y \tag{3.2}$$

Since, both $A^T A$ and AA^T are SPD we can apply CG to either system to obtain an approximate solution of $Ax = f$.

If we solve (3.1) via CG then x_i minimizes $\|r_i\|_2$ over the translated Krylov subspace

$$x_0 + \{A^T r_0, \ldots, (A^T A)^{i-1} A^T r_0\}$$

Algorithm 2.3: The conjugate gradient applied to (3.1) (CGNR)

```
Initial guess $x_0$
Compute $r_0 = f - Ax_0$
Compute $p_0 = A^T r_0$
Compute $(A^T r_0, A^T r_0)$
For $i = 0$ Until Convergence Do
    $a_i = \frac{(A^T r_i, A^T r_i)}{(Ap_i, Ap_i)}$
    $x_{i+1} = x_i + a_i p_i$
    $r_{i+1} = r_i - a_i Ap_i$
    $b_i = \frac{(A^T r_{i+1}, A^T r_{i+1})}{(A^T r_i, A^T r_i)}$
    $p_{i+1} = A^T r_{i+1} + b_i p_i$
EndFor.
```

The work per iteration is two matrix vector product, 5N multiplications and 5N additions. The nonzero entries of A and the vectors x, r, p, Ap need to be stored.

If we solve (3.2) via CG then x_i minimizes $\|h - x_i\|_2$ over the translated Krylov subspace

$$x_0 + \{Ar_0, \ldots, (A^T A)^{i-1} r_0\}$$

the resulting algorithm is:

Algorithm 2.4: The conjugate gradient applied to (3.2) (CGNE)

```
Initial guess $x_0$
Compute $r_0 = f - Ax_0$
Compute $p_0 = A^T r_0$
Compute $(p_0, p_0)$
For $i = 0$ Until Convergence Do
    $a_i = \frac{(r_i, r_i)}{(p_i, p_i)}$
    $x_{i+1} = x_i + a_i p_i$
    $r_{i+1} = r_i - a_i A p_i$ or,    $r_{i+1} = f - A x_{i+1}$
    $b_i = \frac{(r_{i+1}, r_{i+1})}{(r_i, r_i)}$
    $p_{i+1} = A^T r_{i+1} + b_i p_i$
EndFor.
```

The work per iteration is two matrix vector products, 5N multiplications and 5N additions. The nonzero entries of A and the vectors x, r, p, Ap need to be stored.

Since the spectrum of the matrices AA^T and A^TA are the same we should expect that the performance of CGNR and CGNE is the same. However, CGNE minimizes the norm of the error and may yield better performance. The CGNE method is sometimes called Craigs method although it was first proposed by Hestenes.

Similarly to CG we can bound the error functional E(x):

$$E(x_i) \leq 2\left(\frac{1-1/\rho}{1+1/\rho}\right)^i E(x_0)$$

Where ρ is the condition number of A. The fact that this bound depends on the condition number (and not its square root as in CG) indicates that the convergence may be slower. This discourages the use of normal equations without preconditioning.

2.6. Stability Properties of CG

In finite arithmetic the CG process yields residual and direction vectors r_i, p_i which are not mutually conjugate. Nevertheless, the vectors $\{p_i\}$ are linearly independent and, in most cases, the method converges with reasonable accuracy.

The solution of the system $Ax = f$ may be expressed as

$$h = x_0 + \sum_{i=0}^{n-1} a_i p_i$$

Taking inner products with Ap_j we get

$$(x_0, Ap_j) + \sum_{i=0}^{n-1}(Ap_i, p_j)a_i = (h, Ap_j) = (f, p_j)$$

Thus, by applying the CG method to $Ax = f$ we essentially try to solve the transformed system

$$\sum_{i=0}^{n-1}(Ap_i, p_j)a_i = (p_j, r_0) = \|r_j\|_2, \qquad 0 \leq j \leq n-1.$$

In the absence of round–off errors the matrix of the transformed system is diagonal. Otherwise we hope that the diagonal dominates each row

$$\frac{(Ap_i, p_k)}{(Ap_i, p_i)} \ll 1, \text{ for } k \neq i.$$

For $k = i + 1$, these ratios become

$$\frac{(Ap_i, p_{i+1})}{(Ap_i, p_i)} = \frac{a_i}{a_{i-1}} \frac{(Ap_{i-1}, p_i)}{(Ap_{i-1}, p_{i-1})}$$

giving us a propagation formula [HeSt52].

Let b_i be the computed and b'_i the true parameter used in forming p_{i+1}. Then

$$\frac{(Ap_i, p_{i+1})}{(Ap_i, p_i)} = \frac{(Ap_i, r_{i+1})}{(Ap_i, p_i)} + b'_i = b'_i - b_i$$

Now the propagation formula becomes

$$(b'_i - b_i) = \frac{a_i}{a_{i-1}}(b'_{i-1} - b_{i-1})$$

Since $\frac{1}{\lambda_{\max}} < a_i < \frac{1}{\lambda_{\min}}$ we obtain $\frac{a_i}{a_{i-1}} < \frac{\lambda_{\max}}{\lambda_{\min}}$. Hence the condition number of A is a bound on the factor in the propagation formula. This bound means that the only hope for stability is that the matrix should be close to the identity. But the initial residual was ignored in calculating this bound. It is taken into account in the following result.

Proposition 2.1: Let us assume that A has distinct eigenvalues $\lambda_0 < \cdots < \lambda_{n-1}$. Then there exists an initial residual r_0 for which $\frac{a_i}{a_{i-1}} < 1$, and so the CG algorithm is stable.

Proof: [HeSt52] Let us take the eigenvectors as the co-ordinate system and $r_0 = (\delta_0, \delta_1 \epsilon, \ldots, \delta_{n-1}\epsilon^{n-1})$ where ϵ is small. Then the steplengths are

$a_i = \frac{1}{\lambda_i} + O(\epsilon^2)$. Hence $\frac{a_i}{a_{i-1}} = \frac{\lambda_{i-1}}{\lambda_i} + O(\epsilon^2) < 1$ since ϵ is small.

2.7. Preconditioned CG

Let us assume that K is an spd matrix. Then if we apply CG to the preconditioned problem

$$[K^{1/2}AK^{1/2}]\,[K^{-1/2}x] = K^{1/2}f$$

we obtain the preconditioned CG method. The matrix K is usually chosen so that $KA \approx I$.

We remark that in absence of any other preconditioner the use of $K = Diag(\frac{1}{a(i,i)})$ is recommended. The number of steps required by CG to converge equals the degree of minimal polynomial of r_0. So there examples of diagonal matrices of oreder N for which CG takes N steps to converge. However, with diagonal preconditioning it would take only one step.

If we apply Algorithm 2.1 to the preconditioned system we obtain the following algorithm applied to the problem $Ax = f$.

Algorithm 2.5 : The preconditioned conjugate gradient method (PCG).

```
Initial guess $x_0$
Compute $p_0 = Kr_0 = f - Ax_0$
Compute $(r_0, Kr_0)$
```

```
For $i = 0$ Until Convergence Do
    $a_i = \frac{(r_i, Kr_i)}{(p_i, Ap_i)}$
    $x_{i+1} = x_i + a_i p_i$
    $r_{i+1} = r_i - a_i Ap_i$
    $b_i = \frac{(r_{i+1}, Kr_{i+1})}{(r_i, Kr_i)}$
    $p_{i+1} = Kr_{i+1} + b_i p_i$
EndFor.
```

At the i-th step the residual error $E(x) = (h-x,\, A(h-x))$ is minimized over the translated i-dimensional Krylov subspace $x_0 + \{Kr_0, \ldots, KA^{i-1}r_0\}$. The purpose of preconditioning is to decrease the work needed to solve the system. This is accomplished, if K is an approximation to the inverse of A then the matrix of the preconditioned system is closer to the identity matrix than A. Thus the preconditioned system requires fewer steps to converge. This is also good for stability because fewer direction vectors must be generated, although the work per iteration increases.

The preconditioning cost should be such that the total runtime for PCG is less than that of CG. On a scalar processor this can be calculated in terms of the number of flops which are saved minus the operations involved in multiplying by K.

On a vector or parallel system the different parts of CG (inner products, vector updates, multiplications by A) may be running at different speeds and thus the preconditioner must be fast. So its choice may not be the one introducing the fewest number of flops.

Two choices of K seem to be prevalent for the SPD case, the Incomplete Cholesky Factorization and the Polynomial Preconditioner.

2.8. Incomplete Cholesky Preconditioned CG

If A is the matrix resulting from the symmetric 2–D model problem, then the zero entries are given by the set $P = \{(i,j) : i-j \neq 0, 1, n\}$. We demand that the ICCG matrix $K^{-1} = LDL^T$ has zero entries given by P. Then K^{-1} will have the same sparsity structure as A and the nonzero entries of K^{-1} are denoted by $\tilde{a}_i, \tilde{b}_i, \tilde{c}_i$. They can be derived by the recurrences :

$$\tilde{b}_i = b_i, \quad \tilde{c}_i = c_i,$$

$$\tilde{a}_i = a_i - \tilde{b}_{i-1}^2 / \tilde{a}_{i-1} - \tilde{c}_{i-n}^2 / \tilde{a}_{i-n}, \quad 1 \leq i \leq n.$$

We can scale A symmetrically to obtain $\tilde{a}_i = 1$. Then

$$A = K^{-1} + R = (I - F - E)(I - F - E)^T + R$$

where E^T is a matrix consisting of the upper diagonal elements b_i, F^T of the upper diagonal elements c_i, and R is the error in the approximation.

To determine Kv we must solve

$$(I - F - E)z = v$$

or, in block form

$$(I - E_j)z_j = v_j + F_j z_{j-1} \quad j = 1,\ldots,n$$

for the forward step and then do a similar backward step.

Note that there are two dependencies in the previous expression for $z_j(i)$, one on the previous entry $z_j(i-1)$ and one on the entries of the previous z_{j-1}. That is, it is a serial operation. The first dependency can be removed if we use a series expansion for $(I-E_j)^{-1}$ [VDVo82]; the equation becomes

$$z_j = (I + E_j + E_j^2 + \cdots + E_j^m)(v_j + F_j z_{j-1})$$

where $m = 3$ is usually sufficient for a good approximation [VDVo82]. The operation can be block-vectorized (and block-parallelized) since the block dependency remains.

In order to have a parallel preconditioner (PICCG) we must remove the block dependency. This is achieved by using a series expansion for $(I-E_j-F_j)^{-1}$. If we use m=2 then the number of flops needed to perform the multiplication by K is 16N and there is a considerable reduction in the number of iterations as shown by experiments. We can write $A = \overline{K} + S + R$ and $\overline{K}$ is an approximation to K if the series expansion is used.

We now show that the error matrix S is comparable to R, so that we can still hope that the preconditioner is effective. If we write $W = E + F$ then

$$(I + W + \cdots + W^m)^{-1} = (I - W)(I - W^{m+1})^{-1},$$

It follows that $\overline{K} = (I - W)(I - W^{m+1})^{-1}[(I - W)(I - W^{m+1})^{-1}]^T$. Then we get

$$(I - W)(I - W^{m+1}) = I - W + (I - W)W^{m+1}(I - E^{m+1})^{-1} = I - W + \overline{S}.$$

Therefore the matrix S can be expressed as:

$$S = -\bar{S}(I - W) - (I - W)\bar{S}^T - \overline{SS^T}$$

The matrix $I - W$ is bounded: $\|I - W\| \leq 1 + b + c$, where $b = \max\{b_i\}$, $c = \max\{c_i\}$. We can now obtain a bound on the norm of the matrix $\bar{S}$.

$$\|\bar{S}\| \leq (1 + b + c)(b + c)^{m+1}(1 - (b + c)^{m+1})^{-1} \approx (1 + b + c)(b + c)^{m+1}.$$

Neglect the $\overline{SS^T}$ term we obtain S

$$\|S\| \leq 2(1 + b + c)^2(b + c)^{m+1}$$

The set of zero entries of R contains the set $P = \{(i,j) : i - j \neq m - 1\}$. Moreover, the two nonzero diagonals have $b_i c_{i-1}$ as entries. Hence $\|R\| = 2 \max b_i c_{i-1}$.

Suppose A is the 5-point difference operator obtained from the discretization of the 2-D Poisson equation with Dirichlet boundary conditions on the unit square $b_i = c_i = 0.25$, thus $\|R\| \approx .15$. The bound on $\|S\|$ is $\approx .57$. Then the bound on the error in using the series expansion is about four times the error made using the incomplete factorization. Nevertheless, numerical experiments show that the preconditioner is still effective. Since this preconditioner is fully parallelizable, and it is worth using if the number of steps needed for convergence is no more than half that of plain CG. This follows because 16N flops are needed to form Ku compared to the 19N flops of one step of plain CG.

2.9. Polynomial Preconditioned CG

If polynomial preconditioning is used then $K = q(A)$, where q(A) is a polynomial approximation to the inverse of the matrix A [JoMP83]. A stable way to compute Kr_i is

$$\left(\prod_{j=1}^{k} (A - \rho_j) \right) r$$

where ρ_j are the roots of the degree k polynomial $q(\lambda)$.

One choice for q is the Chebyshev polynomial with roots in the interval $[\lambda_{\min}, \lambda_{\max}]$. Then

$$q_k(\lambda) = \frac{(1 - c(\lambda)/c(0))}{\lambda}$$

where

$$c(\lambda) = T\left(\frac{\lambda_{\max} + \lambda_{\min} - 2\lambda}{\lambda_{\max} - \lambda_{\min}} \right)$$

where $T(\lambda)$ is the translated Chebyshev polynomial of degree $k+1$.

When both CG and polynomial PCG are iterated to convergence then the residual polynomials generated by the two methods should have comparable degrees for polynomial PCG to be efficient. This means that approximately the same number of matrix vector products must be performed in either case. On a parallel system, where inner products and not matrix vector products dominate the computation, this may not hold and yet polynomial PCG may still run faster than CG. Johnson Micchelli and Paul [JoMP83] have reported tests where polynomial preconditioning gave results comparable to Incomplete Cholesky.

CHAPTER 3.

s–STEP ITERATIVE METHODS FOR SPD LINEAR SYSTEMS

3.1. Introduction In this chapter we introduce a class of s–step methods and discuss their convergence and stability properties. In the s–step Conjugate Gradient method (s–CG) the s new directions are formed simultaneously from $\{r_i, Ar_i, \ldots, A^{s-1}r_i\}$ and the preceding s directions. All s directions are chosen to be A–orthogonal to the preceding s directions. The approximation to the solution is then advance by minimizing the error functional simultaneously in all s directions. This intuitively means that the progress towards the solution in one iteration of the s–step method equals the progress made over s consecutive steps of the one–step method. This is proven to be true.

The computational work and storage increase slightly (for the s–step methods presented in this chapter) compared to their one step counterparts. However parallel properties and data locality are improved so that the s–step methods are expected to have superior performance on vector and parallel systems. This is due to two attractive properties of the new methods. First the s–step method can be organized so that only sweep through the data is required. This means that the method manages efficiently slower larger levels of memory in systems with memory hierarchy. Second the method can be organized so that the $2s$ inner products required for one s–step iteration are executed simultaneously. This reduces the need for frequent global communication of a parallel system and enhances the performance of the method by pipelining the $2s$ inner products.

In Section 3.2 we present a modification of the CG method which is more parallel and does better memory management than the one in Chapter 2 . This is in effect the form of s*–step* for $s = 1$. In Section 3.3 we review an s–step steepest descent method analyzed by G. Forsythe [Fors68]. This method is modified to obtain the s-step Conjugate Gradient method (s–CG) in Section 3.4. In Sections 3.5–7 we present s–step methods for Conjugate Residual (CR), Conjugate Gradient applied to Normal Equations (s-CGNE), Preconditioned Conjugate Gradient (s–PCG). Finally we discuss the stability of s–CG in Section 3.8.

3.2. A Modified CG Algorithm

In this section we present a modification to the CG method which is more suitable for parallel processing and does better memory management than Algorithm 2.1 .

Algorithm 3.0 : The conjugate gradient method (CG).

```
Initial guess $x_0$
Compute $p_0 = r_0 = f - Ax_0$
Compute $Ar_0$, $a_0$, $b_{-1} = 0$
For $i = 0$ Until Convergence Do
    $p_i = r_i + b_{i-1}p_{i-1}$
    $Ap_i = Ar_i + b_{i-1}Ap_{i-1}$
    $x_{i+1} = x_i + a_i p_i$
    $r_{i+1} = r_i - a_i Ap_i$
    Compute $Ar_{i+1}$
    $b_i = \frac{(r_{i+1}, r_{i+1})}{(r_i, r_i)}$
    $a_{i+1} = (Ar_{i+1}, r_{i+1}) - (\frac{b_i}{a_i})(r_{i+1}, r_{i+1})$
EndFor.
```

We have used the identity $(Ap_i, p_i) = (Ar_i, r_i) - (\frac{b_{i-1}}{a_{i-1}})(r_i, r_i)$. This has increased the number of operation by introducing Ar_i. For the Conjugate Residual no such increase occurs. By doing this, however, we have managed to group the operations so that we can do only one sweep through the memory to obtain the data needed for each step. Also, the two inner products can be performed together.

This algorithm is a stable variant of CG (or CR) and seems more promising than Algorithm 2.1 for both parallel processing because the two inner products required to advance each step can be executed simultaneously. Also, one sweep through the data is required allowing better memory management for large problems.

