Proceedings of **ITC-CSCC 2000**, Pusan, Korea

Efficient Implementation of CG and CR Methods for Linear Systems on a Single Processing Node of the HITACHI SR8000

S. Nishimura, D. Takahashi, T. Shigehara, H. Mizoguchi and T. Mishima

Department of Information and Computer Sciences, Saitama University
Shimo-Okubo 255, Urawa, Saitama 338-8570, JAPAN
Phone: +81-48-858-9035, Fax: +81-48-858-3716
E-mail: seiji@me.ics.saitama-u.ac.jp, {daisuke,sigehara}@ics.saitama-u.ac.jp

Abstract: We discuss the iterative methods for linear systems on a single processing node of the HITACHI SR8000. Each processing node of the SR8000 is a shared memory parallel computer which is composed of eight RISC processors with a pseudo-vector facility. We implement highly optimized codes for basic linear operations including a matrix-vector product and apply them to the conjugate gradient (CG) and the conjugate residual (CR) methods for linear systems. Our tuned codes for both methods score nearly 50% of the theoretical peak performance, which is the best in the sense that it corresponds to an asymptotic performance of the inner product.

1. Introduction

The significance of a large-scale numerical computation is rapidly growing in the various scientific and technological fields such as fluid dynamics and quantum chemistry. In particular, high performance solvers for linear systems are highly desired because the problem is frequently turned into linear systems after a suitable discretization of space and time coordinates.

Automatic tuning is a recent trend of linear algebra library. ATLAS (Automatically Tuned Linear Algebra Software) [1] is a typical example. However, such library is mainly intended for scalar processors. There is no instance intended for vector or pseudo-vector [2, 3] processors. To remedy the situation, we implement in this paper fundamental routines from BLAS (Basic Linear Algebra Subprograms) [4–7] and develop highly optimized code of the conjugate gradient (CG) [8] and conjugate residual (CR) [9] methods on the HITACHI SR8000, which is one of up-to-date parallel supercomputers. We restrict ourselves to a single processing node in this paper. A single processing node of the SR8000 is a shared memory parallel computer which is composed of eight RISC processors with a pseudo-vector facility.

The paper is organized as follows. Experimental environment is summarized in Sect.2. We show in Sect.3 the performance of the basic linear operations including an optimized matrix-vector product, which is applied to the CG and CR methods in Sect.4. The current work is summarized in Sect.5.

2. Experimental Environment

We summarize experimental environment of this work and also give specifications of the HITACHI SR8000. Numerical experiments were performed at the Computer Centre Division, Information Technology Center, the University of Tokyo. The SR8000 system is composed of 128 processing nodes interconnected through a three-dimensional hyper-crossbar network. Each processing node is a shared memory parallel computer with eight RISC processors (Instruction Processor, IP) with a pseudo-vector facility. Each IP has two *multiply-add* arithmetic units with machine cycle of 4[nsec]. As a result, the total theoretical peak performance of each processing node is 8[GFLOPS]. Each IP is designed to achieve a similar performance to a vector processor by adopting a pseudo-vector facility, which serves to suppress a delay caused by cache misses.

We use a single processing node for numerical experiments. The programming language is FORTRAN77. The compile options are "`-64 -nolimit -noscope -Oss -procnum=8 -pvfunc=3`". These options instruct the compiler to use 64-bit addressing mode ("`-64`"), to remove limits of memory and time for compilation ("`-nolimit`"), to forbid dividing a source code into multiple parts when it is compiled ("`-noscope`"), to set the optimize level to the highest ("`-Oss`"), to use eight IP's ("`-procnum=8`") and to set the pseudo-vectorize level to the highest ("`-pvfunc=3`"), respectively. We also give the compiler directives concerning a parallelization among IP's [10].

3. Vector Operations and Optimized Matrix-Vector Product

We begin with basic vector operations in Table 1, which are often used to solve linear systems. In Table 1, $\mathbf{x}$

Table 1: Basic vector operations.

