Parallel Reconstruction for Parallel Imaging SPACERIP on Cellular Computer Architecture

Yanwei Niu
Ziang Hu
Guang R. Gao

CAPSL Technical Memo 57
June 15, 2004

Abstract

Recently the new field of parallel imaging accusation has the potential to revolutionize the field of fast MR imaging. The SPACE RIP technique is one of the parallel imaging methods which uses multiple receiver coil and utilizes the sensitivity profile information from a number of receiver coils in order to reduce the acquisition time. The image reconstruction problem of SPACE RIP is a computation intensive task, which need to be parallelized to further reduce the reconstruction time. In this paper, we analyzed the algorithm and identified the program bottleneck to be parallelized. The loop level parallelization is implemented with Pthread, OpenMP and MPI. Furthermore, since the reconstruction uses Singular Value decomposition (SVD) to solve the matrix pseudoinverse, we implemented the one sided Jacobi parallel SVD on the state-of-art cellular computer architecture Cyclops64 to speedup the problem at the fine grain level. Experimental result shows that cellular computer architecture has very small overhead and thus suitable for fine grain level parallelization.

Contents

List of Figures

List of Tables

1 Introduction

Recently the new field of parallel imaging accusation has the potential to revolutionize the field of fast MR imaging. There are many reasons to further increase the speed of MR image acquisition and image reconstruction. Reduction in acquisition time can reduce or even avoid motion artifacts, make the MR imaging more efficient and make it useful for more potential applications. For instance,dynamic imaging applications of cardiac contraction requires high temporal resolutions without undue sacrifices in spatial resolution, therefore, those applications can be greatly served if the data acquisition time can be reduced with an order of magnitude[20].

The parallel imaging techniques use spatial information contained in the component coils of an array to partially replace spatial encoding which would normally be performed using gradients, thereby reducing imaging acquisition time. The name “parallel” is due to the fact that multiple MR signal data points are acquired simultaneously. In a typical parallel imaging acquisition, only a fraction of the phase encoding lines are acquired compared to the conventional acquisition. A specialized reconstruction is then applied to the data to reconstruct the image. The maximum acquisition time reduction factor would be number of coils used.

A number of parallel imaging methods have been proposed. The SMASH (SiMultaneous Acquisition of Spatial Harmonics) method proposed by Sodickson and Manning [25] is a k-space domain implementation of the parallel imaging. It is based on the computation of the sensitivity profiles of the coils in one direction. These profiles are then weighted appropriately and combined linearly in order to form sinusoidal harmonics which are used to generate the k-space lines that are missing due to undersampling. This technique showed an 8 fold increase in imaging speed.

The SENSE (sensitivity encoding) method proposed by Prussemann et al.[22] is an image domain sensitivity encoding method. It relies on the use of 2D sensitivity profile information in order to reduce image acquisition time. Like SMASH, the cartesian version of SENSE requires the acquisition of equally spaced k-space lines in order to reconstruct sensitivity weighted, aliased versions of the image. It is shown in [22] that the SENSE technique can reduce the scan time to one-half using a two-coil array in brain imaging and double-oblique heart images can be obtained in one-third of conventional scan time with an array of five coils.

The SPACE RIP [20] method proposed by Walid E. Kyriakos is a parallel imaging and reconstruction technique. It generalizes the SMASH approach by allowing the arbitrary placement of RF receiver coils around the object to be imaged , it also allows any combination of k-space lines as opposed to regularly spaced ones.

This paper focuses on the parallel image reconstruction of the SPACE RIP algorithm. In section 2, the SPACE RIP technique will be briefly reviewed to explore the parallelism inherent in the problem. It will be shown that the reconstruction of each column in the image are totally independent of each other, thus making the SPACE RIP technique a perfect candidate for parallel computing. Furthermore, the reconstruction of each column is a pseudoinverse of a matrix, which is solved by singular value decomposition (SVD). Accordingly, the parallelization is implemented at two different levels: the coarse grain level parallelization will be presented in Section 3, the fine grain parallelization of SVD algorithm will be presented in Section 4. The target platform C64 will be introduced in Section 5. The performance experimental results are shown in Section 6. The conclusions are summarized in Section 7.

