Towards Efficient Parallel Implementation of the CG Method Applied to a Class of Block Tridiagonal Linear Systems

A.T.Chronopoulos

Department of Computer Science, University of Minnesota
Minneapolis, Minnesota 55455.

Abstract

An efficient implementation of Conjugate Gradient (CG) methods on vector and parallel machines is presented. The two different architecture models considered are the shared memory machines with memory hierarchy and the message passing private memory machines. For a parametrized vector architecture similar to CRAY-2 we present (theoretically) an implementation of the s-step CG used to solve an elliptic partial differential equation problem is twice as fast as that of the standard CG. For Hypercube parallel computers we show that the performance of s-step CG can be up to $2s$ times the performance of the standard CG.

1 Introduction

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 4-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 the maximum performance. The same is true for communication costs 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. Therefore both the distributed private memory and hierarchical memory models require carefull 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 as long as possible in fast registers or local memories and have many arithmetic operations performed on them. A good first measure of the data locality is for shared memory machines is the size of ratio *(Memory References)/(Floating Point Operations)*. For distributed memory machines the data locality to the amount local or global communications.

Several algorithms which improve the data locality for dense linear algebra problems have been suggested for shared memory systems (e.g.see [2], [4]). These algorithms are based on BLAS3 (Basic Linear Algebra level 3) modules consisting matrix times matrix operations. BLAS3 computations manage efficiently hierarchical shared memory systems. Linear algebra algorithms which improve data locality on distributed memory systems have also been studied (e.g.see [5], [6]). In the area of iterative methods BLAS2 modules implementations consisting of one or more single vector operations have been studied in [7],[8],[9]. The s-step Conjugate Gradient (s-CG) [1] is a generalization of the Hestenes and Stiefel CG [3] which improves both data locality and parallel properties. The s-CG method uses BLAS3 operations. This means that the method makes more efficient use of slower memory in a memory hierarchy system than the standard CG method. Also, 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 in a parallel system and enhances the performance of the method by pipelining the $2s$ inner products.

In section 2, the standard and s-step conjugate gradient methods are presented. In section 3, a model problem in partial differential equations which yields a large sparse matrix after discretization is described. In section 4, the efficient implementation of CG on a vector parallel system is described. In sections 5,6,7 the efficient implementation of s-CG on a CRAY-2 like architecture with memory hierarchy is presented. In sections 8,9 the efficient implementation of s-CG on a distributed memory message passing architecture are suggested.

2 The Conjugate Gradient Methods

The CG algorithm [3] for approximating the solution of a linear system $Ax=f$, where the matrix A is large and sparse is outlined here.

Algorithm 1.1 : The conjugate gradient method (CG).

```
Choose $x_0$
$p_0 = r_0 = f - Ax_0$
**For** $i=0$ **Until Convergence Do**
1.  Compute and Store $Ap_i$
2.  Compute $(p_i, Ap_i)$
3.  $a_i = \frac{(r_i, r_i)}{(p_i, Ap_i)}$
4.  $x_{i+1} = x_i + a_i p_i$
5.  $r_{i+1} = r_i - a_i Ap_i$
6.  Compute $(r_{i+1}, r_{i+1})$
7.  $b_i = \frac{(r_{i+1}, r_{i+1})}{(r_i, r_i)}$
8.  $p_{i+1} = r_{i+1} + b_i p_i$
**EndFor** .
```

Storage is required for the entire vectors solution x, residual r, direction p, and Ap vectors the sparse matrix A. Note that 3. (or 6.) must be completed before the rest of the computations in the same step can start. This forces double access of vectors r, p, Ap from the main memory at each CG step.

In s-CG s direction vectors are computed simultaneously and the the approximation to the solution is improved as much as in s consecutive steps of the standard CG.