Next we will try to generalize this to an algorithm which does one memory sweep per s steps.

3.3. An s–Dimensional Steepest Descent Method

Solving a SPD $Ax = f$ linear system of equations using the CG method is equivalent to minimizing a quadratic function

$$E(x) = \frac{1}{2}(x-h)^T A(x-h)$$

where $h = A^{-1}f$ is the solution of the system. Here we will examine the possibility of forming direction planes instead of single direction vectors (as in CG), and minimizing the error functional over the plane.

Definition 3.1 The s-dimensional affine space

$$L_i^s = \{x_i + \sum_{j=0}^{s-1} a_j A^j r_i \quad : a_j \ scalars \ \text{and} \ r_i = f - Ax_i\} \quad .$$

will be called the *s–dimensional plane of steepest descent* of $E(x)$ at x_i.

Since A is not derogatory, $r_i, Ar_i, \ldots, A^{s-1}r_i$ are linearly independent as long as the minimal polynomial of r_i has degree greater than s. In the *optimum s–gradient method* for minimizing the $E(x)$, the point x_{i+1} is defined to be the unique point in the plane L_i^s for which $E(x)$ assumes a minimum. Existence and uniqueness follows from the positive definiteness of A. This method was first introduced by I. M. Khabaza [Khab63] but analyzed by G. Forsythe [Fors68].

Algorithm 3.1 The optimum s-gradient method

```
Choose $x_0$
Compute $r_0 = f - Ax_0$
```

```
For $i = 0$ Until Convergence Do
    $x_{i+1} = x_i + a_i^1 r_i + \cdots + a_i^s A^{s-1} r_i$
    Choose $a_i^j$ to minimize E(x) over $L_i^s$
    $r_{i+1} = r_i - a_i^1 A r_i - \cdots - a_i^s A^s r_i$ or, $r_{i+1} = f - A x_{i+1}$
EndFor
```

Since x_{i+1} minimizes $E(x)$ over the s–dimensional plane L_i^s and r_{i+1} is the gradient of $E(x)$ it is necessary and sufficient that r_{i+1} be orthogonal to this plane. Equivalently, r_{i+1} must be orthogonal to $\{r_i, A^1 r_i, \ldots, A^{s-1} r_i\}$. Then $a_i^1, \ldots, a_i^s$ are determined by the s conditions

$$(r_i, r_i) + a_i^1 (r_i, A r_i) + \cdots + a_i^s (r_i, A^s r_i) = 0$$

$$\cdots\cdots\cdots\cdots\cdots$$

$$(A^{s-1} r_i, r_i) + a_i^1 (A^{s-1} r_i, A r_i) + \cdots + a_i^s (A^{s-1} r_i, A^s r_i) = 0.$$

Definition 3.2 For $k = 0, \pm 1, \pm 2, \cdots$, let the *moments* μ_i^k of r_i be defined by

$$\mu_i^k = r_i^T A^k r_i .$$

The parameters $a_i^1, \ldots, a_i^s$ can be determined by solving the $s \times s$ system of the "normal equations". Since $(A^p r_i, A^q r_i) = (r_i, A^{p+q} r_i) = \mu_i^{p+q}$, this system has the form

$$\mu_i^0 + \mu_i^1 a_i^1 + \cdots + \mu_i^s a_i^s = 0$$

$$\mu_i^1 + \mu_i^2 a_i^1 + \cdots + \mu_i^{s+1} a_i^s = 0$$

$$\cdots\cdots\cdots\cdots\cdots$$

$$\mu_i^{s-1} + \mu_i^s a_i^1 + \cdots + \mu_i^{2s-1} a_i^s = 0$$

The matrix of this system is the matrix of the moments of r_i

$$M_i = (\mu_i^{j+k}), \quad 1 \le j,k \le s$$

M_i is symmetric positive definite as long as $r_i, \ldots, A^{s-1} r_i$ are linearly independent. Then $a_i^1, \ldots, a_i^s$ are uniquely determined.

Note that the optimum s–gradient method is a steepest descent method and that the first iterate is (in exact arithmetic) equal to the s–th iterate of the CG method. The work for a single step is 4sN multiplications and 4sN additions and s matrix vector products and $O(s^3)$ operations to invert the symmetric matrix of moments. The storage is s+1 vectors and maybe the matrix A. This contrasts with the 5sN multiplications and 5sN additions and s matrix vector products needed for the s steps of CG. If a small number (compared to N) of steps are taken the optimum s–gradient method can be useful if $s \ll N$.

Although in the past the s–optimum gradient has been compared to CG [Khab63] our tests show behavior analogous to one dimensional steepest descent methods. This reasonable because no sequence of conjugate directions was formed. Also, the condition number of the matrix of moments increases prohibitively when s > 10.

The optimum s–gradient method is attractive for parallel processing because we can perform the matrix vector products one after another without halting to calculate parameters. Inner products can be carried out together or coupled with the matrix vector products. Finally linear combinations involving more than two vectors have replaced the vector updates.

Next we try to generalize the optimum s-gradient method to an s-dimensional conjugate gradient method.

3.4. The s-Step Conjugate Gradient Method (s-CG)

One way to obtain an s-step conjugate gradient method is to use the s linearly independent directions $\{r_i, \ldots, A^{s-1}r_i\}$ to lift the iteration s dimensions out of the i-th step Krylov subspace $\{r_0, \ldots, A^{is}r_0\}$. Then these directions must be made A-conjugate to the preceding s directions $\{p_{i-1}, \ldots, p_{i-1}^s\}$. Finally, the error functional $E(x)$ must be minimized simultaneously in all s new directions to obtain the new residual r_{i+1}. This method is outlined in the following algorithm.

Algorithm 3.2 The s-Conjugate Gradient Method (s-CG)

```
Choose $x_0$
Compute $p_0^1 = r_0 = f - Ax_0, \ldots, p_0^s = A^{s-1}r_0$
For $i = 0$ Until Convergence Do
    $x_{i+1} = x_i + a_i^1 p_i^1 + \cdots + a_i^s p_i^s$
    Choose $a_i^j$ that minimize E(x) over the s-dimensional plane
    $L_i^s = \{ x_0 + \sum_{j=1}^{s} a_i^j p_i^j \}$
    Compute $r_{i+1} = f - Ax_{i+1}, A^1 r_{i+1}, \ldots, A^{s-1} r_{i+1}$
```

Form the plane $\{p_{i+1}^1, \ldots, p_{i+1}^s\}$

$pi+1^1 = r_{i+1} + b_i^{(1,1)}p_i + \cdots + b_i^{(1,s)}p_i^s$

$p_{i+1}^2 = Ar_{i+1} + b_i^{(2,1)}p_i + \cdots + b_i^{(2,s)}p_i^s$

$\ldots \ldots \ldots \ldots \ldots \ldots \ldots$

$p_{i+1}^s = A^{s-1}r_{i+1} + b_i^{(s,1)}psubi + \cdots + b_i^{(s,s)}p_i^s$

by choosing the scalars $\{b_i^{(j,l)}\}$ to force

A–conjugacy between the planes $\{p_{i+1}^1, \ldots, p_{i+1}^s\}$, $\{p_i^1, \ldots, p_i^s\}$

EndFor

The parameters $\{b_{i-1}^{(j,l)}\}$ and a_i^j are determined by solving $s+1$ linear systems of equations of order s. In order to describe these systems we need to introduce some notation.

Definition 3.3 Let $M_i = \{(p_i^j, Ap_i^l)\}$, $1 \le j,l \le s$. M_i is symmetric. It is non-singular if and only if $p_i^1, \ldots, p_i^s$ are linearly independent.

Definition 3.4 For $j = 1, \ldots, s$ let $\{b_{i-1}^{(j,l)}\}$, $1 \le l \le s$ be the parameters used in updating the direction vector p_i^j. We use the following s–dimensional vectors to denote them. For simplicity we drop the index i from these vectors.

$$\underline{b}^1 = [b_{i-1}^{(1,1)}, \ldots, b_{i-1}^{(1,s)}]^T$$

$$\ldots \ldots \ldots \ldots \ldots \ldots$$

$$\underline{b}^s = [b_{i-1}^{(s,1)}, \ldots, b_{i-1}^{(s,s)}]^T$$

For p_i^j to be A–conjugate to $\{p_{i-1}^1, \ldots, p_{i-1}^s\}$ it is necessary and sufficient that

$$M_{i-1}\underline{b}^1 + \underline{c}^1 = 0$$

$$\cdots\ \cdots\ \cdots\ \cdots \qquad (3.1)$$

$$M_{i-1}\underline{b}^s + \underline{c}^s = 0$$

where the vectors $\underline{c}^j$, $1 \le j \le s$ are

$$\underline{c}^1 = [(r_i, Ap_{i-1}^1), \ldots, (r_i, Ap_{i-1}^s)]^T$$

$$\cdots\ \cdots\ \cdots\ \cdots\ \cdots\ \cdots$$

$$\underline{c}^s = [(A^{s-1}r_i, Ap_{i-1}^1), \ldots, (A^{s-1}r_i, Ap_{i-1}^s)]^T$$

Definition 3.5 Let $\underline{a} = [a_i^1, \ldots, a_i^s]^T$ denote the steplengths used in updating the solution vector at the i–th iteration of the method. It is uniquely determined by solving

$$M_i\underline{a} - m_i = 0$$

where

$$m_i = [(r_i, p_i^1), \ldots, (r_i, p_i^s)]^T$$

Definition 3.6 Let R_i and P_i be the s–dimensional planes $\{r_i, Ar_i, \ldots, A^{s-1}r_i\}$, $\{p_i^1, \ldots, p_i^s\}$ respectively. Also let $l^k[R_i]$ and $l^k[P_i]$ denote some linear combination of the vectors generating these planes.

Lemma 3.1 The residual r_i at the i–th step is orthogonal to the plane R_{i-1}

Proof: We have that

$$r_{i-1} = p_{i-1}^1 - l^1[P_{i-2}]$$

$$\cdots\ \cdots\ \cdots\ \cdots$$

$$A^{s-1}r_{i-1} = p_{i-1}^1 - l^s[P_{i-2}]$$

Since r_i is orthogonal to the plane P_{i-1} we only need to show that r_i is orthogonal to the plane P_{i-2}. This holds from the fact $r_i = r_{i-1} - \sum_{j=1}^{s} a^j A p_{i-1}^j$ and the fact that r_{i-1} is orthogonal to the plane P_{i-2} and the plane P_{i-1} is A-conjugate to the plane P_{i-2}. ■

Proposition 3.1. Under the assumption that the matrices M_i and M_{i-1} are non-singular the linear systems (3.1) have a nontrivial solution if and only if $r_i \neq 0$.

Proof: It suffices to show that $r_i \neq 0$ implies that $\underline{b}^1, \ldots, \underline{b}^s$ and $\underline{a}$ are non-zero vectors. If $\underline{b}^k = 0$ for some k then $(A^{k-1}r_i, Ap_{i-1}^1) = \cdots = (A^{k-1}r_i, Ap_{i-1}^s) = 0$. This implies that $A^{k-1}r_i$ is orthogonal to $r_i - r_{i-1}$ and by lemma 3.1 we conclude that $A^{k-1}r_i$ is orthogonal to r_i. Hence, $r_i = 0$. Now, $m_i = [(r_i, r_i), \ldots, (r_i, A^{s-1}r_i)]^T$ because r_i is orthogonal to the plane P_{i-1}. Thus $m_i \neq 0$ as long as $r_i \neq 0$. ■

The following theorem guarantees the convergence of the s-CG method in at most N/s steps.

Theorem 3.1 Let m be the degree of the minimal polynomial of r_0, and assume $m > (i+1)s$. Then the direction planes P_i and the residuals r_i generated by the s-CG process for $i = 0, 1, \cdots$ satisfy the following relations

(1) (a) P_i is A-conjugate to P_j

(b) P_i is A-conjugate to R_j, for $j = 0, 1, \ldots, i-1$ and $i = 1, 2, \cdots$

(2) (a) r_i is orthogonal to R_j and P_j, for $j = 0, 1, \ldots, i-1$ and $i = 1, 2, \cdots$

(b) $a_j^s \neq 0$, for $j = 0, 1, \ldots, i-1$ and $i = 1, 2, \cdots$

(c) R_i is A–conjugate to R_j and P_j, for $j = 0, 1, \ldots, i-2$ and $i = 1, 2, \cdots$

Proof: We use induction. For the plane P_0 and the vectors r_0, r_1 by the definition of x_1 we have that (1) and 2(c) are empty and (2)(a) holds. Since the vectors $r_0, Ar_0, \ldots, A^{s-1}r_0$ are linearly independent $a_0^s \neq 0$. Let us assume that (1) and (2) hold for P_j, $j = 0, \ldots, i-1$ and r_j, $j = 0, \ldots, i$, then we attach P_i to this set. For (1) (a) we have

$$p_i^1 = r_i + l^1[P_{i-1}]$$
$$\cdots \cdots \cdots \cdots$$
$$p_i^s = A^{s-1}r_i + l^s[P_{i-1}]$$

By the induction hypothesis for (1) the linear combinations $l^1[P_{i-1}], \ldots, l^s[P_{i-1}]$ are A–conjugate to the planes P_j, $j = 0, \ldots, i-2$. By (2) the vectors $r_i, Ar_i, \ldots, A^{s-1}r_i$ are A–conjugate to the planes R_j, $j = 0, \ldots, i-2$. Also,

$$p_i^1 = r_i + \sum_{k=0}^{j-1} l_k^1[P_{i-1}]$$
$$\cdots \cdots \cdots \cdots \cdots \qquad (3.2)$$
$$p_i^s = A^{s-1}r_i + \sum_{k=0}^{j-1} l_k^s[P_{i-1}]$$

Thus $r_i, Ar_i, \ldots, A^{s-1}r_i$ are A–conjugate to the planes P_j, $j = 0, \ldots, i-2$. This implies that P_i is orthogonal to P_j, $j = 0, \ldots, i-2$. By definition P_i is A–conjugate to P_{i-1}, and this proves (1) (a).

To prove (1) (b) we write

$$r_j = p_j^1 - l^1[P_{j-1}]$$

$$\dots \quad \dots \quad \dots \quad \dots$$

$$A^{s-1} r_j = p_j^s - l^s [P_{j-1}]$$

Since P_i is A-conjugate to P_j, $j = 0, \dots, i-1$ we get that P_i is A conjugate to r_j, $j = 0, \dots, i-1$.

It remains to be shown that if $r_{i+1} \neq 0$ then it can be attached to the set $\{P_j, r_j, j = 0, \dots, i\}$. To this end we must prove (2) for r_{i+1}.

For (2) (a) we have

$$r_{i+1} = r_i - a^1 A p_i^1 - \cdots - a^s A p_i^s$$

By the induction hypothesis we get that r_{i+1} is orthogonal to R_j, $j = 0, \dots, i-1$ and by lemma 3.1 it is orthogonal to R_i. By the identities (3.2) r_{i+1} is orthogonal to P_j, $j = 0, \dots, i$.

(b) Since

$$p_j^1 = q_j^1(A) r_0$$

$$\dots \quad \dots \quad \dots \quad \dots$$

$$p_j^s = q_j^s(A) r_0$$

where $q_j^l(\lambda)$ are polynomials of degree js and the planes P_j, $j = 0, \dots, i$ are mutually A-conjugate they form a basis for the Krylov subspace $V_i = \{r_0, Ar_0, \dots, A^{(i+1)s-1} r_0\}$. If $a_i = 0$ and $r_{i+1} \neq 0$, then $r_{i+1} = r_i - A \left(\sum_{j=1}^{s-1} a_i^j p_i^j \right)$.

Thus r_{i+1} is in V_i. By (2) (a) r_{i+1} is orthogonal to V_i. Hence, $r_{i+1} \equiv 0$.

For (c) we must show that R_{i+1} is A-conjugate to R_j, $j = 0, \ldots, i-1$, or equivalently that r_{i+1} is orthogonal to $\{r_j, Ar_j, \ldots, A^{2s-1}r_j\}$. This holds if $\{r_j, Ar_j, \ldots, A^{2s-1}r_j\} \subset V_i$. And this holds if the $degree(A^{2s-1}r_j) \leq (i+1)s-1$, or $j \leq i-1$. Now by equations (3.2) we get that orthogonality to P_j. ■

The following corollary simplifies the computation of the vectors $\underline{c}^j$.

Corollary 3.1 The right-hand side vectors $\underline{c}^1, \ldots, \underline{c}^s$ for the linear systems (3.1) become

$$\underline{c}^1 = [0, \ldots, 0, (r_i, A^s r_{i-1})]^T$$

$$\underline{c}^2 = [0, \ldots, 0, (Ar_i, A^{s-1} r_{i-1}), (Ar_i, A^s r_{i-1})]^T$$

$$\cdots \; \cdots \; \cdots \; \cdots \; \cdots \; \cdots \; \cdots \; \cdots$$

$$\underline{c}^s = [\,(A^{s-1}r_i, A^s r_{i-1}), \ldots, (A^{s-1}r_i, A^s r_{i-1})]^T$$

Proof: We use the definition of $\underline{c}^1, \ldots, \underline{c}^s$ and $p_{i-1}^1, \ldots, p_{i-1}^s$ and (2)(c), then (2)(a). ■

Using this result and the fact that A is symmetric we get that the vectors can be obtained from the 2s inner products

$$(A^s r_i, r_{i-1}), (A^{s+1} r_i, r_{i-1}), \ldots, (A^{2s-1} r_i, r_{i-1})$$

The following proposition reduces the computation of the vectors $\underline{c}^j$ to the first s moments of r_i.