2 Encoding Scheme and Reconstruction Algorithm of SPACE RIP

MRI uses gradient coils to encode each voxel with a different frequency and phase. Frequency and phase correspond to a given location in k-space. Taking the Fourier Transform will convert the acquired data from k-space to coordinate space. The concept of parallel imaging is based on using multiple receiver coils, each providing independent information about the image.

Mathematically, the MR signal received in a coil having $W_k(x, y)$ as its complex 2D sensitivity profile can be written as:

$$s_k(G_y^g, t) = \int\int r(x, y)W_k(x, y)e^{j2\pi(G_x xt + G_y^g y\tau)}dxdy, \tag{1}$$

where $r(x, y)$ denotes the proton density function, $W_k(x, y)$ is the complex 2D sensitivity profile of this coil, G_x represents the readout gradient amplitude applied in the x direction, G_y^g represents the phase encoding gradient applied during the g^{th} acquisition, x and y represent the x and y positions, respectively, and τ is the pulse width of the phase encoding gradient G_y^g.

In the conventional serial imaging sequences, only one receiver coil is used to collect all the data required to reconstruct a digitized version of $r(x, y)$, assuming $W_k(x, y) = 1$. To achieve this, the phase encoding gradient G_y is varied in order to cover all of the k-space with the desired resolution. One echo is needed for each value of G_y^g, making sequential imaging a time consuming procedure.

There are some ways to reduce the acquisition time for sequential imaging. For instance, multi-echo imaging EPI (Echo Planar Imaging) can achieve higher speed by optimizing strengths, switching rates, and patterns of gradients and RF (Radio Frequency) pulses. However, these approaches sometimes will decrease SNR (Signal to Noise ratio) or spatial resolution, besides, they tend to require higher magnetic field strengths and increased gradient performance, thus reaching the technical limits[22].

The SPACE RIP technique [20] uses multiple receiver coils and utilizes the sensitivity profile information from a number of receiver coils in order to minimize the number of acquisitions needed to estimate and reconstruct $r(x, y)$.

In the SPACE RIP technique, if we take the Fourier transform of Eq.[1] along the x direction when a phase encoding gradient G_y^g is applied, we can get:

$$S_k(G_y^g, x) = \int r(x, y)W_k(x, y)e^{j2\pi(G_y^g y\tau)}dy, \tag{2}$$

which is the phase modulated projection of the sensitivity weighted image onto the x axis. Here the x and y are continuous value. In order to get discrete version of $r(x, y)$ the $r(x, y)$ and $W_k(x, y)$ are expanded along the y direction in terms of a set of orthonormal sampling functions $\Psi_n(y)$, with further

$$\begin{pmatrix} S_1(G_y^1,x) \\ \cdot \\ S_1(G_y^F,x) \\ S_2(G_y^1,x) \\ \cdot \\ S_2(G_y^F,x) \\ \cdot \\ \cdot \\ S_K(G_y^1,x) \\ \cdot \\ S_K(G_y^F,x) \end{pmatrix} = \begin{pmatrix} W_1(x,1)e^{j2\pi(G_y^1 1\tau)} & \cdots & W_1(x,N)e^{j2\pi(G_y^1 N\tau)} \\ \cdot & \cdots & \cdot \\ W_1(x,1)e^{j2\pi(G_y^F 1\tau)} & \cdots & W_1(x,N)e^{j2\pi(G_y^F N\tau)} \\ W_2(x,1)e^{j2\pi(G_y^1 1\tau)} & \cdots & W_2(x,N)e^{j2\pi(G_y^1 N\tau)} \\ \cdot & \cdots & \cdot \\ W_2(x,1)e^{j2\pi(G_y^F 1\tau)} & \cdots & W_2(x,N)e^{j2\pi(G_y^F N\tau)} \\ \cdot & \cdots & \cdot \\ \cdot & \cdots & \cdot \\ W_K(x,1)e^{j2\pi(G_y^1 1\tau)} & \cdots & W_K(x,N)e^{j2\pi(G_y^1 N\tau)} \\ \cdot & \cdots & \cdot \\ W_K(x,1)e^{j2\pi(G_y^F 1\tau)} & \cdots & W_K(x,N)e^{j2\pi(G_y^F N\tau)} \end{pmatrix} \cdot \begin{pmatrix} \eta(x,1) \\ \eta(x,2) \\ \eta(x,3) \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ \eta(x,N) \end{pmatrix} \tag{4}$$