Here we outline the algorithm in block vector form. We will use the notation

$$P = [\underline{p}^1, \dots, \underline{p}^s]$$

$$Q = [\underline{q}^1, \dots, \underline{q}^s]$$

denote the direction vectors in the odd and even iterates respectively. The parameters $\{b_{i-1}^{(j,l)}\}$ and $\{a_i^j\}$ are determined by solving $s+1$ linear systems of equations of order s. The scalar work subroutine computes the matrices $W_i = [(Ap_i^l, p_i^k)]\ 1 \le l,k \le s$ and the right hand side vectors $\underline{c}^j, \underline{b}^j$ and solves $s+1$ linear systems of size s. The scalars $\mu^i = (r_i, A_i r_i)$ are the moments of the vector r_i with respect to the matrix A.

Algorithm 2.2 The s-step Conjugate Gradient Method (s-CG)

```
Select $x_0$
Set $P = 0$
Compute $Q = [r_0 = f - Ax_0, Ar_0, \dots, A^{s-1}r_0]$
Compute $\mu^0, \dots, \mu^{2s-1}$
**For** $i=0$ **Until Convergence Do**
    **Call ScalarWork**
    **If ( i even ) then**
1.  $Q = Q + P[\underline{b}^1, \dots, \underline{b}^s]$
2.  $x_{i+1} = x_i + Q\underline{a}$
3.  $P = [r_{i+1} = f - Ax_{i+1}, Ar_{i+1}, \dots, A^{s-1}r_{i+1}]$
4.  Compute $\mu^0, \dots, \mu^{2s-1}$
    **Else**
1.  $P = P + Q[\underline{b}^1, \dots, \underline{b}^s]$
2.  $x_{i+1} = x_i + P\underline{a}$
3.  $Q = [r_{i+1} = f - Ax_{i+1}, Ar_{i+1}, \dots, A^{s-1}r_{i+1}]$
4.  Compute $\mu^0, \dots, \mu^{2s-1}$
    **EndIf**
**EndFor**
**ScalarWork Routine**
 **If** ( $i=0$ ) **then**
```

```
    Form and Decompose $W_0$
    Solve $W_0\underline{a} = m_0$
  Else
    Solve $W_{i-1}\underline{b}^j + \underline{c}^j = 0, \; j = 1, \ldots, s$
    Form and decompose $W_i$
    Solve $W_i\underline{a} = m_i$
  EndIf
  Return
  End
```

Vector products needed to advance one iteration in the s-step CG are performed by multiplying one vector by powers of the matrix. It does not seem possible to take advantage of this fact for efficient use of local memories unless the matrix has a regular sparsity structure. We will demonstrate this possibility for a linear system arising from the numerical solution of a model problem.

3 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:

$$-(au_x)_x - (bu_y)_y + cu = g \qquad (3.1)$$

where $u = H$ on $\partial\Omega$, and $a(x,y), b(x,y), c(x,y)$ and $g(x,y)$ are sufficiently smooth functions defined on Ω, and $a, b > 0$, $c \geq 0$ on Ω. If we discretize (3.1) 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 $N = n^2$. Since the PDE 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

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

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

$1 \leq i \leq n$,

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 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:

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

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

4 Efficient 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/80) 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 11/9. For example, on the CRAY-2 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 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 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-2 at half the maximum rate. (The CRAY X-MP has two channels for vector LOAD and one for vector STORE whereas the CRAY-2 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 [9]

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

This provides a ratio $(k+2)/(2k) \simeq 1/2$, 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.

5 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 a local memory or cache. Examples of such systems are the ALLIANT FX/80 and the CRAY-2 computers.

We first discuss the implementation of the matrix product for the 2-D and 3-D model problem. 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 simultaneously accommodate two full sections of A and seven full subvectors, we can carry out the computation

```
$\underline{v}_1 = A_1\underline{u}_1$
 **Do**  $k = 1, n-1$
    $\underline{v}_{k+1} = A_{k+1}[\underline{u}_k^T, \underline{u}_{k+1}^T, \underline{u}_{k+2}^T]^T$
    $\underline{w}_k = A_k[\underline{v}_{k-1}^T, \underline{v}_k^T, \underline{v}_{k+1}^T]^T$
 **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 $Au, \cdots, A^s u$ 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: $A_k = [D_k, \bar{T}_k, D_k]$, $1 \le k \le n$. 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/80).

(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 $(sm + 2sl) \le$ local memory size. The "DO" loop for all the inner products consists of an outer loop of 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 $(2s+2)m \le$ local memory size. The "DO" loop for all the linear combinations consists of an outer loop of N/m steps and an inner loop of m steps. By using the matrix notation 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
```

6 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 VRs 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-2 with 11 VRs (instead of 8).