Proposition 3.2 The following recurrence formulae hold true

$$(A^{(s+k)}r_i,r_{i-1}) = -(\frac{1}{a_{i-1}^s})\{(A^k r_i,r_i) + a_{i-1}^{(s-k)}(A^s r_i,r_{i-1})$$

$$+ a_{i-1}^{(s-k+1)}(A^{(s+1)}r_i,r_{i-1}) + \cdots + a_{i-1}^{(s-1)}(A^{(s+2)}r_i,r_{i-1})\}$$

for $k = 0, \ldots, s-1$

Proof: By Theorem 3.1 $(r_i,A^k r_{i-1}) = 0,\ k = 0, \ldots, s-1$ Hence, r_i is orthogonal to $\{Ap_{i-1}^1, \ldots, Ap_{i-1}^{s-1}\}$. Thus

$$(r_i,r_i) = -a_{i-1}^s(r_i,Ap_{i-1}^s) = -a_{i-1}^s(A^s r_i,r_{i-1}).$$

Therefore $(A^s,r_{i-1}) = -\frac{(r_i,r_i)}{a_{i-1}^s}$. The case k > 1 follows inductively. ■

The following corollary reduces the computation of M_i to the the first $2s$ moments of r_i and scalar work.

Corollary 3.2 The matrix of inner products $M_i = (p_i^l,Ap_i^j),\ 1 \le l,j \le s$ is symmetric and it can be formed from the moments of r_i and the s-dimensional vectors $\underline{b}_{j-1}^1, \ldots, \underline{b}_{j-1}^s$ and $\underline{c}_j^1, \ldots, \underline{c}_j^s$.

Proof: If we write out p_i^l and p_i^j then since p_i^l is A-conjugate to the plane P_{i-1} we get

$$(p_i^l,Ap_i^j) = (A^l r_i,A^j r_i) + \underline{b}_{j-1}^{l\,T}\underline{c}_j^j. \ ■$$

The following corollary reduces the vector m_i to the first s moments of r_i.

Corollary 3.3 The vector m_i can be derived from the moments.

Proof:

$$m_i = [(r_i,p_i^1), \ldots ,(r_i,p_i^s)]^T = [(r_i,r_i), \ldots ,(r_i,A^{s-1}r_i)]^T. \blacksquare$$

Next we show that the s–step CG minimizes the error functional on the whole translated Krylov subspace formed by the i–th iteration. This proves that s–CG is an s–step conjugate gradient method.

Theorem 3.2 The approximate solution x_i given by s–CG minimizes the error functional $E(x)$ on the plane

$$L_i^s = \{x_i + a^1 p_i^1 + \cdots + a^s p_i^s\}.$$

It also minimizes $E(x)$ on the $(i+1)s$–dimensional plane

$$P = \{x_0 + \sum_{j=0}^{i}(\alpha_j^1 p_j^1 + \cdots + \alpha_j^s p_j^s)\}$$

where $\alpha_j^1, \ldots ,\alpha_j^s$ are scalars. This plane contains the points x_0, $x_1, \ldots ,x_i$.

Proof: Since $E(x) = (h-x,A(h-x))$

$$E(x+q) = E(x) - 2(q,r) + (q,Aq)$$

where $r = (f - Ax)$. For $x \in P$ and $q_j = \alpha_j^1 p_j^1 + \cdots + \alpha_j^s p_j^s$

$$E(x) = E(x_0) - \sum_{j=0}^{i}[2(q_j,r_j) - (q_j,Aq_j)]$$

by the fact that the planes P_j, $j = 0, \ldots ,i$ are A–conjugate. To minimize the positive quadratic function $E(x)$, it suffices to find the extrema of the functions

$$G_j(\alpha_j^1, \ldots ,\alpha_j^s) = [2(q_j,r_j) - (q_j,Aq_j)]$$

Thus we need to solve $\left\{\frac{\partial G_j}{\partial \alpha_j^1}, \ldots , \frac{\partial G_j}{\partial \alpha_j^s} = 0\right\}$ $j = 0, \ldots ,i$. Since

$$G_j = 2(\sum_{l=1}^{s} a_j^l p_j^1 \, , r_j) - \sum_{l=1}^{s}\sum_{k=1}^{s} a_j^l a_j^k (Ap_j^l, p_j^k)$$

we must solve the systems

$$[(Ap_j^l, p_j^k), \ 1 \le l,k \le s][a_j^1, \ldots, a_j^s]^T = [(p_j^1, r_j), \ldots, (p_j^s, r_j)]^T, \ j = 0, \ldots, i$$

which is precisely what the s–CG algorithm does. ■

Corollary 3.4 If the initial vector x_0 is the same for CG and s–CG then the approximate solution x_i given by s–CG is the same (in exact arithmetic) as the iterate $\tilde{x}_{is}$ given by CG. ■

We now reformulate the s–CG algorithm taking into account the theory developed above. We will denote by

$$P = [\underline{p}^1, \ldots, \underline{p}^s]$$
$$Q = [\underline{q}^1, \ldots, \underline{q}^s]$$

alternately the i–th and (i–1)–th direction planes.

Algorithm 3.3 The s–Conjugate Gradient Method (s–CG)

```
Choose $x_0$

Compute

$P = [r_0 = f - Ax_0, Ar_0, \ldots, A^{s-1}r_0]$

$\mu_0, \ldots, \mu_{2s-1}$

Call Scalar Work Routine

$x_1 = x_0 + P\underline{a}$

For $i = 1$ Until Convergence Do

     If( i odd ) then
```

```
            $Q = [r_i = f - Ax_i, Ar_i, \ldots, A^{s-1}r_i]$
            $\mu_0, \ldots, \mu_{2s-1}$
            Call Scalar Work Routine
            $Q = Q + P[\underline{b}^1, \ldots, \underline{b}^s]$
            $x_{i+1} = x_i + Q\underline{a}$
        Else
            $P = [r_i = f - Ax_i, Ar_i, \ldots, A^{s-1}r_i]$
            $\mu_0, \ldots, \mu_{2s-1}$
            Call Scalar Work Routine
            $P = P + Q[\underline{b}^1, \ldots, \underline{b}^s]$
            $x_{i+1} = x_i + P\underline{a}$
        EndIf
    EndFor
**Scalar Work Routine**
    If( i=0 ) then
        Solve $M_i\underline{a} = m_i$
    Else
        Solve $M_{i-1}\underline{b}^j + \underline{c}^j,\ j = 1, \ldots, s$
        Form and Decompose $M_i$
        Solve $M_i\underline{a} = m_i$
    EndIf
    Return
```

End

Let us examine the work and storage for one iteration of the algorithm. Storage is required for P, Q, x, f, A; The $O(s^2)$ storage for the scalar work routine is negligible.

The work is: (1) Scalar: $O(s^3)$ flops to form and decompose the symmetric matrix M_i for $(s^2+s)s$ and solve $(s+1)$ linear systems. Forming the right hand side vectors $\underline{c}^1, \ldots, \underline{c}^s$ requires $\sum_{k=1}^{s}(2k+1) = (s+1)^2$ flops.

(2) Vector: $(s+1)$ matrix vector products, $2s$ inner products and $(s+1)$ linear combinations of the form $v + \sum_{j=1}^{s} c_j u_j$ for $2s(s+1)N$ flops.

The additional work (compared to s iterations of CG) is: For linear linear combinations $2s(s+1)N - 6sN$ and one matrix vector product. The extra matrix vector product is introduced because the residual vector is computed directly unlike CG where it is the result of a vector update. Since, $s \leq 10$ for stability the additional $O(s^3)$ scalar work is negligible. When $s \leq 5$ at most twice as many flops as in CG are needed to form the linear combinations. As we will see this is not too costly an overhead .

Remark This algorithm can be modified to obtain one which does one sweep through the data per step. This is achieved by computing first the direction plane P (or, Q) and solution update x_i, while simultaneously computing $r_i, Ar_i, \ldots, Ar^{s-1}r_i$. This can be useful when efficient use of slow secondary storage is necessary for very large problems.

3.5. s–Step Conjugate Residual Method (s–CR)

As in the one dimensional CR case we can minimize the error functional $\|f - Ax_{i+1}\|_2$, where $x_{i+1} = x_i + a_i^1 p_i^1 + \cdots + a_i^s p_i^s$, over the (i+1)s–dimensional translated Krylov subspace $x_0 + \{r_0, Ar_0, \ldots, A^{(i+1)s-1} r_0\}$. This gives the s–dimensional Conjugate Residual Method.

Algorithm 3.4 The s–Conjugate Residual Method (s–CR)

```
Choose $x_0$
Compute
$P = [r_0 = f - Ax_0, Ar_0, \ldots, A^{s-1} r_0]$
$\mu_1, \ldots, \mu_{2s}$
Call Scalar Work Routine
$x_1 = x_0 + P\underline{a}$
For $i = 1$ Until Convergence Do
        If( i odd ) then
          $Q = [r_i = f - Ax_i, Ar_i, \ldots, A^{s-1} r_i]$
          $\mu_1, \ldots, \mu_{2s}$
          Call Scalar Work Routine
          $Q = Q + P[\underline{b}^1, \ldots, \underline{b}^s]$
          $x_{i+1} = x_i + Q\underline{a}$
        Else
          $P = [r_i = f - Ax_i, Ar_i, \ldots, A^{s-1} r_i]$
        $\mu_1, \ldots, \mu_{2s}$
        Call Scalar Work Routine
        $P = P + Q[\underline{b}^1, \ldots, \underline{b}^s]$
        $x_{i+1} = x_i + P\underline{a}$
      EndIf
  EndFor
```

The only difference between s–CR and s–CG is that different moments are computed. For CR we need both Ar_i and Ap_i, computing the latter via an extra vector update: $Ap_i = Ar_i + b_{i-1}Ap_{i-1}$. Since s–CG and s–CR involve the same amount of work, we expect the gap in speed between CR and s–CR to be larger than the one between CG and s–CG.

3.6. s–Step Preconditioned Conjugate Gradient Method

If K is an SPD matrix then applying the s–CG to the $[K^{1/2}AK^{1/2}]K^{-1/2}x = K^{1/2}f$ gives rise to the following algorithm. Here $\mu_0, \ldots, \mu_{2s-1}$ denote the moments of the vector $r_i = f - Ax_i$ with respect to the matrix AK and inner product $(., K.)$. For example μ_0, μ_1, μ_2 are (r_i, Kr_i), (AKr_i, Kr_i), $((AK)^2r_i, Kr_i)$.

Algorithm 3.5 The Preconditioned s–Conjugate Gradient Method (s–CG)

```
  Choose $x_0$
  $P = [Kr_0 = K(f - Ax_0), (KA)Kr_0, \ldots, (KA)^{s-1}Kr_0]$
```

```
$\mu_0, \ldots, \mu_{2s-1}$
Call Scalar Work Routine
$x_1 = x_0 + P\underline{a}$
For $i = 1$ Until Convergence Do
    If (i odd ) then
        $Q = [Kr_i = K(f - Ax_i), (KA)Kr_i, \ldots, (KA)^{s-1}Kr_i]$
        $\mu_0, \ldots, \mu_{2s-1}$
        Call Scalar Work Routine
        $Q = Q + P[\underline{b}^1, \ldots, \underline{b}^s]$
        $x_{i+1} = x_i + Q\underline{a}$
    Else
        $P = [Kr_i = K(f - Ax_i), (KA)Kr_i, \ldots, (KA)^{s-1}Kr_i]$
        $\mu_0, \ldots, \mu_{2s-1}$
        Call Scalar Work Routine
        $P = P + Q[\underline{b}^1, \ldots, \underline{b}^s]$
        $x_{i+1} = x_i + P\underline{a}$
    EndIf
EndFor
```

Although $(s+1)$ matrix vector products with A are needed, only s such products with K are needed. Thus the overhead of the preconditioned s–CG is the same as that in s–CG.

3.7. The s–Step Conjugate Gradient Applied to Normal Equations

Here, as in the one dimensional CG applied to the the normal equations, we can minimize either $\|r_i\|_2$ or $\|h - x_{i+1}\|_2$, over the (i+1)s–dimensional translated Krylov subspace $x_0 + \{A^T r_0, (AA^T)A^T r_0, \ldots, (AA^T)^{(i+1)s-1} A^T r_0\}$. We then obtain the s–dimensional Conjugate Gradient Applied to the Normal Equations s–CGNR and s–CGNE respectively.

The s–CGNR method is similar to s–CGNE except that we need to the moments $\mu_1, \ldots, \mu_{2s}$ (instead of $\mu_0, \ldots, \mu_{2s-1}$) for s–CGNE. In s–CGNR the extra operations per step over s steps of CGNR are the same as in s–CG and CG. For s–CGNE they are reduced as we shall see. Consequently, we only present an algorithm for s–CGNE. We denote by μ_k the moments of r_i with respect to AA^T.

Algorithm 3.6 The s–step CGNE

```
Choose $x_0$
Compute
$P=[A^T r_0=A^T(f - Ax_0), (AA^T)A^T r_0, \ldots, (AA^T)^{s-1}A^T r_0]$
$\mu_0, \ldots, \mu_{2s-1}$
Call Scalar Work Routine
$x_1 = x_0 + P\underline{a}$
For $i = 1$ Until Convergence Do
    If (i odd ) then
        $Q = [A^T r_i = A^T(f - Ax_i), (AA^T)A^T r_i, \ldots, (AA^T)^{s-1}A^T r_i]$
```

```
            $\mu_0, \ldots, \mu_{2s-1}$
            Call Scalar Work Routine
            $Q = Q + P[\underline{b}^1, \ldots, \underline{b}^s]$
            $x_{i+1} = x_i + Q\underline{a}$
        Else
            $P = [A^T r_i = A^T(f - Ax_i), (AA^T)A^T r_i, \ldots, (AA^T)^{s-1}A^T r_i]$
            $\mu_0, \ldots, \mu_{2s-1}$
            Call Scalar Work Routine
            $P = P + Q[\underline{b}^1, \ldots, \underline{b}^s]$
            $x_{i+1} = x_i + P\underline{a}$
        EndIf
    EndFor
```

Since $M_i = \{(p_i^l, p_i^j)\}$, $1 \leq l,j \leq s$ only s matrix vector products are needed in one step of s–CGNE. Thus in s–CGNE the overhead results entirely from the linear combinations.

3.8. Stability of the s–CG Method

As in CG when we apply s–CG on the system $Ax = f$, we essentially solve the transformed system

$$\sum_{l=1}^{n/s} \sum_{j=1}^{s} (Ap_l^j, p_i^k)\, a_l^j = (A^k r_i, r_i)$$

where $1 \leq i \leq n/s$ and $1 \leq k \leq s$. Now the diagonal $s \times s$ blocks of the matrix are the matrices M_i. Since $\underline{a} = M_i^{-1} m_i$ we hope to have a good approximate solution at termination if the diagonal blocks dominate. Let the matrix $\overline{M_i}$ denote the $s \times s$ block

$$\begin{array}{c} (Ap_i^1, p_{i+1}^1), \ldots, (Ap_i^s, p_{i+1}^1) \\ \cdots\ \cdots\ \cdots\ \cdots\ \cdots\ \cdots \\ (Ap_i^1, p_{i+1}^s), \ldots, (Ap_i^s, p_{i+1}^s) \end{array}$$

then a weaker requirement is $\| M_i^{-1} \overline{M_i} \| \ll 1$ in some operator norm. Since, $(Ap_i^1, p_{i+1}^j) = (Ap_i^1, A^{j-1} r_{i+1}) + \sum_{k=1}^{s} (p_i^k, Ap_i^1)\, b_k^j$ we can write the matrix $\overline{M_i}$ in the column form

$$[\, (M_i \underline{b}^1 + \underline{c}^1), \ldots, (M_i \underline{b}^s + \underline{c}^s) \,].$$

The condition above becomes

$$\| (\underline{b}^1 - \underline{\hat{b}}^1), \ldots, (\underline{b}^s - \underline{\hat{b}}^s) \| \ll 1$$

in some vector norm, where $\underline{b}^j, \underline{\hat{b}}^j, j = 1, \ldots, s$ are the true and computed scalars.

So essentially the stability of s–CG is closely related to the accuracy the scalars in updating the direction planes are computed. Computing these scalars involves computing the right–hand side vectors $\underline{c}^j$ via the recurrence formulae (in Proposition 3.2) and solving of s linear systems each having coefficient matrix M_i. Since this matrix can be near the matrix of moments of r_i, it may have a relatively large condition number. However, for $s \leq 10$ experiments indicate it is not too large and so s–

CG is stable.

CHAPTER 4.

IMPLEMENTATION OF S–STEP METHODS ON PARALLEL ARCHITECTURES

4.1. Introduction In this chapter we discuss how the s–CG can be efficiently implemented vector and parallel machines. The two different architecture models considered are the shared memory machines with memory hierarchy and the message passing private memory machines. The discuss separately the matrix vector multiplications, inner products, and linear combinations. So the implementation of the s–CR, s–CGNE, and s–PCG (to some extent) are similar.

For an architecture similar to CRAY–1 we show that s–CG applied to the model problem is twice as fast as CG. For the message passing architectures a speedup up to $2s$ over CG can be achieved.