Name in BLAS	Function
daxpy	$\mathbf{y} := \mathbf{y} + \alpha\mathbf{x}$
ddot	$(\mathbf{x}, \mathbf{y})$
dnrm2	$\|\|\mathbf{x}\|\|_2 = \sqrt{(\mathbf{x}, \mathbf{x})}$
dscal	$\mathbf{x} := \alpha\mathbf{x}$

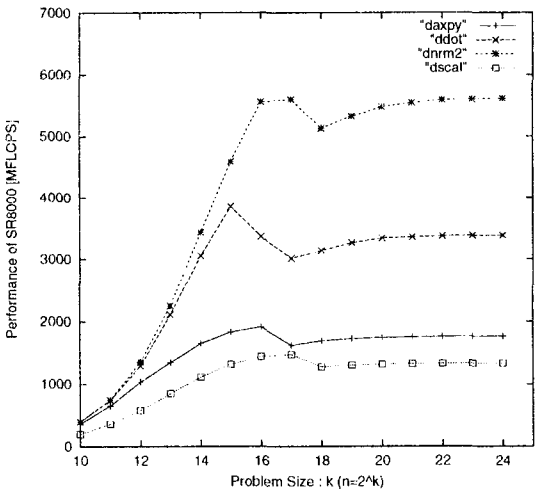

Figure 1: Performance of basic vector operations in Table 1.

and $\mathbf{y}$ are n-dimensional real vectors and α is a real scalar. The source code of these routines is implemented with the same interface as in BLAS. The vector operations in Table 1 are written in a single loop which is pseudo-vectorized and parallelized with a block distribution [10]. Fig.1 shows the performance of the basic linear operations. The vertical axis shows the performance in units of MFLOPS, while the horizontal axis is $k = \log_2 n$ with the problem size n. Note that since the kernel loop is performed on eight IP's in parallel, the loop length on each IP is $n/8$.

It is clear that the ratio between arithmetic operations and data operations in the kernel is one of the most important factors to keep good performance. From a viewpoint of the arithmetic operations, the Euclidean norm dnrm2 is the same as the inner-product ddot. However, dnrm2 is about 1.6 times faster than ddot. This is due to the fact that the statement $s := s + x_i y_i$ requires two *load* and one *multiply-add* operations. As a result, the performance of ddot is at most 4[GFLOPS], namely 50% of the theoretical peak performance of a single processing node. On the other hand, dnrm2 requires only one *load* operation for a single *multiply-add* operation. This explains the ratio of the performance between dnrm2 and ddot. One can also observe that

```
*  at(j,i)=a(i,j), (i=1,...,m, j=1,...,n)
   do 10 i=1,m-1,2
      dtmp1=0.d0
      dtmp2=0.d0
      do 20 j=1,n
         dtmp1=dtmp1+at(j,i  )*v(j)
         dtmp2=dtmp2+at(j,i+1)*v(j)
20    continue
      u(i  )=dtmp1
      u(i+1)=dtmp2
10 continue
```

Figure 2: FORTRAN source code for matrix-vector product. Loop-unrolling to a depth of two is employed for outer loop. Additional statements are required if m is odd.

ddot is almost twice faster than daxpy. This is because daxpy requires a *store* operation after two *load* and one *multiply-add* operations, which is unnecessary for ddot. Similarly, dscal requires a *store* operation after one *load* and one *multiplication* operations. Thus, the ratio of arithmetic operations to data operations is the smallest in dscal. This is the reason why the score of dscal is the poorest in Fig.1.