mathematical simplification detailed in [20], we can finally get

$$S_k(G_y^g,x) = \sum_{n=1}^{N} \eta(x,n) W_k(x,n) e^{j2\pi(G_y^g n\tau)}. \tag{3}$$

where N is the number of pixels in the y direction. The $\eta(x,n)$ is the discretized version of $r(x,y)$. The symbol k is used to denote the different coils with $k = 1, K$, where K is the total number of coils. The symbol g is used to denote different phase encodes, the value of g can be from 1 to F, where F is the number of phase encode used in the experiments. This expression can be converted into the matrix form for each position x along the horizontal direction of the image, as shown in Eq.[4].

Essentially we can simplify the Eq.[4] as:

$$A(x) = G(x) \times I(x), x = 1 \quad to \quad M; \tag{5}$$

Where the A(x), G(x), I(x) represents the left, middle and right item in the Eq.[4]. Their dimensions are $KF \times 1$, $KF \times N$, $N \times 1$ respectively. K is the number of coils , F is the number of phase encodes for each coil. The M and N is the resolution of the reconstructed image. Normally the M and N will be 256 by 256 or 128 by 128.

A(x) contains the F phase encoded values for all K coils, it is essentially one dimensional DFT of the chosen k-space data. I(x) is an N-element vector representing one column of the image to be reconstructed, x is the horizontal coordinate of that column. G(x) can be constructed based on the sensitivity profiles and phase encodes used. If a image has M columns, then x could be from 1 to M, for each particular x, we have such a equation as Eq.[5]. Those M equations can be constructed and solved independently from each other, which means each column of the image can be reconstructed independent of each other. Increasing of M and N will increase the computation load. It can also be seen that the Gain matrix G(x) will become larger when K and F increase, thus increasing the computation load.

3 Loop Level Parallelization

In this section, the coarse grain parallelization of the image reconstruction is presented. As shown in the previous section, the SPACE RIP reconstruction algorithm can be done column by column independently of each other. The actual program begins with reading k-space data from the data file, then 1D DFT is computed along the x direction, followed by a major loop reconstructing the columns one by one. This loop has M iterations, where M is the x dimension of the reconstructed image. Inside each iteration, a matrix $G(x)$ as in Eq.[4] is constructed (we call it Gain matrix), then the pseudoinverse of this matrix is computed and one column of the image is finally reconstructed by multiplying the inverse matrix with the vector $A(X)$ as in Eq.[4]. Timing profiling of the program for typical data set shows that the major loop occupies about 98.79 % of the total execution time. So this loop is the bottleneck to be parallelized.

Both Pthread and OpenMP version parallelization at the loop level are implemented. The speedup result on a 12 CPUs Sunfire workstation will be shown in section 6. On a shared-memory multiprocessor computer, All CPUs share the same main memory and can work on the same data concurrently. The major advantage of the shared-memory machine is that no explicit message passing is needed, thus making it easier for programmer to parallelize the sequential code of the application compared to message passing based parallel languages, such as PVM or MPI.

Multithreaded programming is a programming paradigm tailored to shared-memory multiprocessor system. Multi-threaded programming offers an alternative to multi-process programming that is typically less demanding of system resources – here the collection of interacting tasks are implemented as multiple threads within a single process. The programmer can regard the individual threads as running concurrently and need not implement task switching explicitly, which is instead handled by the operating system or thread library in a manner similar to that for task switching between processes. Libraries and operating system support for multithreaded programming are available today on most platforms, including almost all available Unix variants. However, it is worth noting that there is a certain amount of overhead for handling multiple threads, so the performance gain archived by parallelization must outweigh this overhead. In our application, the loop level parallelization are at the coarse grain level, thus justifying the overhead.