In Section 4.2 we discuss the vector and parallel implementation of CG. In Section 4.3 we show how s–CG can be efficiently implemented on a system with local memory. The implementation of the matrix vector multiplication for bordered systems is also discussed. In Sections 4.4–5 s–CG is implemented on a vector system for efficient memory management and it is compared to CG. In Section 4.6 the implementation of s–PCG is discussed. Finally in Section 4.7 an implementation of s–CG on a message passing system is presented.

4.2. Vector and Parallel Implementation of CG for the Model Problem

Let us first consider the CG case. At each iteration one matrix vector product, two inner products, and three vector updates are performed in a certain order.

(i) The matrix vector product can be written in vector form :

$$Ap(i) = c(i-n) * p(i-n) + c(i) * p(i+n) +$$
$$b(i-1) * p(i-1) + b(i) * p(i+1) + a(i) * p(i)$$

In some machines with vector registers (CRAY X-MP, ALLIANT FX/8) the restructuring software does not take advantage of the shift and so 11 vectors of data are transferred and $9N$ operations are performed, giving a ratio of $\frac{11}{9}$. For example, on the CRAY-1 this operation takes approximately $11N$ clock cycles.

(ii) The inner products are (r_i, r_i) and (p_i, Ap_i), giving a ratio of $\frac{1}{2}$ and 1, respectively. They cannot be performed simultaneously because they are separated by a SAXPY.

(iii) The three vector updates are

$$x_i = x_{i-1} + a_i p_{i-1}$$
$$r_i = r_{i-1} - a_i Ap_i$$
$$p_i = r_i + b_{i-1} p_{i-1}$$

Here the ratio is $\frac{3}{2}$. These operations are memory intensive; consequently, they can be slow unless the communication between the vector functional units is faster than the vector operations. For example, on CRAY-X-MP these operations are executed at the maximum rate whereas on CRAY–1 at half the maximum rate. The CRAY–X–MP has two channels for vector LOAD and one for vector STORE whereas the CRAY–1 has one bidirectional channel.

This situation can be improved slightly if (ii) is combined with (i) or (iii). For example, the update of p_i may be combined with Ap_i and (Ap_i, p_i), saving two vector memory references. The update of r_i may be combined with (r_i, r_i), saving one reference. We can also update the solution only every k steps using the linear combination [VDVo86]

$$x_{i+k} = x_i + a_i p_i + a_{i+1} p_{i+1} + \cdots + a_{i+k-1} p_{i+k-1}$$

This provides a ratio $\frac{(k+1)}{(2k)} \approx \frac{1}{2}$, thus improving data locality, but increases the storage requirements by $k-1$ vectors.

Although the data locality is not good this algorithm is fully vectorizable. Parallelization may be problematic for systems with a large number of processors (e.g. a Hypercube architecture). This is because the inner products may constitute a bottleneck if the interprocessor communication is much slower than the speed of the processors.

The data locality of the modified CG Algorithm 3.0 is much better. The matrix vector product has the same ratio, but the ratios for the vector updates and inner products are 1 and 1/2 respectively. This is because can be performed simultaneously and the two inner products can be combined.

4.3. Implementation of s–CG with Efficient Use of Local Memory

In this section we show how the different parts of s–CG can be implemented efficiently on a vector processor with local memory. Such a system can have either a "register–to–register" (e.g. ALLIANT FX/8) or a "memory–to–memory " (e.g. ETA–10) organization. In the first case the functional units are supplied with operands from the registers, in the second case operands are brought directly from the memory of the system. For CRAY–2 the functional units can directly communicate to the local memory of the processor.

We first discuss the implementation of the matrix product for the 2–D and 3–D model problem, for bordered systems. Second we consider how inner products and linear combinations are implemented.

(i) *Matrix vector products:*

Let a horizontal section of order n be the submatrix $A_k = [C_{k-1}, T_k, C_k]$, $k = 1, \ldots, n$. Also, let $\underline{u_k}$, $\underline{v_k}$ be subvectors (of order n) of u, v corresponding to the block A_k. If the local memory can accommodate simultaneously two full sections of A, seven full subvectors we can carry out the computation

```
$\underline{v_1} = A_1 \underline{u_1}$
Do $k = 1, n-1$
    $\underline{v_{k+1}} = A_{k+1}[\underline{u_k}, \underline{u_{k+1}}, \underline{u_{k+2}}]$
    $\underline{w_k} = A_k[\underline{v_{k-1}}, \underline{v_k}, \underline{v_{k+1}}]$
EndDo
```

Compute $\underline{w}_n$

while keeping the matrix in the local memory. If these blocks do not fit in the memory they must be further sectioned.

This idea can be generalized to do $Au, \ldots, A^s u$ together while A is in the local memory. However, s sections of the matrix and 3s subvectors must fit in the local memory. This can be useful even on a sequential machine when the two levels of memory that must be used efficiently are the main memory and a slow secondary storage device.

The same idea can be applied to the 3-D problem. Here the horizontal sections of order n^2 are: $A_k = [D_k, \overline{T}_k, D_k]$, $1 \leq k \leq n$. For a reasonable resolution without need of secondary storage $n^3 = 10^6$ (because of the main memory limits). So $n^2 = 10^4$ and a good portion of a section can be kept in a local memory of size 16K (CRAY-2, ALLIANT FX/8).

Linear systems of bordered form often arise in the numerical solution of boundary value problems in nonlinear PDEs and the transient analysis of VLSI or other massive circuits to mention just a few. Let us assume that A has the form

$$A = \begin{bmatrix} \tilde{A} & B \\ B^T & G \end{bmatrix}$$

where $\tilde{A}$ is order $N \times N$, B is $N \times k$ and G is a $k \times k$ symmetric matrix; k is a small number. For simplicity assume that $\tilde{A}$ is the matrix of the 2-D model problem. We can then section A in accordance with the sections of $\tilde{A}$ and section B as well. We first perform k inner products to compute the last k entries of v_n, and then the computation proceeds as in the 2–D model problem. We have to keep the last k entries of u_n and v_n in the local memory throughout the computation.

(ii) *Inner Products:*

We must compute 2s inner products involving the vectors $p_i^1, \ldots, p_i^s$ by efficiently using the local memory. We partition the vectors in N/m equal subvectors of length m. The 2s subvectors (of length l) holding the partial results of the inner products must remain in the local memory. Thus $(s*m + 2s*l) \leq$ local memory size. The "DO" loop for all the inner products consists of an outer loop of $\frac{N}{m}$ steps and an inner loop of m steps.

(iii) *Linear Combinations:*

For the linear combinations we partition the vectors $p_i^1, \ldots, p_i^s$, $p_{i-1}^1, \ldots, p_{i-1}^s$ and x_{i-1}, x_i into equal subvectors of length m such that $(2s+2)*m \leq$ local memory size. The "DO" loop for all the linear combinations consists of an outer loop of $\frac{N}{m}$ steps and an inner loop of m steps. By using the matrix notation as in Chapter 3 we can describe this as follows

```
Do  $k = 1, N/m$
        $P_k = P_k + Q_k[\underline{b}^1, \ldots, \underline{b}^s]$
        $(x_{i+1})_k = (x_i)_k + P_k\underline{a}$
EndDo
```

4.4. Implementation of s–CG with Efficient Use of Vector Registers

In this section we will demonstrate how the computations in 5–CG (i.e. s=5) can organized to achieve a speedup ≈ 2 with respect to CG. We assume that each processor in a multiprocessor system has a sufficient number of Vector Registers (VRs). This assumption excludes systems such as the CRAY–X–MP because it has only 8 VRs. The FUJITSU VP–200 system has a total vector capacity of 8K–bytes which can dynamically reconfigured as different sets of varying length vector registers. For example, 32 VRs, each of length 256 and width 64 bits, is one possible arrangement.

The model vector processor we consider has at least 11 VR and one bidirectional port to the memory, one pipelined multiplier (adder). We also assume that vector operations can be going on simultaneously and they take the same number of clock cycles to execute. For simplicity we also assume that they take N cycles for vectors of length N. Finally, we assume that a subvector of operands can be extracted from a vector register. A good example for such a system would be a CRAY–1 with 11 VRs (instead of 8).

Vectorization:

(i) *Matrix vector products:* $r_i = f - Ax_i$, Ar_i, . . . ,$A^4 r_i$, $(A^4 r_i, A^5 r_i)$. We will compute $v = Au$ and $w = Av$ simultaneously keeping A and v local.

In terms of grid lines the idea is to partition the square region into p equal horizontal regions and distribute the section amongst the p processors. The data of one section A_k come from three consecutive lines. If the the grid line is longer than the register length, it is partitioned into segments of length equal to that of the vector registers. Each processor gets segments of data from its region. Segments from four consecutive grid lines to compute $v = Au$ and $w = Av$. So each processor sweeps its horizontal region in a vertical fashion. For the first line of its region each processors will have to do v and Av separately. This is illustrated in Figure 1 for two processors.

We will perform the computation as follows: (1) r_i, Ar_i, (2) $A^2 r_i$, $A^3 r_i$, (3) $A^4 r_i$, $A^4 r_i$,$A^5 r_i$. We will demonstrate how to compute (2) with the least number of memory transfers. Let l be the length of the Vector Register (VR). We use the following notation to denote the contents of a VR.

$$v^{+n} \equiv v(i+n : i+n+l), \quad v^{+} \equiv v(i+1 : i+1+l)$$
$$v^{-n} \equiv v(i-n : i-n+l), \quad v^{-} \equiv v(i-1 : i-1+l)$$
$$v \equiv v(i : i+l)$$

Similarly for a, b, c, u. For w, $w \equiv w(i+1 : i+l-1)$; and we use similar notations for w^{+} , w^{-}, w^{-n}, w^{+}.

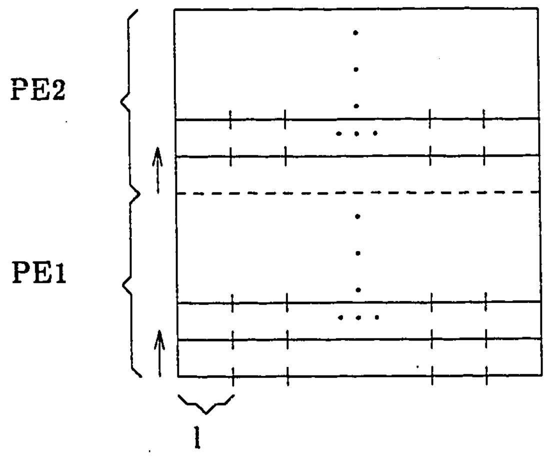

Figure 4.1 Processor assignment to regions of the square grid.

Let us assume that a subvector like $a(i+1 : i+l-1)$ can be extracted from a VR containing $a(i : i+l)$, and that all the subvectors components are 0 for indices $-n+1 : 0$, $n^2 : n^2+n$. The computation of (2) proceeds as follows:

```
$v = c^{-n} * u^{-n} + c * u^{+n} + b^{-} * u^{-} + b * u^{+} + a * u$
[Keep $c^{-}$, $c$, $b^{-}$, $b$, $a$, $v$ in VR ]
$w = c^{-n} * v^{-n} + b^{-} * v^{-} + b * v^{+} + a * v$
[Keep $c$, $v$, $w$, as $c^{-}$, $v^{-}$, $w^{-}$ in VR]
Do $i = l, n^2, l$
    $v = c^{-n} * u^{-n} + c * u^{+n} + b^{-} * u^{-} + b * u^{+} + a * u$
    $w^{-n} = w^{-n} + c^{-n} * v$
    [Keep $c^{-n}$, $c$, $b^{-}$, $b$, $a$, $v$ in VR ]
    [ $v^{-}$, $v^{+}$ are extracted from $v$ ]
    $w = c^{-n} * v^{-n} + b^{-} * v^{-} + b * v^{+} + a * v$
    [Keep $c$, $v$, $w$, as $c^{-n}$, $v^{-n}$, $w^{-n}$ in VR]
EndDo
    $w^{-n} = w^{-n} + c^{-n} * v$
Do $i = 1, n^2/l$
    Compute
    $w(l * i + 1)$
    $w((l-1) * i + 1)$
EndDo
```

The second "DO" loop is executed in vector mode in $(\frac{n^2}{l^2})$ vector steps. This part is only $\frac{1}{l}$ of the total time required to compute (2) and is rather small since $64 \leq l$ in most computers. Thus, we need only concentrate in the work involved in the main "DO" loop.

Computing (2) requires $18N$ flops for v, w and 10 vector transfers. This gives a ratio : $\frac{10}{18} \approx \frac{1}{2}$. We obtain a similar ratio from (1) and (2). We need seven VRs to keep w^{-n}, c^{-n}, c, b^{-}, b, a, v in VRs. During the last multiply and add in computing v two subvectors needed for computing v in the next step are brought in. This requires two additional VRs. Two additional VRs are needed for storing intermediate results (e.g. $c^{-n} * v^{-n}$). So a total of 11 VRs are required.

On our model vector processor $v = A * u$ takes approximately $11N$ cycles. Computing $v$and w for the main "DO" loop above takes approximately $11N$ cycles because vector loads, stores and additions may be completely overlapped with the multiplications. Therefore, on such a processor, two matrix vector products of s–CG take approximately the same time as one in CG.

(ii) *Inner Products:* We choose to group the operations in two sets so that the number of VR needed is not too large.

(1) (r_i,r_i), (r_i,Ar_i), (Ar_i,Ar_i), (A^2r_i,Ar_i), (A^2r_i,A^2r_i),

(2) (A^3r_i,A^2r_i), (A^3r_i,A^3r_i), (A^4r_i,A^3r_i), (A^4r_i,A^4r_i)

Denoting $iuv = iuv(i\colon i+l)$, an inner product is performed as follows:

```
Do $i$=1, $N$, $l$
    $iuv = iuv + u * v$
EndDo
```

Sum the components of iuv

To execute (1) we need 3 VR to store the vectors involved, 5 VR for the intermediate inner product vectors (e.g. iuv), 3 VR for temporary storage. Thus we have a ratio of $\frac{3}{10}$ for (2) and $\frac{3}{8}$ for (1). Hence the ratio for the inner products is $\approx \frac{1}{3}$ provided that there are 11 VRs at our disposal.

On the model vector processor with 11 VR these nine inner products take about $10N$ cycles, whereas a single inner product and a norm take about two and one cycles respectively. Thus 5 consecutive steps of CG require about $15N$ cycles for inner products.

(iii) *Linear Combinations:* We need $p_i^j = A^{j-1} r_i + l^1[p_{i-1}^1, \ldots, p_{i-1}^5]$ for $j = 1, \ldots, 5$ and $x_{i+1} = x_i + l[p_{i-1}^1, \ldots, p_{i-1}^5]$. These 17 vector transfers and $60N$ flops give a ratio of $\frac{17}{60} \approx \frac{1}{3}$. As for CG we can update the solution vector less frequently, improving this ratio to about $\frac{1}{4}$. For example this could be done by

$$x_{i+1} = x_i + l[p_{i-1}^1, \ldots, p_{i-1}^5] + l[p_i^1, \ldots, p_i^5].$$

This does not increase the storage requirements.

We need 5 VRs to store $p_{i-1}^1, \ldots, p_{i-1}^5$, 2 VRs for any two of $p_i^1, \ldots, p_i^5$, 1 VR for x_i, and 3 VR for temporary storage. On our model vector processor the computation can be performed in about $30N$ cycles. This is because the vector loads, stores and additions completely overlap with the multiplications. We note that a vector update can be computed in $3N$ cycles. Thus for 5 consecutive steps of CG vector updates require $45N$ cycles.

Parallelization:

Parallelization for (i) is realized by partitioning the matrix into p equal horizontal sections, assigning each to one of p processors. If $\frac{N}{p}$ is integer then each section consists of entire blocks of order n and the communication needed between every 2 processors working on adjacent sections will be a subvector of v of dimension n. Since this will be formed initially by the processor whose starting point is this vector this causes no delay.

In cases (ii) and (iii) the vectors are divided into p equal subvectors and distributed to the p processors.

4.5. Comparison of s–CG with CG on a CRAY–1–like architecture

In both CG and s–CG the data locality can be improved by combining some of the 3 types of operations. For example matrix vector multiplications and inner products in s–CG can be executed simultaneously. Here we consider them separately because it is easier to compare. The number of flops and the critical ratios for 5–CG and CG are shown on Table 4.1 .

Assume for a vector system the time for a vector transfer equals the vector multiplication (addition) times a factor $1 \leq \alpha$. Also, assume one port for vector transfers from the memory to the VRs.

Vector Operation	5–CG/flops	5–CG/ratio	CG/flops	CG/ratio
Inner Products	10N	1/3	10N	3/4
Matrix Vector Products	54N	10/18	45N	11/9
Linear Comb.	60N	17/60	30N	3/2

Table 4.1 Flops for 5 steps of CG (1 for 5–CG) and (mem. transfers)/flops.

When $\alpha = 1$ and an architecture model like the one described in the previous section is adopted we obtain a speedup of 1.5. This is because the execution of one step of s–CG requires approximately $33 + 30 + 10 = 73$N cycles compared to $55 + 45 + 15 = 110$N cycles for 5 consecutive steps of CG.

If $\alpha > 1$ then the vector transfers essentially overlap with almost all the vector operations, and so the execution time is equal to the "vector transfer time ". The number of vector transfers for 1 step of 5–CG is $17 + 6 + 30 = 53$ and for 5 steps of CG $45 + 15 + 55 = 115$. Thus, in this case, 5–CG will run twice as fast as CG on the model problem.