Vectorization:

(i) *Matrix vector products:* $r_i = f - Ax_i, Ar_i, \ldots, A^4r_i, (A^4r_i, A^5r_i)$. 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 Fig. 6.1 for two processors.

We will perform the computation as follows: (1) r_i, Ar_i, (2) A^2r_i, A^3r_i, (3) $A^4r_i, (A^4r_i, A^5r_i)$. 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.

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

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^+. Use of the operator "$*$" or ("+") here means that the contents of two VR go through the vector functional unit pipeline (so that the elements of the vectors are arithmetically combined componentwise).

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 $v$ ]
```

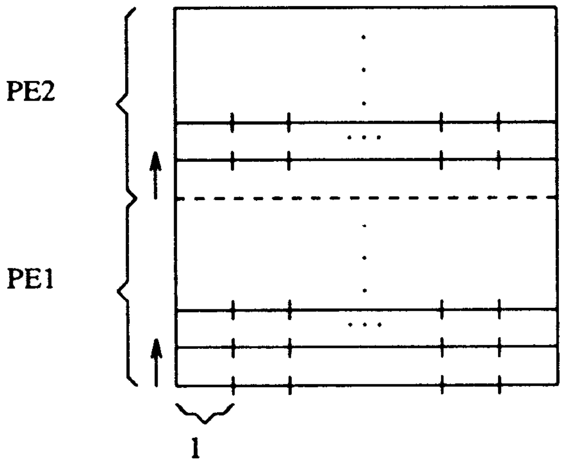

Fig. 6.1. Processor mapping to square grid.

```
    $w = c^{-n} * v^{-n} + b^{-} * v^{-} + b * v^{+} + a * v$
    [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 (n^2/l^2) vector steps. This part is only $1/l$ 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 $18N$ floating point operations (*Ops*) for v, w and 10 vector transfers. This gives a ratio : $10/18 \approx 1/2$. 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, $v = Au$ 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 $iuv = iuv(i: i+l)$, 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 3/10 for (2) and 3/8 for (1). Hence the ratio for the inner products is $\approx 1/3$ 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 $60N$ *Ops* give a ratio of $17/60 \approx 1/3$. As for CG we can update the solution vector less frequently, improving this ratio to about 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 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.

8 Comparison of s-CG with CG on a CRAY-2-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 the comparison is easier. The number of *Ops* and the critical ratios for 5-CG and CG for a single iteration of algorithms 2.1 and 2.2 are shown on Table 7.1. We use the notation: Vops (Vector operations), Dotpr (Dot products), Mv (Matrix times vectors), Lc (Linear combinations)

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

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 $33 + 30 + 10 = 73N$ cycles compared to $55 + 45 + 15 = 110N$ 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. The local memory of the CRAY-2 processors can be efficiently programmed to provide memory space for three additional vector registers. Thus the implementation of s-CG for the CRAY-2-like architecture can be realized on the CRAY-2 system.

9 Description of a Message Passing Computer Architecture

Vops	*Ops*	ratio	*Ops*	ratio
Dpr	20N	1/3	20N	3/4
Mv	54N	10/18	45N	11/9
Lc	60N	17/60	30N	3/2

Table 7.1 *Ops* and (mem. transfers)/*Ops* (5-CG in columns 1,2 and 5 consecutive steps of CG in columns 3,4).

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).

A hypercube model is a particular example of a distributed-memory message passing parallel computer. In a hypercube of dimension d, there are 2^d processors. Assume that these are labeled $0,1,...,2^d-1$. Two processors i and j are directly connected iff the binary representation of i and j differ in exactly one bit. Each edge of Figure 8.1 represents a direct connection of a dimension 4 hypercube (lines and dotted lines are communication links).