Pthread [5] is a standardized model for dividing a program into subtasks whose execution can be interleaved or run in parallel. The OpenMP Application Program Interface (API) [7] supports multi-platform shared-memory parallel programming in C/C++ and Fortran on almost all architectures, it is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications.

It is worth noting that static variable are shared all across all threads for both Pthread and OpenMP programming. In the SPACE RIP code, some CLAPACK [1] routines are used, however, the CLAPACK [1] routines has many unnecessary static local variable, which are not thread-safe since they will cause some unwanted sharing. If left undealt, these unintended variable sharing will cause false result or may affect performance.

In the current implementation, the memory for A(x), G(x) and I(x) as shown in Eq.[5] are preallocated, thus the program structure is quite simple, all the threads can work on totally independent memory locations and return the result to also totally independent memory locations. No communication issue need to be considered due to the problem property. In our implementation, dynamic load balancing strategy is used for task distribution. As a matter of fact, the load balancing is not a big issue for our test platform because all the slave nodes has similar performance and task computation load are similar according to our observation.

Furthermore, an MPI version is implemented on Linux Cluster. Nowadays building clustering servers for high performance computing is gaining more and more acceptance. Assembling large Beowulf clusters [26] is easier than ever and the performance is increasing dramatically. So we also implemented an MPI version of the loop body parallelization. The difference from the above SMP based solution is the MPI version need explicit message passing. Specifically, the M iterations in the loop is distributed to slave nodes dynamically, after the computation of the pseudoinverse for each column, the slave nodes will send back the result (Pseudo inverse of the Gain matrix) to the master nodes, the master will then send a new column index to this slave nodes. Such process will continue till all the iterations are completed. At the beginning, the master nodes will send necessary information to slaves, including the Phase Encodes data and necessary information about the image such as image dimension. Also for each iteration, the slave need to send back $KF \times N$ double precision complex number as the result, which will cause relatively heavy communication overhead.

4 Parallel SVD for Complex matrices

The pseudoinverse of the Gain matrix G(x) is solved by the Singular Value Decomposition. In this section, we will present the parallelization of the one sided Jacobi SVD algorithm. First the current existing algorithm for SVD are briefly reviewed. Then a one sided Jacobi update algorithm for complex matrix is proposed because the Gain matrix is complex matrix in this particular application. Then our parallel implementation will be presented with the parallel ordering of GaoThomas [14]. GaoThomas parallel ordering will be briefly reviewed and related implementation issue on SMP will be discussed. The parallelization is implemented both on the current SMP and cellular architecture which is under development. The speedup result will be presented in the Section 6.

4.1 Singular Value Decomposition and existing methods

One of the important problems in the mathematical science and engineering is singular value decomposition (SVD). SVD is one of the most important factorization of a real or complex matrix and is a very computationally intensive problem. A SVD of a real or complex m by n matrix is its factorization of this matrix into the product of three matrices:

$$A = U\Sigma V^H \tag{6}$$

where U is an m by n matrix with orthogonal columns, Σ is an n by n matrix non-negative diagonal matrix, and V is an n by n orthogonal matrix. Here we use H to denote the complex conjugate transpose of a matrix, if a matrix is real matrix, then H is just a transpose operation.

There are many algorithms for solving the SVD problem. Firstly, the QR algorithm, this algorithm is used to solve singular value decomposition of a bidiagonal matrix. QR is used to compute singular vectors in LAPACK's [1] computational routine xBDSQR, which is used by driver routine xGESVD to compute the SVD of dense matrices. The xGESVD routine will first reduce a matrix to bidiagonal form, then call QR routine xBDSQR to find the SVD of the bidiagonal matrix. Originally the SPACE RIP sequential code uses zgesvd routine to solve the SVD problem of a complex matrix. It is worth noting that Matlab SVD routine uses LAPACK routines DGESVD (for real matrix) and ZGESVD (for complex matrix) to compute the singular value decomposition.