4.6. Implementation of the Preconditioned s–CG

Here only the implementation of the matrix vector products and the inner products are affected. Again we assume s=5. We need to form the following matrix vector products:

$$Kr_i,\ (KA)Kr_i,\ (KA)^2Kr_i,\ (KA)^3Kr_i,\ (KA)^4Kr_i,\ A(KA)^4Kr_i$$

and the inner products

$$(r_i,Kr_i),\ (AKr_i,Kr_i),\ ((AK)^2r_i,Kr_i)\ ,\ \ldots\ ,\ ((AK)^4r_i,(KA)^4Kr_i),\ ((AK)^5r_i,(KA)^4Kr_i).$$

Note that the ratio for executing the inner products is about $\frac{1}{2}$ because ten vector references are needed. This is about half the ratio for the inner products of the preconditioned CG. There four vectors are transferred to form two inner products, yielding a ratio of 1 . For vectorized ICCG the data locality is poor. For example, the ratio for the forward step

$$u = y_j + F_j z_{j-1}$$

is 2. Starvation of the processors can occur if the memory ports are slow. For s–CG the matrix vector multiplication by A can be combined with the forward of the multiplication by K, thus improving data locality and increasing the speed on architectures with memory hierarchy. However the backward step cannot be combined and the bad data locality remains for this part.

If PICCG is used then

$$z_i = (I - E - F)^{-1} r_i$$

where the inverse is approximated by the matrix

$$I - (E + F) + (E + F)^2$$

Since the entries of this matrix are those of A we can organize the computation so that multiplications by K and A can be performed while the data remain in the local storage.

For polynomial preconditioning the multiplication by A can be combined with the last multiplication by the last factor $(A - \rho I)$ of the $K = q(A)$ (see 2.9) in a way similar to the implementation above. Also, the multiplication of two factors can be combined to give the same data locality as in the plain s–CG. Parallelization for the multiplication by K is similar to multiplication by A.

4.7. Implementation of s–CG on Message Passing Architectures

We adopt the Hypercube architecture as our model. A similar discussion can be carried out for any ensemble architecture system (e.g. the ring architecture). Each node of the system can be a scalar (or vector) processor for computations and it has its own memory. It must also have a processor which sends and receives messages to neighbor nodes. This processor can coincide with the processor dedicated for computations (e.g. INTEL iPSC). The time required for a message package of size k bits to be communicated between two adjacent nodes is

$$\omega = \alpha + \beta\, k$$

This is because the network requires a startup time α and then using a pipeline it can deliver one bit per β seconds. Assume that there are p nodes in the system. We can distinguish two types of communications needed to carry out computations. The local communication involves only $m \ll p$ neighboring nodes and the global one involves $m = p$ nodes. The time required for a message to be communicated globally is

$$\omega \log_2(p)$$

CG and can be implemented on such a system by dividing all vectors in p equal subvectors (and the matrix A into p horizontal sections) and storing them at the p nodes of the system. The vector updates and linear combinations require no communication of the nodes. The scalars computed in s–CG require the moments of the residual vector. So they do not introduce any communication of the nodes. We now consider separately the inner products and the matrix vector products.

(i) *Inner products:* An inner product requires global communication of the nodes in order to sum up the vector product and communicate the result to all the processors. Denote by t_a the time to perform a scalar addition on a single processor. Assume that the time to transmit a double precision number to a neighbor is greater than the time for scalar addition. This is a reasonable assumption. For example this is valid for the INTEL iPSC Hypercube machine.

$$t_a < \omega \, (= \alpha + \beta \, 64 \,)$$

One way to perform an inner product so that the result remains in all processors is the following. Assume that the Hypercube is a square with nodes a_1, a_2, a_3, a_4 (Fig. 4.2) and we have to perform

$$a_1 + a_2 + a_3 + a_4$$

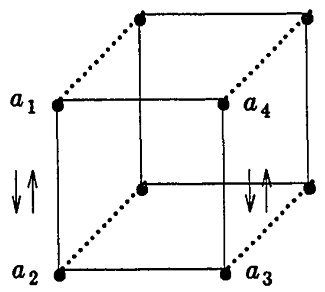

Figure 4.2 Inner Products on the Hypercube

Firstly $a_1 + a_2$ and $a_3 + a_4$ are computed at nodes a_1 and a_3 respectively and transmitted to nodes a_2 and a_4 for time equal to $2\omega + t_a$. Secondly these results can be added to obtain the final result in all four nodes for total time equal to $4\omega + 2t_a$. For $p = 2^k$ it takes time equal to

$$\log_2(p)(2\omega + t_a)$$

A more expensive way is to collect all numbers to a single node for the addition and then transmit the result to all nodes. For $p = 4$ it takes time equal to $6\omega + 3t_a$.

Performing $2s$ inner products simultaneously requires time equal to

$$\log_2(p)(2\tilde{\omega} + \tilde{t}_a)$$

where $\tilde{\omega} = \alpha + 2s(64)\beta$ and $\tilde{t}_a$ is the time to add two vectors of length $2s$. If we assume that every node has a separate processor for passing messages, then $\tilde{t}_a$ overlaps with $2\tilde{\omega}$. This is because of the assumption that the time for scalar addition is smaller than the time to send a number to a neighbor node. Neglecting the time spent for the multiplication part of the inner products we obtain the following speedup (by performing $2s$ inner product versus 1)

$$\frac{2s(2\omega + t_a)}{2\tilde{\omega}} \approx \frac{2s(\alpha + 64\beta)}{\alpha + 2s(64)\beta}.$$

When the transmission startup time α is much greater than $2s(64)\beta$ the speedup is of order $2s$. To this gain we should add the loss of speed because the processors (if the nodes have vector processors) during $2s$ global communications (in CG) versus one (in s–CG).

For the INTEL iPSC Hypercube machine $\alpha \approx 1831\ 10^{-6}$ and $\beta \approx (3/8)10^{-6}$ if the time is measured in *sec*. Also, the double precision addition takes 43.0 $10^{-6}sec$ and $\omega = 19\ 10^{-3}sec$. This essentially means that for this machine the global communication may dominate the whole computation in the CG iteration. If this is true then the speedup obtained by using s-CG can be of order $2s$.

In Tables 4.2 and 4.3 we show the transmission and flop times for the INTEL iPSC Hypercube [GrRe86].

Bytes	Time/sec	Bytes	Time/sec
2	0.001837	128	0.002040
4	0.001831	256	0.002444
8	0.001849	512	0.002763
16	0.001853	1024	0.003658
32	0.001900	2048	0.007148
64	0.001954	4096	0.013531

Table 4.2 Transmission times of a message (in Bytes) between adjacent nodes for INTEL iPSC.

(ii) *Matrix Vector Products:* Consider the 2–D model problem first. The matrix A is partition into p horizontal sections. Each section is stored in the private memory of a node. Assume for simplicity that $N = p^2$. Then each section

$$A_k = [C_{k-1}, T_k, C_k], \quad k = 1, \ldots, p$$

Precision	$(*)/\mu sec$	Mflops	$(+)/\mu sec$	Mflops
Single	40.4	.024	39.5	.025
Double	43.5	.023	43.0	.023

Table 4.3 Operations times and speed for INTEL iPSC.

is a full block of the matrix A. A certain numbering of the nodes must followed so that adjacent sections are stored in adjacent nodes. The multiplications

$$r_i = f - Ax_i \ , \ \ldots \ , \ A^s r_i$$

are carried out sequentially in each node. To compute the subvector

$$\underline{v}_k = A_k [\underline{u}_{k-1}, \underline{u}_k, \underline{u}_{k+1}]$$

the subvectors $\underline{u}_{k-1}$ and $\underline{u}_{k+1}$ must be transferred to the node holding the section A_k from neighbor nodes. Therefore to carry out one matrix vector multiplication every node must send one subvector to two neighbors and must receive two subvectors. Therefore only local communication of the processors is necessary to carry out this part of the computation.

Assume for the 3–D problem that $p = n$ then n horizontal sections of order n^2

$$A_k = [D_k, \bar{T}_k, D_k], \quad 1 \leq k \leq n$$

must be stored each in each of the p nodes. The matrix vector multiplications can be performed similarly to the 2–D problem. The communication still consists of sending one subvector to two neighbors and receiving two subvectors of order n^2.

Next we present an example of matrix which requires global communication to carry out a matrix vector multiplication. Consider the bordered matrix

$$\begin{bmatrix} \tilde{A} & B \\ B^T & G \end{bmatrix}$$

of section 4.3 . The matrix B^T can be in column form

$$B^T = [b_1, \ldots, b_n]$$

where the column vectors b_i, $1 \leq i \leq n$ are of dimension k. If only a small (compared to the size of n) number of these short vectors are nonzero, then we can still carry out the matrix vector multiplication with local communication only. Otherwise global communication is necessary. Let us call the latter matrix fully bordered.

We remark that it seems intuitively unlikely that in modeling a physical phenomenon the matrix of the model (no transformations applied to) is fully bordered. This is because fully bordered essentially means that all almost all the nodes of the discrete model interact directly with a small set of nodes.

Assume that the bordered matrix has the form

$$A = \begin{bmatrix} \tilde{A} & C \\ B^T & G \end{bmatrix}$$

where C is $N \times k$ matrix of zero entries and $\tilde{A}$, B, G as above. We can the gain by applying a nonsymmetric s-step method on the this problem versus a one-step method. This is because the matrix vector products $Au, \ldots, A^{(s+1)}u$ can be carried out in two steps. Firstly the computations

$$\tilde{A}\tilde{u}, \ldots, \tilde{A}^{(s+1)}\tilde{u}$$

are carried out. Then the multiplication by matrix $[B^T\ G]$ constitutes k inner products. All $(s+1)k$ inner products needed for $(s+1)$ matrix vector products and the $2s$ inner products required for the iteration of the s-step method can be performed simultaneously. This eliminates completely the need for global communication for carrying out the matrix vector products. This because it combines it with the required global communication due to the s-step iteration. The need for global communication for the full bordered system disappears if we eliminate the entries of B^T. This is feasible if the lowest subdiagonal of $\tilde{A}$ has all nonzero elements. This would create no fill-ins in the matrix besides the $k \times k$ submatrix G.

CHAPTER 5.

NUMERICAL EXPERIMENTS ON THE ALLIANT FX/8 MULTIPROCESSOR

5.1. Introduction In this chapter we measure the performance of the Conjugate Gradient, the Conjugate Residual, the Vectorized Incomplete Cholesky Preconditioned CG, their 5-steps counterparts, and finally the Parallelized Incomplete Cholesky Preconditioned CG on a multiprocessor system. Although the test problems do not necessarily represent the general case of the model problem these measurements are still be valid because the matrix vector multiplication simulated the general five-point operator matrix vector product.

A description of the multiprocessor system is given. All codes were written in Fortran. The implementation on a system with local memory given in Chapter 4 is followed for the inner products and the linear combinations. The simultaneous implementation of more than one matrix vector multiplications is not carried out. This is because tests showed that the matrix vector multiplication is so slow (even when all the data lie in the local memory). Consequently these data could be brought in directly from the main memory without further lowering the rate of the computation. Finally the vector register implementation in Chapter 4 is not possible because there is not a sufficient vector registers in the system.

The results obtained show that the s–step methods are up to fifty percent faster than the one–step methods. However, the inner products and the linear combinations are one hundred percent faster for the s–step methods compared to the one–step methods. The matrix vector multiplication (even when the matrix is in the local memory) is slow and this reduces the speedups obtained for the inner products and linear combinations. If the matrix consists of groups of constant entries (e.g. the Laplace operator) then the speedup would be about one hundred percent.

In Section 5.2 w describe the shared memory system that was used to run the experiments. In Section 5.3 the computational rates for the matrix vector multiplication, the inner products, and the linear combinations are presented and discussed. The results of the experiments are contained in Section 5.4 .

5.2. The Experimental Environment

The experiments were conducted on the ALLIANT FX/8 multiprocessor system at the Center for Supercomputing Research and Development of the University of Illinois.

The FX/8 is an example of a supercomputer architecture with memory hierarchy. The configuration of the FX/8 contains 8 Computational Elements (CEs), which communicate to each other via a concurrency control bus used as a synchronization device. Each CE has a computational clock cycle of 170 ns . The maximum performance of one CE is 11 Mflops (million flops/sec) for single precision and 5.9

Mflops for double precision computations. Thus when the 8 CEs run concurrently the peak performance can reach 47.2 Mflops. Each CE is connected via a crossbar switch to a shared cache of 16K (64 bit) words, implemented in four quadrants. This connection is interleaved and provides a peak bandwidth of 47.12 MW/sec. The cache is connected to an 8 MW interleaved global memory via a bus with a bandwidth of about 23.5 MW/sec for sequential read and about 19 MW/sec for sequential write access. The system also has 6 interactive processors (IPs) used for operating system related functions and I/O operations.

Each CE is a pipelined vector processor with 8 64-bit vector registers of length 32, 8 64-bit scalar registers, as well as 8 address registers. Operands for vector instructions come from vector registers or vector and scalar registers. The vector multiply and add instructions can be overlapped. It is worth noting that the vector multiply takes 2 cycles per element while the vector add/subtract/convert take 1 cycle and division take 8 cycles. Multiprocessing is realized by concurrency instructions which permit a loop to be executed concurrently across more than one processors in an interleaved mode or by concurrent call of a subroutine which assign one task to each CE.

The ALLIANT FX/8 optimizer and compiler restructures a FORTRAN code based on data dependency analysis for scalar, vector, and concurrent execution. A FORTRAN program can execute in one of the following modes: scalar, vector, scalar concurrent, vector-concurrent, or concurrent-outer /vector inner. We illustrate the modes of execution on the following loop:

```
DO 1 I = 1, N
1     A(I) = A(I) + S
```

For N = 8192 we have,

Scalar: $A(1), A(2), \ldots, A(8192)$

Vector: $A(1{:}32), A(33{:}64), \ldots, A(8161{:}8192)$

Concurrent:

$CE_{1:}$ $A(1), A(9), \ldots, A(8185)$

$CE_{2:}$ $A(2), A(10), \ldots, A(8186)$

. . .

$CE_{8:}$ $A(8), A(16), \ldots, A(8192)$

Vector Concurrent:

$CE_{1:}$ $A(1{:}249{:}8), \ldots, A(7937{:}8185{:}8)$

$CE_{2:}$ $A(2{:}250{:}8), \ldots, A(7938{:}8186{:}8)$

. . .

$CE_{8:}$ $A(1{:}256{:}8), \ldots, A(7944{:}8192{:}8)$

Concurrent Outer/Vector Inner:

$CE_{1:}$ $A(1{:}32), \ldots, A(992{:}1024)$

$CE_{2:}$ $A(1025{:}1056), \ldots, A(2017{:}2048)$

. . .

$CE_{8:}$ $A(7169{:}7200), \ldots, A(8161{:}8192)$

There is a timing facility which is accessible via a FORTRAN subroutine call and seems to give stable results with a resolution of $10^{-1} sec$ measurements. The computational rates which will be presented were run with a large serial (non-concurrent) outer loop in order to obtain reliable timing data. To do time measurements the programs were run more than three times. Although the execution was not in single user mode no other sizable jobs were running at the same time, and the timing variations were of the order of one percent.

5.3. Computational rates of CG and s–CG parts

Vector Operation	5–CG	CG
Vector Update	–	5.5 Mflops
Inner Products	15 Mflops	8 Mflops
Vector Norm	–	14 Mflops
Matrix Vector Products (A)	8 Mflops	8 Mflops
Matrix.Vector Products (K)	7 Mflops	7 Mflops
Linear Comb.	22 Mflops	-

Table 5.1 . Computational rates for the 5–CG and CG parts.

All the rates cited are for vector lengths between 10^4 and 10^5. The matrix vector product runs at ≈ 14 Mflops from the cache. Thus, for the two matrix products Av and A^2v, one can get a rate slightly higher than 8 Mflops if the vectors exceed the cache and proper management of the cache is done in a way similar to the implementation of 5–CG in Chapter 4 . The low rate of the matrix product (even from the cache) is mainly due to the low rate of the vector product operation. This part involving half the flops of CG applied to the model problem has not been sped up and it is a slow part. Thus, the speedup of 5–CG compared to CG is not expected to be very good, taking into account the fact that there is an extra matrix vector product (in 5–CG) per five steps of CG.

It is worth noting that the linear combinations run at 16 Mflops if Vector Concurrent mode is used and at 22 Mflops if Concurrent outer/Vector inner mode is used. Since there are twice as many flops in the linear combinations as in the vector updates and (the rate for the linear combinations is four times that of the vector updates) we have sped up this part by a factor of 2.

The inner products were computed by assigning each one of 8 inner products to one processor and computing each one of the 2 remaining norms separately. If all 10 inner products are carried out together in Concurrent outer/vector in mode then this may yield a higher rate. It is worth noting that there is a primitive function DOTPRODUCT, which is designed to compute only one inner product at time in vector mode on a single processor, or Vector Concurrent mode. The inner products rate may have been higher if two inner products were computed by keeping the same data in the vector registers as described in the implementation part of Chapter 4 . Since the rate for the two inner products for CG is 11 Mflops we have sped up this part only about fifty percent.

5.4. Test Problems We include test results on two problems with the matrix being the Laplace operator. The first problem is a discretized PDE with known solution. The second problem is a linear system. The matrix was stored in three diagonals of order N to simulate the general five-point difference operator.

Problem 1 : $-(au_x)_x - (bu_y)_y = g$ on the unit square with homogeneous boundary conditions and $a \equiv b \equiv 1$ and $u(x, y) = e^{xy} sin(\pi x) sin(\pi y)$.