One can see that the performance of the operations for a single vector (dnrm2 and dscal) is saturated at $k = 17$, since the data cache memory for each processing node is 128[KB/IP] $\times$ 8[IP] in the SR8000. For the operations for two vectors (dnrm2 and dscal), the saturation occurs around a half of the problem size; $k = 15 \sim 16$. For each operation, the performance is kept at a high level even for a larger problem size, owing to a pseudo-vector facility In an asymptotic region ($k = 24$), the performance of daxpy, ddot, dnrm2 and dscal is 1755.6[MFLOPS], 3359.8[MFLOPS], 5565.5[MFLOPS] and 1322.6[MFLOPS], respectively.

In order to keep good performance at each step of iterative solutions for linear systems, it is crucial to develop a highly optimized code for matrix-vector product. Fig.2 shows a double loop for matrix-vector product. We parallelize the outer loop with a block distribution and also pseudo-vectorize the inner loop for vector inner product. In Fig.2, we use the transposed matrix A^T instead of A. (The elements of A^T can be overwritten on the original A in an actual implementation, since one uses A only in a form of the matrix-vector product.) Since the matrix is stored by columns in FORTRAN, the continuous memory access to the matrix elements of A is ensured by using the transposed matrix A^T. We also employ loop-unrolling to a depth of two for the outer loop. For the inner loop, we leave the unrolling to the compiler, as in vector operations. As a result, the inner loop is unrolled to a depth of four. The loop-unrolling to a depth of two of the outer loop makes it

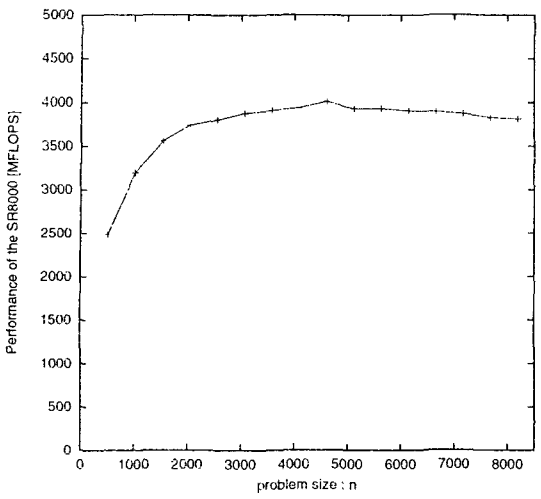

Figure 3: Performance of optimized matrix-vector product in Fig.2. Outer loop is performed among 8 IP's in parallel. Inner loop is pseudo-vectorized on each IP.

possible to reduce the number of *load* operations for the vector $\mathbf{v}$ to the half in the inner loop. As well, the length of the outer loop is reduced by 50%. As a result, the performance is improved by about 10%.

Fig.3 shows the performance of the tuned matrix-vector product. One can see that the curve in Fig.3 reproduces the performance of the inner product ddot in Fig.1. This is mainly due to the use of the transposed matrix to keep a continuous memory access. Recall that since the loop for vector operations in Fig.1 is parallelized among eight IP's, each IP processes only one eighth of the vector elements. Thus, Fig.3 should be compared with the performance of ddot for $k = 12 \sim 16$ in Fig.1.

We have examined unrolling of the outer loop to depth greater than two in numerical experiment. The performance for a depth of four is almost the same as for a depth of two, while a depth of eight gives rise to only 10% performance compared to the case of a depth of two. A depth of eight is too large to store relevant elements in the floating-point registers on each IP.

4. CG and CR Methods

In this section, we discuss the implementation of CG and CR methods on a single processing node of the SR8000. The CG method [8] is one of typical iterative methods for a linear system $A\mathbf{x} = \mathbf{b}$. The algorithm of CG method is shown in Fig.4. It can be verified that if the coefficient matrix A is symmetric and positive definite, then the approximate solution converges to the exact solution within n iterations for any initial guess. Here, n is the number of unknowns. Owing to this theorem, the CG method is usually applied to the case