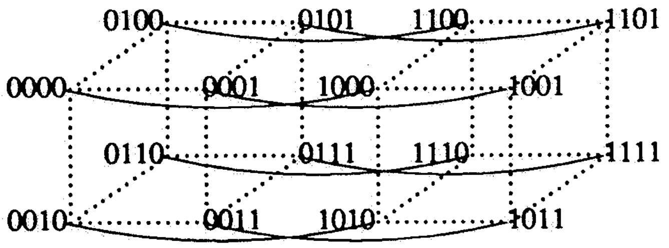

Fig. 8.1. Four dimensional Hypercube.

Thus in a hypercube of dimension d, each processor is connected to d others, and 2^d processors may be interconnected such that the maximum distance between any two is d.

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 may coincide with the processor dedicated for computations (e.g. INTEL iPSC/1). The time required for a message packet of size k bytes to be communicated between two adjacent nodes is

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

where α is the communication latency (startup time for a transmission) and β is the transmission time per byte and k the number of bytes in the message packet. Assume that there are 2^d nodes in the system. We can distinguish two types of communications needed to carry out computations on an ensemble architecture. The local communication involves only $m \ll 2^d$ neighboring nodes and the global one involves $m = 2^d$ nodes. The time required for a message to be communicated globally is

$$\omega d$$

Table 8.1 shows the times for single 8 Byte neighbor transfer, the times to perform an 8 Byte Add and Multiply operations and the ratio Communication/Computation (Comm./Comp.) time on NCUBE/7. We see that a single 8 byte message transfer between two directly connected processors takes 42 times the time for an 8 byte real addition and 32 times that of an 8 byte real multiplication. Furthermore, longer messages are transferred at a higher rate (i.e. bytes per second) than shorter ones going the same distance. In a linear time model of nearest neighbor communication a message of length M bytes requires approximately 446.7+2.4M microseconds where the constant term is a startup time which consists of the software overhead at each end and the time to set up the circuit and the second term is the actual transmission time. The startup time for short messages is the dominating factor in the communication cost on the NCUBE.

Operation	time	Comm./Comp.
8 B transfer	470μs	
'+'	11.2μs	42 times
'*'	14.7μs	32 times

Tab. 8.1. 8-Byte Oper. times on NCUBE/7.

10 Implementation of CG and s-CG on Message Passing Architectures

The CG and s-CG methods can be implemented on such a system by dividing all vectors into 2^d equal subvectors (and the matrix A into 2^d horizontal sections) and storing them at the 2^d 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 is computed by assigning an equal part of the vector (if possible) to each node. This allows each processor to work on local segments independently of other processors for most operations. Each node computes in parallel the sum of squares of its part of the vector. Then an Exchange-Add algorithm [5] can be used. Processors $P_{(z_{d-1},\ldots,z_{i+1},0,z_{i-1},\ldots,z_0)}$, ($i$=0,...,$d$−1) concurrently exchange their most recent partial sum with their neighbor $P_{(z_{d-1},\ldots,z_{i+1},1,z_{i-1},\ldots,z_0)}$ and then concurrently form their new partial sum. At the end of $2d$ concurrent nearest neighbor communication steps, each processor has its own copy of the inner product. The Exchange-Add algorithm is illustrated in Fig. 9.1(d=3).

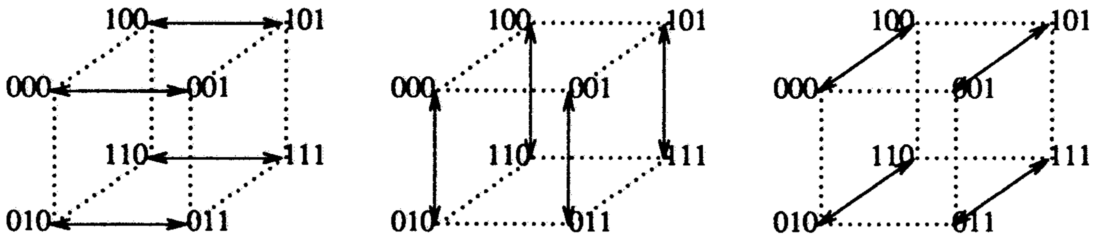

Fig. 9.1. Dot product computation on Hypercube

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 NCUBE machine.