Problem 2 : The linear system $Ax = f$ where A is the pentadiagonal matrix of problem 1 and $x_i = \sqrt{i}$.

The termination criterion used for preconditioned CG was $(r_i, Kr_i)^{1/2} < 10^{-6}$ and $(r_i, r_i)^{1/2} < 10^{-6}$ for all the other cases. The significance of the tests is the following:

We can compute the speedup factor for the various methods from the results in Tables 5.2-5.7. These factors are

$$CG/5\text{–}CG \approx 1.3, \quad CR/5\text{–}CR \approx 1.5, \quad ICCG/5\text{–}ICCG \approx 1.15 .$$

The matrix vector multiply by K has a rate of about 7 Mflops and 12N flops and in the preconditioned CG the slow parts involve about seventy percent of the total number of flops and this account for the drop in the speedup factor. For CR one extra vector update is required per step while 5–CG and 5–CR require the same number of operations (if the residual norm is not computed in CR).

Table 5.7 shows the performance of the proposed Parallel ICCG method (PICCG). Compared to the CG method (both number of steps and execution times) we conclude that it is a reasonably good preconditioner. The performance of the vectorizable ICCG is far better mainly because PICCG involves 16N flops (to compute Kv) compared to 12N for ICCG. Also, block parallelization is sufficient for 8 CE with vector register length 32 (e.g. if $N = 256^2$) then there is a complete set of data for all CEs. Finally interprocessor communication is not a problem because of the shared cache and the fact that multiplication by the preconditioner K is slow (7 Mflops).

	Problem 1		Problem 2	
$\sqrt{N}$	Steps	Time/sec	Steps	Time/sec
64	136	1.08	196	1.5
100	209	4.66	307	7.2
128	266	10.21	395	15.66
160	331	21.26	496	32.28
200	412	41.39	621	62.26
256	525	92.2	797	140.51
300	613	145.01	936	221.22

Table 5.2. Execution times for the CG on Problem 1 and Problem 2

	Problem 1		Problem 2	
$\sqrt{N}$	Steps	Time/sec	Steps	Time/sec
64	27	1.1	39	1.58
100	42	4.24	62	6.13
128	53	8.68	79	13.19
160	66	16.92	99	25.1
200	83	32.91	124	49.26
256	107	70.48	160	105.62
300	123	110.98	187	168.15

Table 5.3. Execution times for the 5-CG Problem 1 and Problem 2

	Problem 1		Problem 2	
$\sqrt{N}$	Steps	Time/sec	Steps	Time/sec
64	52	1.08	75	1.45
100	72	3.44	115	5.18
128	90	7.17	147	11.37
160	111	14.02	182	22.56
200	137	25.32	217	39.14
256	174	52.26	276	85.23
300	203	84.62	324	133.8

Table 5.4. Execution times for the ICCG on Problem 1 and Problem 2

	Problem 1		Problem 2	
$\sqrt{N}$	Steps	Time/sec	Steps	Time/sec
64	11	1.27	16	1.54
100	15	3.4	23	5.45
128	18	6.58	30	10.47
160	22	11.64	37	19.43
200	28	22.15	44	34.98
256	35	45.74	55	72.2
300	41	75.55	65	120.01

Table 5.5. Execution times for the 5–ICCG on Problem 1 and Problem 2

$\sqrt{N}$	Steps	Time/sec	Steps	Time/sec
64	133	1.3	28	1.2
100	201	5.66	40	4.17
128	252	12.24	52	8.95
160	307	23.63	62	16.59
200	376	46.6	76	31.54
256	471	97.8	94	64.07
300	544	158.8	110	103.17

Table 5.6. Execution times for the CR and 5–CR for Problem 1.

	Problem 1		Problem 2	
$\sqrt{N}$	Steps	Time/sec	Steps	Time/sec
64	56	.99	85	1.61
100	85	4.31	132	6.85
128	107	9.29	160	14.18
160	133	18.4	198	27.97
200	165	36.07	247	55.5
256	209	75.39	316	115.3
300	245	122.96	371	189

Table 5.7. Execution times for the PICCG Problem 1 and Problem 2

CHAPTER 6.

s–STEP ITERATIVE METHODS FOR NONSYMMETRIC LINEAR SYSTEMS

6.1. Introduction Consider the linear system of equations

$$Ax = f$$

where A is a nonsymmetric matrix of order N with symmetric part $M = \frac{(A + A^T)}{2}$ being positive definite. In this chapter we review some generalizations of CR which can be used to solve this system. We then derive s–step iterative methods similar to the methods for the symmetric problem. These methods are shown to converge. However, we offer no tests to guarantee that they are stable.

In Section 6.2 we review the Generalized Conjugate Residual (GCR) and Orthomin(k) methods. In Section 6.3 we introduce the s–step Generalized Conjugate Residual (s–GCR) and the s–step Orthomin(k) (s–Orthomin(k)) methods. In Section 6.4 we show the convergence of the new s–step methods. In Section 6.5 we discuss the work and storage requirements for the new s–step methods. It turns out that there is a modest increase in the storage but they require less work than the the one–step methods. Thus these methods may be useful even for sequential computing if proved to be stable. Orthodir is a variant of GCR that converges for general matrices. In Section 6.6 we introduce the s–step Orthodir and Orthodir(k).

6.2. Generalizations of the Conjugate Residual Method CR applied to the SPD problem minimizes $\|r_{i+1}\|$ along the direction p_i in order to determine a_i in

$$x_{i+1} = x_i + a_i p_i .$$

Also, p_i is made $A^T A$-orthogonal to p_{i-1}. Symmetry is used to obtain

$$(Ap_i, Ap_j) = 0, \text{ for } i \neq j.$$

Positive definiteness is necessary to guarantee that $a_i = \dfrac{(r_i, Ar_i)}{(Ap_i, Ap_i)}$ is positive and so there is progress towards the solution in every step. The orthogonality and the norm reducing property of CR guarantee its convergence in at most N iterations.

If A is nonsymmetric but definite then the norm reducing property of CR is still valid but the orthogonality only holds locally. That is p_i is guaranteed to be $A^T A$-orthogonal only to p_{i-1}. This shortcoming is ameliorated in some of the generalizations of CR.

Algorithm 6.1: Generalization of CR

```
Initial guess $x_0$
Compute $p_0 = r_0 = f - Ax_0$
For $i = 0$ Until Convergence Do
    $a_i = \dfrac{(r_i, Ar_i)}{(Ap_i, Ap_i)}$
    $x_{i+1} = x_i + a_i p_i$
    $r_{i+1} = r_i - a_i Ap_i$
```

```
        Compute $p_{i+1}$, $Ap_{i+1}$
EndFor.
```

Since $(r_i, Ar_i) \geq 0$ the norms of the residuals is a decreasing sequence. The direction vectors must be constructed to significantly reduce the norm at each step.

(i) Generalized Conjugate Residual Method (GCR) :

$$p_{i+1} = r_{i+1} + \sum_{j=0}^{i} b_j^i p_j$$

$$b_j^i = -\frac{(Ar_{i+1}, Ap_j)}{(Ap_j, Ap_j)}, \quad j \leq i.$$

Here $\|r_{i+1}\|_2$ is minimized over $x_0 + \{r_0, Ar_0, \ldots, A^i r_0\}$. GCR gives the exact solution in at most N iterations. However, if more than a few iterations are needed then the storage requirements become prohibitive. To circumvent this GCR can restart periodically. This method is called GCR(k). An alternative is to orthogonalize to over k directions. This gives Orthomin(k).

(ii) Orthomin(k):

$$p_{i+1} = r_{i+1} + \sum_{j=i-k+1}^{i} b_j^i p_j$$

where the $\{b_j^i\}$ are defined as in (i). Both methods coincide with CR in the symmetric case. Note that CR applied to the nonsymmetric problem is Orthomin(1). Orthomin(0) is a one dimensional steepest descent method called sometimes Minimal Residual Method (MR).

In both (i) and (ii) we need to compute the Ap_{i+1}. This can be done either directly or via the recursion

$$Ap_{i+1} = Ar_{i+1} + \sum_{j=j_i}^{i} b_j^i Ap_j \tag{6.1}$$

where $j_i = 0$ for CGR and $j_i = \max(0, i-k+1)$ for Orthomin(k). Assuming (6.1) is used, the work and storage for the two methods on a sequential machine is shown in Table 1 .

Note that storage is required for x_i, r_i, Ar_i, $\{p_j\}_{j=j_i}^{i+1}$ and $\{Ap_j\}_{j=j_i}^{i+1}$ and may be A.

6.3. The s–step GCR and s–step Orthomin(k)

In this section we first present a modification of the GCR or Orthomin(k) method. In this method the directions are A^TA-orthogonal as in the GCR or Orthomin(k). However there is look–ahead because two new directions are formed and then they are orthogonalized. However this requires that some inner products are computed from others and could be an unstable method. We then derive an s–step method for GCR and Orthomin(k) that does not require s directions to be A^TA–orthogonal to each other.

Algorithm 6.2: The Modified (GCR) Orthomin(k) Method.

Per Loop	GCR	Orthomin(k)
Work	(6(i+1)+8)N + 1 MatVec	(6k+8)N + 1 MatVec
Storage	2(i+2)+2	(2k+3)

Table 6.1 . Work and Storage for GCR and Orthomin(k).

```
Initial guess $x_0$
Compute $p_0 = r_0 = f - Ax_0$, $Ar_0$, $A^2 r_0$
Set $v = r_0$, $w = Ar_0$
$A^T A$-orthogonalize to get $p_1$, $p_2$, $Ap_1$, $Ap_2$
For $i = 0$ Until Convergence Do
    $a_i^1 = \frac{(r_{2i}, Ap_{2i})}{(Ap_{2i}, Ap_{2i})}$
    $a_i^2 = \frac{(Ar_{2i}, Ap_{2i+1})}{(Ap_{2i+1}, Ap_{2i+1})}$
    $x_{i+1} = x_i + a_i^1 p_i^1 + a_i^2 p_i^1$
    $r_{i+1} = r_i - a_i^1 p_i^1 - a_i^2 p_i^1$
    $v = r_{i+1} + \sum_{j=j_{(1,i)}}^{2i+1} b_j^{(1,i)} p_j$
    $w = Ar_{i+1} + \sum_{j=j_{(2,i)}}^{2i+1} b_j^{(2,i)} p_j$
    Compute $Av$, $Aw$
    $b_j^{(1,i)} = -\frac{(Ar_{i+1}, Ap_j)}{(Ap_j, Ap_j)}, \quad j = j_{(1,i)}, \ldots, 2i+1$
    $b_j^{(2,i)} = -\frac{(A^2 r_{i+1}, Ap_j)}{(Ap_j, Ap_j)}, \quad j = j_{(2,i)}, \ldots, 2i+1$
    $A^T A$-orthogonalize $v$, $w$ to get $p_{2i+2}$, $p_{2i+3}$
EndFor.
```

To $A^T A$-orthogonalize the vectors v and w we need the inner products (Av, Aw), (Av, Av) and (Aw, Aw). These in turn require

(Ar_{i+1}, Ar_{i+1}), (Ar_{i+1}, A^2r_{i+1}), (A^2r_{i+1}, A^2r_{i+1}), $\{(Ar_{i+1}, Ap_j), (A^2r_{i+1}, Ap_j)\}_{i_j}^{2i+1}$, and $b_j^{(1,i)}$, $b_j^{(2,i)}$.

If we set $i1 = 2i+2$, $i2 = 2i+3$, then we get the two new direction vectors

$$p_{i1} = v$$

$$p_{i2} = w - \frac{(Av, Aw)}{(Av, Av)} v$$

We can also derive the inner products

$$(Ap_{i1}, Ap_{i1}), (Ap_{i2}, Ap_{i2}), (r_{i1}, Ap_{i1}), (Ar_{i2}, Ap_{i2})$$

from

$$(Av, Aw), (Av, Av), (Aw, Aw), (r_{i1}, Av), (Ar_{i1}, Av), (Ar_{i1}, Aw).$$

Next we will present for both s–step GCR and s–step Orthomin(k) in one algorithm. The following definitions are similar to those in chapter 3 .

Definition 6.1: (1) Let us denote by $M_i = [(Ap_i^j, Ap_i^l)]$, where $1 \le j, l \le s$

(2) $\underline{a_i} = [a_i^1, \ldots, a_i^s]^T$ be the steplengths in updating x_i and $\underline{m_i} = [(r_i, Ap_i^1), \ldots, (r_i, Ap_i^s]^T$.

(3) $\underline{c_j^i} = [(Ar_{i+1}, Ap_j^1), \ldots, (A^s r_{i+1}, Ap_j^s)]^T$, for $j = j_i, \ldots, i$.

(4) $P_i = [p_i^1, \ldots, p_i^s]$ and $\underline{b_j^i}$ the constants in updating P.

(5) $R_i = [r_i, Ar_i, \ldots, A^{s-1}r_i]$

The following linear systems of order s must be solved in executing one step of s–step GCR (Orthomin(k)):

$$M_i \underline{a_i} - \underline{m_i} = 0$$

$$M_j \underline{b_j^i} + \underline{c_j^i} = 0, \text{ for } j = j_i, \ldots, i$$

Note that M_i is invertible iff $p_i^1, \ldots, p_i^s$ are linearly independent. This follows from the fact that the bilinear form $(A^T A \cdot, \cdot)$ is an inner product.

Algorithm 6.3 The s-step GCR, Orthomin(k) algorithm.

```
Choose $x_0$
Compute
$P = [r_0 = f - Ax_0, Ar_0, \ldots, A^{s-1} r_0]$
For i=0 Until Convergence Do
        Compute $\underline{m_j}$, $M_i$
        Call Scalar1
        $x_{i+1} = x_i + P_i \underline{a_j}$
        $r_{i+1} = r_i - AP_i \underline{a_j}$
        Compute $\underline{c_j^i}$, $j = j_i, \ldots, i$
        Call Scalar2
        $P_{i+1} = R_{i+1} + \sum_{j=j_i}^{i} P_j \underline{b_j^i}$
        Compute $AP_{i+1}$ or,
        $AP_{i+1} = AR_{i+1} + \sum_{j=j_i}^{i} AP_j \underline{b_j^i}$
EndFor
Scalar1 : Decomposes $M_i$ and solves $M_i \underline{a_j} = \underline{m_j}$
Scalar2 : Solves $M_j \underline{b_j^i} = -\underline{c_j^i}$, for $j = j_i, \ldots, i$.
```

where $j_i = 0, i-k+1$ for s–step GCR and s–step Orthomin(k) respectively. Notice that in s–step Orthomin(0) s directions are used to improve the solution. The s–step Orthomin(0) is the s–step MR method. The s–step Orthomin(1) method coincides with s–CR for A SPD. In general we will have $k > s$, because s must be small for stability as in the SPD case. However, all cases are interesting and they may yield methods useful even for sequential processing.

6.4. Convergence of s–step GCR and s–step Orthomin(k)

In this section we give convergence proofs for the s–step methods and discuss their relation to their one–step counterparts.

Theorem 6.3: Assume that the degree of the minimal polynomial r_0 is greater than $s*i$. The solution vectors x_i and the planes R_i, P_i generated by s–GCR satisfy the following relations:

(i) P_i is A^TA-orthogonal to P_j, for $i \neq j$

(ii) r_i is orthogonal to AP_j, for $i > j$

(iii) $(r_i, Ap_i^l) = (r_i, A^l r_i)$, for $l = 1, \ldots, s$

(iv) r_i is orthogonal to AR_j, for $i > j$

(v) AP_i is orthogonal to AR_j, for $i > j$

(vi) $(Ap_i^l, Ap_i^j) = (Ap_i^l, A^j r_i)$, for $1 \leq l,j \leq s$

(vii) $(r_j, Ap_i^l) = (r_0, Ap_i^l)$, for $j \leq i$ and $1 \leq l \leq s$.

(viii) $\{R_0, R_1, \ldots, R_i\} = \{P_0, P_1, \ldots, P_i\} = \{r_0, Ar_0, \ldots, A^{(i+1)s-1}r_0\}$

(ix) If $r_i \neq 0$, then $a_{i-1}^s \neq 0$.

(x) x_{i+1} minimizes $\|r_{i+1}\|$ over the translated subspace $x_0 + \{P_0, \ldots, P_i\}$.

Proof: By the the definition of the direction planes P_i we get (i). From $r_i = r_{i-1} - AP_{i-1}\underline{a}_{i-1}$ and (i) we get (ii) by induction. The defining relations for $\{p_i^1, \ldots, p_i^s$ and (ii) give (iii).

To prove (iv) we rewrite the defining identity for AP_j

$$AR_j = AP_j - l\{AP_{j-1}, \ldots, AP_0\}$$

where $l\{\ \}$ is a linear combination of the planes involved. We then obtain (iv) by use of (ii). The same equation and (i) gives (v). We show (vi) from the definition of p_i^j, $j = 1,\ldots,s$ and (i). The identity $r_j = r_{j-1} - AP_{j-1}\underline{a}_{j-1}$ and induction give (vii).

To prove (viii) we note that the Krylov subspace contains the other two sets. Also, it is easy to check that $\{P_0, \ldots, P_i\}$ is contained in $\{R_0, \ldots, R_i\}$ because every direction can be written in terms of $A^l r_i$, $l = 1, \ldots, s$. By (i) the dimension of $\{P_0, \ldots, P_i\} = (i+1)s$, which is the dimension of the Krylov subspace. Therefore all the subspaces are equal.