Take an initial guess $\mathbf{x}_0$;
$\mathbf{r}_0 := \mathbf{b} - A\mathbf{x}_0$; $\mathbf{p}_0 := \mathbf{r}_0$;
for $k := 0, 1, 2, \cdots$ **until** $||\mathbf{r}_k|| < \epsilon||\mathbf{b}||$ **do**
begin

$$\begin{aligned} \alpha_k &:= (\mathbf{r}_k, \mathbf{p}_k)/(\mathbf{p}_k, A\mathbf{p}_k); \\ \mathbf{x}_{k+1} &:= \mathbf{x}_k + \alpha_k \mathbf{p}_k; \\ \mathbf{r}_{k+1} &:= \mathbf{r}_k - \alpha_k A\mathbf{p}_k; \\ \beta_k &:= -(\mathbf{r}_{k+1}, A\mathbf{p}_k)/(\mathbf{p}_k, A\mathbf{p}_k); \\ \mathbf{p}_{k+1} &:= \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k; \end{aligned}$$

end

Figure 4: Algorithm of CG method.

Take an initial guess $\mathbf{x}_0$;
$\mathbf{r}_0 := \mathbf{b} - A\mathbf{x}_0$; $\mathbf{p}_0 := \mathbf{r}_0$; $\mathbf{q} := A\mathbf{p}_0$;
for $k := 0, 1, 2, \cdots$ **until** $||\mathbf{r}_k|| < \epsilon||\mathbf{b}||$ **do**
begin

$$\begin{aligned} \mu &:= (\mathbf{q}, \mathbf{q}); \\ \alpha_k &:= (\mathbf{r}_k, \mathbf{q})/\mu; \\ \mathbf{x}_{k+1} &:= \mathbf{x}_k + \alpha_k \mathbf{p}_k; \\ \mathbf{r}_{k+1} &:= \mathbf{r}_k - \alpha_k \mathbf{q}; \\ \mathbf{a} &:= A\mathbf{r}_{k+1}; \\ \beta_k &:= -(\mathbf{a}, \mathbf{q})/\mu; \\ \mathbf{p}_{k+1} &:= \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k; \\ \mathbf{q} &:= \mathbf{a} + \beta_k \mathbf{q}; \end{aligned}$$

end

Figure 5: Algorithm of CR method.

when A is a positive definite symmetric matrix. The CR method [9] is another efficient algorithm for solving linear systems. It can be applied even for asymmetric problems. The algorithm of CR method is shown in Fig.5. A remarkable feature of the CR method is that the residual decreases monotonically if the symmetric part of the coefficient matrix A, i.e. $(A + A^T)/2$, is positive or negative definite. One can see from Figs.4 and 5 that a dominant part in calculation at each iteration is a single matrix-vector product both for the CG and CR methods.

In this study, we restrict ourselves to a dense linear system. The iterative methods are important even for dense linear systems, since the convergence is often attained within a relatively small number of iterations [11,12]. In numerical experiment, we use $A = (a_{ij})$ with $a_{ij} = \max\{i, j\}$, $(i, j = 1, 2, \cdots, n)$ as the coefficient matrix. Fig.6 shows the performance of the CG and CR methods for the problem size $n = 256 \times i$; $i = 1, 2, \cdots, 32$. The behavior of both curves is quantitatively explained by the performance of matrix-vector

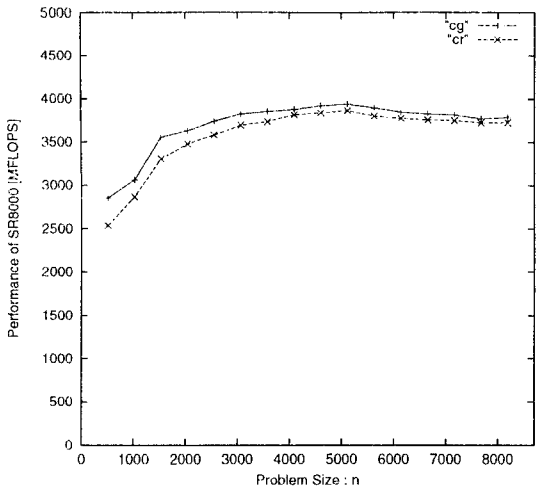

Figure 6: Performance of CG and CR methods.