$$t_a < \omega\,(=\alpha + 8\,\beta)$$

The exchange-add algorithm described in the previous section (for 2^d processors) requires time equal to

$$d\,(2\omega + t_a)$$

(A more expensive way is to collect all numbers to a single node for the addition and then transmit the result to all nodes)

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

$$d\,(2\bar{\omega} + \bar{t}_a)$$

where $\bar{\omega} = \alpha + 2s8\beta$ and $\bar{t}_a$ is the time to add two vectors of length $2s$. If we assume that every node has a separate processor for passing messages, then $\bar{t}_a$ (or t_a) overlaps with $2\bar{\omega}$ (or 2ω). This is because of the assumption that the time for scalar addition is much less 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 products versus 1)

$$\frac{2s(2\omega)}{2\bar{\omega}} = \frac{2s(\alpha + 8\beta)}{\alpha + 2s8\beta}\ .$$

When the transmission startup time α is much greater than $16s\beta$ the speedup is of order $2s$. This essentially means that for such a 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 $2s$.

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

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

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

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

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. To carry out one matrix vector multiplication every node must send one subvector to two neighbors and must receive two subvectors. So only local communication of the processors is necessary to carry out this part of the computation. In Fig. 9.2 we demonstrate how to assign the matrix blocks and subvectors to the processors of a hypercube of dimension 3 for the single matrix vector multiplication.

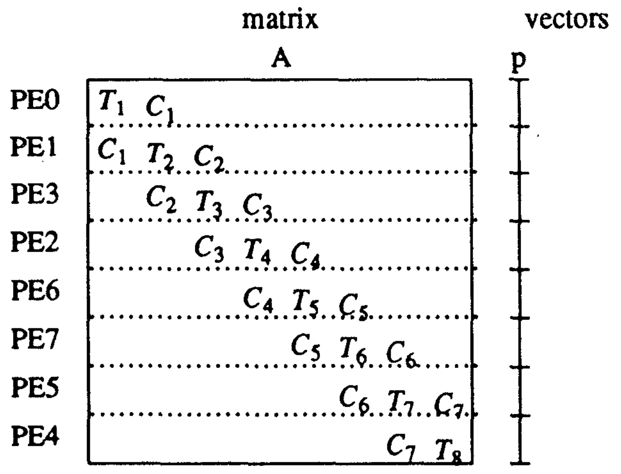

Fig. 9.2. Matrix and vector mapping to processors

Assume for the 3-D problem that the number of processors if $n = 2^d$ then n horizontal sections of order n^2

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

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.

11 Summary and future work

We have presented an efficient implementation of the s-step conjugate gradient method on a vector parallel system with memory hierarchy and on a message passing distributed memory architecture. We can conclude that the s-step methods give significant performance enhancement over the standard methods.

References

[1] A.T.Chronopoulos and C.W.Gear, "s-step iterative methods for symmetric linear systems", *Journal of Computational and Applied Mathematics 25, pp. 153-168, 1989.*

[2] J. J. Dongarra, D. C. Sorensen, "Linear Algebra on High-Performance Computers" *Parallel Computing 85, M. Feilmeier, G. Joubert and U. Schendel (eds.), Elsevier Science Publishers B.V.*

[3] M. Hestenes and E. Stiefel, "Methods of conjugate gradients for solving linear systems," *J. Res. Nat. Bureau of Stand. 49, pp. 409-436, 1952.*

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

[5] R.F.Lucas, T.Blank, and J.J.Tiemann, "A Parallel Solution Method for Large Sparse Systems of Equations", *IEEE Transactions on Computer-Aided Design, Vol. CAD-6, No. 6, November 1987*

[6] O.A.McBRYAN and E.F.van de Velde, "Matrix and vector operations on hypercube parallel processors", *Parallel Computing 5, pp. 117-125, 1987.*

[7] Gerard Meurant, "Multitasking the conjugate gradient method on the CRAY X-MP/48", *Parallel Computing 5, pp. 267-280, 1987.*

[8] M. K. Seager, "Parallelizing conjugate gradient for the CRAY X-MP " *Parallel Computing 3, pp. 35-47, 1986.*

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