(ix) states that the new (nonzero) residual lifts the iteration out of the current Krylov subspace. Then the assumption on the degree of the minimal polynomial of r_0 proves that the directions $\{p_i^1, \ldots, p_i^s\}$ are independent.

Since the symmetric part M of A is positive definite

$$(r_{i-1}, Ap_{i-1}^1) = (r_{i-1}, Ar_{i-1}) = (r_{i-1}, Mr_{i-1}) > 0$$

Thus the system $M_{i-1}\underline{a}_{i-1} = m_{i-1}$ has a nontrivial solution. Now if $a_{i-1}^s = 0$, then $r_i = r_{i-1} - AP_{i-1}\underline{a}_{i-1}$ belongs to the Krylov subspace $\{r_0, Ar_0, \ldots, A^{si-1}r_0\}$ and at the same time is A–orthogonal to all its vector. Thus $r_i = 0$, which is absurd.

To prove (x) we expand the norm of the residual as follows:

$$\|r_{i+1}\| = (r_0, r_0) - 2\sum_{j=0}^{i}\sum_{l=1}^{s} a_j^l\,(r_0, Ap_j^l) + \sum_{j=0}^{i}\sum_{m=1}^{s}\sum_{l=1}^{s} a_j^l a_j^m (Ap_j^m, Ap_j^l)$$

Since $(r_0, Ap_j^l) = (r_j, Ap_j^l)$ by (vii), we can rewrite the above expression in matrix form:

$$\|r_{i+1}\| = (r_0, r_0) - 2\sum_{j=0}^{i}\underline{a}_j^T\underline{m}_j + \sum_{j=0}^{i}\underline{a}_j^T M_j \underline{a}_j \,.$$

Now, as in s–step CG we can see that minimizing $\|r_{i+1}\|$ over the affine subspace $x_0 + \{P_0, \ldots, P_i\}$ is equivalent to solving the linear systems

$$M_j\underline{a}_j = \underline{m}_j, \quad j = 0, \ldots, i \,.$$

This is exactly what s–step GCR does. ■

This theorem shows the method is indeed an s–step GCR. That is, the iterate x_i of s–GCR is the equal to the iterate x_{is} of GCR. Thus GCR converges to the true solution in at most N steps.

The following theorem, which we present without proof shows the relations satisfied by the vectors generated by s–Orthomin(k).

Theorem 6.4: Assume that the degree of the minimal polynomial of r_0 is greater than $s*i$. Then the vectors x_i and the planes R_i, P_i generated by s–Orthomin(k) satisfy the following relations:

(i) P_i is A^TA-orthogonal to P_j, for $i = i-k, \ldots, i-1$, $i \geq k$

(ii) r_i is orthogonal to AP_j, for $i = i-k-1, \ldots, i-1$, $i \geq k+1$

(iii) $(r_i, Ap_i^l) = (r_i, A^l r_i)$, for $l = 1, \ldots, s$

(iv) r_i is orthogonal to AR_{i-1}

(vi) $(Ap_i^l, Ap_i^j) = (Ap_i^l, A^j r_i)$, for $1 \leq j,l \leq s$

(vii) $(r_j, Ap_i^l) = (r_{i-k}, Ap_i^l)$, for $1 \leq l \leq s$

(viii) If $r_i \neq 0$, then $a_{i-1}^s \neq 0$.

(ix) x_{i+1} minimizes $\|r_{i+1}\|$ over the space $x_0 + \{P_{i-k}, \ldots, P_i\}$. ■

Next we prove that s–Orthomin(k) converges but may require an infinite number of steps. The following theorem gives a bound on the norm of the residual error for all the s–step methods considered here.

Theorem 6.5: If $\{r_i\}$ are the residual vectors generated by s–Orthomin(k), s–GCR and s–MR, then

$$\|r_i\|_2 \leq \left[1 - \frac{\lambda_{\min}(M)^2}{\lambda_{\max}(A^TA)} \right]^{s/2} \|r_i\|_2$$

Proof: Consider the s–step Minimal Residual Method at the i–th iterate x_i of s–

Orthomin(k). The iterate and the residual given by s–MR are

$$\tilde{x}_{i+1} = \tilde{x}_i + a_i^1 r_i + \cdots + a_i^s A^{s-1} r_i$$
$$\tilde{r}_{i+1} = \tilde{r}_i - a_i^1 A r_i - \cdots - a_i^s A^s r_i$$

where $P_i = \{r_i , \ldots , A^{s-1} r_i\}$ is the direction plane of s–dimensional steepest descent and $M_i \underline{a_i} = \underline{m_i}$. The matrix M_i of inner products of the plane has the special form

$$(A^{l+1} r_i , A^{k+1} r_i), \qquad 1 \leq l,k \leq s$$

and $\underline{m_i} = [(r_i , A r_i) , \ldots , (r_i , A^s r_i)]^T$. The symmetric matrix M_i is not a matrix of moments because A is not symmetric. Thus it is not positive definite. Nevertheless, it is nonsingular as long as degree of the minimal polynomial of r_0 is greater than s. Since the residual r_{i+1} generated by s–Orthomin(k) is orthogonal to AR_i we obtain the inequality $\|r_{i+1}\|_2 \leq \|\tilde{r}_{i+1}\|$. The norm of the residual of s–MR is

$$\|\tilde{r}_{i+1}\|_2 = (r_i , r_i) \left[1 - \frac{\underline{m_i}^T M_i^{-1} \underline{m_i}}{(r_i , r_i)} \right]$$

Notice that the first iterate of s–MR is the same as the s–th iterate of GCR. This is because the two methods minimize the same error functional on the same translated Krylov subspace $x_0 + \{r_0 , \ldots , A^{s-1} r_0\}$. Using s iterations of 1–MR we obtain the following bound:

$$\|r_{i+1}\|_2 \leq \|\tilde{r}_{i+1}\|_2 \leq \|r_i\| \left[1 - \frac{\lambda_{\min}(M)}{\lambda_{\max}(A^T A)} \right]^{s/2} .$$

Next we compare the s–step methods their one–step counterparts.

6.5. Comparison of s–Step and one–Step Methods

Next we present the work for vector operations and storage requirements for the s–step GCR and Orthomin(k) when the x_{i+1} iterate is formed. At the same time we make comparisons with the one–step methods.

For one step of s–GCR we need s matrix vector products, $(i+1)s^2 + s(s+1)/2 + s$ inner products, and $4s^2iN + 4sN$ flops for the linear combinations. The same number of matrix vector products and inner products are needed by the last s steps of GCR to reach the same approximate solution. However we need $(4is + 4)sN + 2s(s+1)N$ flops to perform the linear combinations in GCR. This is due to the fact that in s–GCR we do not have to orthogonalize the directions in the s–dimensional planes P_i. Thus GCR needs $2s(s+1)N$ flops per s iterations more than s–GCR, ignoring the $O(is^3)$ flops needed in s–GCR to solve the $(i+1)$ linear systems.

For either s–GCR or Orthomin(k) we must store x_i, R_i, $\{P_j\}_{j=j_i}^{i}$ and $\{AP_j\}_{j=j_i}^{i}$. Thus compared to GCR or Orthomin(k) we have increased the storage by at most s–1 vectors.

The s–Orthomin(k) method minimizes the error functional over an affine space of dimension $(k+1)s$. By Theorem 6.5 it reduces the norm of the error by at least as much as s consecutive steps of MR. Therefore we should compare the computational work to that of s consecutive steps of Orthomin(ks). While the number of matrix vector products and inner products is the same for the two cases, the vector operations for the linear combinations are $s(4ks+4)N$ for s–Orthomin(k) and $s(4ks+4)N + 2s(s+1)$ for Orthomin(ks).

The data locality and the parallel properties of the two s-step methods $2 \leq s$ are far superior to their one–step counterparts. The critical ratios (memory references / flops) are approximately divided by s. Moreover, since the inner products can be performed together, they can be pipelined on a massively parallel system and gain a speedup up to s. An implementation similar to s–CG on a system with memory hierarchy is possible.

6.6. The s–Step Orthodir Method

The Orthodir method [YoJe80] is a variant of GCR that is guaranteed to converge even if the symmetric part of A is indefinite. It only differs from GCR in the recursion defining the direction vectors. However the truncated version Orthodir(k) does not converge for general matrices. It is not known if it converges even when A has positive definite symmetric part.

The algorithm for Orthodir and Orthodir(k) is similar to Algorithm 6.1 except that the set of A^TA- orthogonal direction vectors are

$$p_{i+1} = Ap_i + \sum_{j=j_i}^{i} b_j^i p_j$$

$$b_j^i = -\frac{(A^2p_i, Ap_j)}{(Ap_j, Ap_j)}, \quad j \leq i.$$

where $j_i = 0$ for Orthodir and $j_i = i-k+1$ for Orthodir(k). The work and storage for a single iteration of Orthodir and Orthodir(k) is described by Table 6.1 .

The s-step Orthodir and Orthodir(k) are similar to Algorithm 6.3 except that the s-dimensional direction planes are defined by

$$P_{i+1} = \overline{R_i} + \sum_{j=j_i}^{i} P_j \, \underset{\sim}{b}_j^i$$

where

$$\overline{R_i} = [Ap_i^s, A^2p_i^s, \cdots, A^{s+1}p_i^s]$$

$$\underset{\sim}{c}_j^i = [(A^2p_i^s, Ap_j^1), \cdots, (A^{s+1}p_i^s, Ap_j^s)]^T, \quad j = j_i, \ldots, i$$

The convergence of s-Orthodir is shown by proving a theorem similar to Theorem 6.4 . The work and storage requirements for s-Orthodir and s-Orthodir(k) are same as for s-GCR and s-Orthomin(k) respectively. These methods require more storage but less work than the corresponding one-step methods. .he "%"

CHAPTER 7.

s–STEP METHODS APPLIED TO ORDINARY DIFFERENTIAL EQUATIONS

7.1. Introduction

Systems of Ordinary Differential Equations (ODEs) arise frequently when modeling physical phenomena in sciences and engineering. When the problems involve a very large number of equations solving them on multiprocessors is imperative. Examples of such problems are the solution of time dependent PDEs semi–discretized by the method of lines and the transient analysis of VLSI or other massive circuits to mention just a few. In this chapter we discuss how the ODE integration methods can efficiently be implemented on systems with memory hierarchy and message passing multiprocessors. In Section 7.2 we discuss the implementation of nonstiff methods. Stiff methods require that a system of nonlinear equations be solved at every integration step. In Section 7.3 we show how the s–step iterative methods can be used in stiff methods improving the speed of the integration. We note that a large part (eighty percent) of the work involved for solving the systems of nonlinear equations. Thus speeding up this part of the computation could yield almost equal speedup of the whole process.

7.2. Nonstiff methods

Consider the system of Ordinary Differential Equations (ODEs)

$$y' = f(t, y)$$

where $f : R^{N+1} \longrightarrow R^N$ satisfies a Lipschitz condition. In this section we describe the MDD implementation of the variable–step size variable–order Adams' integration formulae. We then indicate the modifications required to make efficient use of local memory.

Assume that the solution has been approximated at the time points $t_n, t_{n-1}, \ldots, t_{n-k+1}$ and the values $y_n, y_n', \ldots, y_{n-k+1}'$ are kept. There is a unique $k+1$ degree polynomial which interpolates these values. The polynomial can be uniquely determined by the modified divided differences $\phi_j^*(n)$, for $j = 0,\ldots,k-1$ of the derivatives. For constant step size these differences coincide with the backward differences $\nabla_n^j y_n'$, for $j = 0,\ldots,k-1$. At the prediction for $t_{n+1} = t_n + h$ the value of the solution and its derivative are computed by

$$p_{n+1} = y_n + h_{n+1} \sum_{j=0}^{k-1} g_{i,1} \phi_i^*(n) \tag{7.1}$$

$$p_{n+1}' = h_{n+1} \sum_{j=0}^{k-1} \phi_i^*(n) \tag{7.2}$$

where the $g_{i,1}$ are the Adams–Bashforth coefficients(which are functions of ratios of the last k step sizes). The prediction is essentially the calculation of the value and the derivative of the polynomial interpolating $y_n, y_n', \ldots, y_{n-k+1}'$. Assume that the corrector of order $k+1$. The correction computes the value y_{n+1} such that $\tilde{y}_{n+1}'$ is an approximation to $f(t, y_{n+1})$ where $\tilde{y}'_{n+1}$ is the derivative of the polynomial interpolating $y_{n+1}, y_n, y'_n, \ldots, y'_{n-k-1}$. If the corrector is iterated to convergence, $\tilde{y}'_{n+1} = f(t, y_{n+1})$ and $y'_{n+1} = \tilde{y}'_{n+1}$. For PECE

$$\tilde{y}'_{n+1} = f(t, p_{n+1})$$
$$y'_{n+1} = f(t, y_{n+1})$$

The computations involved in advancing the step from t_n to t_{n+1} (for PECE) are [ShGo75]

```
    Compute scalars $g_{j,1}$,   $j = 0,1, \ldots, k$;
P   Compute $\phi_j^*(n) = \beta_j(n+1)\phi_j(n)$,   $j = 0,1, \ldots, k-1$,
        $\phi_k^e(n+1) = 0$,
        $\phi_j^e(n+1) = \phi_{j+1}^e(n+1) + \phi_j^*(n)$,   $j = k-1, \ldots, 0$
    Predict by (7.1);
E   Evaluate $f_{n+1}^p = f(t_{n+1}, p_{n+1})$;
C   Correct $y_{n+1} = p_{n+1} + h_{n+1}g_{k,1}(f_{n+1}^p - \phi_0^e(n+1))$;
E   Evaluate $f_{n+1} = f(t_{n+1}, y_{n+1})$,
        $\phi_{k+1}(n+1) = f_{n+1} - \phi_0^e(n+1)$,
        $\phi_j(n+1) = \phi_j^e(n+1) + \phi_k(n+1)$,   $j=k-1, \ldots, 0$.
```

where $\beta_j(n+1)$ and $g_{j,1}$ are scalar functions of the k most recent step sizes. The differences $\phi_j^e(n+1)$ are extensions of $\phi_j(n)$ by including p'_{n+1} as part of the extrapolation to the point t_{n+1}. On completing the correction step the norms of $\phi_j^e(n+1) + (f_{n+1}^p - \phi_1^e(n+1)$, $j=k-1,k,k+1$ are computed. They are required for computing the local truncation error and decide the next step size and order of the method.

Implementation with Efficient Use of Local Memory: If the above computation is performed on a system with memory hierarchy then it must be blocked according to the local memory size. The differences $\phi_j(n+1)$ must be updated simultaneously with $\phi_j^*(n+1)$, p_{n+2}, and $\phi_j^e(n+2)$ transferring the vectors to the local memory only once. The vectors $\phi j^*(n+1)$ do not need to be written in the global memory. Updating these differences gives a ratio of

$$\frac{3k+4}{5k+1}$$

This ratio is greater than one for $k=1$ and it is 19/26 for $k=5$. This suggests that the higher order nonstiff methods may run faster than lower order methods on such systems. Since the higher order methods involve more operations this implies that we should not force the integrator to change to lower order because this is not going to speedup the integration.

The computation of f_{n+1}^p, y_{n+1} and f_{n+1} can be carried out keeping the data local as two matrix vector products were combined in Chapter 4. We show how to combine $v=F(u)$, $w=F(v)$. Assume for simplicity that $J=\dfrac{\partial F}{\partial x}$ has the same structure as A (of the 2-D model problem). We partition u, v, and w into subvectors (as in Chapter 4). We can bring in the local memory the vector instructions for computing the subvector $v_k=F(u_k)$ and keep them local computing in the next block step $w_k = F(v_k)$. The vector instructions for computing $F(u)$ are basic functions such as constants, exp, $\surd$, trigonometric functions etc. The structure of J represents the dependence of F on u. So if for example for a nonlinear function F, J has the structure of A in the model problem then the implementation of the $v = F(u)$ and $F(v)$ is exactly the one in Chapter 4. Sometimes the function evaluation requires no transfer of data. Multiplication by the discrete Laplacian is such an example.

Implementation on Message Passing Systems: Function evaluations for most sparse problems arising from modeling physical phenomena only local communication of a parallel system is required. This a result of the previous paragraph and the discussion on implementing matrix vector multiplication (by sparse matrices) in Chapter 4. How much local communication is needed depends on how well the topology of the system approximates the nodes of the discretized model.

We partition every vector involved in PECE into p equal subvectors and assign each to the p processors. The scalars $\beta_j(n+1)$, t_{n+1}, and $g_{j,1}$ all depend on the step size and order. Thus they are computed at each node simultaneously. The step size h_n and order k depend on the norm of the local truncation error. Thus the only part of integration step of a nonstiff method which requires global communication of a parallel system is the computation of three norms after the correction step.

This suggests that we could overcome the frequent global communication problem if we kept the step size and order constant for a *time window*. This requires the choice of step size and order be made for the whole window. This seems reasonable because the step size is small and and increases infrequently during the integration of nonstiff problems. Thus we need a good estimate of the Lipschitz constant. The size of the window should depend on the step size (to satisfy the accuracy and stability conditions) and the global communication delay $\omega \log_2(p)$ (to satisfy a processor non-starvation condition). This means that a sufficient number of steps must be taken in a window in order to overlap the communication cost of computing the Lipschitz constant, step size and order for the next window. For example after one or two steps of constant step size the Lipschitz constant and local truncation errors (to be used in the next window) can be computed simultaneously with the integration in the current window. This would allow the global communication to be overlapped with the computations of several steps.