product in Fig.3. For $n = 8192$, the performance of the CG and CR methods is 3785.8[MFLOPS] and 3729.2[MFLOPS], respectively. They are close to the asymptotic performance of `ddot` subroutine, nearly 50% of the peak performance of a single processing node of the SR8000. If we use a standard matrix-vector product without using the transposed matrix, the performance is at most 2700[MFLOPS], which we have checked in experiment.

5. Summary

We have implemented the highly optimized codes of matrix-vector product and applied it to the CG and CR methods for linear systems on a single processing node of the HITACHI SR8000, which is a shared memory parallel computer composed of eight IP's with a pseudo-vector facility. The performance of the tuned CG and CR methods is nearly 4[GFLOPS], which amounts to the peak performance for the vector inner product on a single processing node of the SR8000. In a future work, we shall (1) develop the optimized code for iterative methods with compressed row storage [13] on the SR8000, and (2) extend the present work to multiple processing nodes.

References

[1] R. C. Whaley and J. J. Dongarra, "Automatically Tuned Linear Algebra Software," `http://www.netlib.org/atlas/atlas.ps.gz`.

[2] K. Nakazawa, H. Nakamura, H. Imori and S. Kawabe, "Pseudo vector processor based on register-windowed super-scalar pipeline," Proceedings of Supercomputing '92, pp.642-651, 1992.

[3] K. Nakazawa, H. Nakamura, T. Boku, I. Nakata and Y. Yamashita, "CP-PACS: A massively parallel processor at the University of Tsukuba," Parallel Computing, vol.25, pp.1635-1661, 1999.

[4] C. Lawson, R. Hanson, D. Kincaid and F. Krogh, "Basic Linear Algebra Subprograms for Fortran Usage," ACM Trans. on Math. Soft., vol.5, pp.308-325, 1979.

[5] J. J. Dongarra, J. DuCroz, S. Hammarling and R. Hanson, "An Extended Set of Fortran Basic Linear Algebra Subprograms," ACM Trans. on Math. Soft., vol.14, no.1, pp.1-32, 1988.

[6] J. J. Dongarra, I. S. Duff, J. DuCroz and S. Hammarling, "A Set of Level 3 Basic Linear Algebra Subprograms," ACM Trans. on Math. Soft., vol.16, no.1, pp.1-17, 1990.

[7] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst, "Solving Linear Systems on Vector and Shared Memory Computers," SIAM, Philadelphia, 1991.

[8] M. R. Hestenes and E. Stiefel, "Methods of conjugate gradients for solving linear systems," Journal of Research of the National Bureau of Standards, vol.49, pp.409-436, 1952.

[9] A. Greenbaum, "Iterative methods for solving linear systems," SIAM, Philadelphia, 1997.

[10] S. Nishimura, D. Takahashi, T. Shigehara, H. Mizoguchi and T. Mishima, "A Performance Study on a Single Processing Node of the HITACHI SR8000," Proceedings of Second Conference on Numerical Analysis and Applications, Russe, Bulgaria, June, 2000, to appear.

[11] M. Yokoyama, T. Shigehara, H. Mizoguchi and T. Mishima, "Efficiency of the CR method for solving large-scale dense linear system on distributed memory parallel computers," Proceedings of ITC-CSCC'98, pp.1219-1222, Sokcho, Korea, 1998.

[12] M. Yokoyama, T. Shigehara, H. Mizoguchi and T. Mishima, "Iterative methods for dense linear systems on distributed memory parallel computers," IEICE Trans, vol.E82-A, no.3, pp.483-486, March, 1999.

[13] R. Barrett et al., "Templates for the solution of linear systems: building blocks for iterative methods," SIAM, Philadelphia, 1996.