This is one fo the reasons why so-called waveform relaxation methods are fast on parallel systems. In these methods function which approximates the solution globally (e.g. Picard Iteration) is integrated over a time window (using uniform mesh) and this process is repeated until a good approximation is reached. The size of the time window depends on the Lipschitz constant and the accuracy requirements. The integration over a time window can be assigned to different processors. However this is not practical because the window size changes and so the number of processors per integration mesh point must change. Moreover, this would require that each node of a parallel system has stored the same data which is inefficient use of memory. Therefore it seems reasonable that parallelization of this process must be carried out by partitioning the matrix (function) into p equal sections and assigning each to the p processors. Then each processor integrates the waveform approximation on the whole window. Hence the communication required for function evaluations is the same as the one in the nonstiff methods.

7.3. Stiff Methods

Predictor Corrector integration schemes are used to solve stiff problems. The only difference is that the Corrector must have infinite stability region because large step size is required. Such methods are for example the BDF methods. At every integration step of such a method a nonlinear system of equations must be solved to retain stability. This system has the form

$$F(y_{n+1}) = y_{n+1} - \sum_{n+1} + h_n \beta_0 f\,(t_{n+1}, y_{n+1}) = 0$$

where h_{n+1} the step size, β_0 and $\sum_n$ are already computed constants. This nonlinear system cannot be solved via functional iteration (as for stiff problems), the function is not contractive. This is because the step size is large. Inexact Newton methods coupled with a linear solver can be used efficiently to solve this nonlinear system.

For large sparse Jacobians linear iterative solvers are superior to direct solvers in terms of computational work and storage. Gear and Saad [GeSA82] have used successfully CG–like methods coupled with the Newton–Rapson scheme in a BDF code. Assume $J = \dfrac{\partial F}{\partial x}$ then the outer iteration is:

$$y_{n+1}^{i+1} = y_{n+1}^{i} + \Delta_i$$

$$J\Delta_i = -F(y_{n+1}^{i})$$

where $y_{n+1}^{0} = y_{n+1}$. The inner iteration (e.g. CG method) gives the solution Δ_i. The multiplication by the Jacobian is done using the Taylor's expansion approximation

$$Jv = G(v) \equiv \frac{(F(y_{n+1}^{i} + \epsilon v) - F(y_{n+1}^{i})}{\epsilon}$$

in order to avoid computing the Jacobian explicitly. The s–step methods can be used to carry out the inner iteration more efficiently. Thus we need to perform together $v = G(u)$ and $w = G(v)$.

Implementation with Efficient Use of Local Memory: Implementation of the Prediction is similar to the nonstiff case. The correction consists of a few Newton steps coupled with an s–step method. The implementation of the s–step methods was discussed in Chapter 4. The function evaluations can be carried out by keeping the data in the local memory (as in the previous section).

Implementation on Message Passing Systems: Unlike the nonstiff methods here the solution of the nonlinear system requires the global communication of the parallel system. However the inner products in the last step of the inner (s–step CG) iteration can be computed simultaneously with the local truncation error. Pipelining (as in Chapter 4) all these inner products could eliminate (perhaps partially) the global communication bottleneck. The idea of using a time window for stiff methods seems impractical because the step changes frequently. Also, the step size is so large that good estimation (locally) of the Lipschitz is difficult even for a single step.

CHAPTER 8.

CONCLUSIONS

Iterative methods for large, sparse systems of linear equations can be used efficiently to obtain an approximation to the solution. The Conjugate Gradient method is an iterative method that can be applied to symmetric and positive definite systems. It requires no a priori information (e.g. eigenvalues) about the coefficient matrix of the system. A whole class of methods based on the Conjugate Gradient (and its variants) have been developed for solving nonsymmetric problems.

At each iteration of the Conjugate gradient one matrix vector multiplication, two inner products and three vector updates are computed. These operations can be carried out efficiently on a sequential machine. On a parallel system (e.g. Hypercube, Ring architecture etc.) the computation of a single inner product constitutes a bottleneck. This is because global communication of the system is required for the reduction part. Vector updates and single matrix vector multiplication have poor data locality. This means that a lot of vector data are tranferred and few vector operations are performed on them. Poor data locality results in poor memory utilization (and hence performance degradation) for vector processors or multiprocessors with memory hierarchy.

In this work we introduce a class of s-step CG-like iterative methods for symmetric and nonsymmetric systems of linear equations. The progress made towards the solution in one iteration of an s–step method is the same as the one made in s consecutive iteration of its one–step counterpart. In the s–step methods s matrix vector multiplications $Ar, A^2r, \ldots, A^s r$, $(s+1)$ linear combinations involving $2s$ vectors and $2s$ inner products are performed at each iteration. This means that the data locality is improved because the

$$Ratio = (Memory\ References)/(Floating\ Point\ Operations).$$

is lower than the corresponding ratio in the one–step methods. Thus these methods can be shown to have better performance on vector and multiprocessor systems with memory hierarchy. This also allows efficient use of slow secondary storage devices because one sweep through the data is required per iteration. Performing $2s$ inner products simultaneously not only improves data locality but also allows the pipelining of the reduction part on ensemble architectures (e.g. Hypercube) eliminating the global communication bottleneck of single inner product computations.

The s–step iterative methods introduced are stable. This is also demonstrated by experiments of large problems carried out on a shared memory multiprocessor system with memory hierarchy. The experiments show that the s–step methods are faster than the corresponding one–step methods. We include data on the communication speed of a state of the art message passing parallel system (Hypercube) indicating that the s–step methods implemented on such a system could be up to $2s$ times faster than the corresponding one–step methods.

To solve nonlinear systems of algebraic equations via the Newton–Rapson method requires that a sequence of systems of linear equations be solved. The coefficient matrix for these linear systems is the Jacobian of the the nonlinear system. A linear iterative solver can be used as the inner iteration of the Newton–Rapson procedure when the Jacobian is large and sparse. Replacing the one–step methods by the corresponding s–step speedups similar to the linear problems can be achieved. This can be used for example to improve the efficiency of Stiff ODE codes.

The proposed s–step iterative methods for symmetric positive definite problems have additional vector operations compared to their one–step counterparts. It remains an open question to design stable s–step methods with no extra vector operations. The design of stable and efficient s–step analogs for other nonsymmetric CG–like methods remains to be done. Efficient implementation on state of the art message passing architectures remains to be done.

REFERENCES

[AbMa86] W. Abu–Sufah and A.D. Malony. "Experimental Results for Processing on the Alliant FX/8." *CSRD Rep. #539, Univ. of Illinois, Jan., 1986.*

[Axel80] O. Axelson "Conjugate gradient type methods for unsymmetric and inconsistent systems of linear equations" *Lin. Algebra and its Applications, 29, pp. 1-16, 1980.*

[BaHo60] F. L. Bauer, and A. S. Householder "Moments and characteristic roots." *Numer. Math. 2, 42-53 (1960)*

[BrHi84] P. N. Brown and A. C. Hindmarsh "Matrix–free methods in the solution of stiff systems of ODEs," *Tech. Rep. UCID-19937, Lawrence Livermore Lab., Univ. California, Livermore, CA, 1984.*

[Cala85] D. A. Calahan "Task Granularity Studies on a many–Processor CRAY X–MP" *Parallel Computing 2 (1985), pp. 109-118, North-Holland.*

[ChJa67] T. Chan and K. Jackson
"The use of iterative linear–equation solvers in codes for large systems of stiff IVPs for ODEs," *SIAM J. SCI. STAT. COMPUT. Vol. 7, No. 2, April 1986*

[CoGO76] P. Concus, G. H. Golub and D. P. O'Leary,
"A generalized conjugate gradient method for the numerical solution of elliptic partial differential equations," *in Sparse Matrix Computations, J. R. Bunch and Rose, eds Tech. Rep. STAN-CS-76-585, Dept. Computer Science, Stanford Univ., Stanford,CA, 1976.*

[ChKS78] S. C. Chen, J. Kuck, A. H. Sameh "Practical Parallel Band Triangular System Solvers" *ACM Trans. Math. Software, Vol.4, No.3, Sep. 1978, pp. 270-277.*

[CoGM85] P. Concus, G. H. Golub and G. Meurant "Block preconditioning for the conjugate gradient method," *Siam J. Sci. Stat. Comput., Vol. 6, No. 1, January 1985.*

[Dong85] J. J. Dongarra,
"Comparison of the Cray X–MP–4, Fujitsu VP–200, and Hitachi S–810/20; An Argonne Perspective", *Argonne National Laboratory, ANL-85-19, October*

1985

[DoDu85] J. J. Dongarra and I. S. Duff. "ADVANCED ARCHITECTURE COMPUTERS" *Tech. Memo #57, Mathematics and Computer Science Division, Argonne National Lab., Oct. 1985.*

[DGRo79] P. F. Dubois, A. Greenbaum, and G. H. Rodrigue "Approximating the inverse of a matrix for use in iterative algorithms on vector processors". *Computing, 22,(1979) pp. 257-268.*

[Elma82] H. C. Elman, "Iterative methods for large, sparse, nonsymmetric systems of linear equations," *Ph. D. thesis, Tech. Rep. 229, Dept. Computer Science, Yale Univ., New Haven, CT, 1982.*

[EnHL75] W. H. Enright, T. E. Hull and B. Lindberg, "Comparing numerical methods for stiff systems of ODEs," *BIT, 15 (1975),pp. 10-48.*

[Fors68] G. E. Forsythe, "On the Asymptotic Directions of the s–Dimensional Optimum Gradient Method," *Numerische Mathematik 11, 57-76 (1968).*

[Gear80] C. W. Gear "Unified Modified Divided Difference Implementation of Adams and BDF Formulas" *Rep. UIUCDCS-R-80-1014, Univ. of Illinois, Dept. of Comp. Scie. Urbana, IL 61801 (1980).*

[Gear86] C. W. Gear "The Potential for Parallelism in Ordinary Differential Equations" *Presented at the Second Intern. Conf. of Comput. Math., Univ. of Benin, Benin City, Nigeria, Jan. 1986.*

[GeSa83] C. W. Gear and Y. Saad, "Iterative solution of linear equations in ODE codes," *Siam. J. Sci. Stat. Comput., 4 (1983) pp. 583-601.*

[GrRe86] D. G. Grunwald and D. A. Reed, "Benchmarking Hypercube Hardware and Software" *To Appear In "Second Conf. On Hypercube Multiprocessors" SIAM (1986).*

[HeSt52] M. Hestenes and E. Stiefel, "Methods of conjugate gradients for solving linear systems," *J. Res. NBS 49 (1952) pp. 409-436.*

[HwBr84] K. Hwang and F. Briggs
"Computer Architecture and Parallel Processing" *McGraw-Hill, New York, 1984.*

[JaMa86] W. Jalby and U. Meier.
"Optimizing Matrix Operations on a Parallel Multiprocessor with a Memory Hierarchy." *Inter. Conf. on Parallel Proc., 1986.*

[John85] L. Johnsson
"Data Permutations and Basic Linear Algebra Computations on Ensemble Architectures" *Research Rep. YALEU/DCS/RR-367, Feb. 1985.*

[JoMP83] O. G. Johnson, C. A. Micchelli and G. Paul "Polynomial preconditioners for conjugate gradient calculations," *Siam J. Num. Anal., Vol. 20, No. 2, April 1983.*

[Jord82] T. L. Jordan, "A guide to parallel computation and some CRAY-1 experiences," *G. Rodrigue, ed., Parallel Computations (Academic Press, New York, 1982).*

[KaSa84] C. Kamath and A. Sameh
"The preconditioned Conjugate Gradient Method on a Multiprocessor". *ANL/MCS-TM-28, Argonne National Lab., Mathematics and Computer Science Division, 1984.*

[Khab63] I. M. Khabaza
"An iterative least-square method suitable for solving large sparse matrices" *Comp. J. 6, 202-206 (1963)*

[Krog74] F. T. Krogh,
"Changing Stepsize in the Integration of Differential Equations using MDDs," *in Lecture Notes in Mathematics 362, Springer-Verlag, NY, 1974, pp. 22-71.*

[Kuck78] D. Kuck "The Structure of Computers and Computation" *John Wiley and Son, 1978*

[Lars84] J. L. Larson, "Multitasking on the X-MP-2 Multiprocessor," *IEEE Computer, Jul. 1984, pp.62-69.*

[Lawr75] D. H. Lawrie "Access and Alignment of data in an Array Processor" *IEEE Trans. Computer C-24 (Dec. 1975), pp. 1145-1155.*

[LaSa84] D. H. Lawrie, A. H. Sameh *ACM Trans. Math. Software 10, 185 (1984)*

[LuMM85] O. Lubeck and J. Moore, P. Mendez
"A Benchmark Comparison of Three Supercomputers: Fujitsu VP-200, Hitachi S810/20, and CRAY X-MP/2" *IEEE Computer, Feb. 1985, pp. 10-23.*

[MiNe85] U. Miekkala and O. Nevanlinna,
"Convergence of Dynamic Iteration Methods for Initial Value Problems" *Inst. of Math., Helsinki Univ. of Technology, Report #MAT-A230, Espoo, Finland, (1985).*

[MiLi67] W. L. Miranker and W. Liniger,
"Parallel Methods for the Numerical Integration of Ordinary Differential Equations," *Math. Comput.(1967), Vol.21, pp. 303-320.*

[Mitr85] "Asynchronous Relaxations for the Numerical Solution of Differential Equations by Parallel Processors", *AT&T Bell Laboratories, Murray Hill, N.J. 07974, (1985).*

[NeSV83] A. R. Newton and A. L. Sangiovanni–Vincentelli, "Relaxation–based electrical simulation," *Siam J. Sci. Stat. Comput. Vol. 4, No. 3, September 1983.*

[Newt79] A. R. Newton "The simulation of large-scale integrated circuits" *IEEE trans. circuits syst., Vol. CAS-26, pp. 741-749, Sept. 1979.*

[Nage75] W. Nagel, "SPICE2, A computer program to simulate semiconductor circuits," *Univ. of California, Berkeley, Memo No. ERL-M520, May 1975.*

[PaTL85] B. N. Parlett, D. R. Taylor and Z. A. Liu, "A Look–Ahead Lanczos Algorithm for Unsymmetric Matrices," *Mathematics of Computation, Vol. 44, Num. 169, Jan. 1985, pp. 105-124.*

[ReAP86] D. A. Reed, L. M. Adams, M. L. Patrick,
"Stencils and Problem Partitioning: Their Influence on the Performance of Multiple Processor Systems," *NASA/ICASE Rep. No. 86-24, May 1986*

[SaSa81] Y. Saad, A. H. Sameh
" Iterative Methods for the Solution of Elliptic Differential Equations on Multiprocessors." *Proc. of the CONPAR 81 Conf., 1981, pp 395-441*

[SaKU78] A. H. Sameh, D. J. Kuck "On Stable Parallel Linear System Solvers", *J. of the ACM 25, No.1, January 1978, pp. 81-91*

[Seag85] M. K. Seager "Parallelizing conjugate gradient for the CRAY X-MP " *Parallel Computing 3 (1986) 35-47, North-Holland.*

[Seit85] C. L. Seitz "The Cosmic Cube" *Communications of the ACM 28, 1 (1985), 22-33.*

[ShGo75] L. F. Shampine and M. K. Gordon "Computer Solution of Ordinary Differential Equations" *W. H. Freeman and Co., San Fransisco, (1985).*

[SDEE79] R. F. Sincovec, B. Dembart, M. A. Epton, A. M. Erisman, J. W. Manke, and E. L. Yip, 1979, "Solvability of large-scale descriptor systems" *Report, Boeing Computer Services Company, Seattle, WA.*

[Stie52] E. Stiefel "Uber einige Methoden der Relaxionsrechnung" *Z. Angew. Math. Physik 3, 1-33 (1952)*

[Varg62] R. Varga "Matrix Iterative Analysis", *Prentice Hall, Engledwood Cliffs, NJ, 1962.*

[VDVo82] H. A. Van Der Vorst "A vectorizable variant of some ICCG methods," *Siam J. Sci. Stat. Comput. Vol. 3, No. 3, September 1982.*

[VDVo86] H. A. Van Der Vorst "The Performance of FORTRAN Implementations for Preconditioned Conjugate Gradients on Vector Computers" *Parallel Computing 3 (1986) 49-58, North-Holland.*

[YoJe80] D. M. Youn g and K. C. Jea " Generalized Conjugate-Gradient Acceleration of Nonsymmetrizable Iterative Methods" *Linear Algebra and it Applications, 34, pp.159-194 (1980).*

[WhSV83] J. White and Sangiovanni-Vincentelli, "RELAX2, A new waveform relaxation approach for the analysis of " LSI MOS," *in Proc. 1983 Int. Symp. Circuits Syst., May 1983.*

VITA

Anthony Chronopoulos was born in Pylos, Messinia, Greece on October 18, 1956. He entered The National and Kapodistrian University of Athens in 1974 and received a B.S. in mathematics, with high honors, in June 1979. He entered graduate studies at the University of Minnesota in the fall of 1979 and received an M.S. in Applied Mathematics in August 1981. He continued his graduate studies at the University of Illinois at Urbana-Champaign completing his Ph. D. in Computer Science in the Fall of 1986. He is at the Department of Computer Science the University of Minnesota in Minneapolis.