Secondly, Divide-and-conquer method. It divides the matrix into two halves, computes the SVD of each half, and glues the solutions together by solving a special rational equation. Divide-and-conquer is implemented by LAPACK [1] computational routine xBDSDC, which is used by LAPACK driver routine xGESDD to compute the SVD of a dense matrix. It is currently the fastest method available in LAPACK to solve the SVD problem of a bidiagonal matrix larger than about 25 by 25 [16]. xGESDD is currently the LAPACK algorithm of choice for the SVD of dense matrices. However, to our best knowledge, there is no current parallel version of ZGESVD routine or ZGESDD routine in the ScaLAPACK [2], a parallel version of LAPACK.

Finally, Jacobi's method [9, 15]. It is most suitable for parallel computing. This transformation method repeatedly multiplies on the right by elementary orthogonal matrices (Jacobi rotations) until it converges to $U\Sigma$, the product of the Jacobi rotations is V . Jacobi is slower than any of the above transformation methods, but has the useful property that it can deliver the tiny singular values, and their singular vectors, much more accurately than any of the above methods provided that it is properly implemented [11, 10]. Especially it is shown that Jacobi method is more accurate than QR algorithm [12].

4.2 One sided Jacobi algorithm

In our implementation, we will focus on the one sided Jacobi SVD method since it is most suitable for parallel computing. In the one sided Jacobi algorithm, in order to compute an SVD of an $m \times n$ matrix A, most of the algorithms uses the Jacobi rotations. The idea is to generate an orthogonal matrix V such that the transformed matrix $AV = W$ has orthogonal columns. Normalizing the Euclidean length of each nonnull column of W to unity, we will get the relation:

$$W = U\Sigma, \tag{7}$$

where the U is a matrix whose nonnull columns form an orthonormal set of vectors and Σ is a nonnegative diagonal matrix. Since $V^H V = I$, where I is the identity matrix, we have the SVD of A given by $A = U\Sigma V^H$.

Hestenes [18] uses plane rotations to construct V. In the rest of this subsection, first we review the Hestenes's method for real matrices, then we extend the method for complex matrices.

Hestene generates a sequence of matrices $\{A_k\}$ using the rotation

$$A_{k+1} = A_k Q_k \tag{8}$$

where the initial $A_1 = A$ and Q_k is a plane rotation. Let $A_k \equiv (\vec{a}_1^{(k)}, \vec{a}_2^{(k)}, \cdots, \vec{a}_n^{(k)})$, and $Q_k \equiv q_{rs}^{(k)}$, suppose the Q_k represents a plane rotation in the (i, j) plane, with $i < j$, Let us define:

$$\begin{aligned} q_{ii}^{(k)} &= c, & q_{ij}^{(k)} &= s, \\ q_{ji}^{(k)} &= -s, & q_{jj}^{(k)} &= c. \end{aligned} \tag{9}$$

The postmultiplication by Q_k affect only two columns:

$$(\vec{a}_i^{(k+1)}, \vec{a}_j^{(k+1)}) = (\vec{a}_i^{(k)}, \vec{a}_j^{(k)}) \begin{pmatrix} c & s \\ -s & c \end{pmatrix}. \tag{10}$$

To simplify the notation, let us define:

$$\begin{aligned} \vec{u}' &\equiv \vec{a}_i^{(k+1)}, & \vec{u} &\equiv \vec{a}_i^{(k)} \\ \vec{v}' &\equiv \vec{a}_j^{(k+1)}, & \vec{v} &\equiv \vec{a}_j^{(k)}. \end{aligned} \tag{11}$$

Then we have:

$$(\vec{u}', \vec{v}') = (\vec{u}, \vec{v}) \begin{pmatrix} c & s \\ -s & c \end{pmatrix}. \tag{12}$$

For real matrices, to make the two new columns orthogonal, we have to satisfy $(\vec{u}')^T \vec{v}' = 0$, further mathematical manipulation will yield:

$$(c^2 - s^2)w + cs(x - y) = 0, \tag{13}$$

where $w = \vec{u}^T \vec{v}$, $x = \vec{u}^T \vec{u}$, $y = \vec{v}^T \vec{v}$.

Rutishauser[24] proposed the formulas as in Eq.[14] to solve the Eq.[13]. They are in use because they can diminish the accumulation of rounding errors:

$$\begin{aligned} \alpha &= \frac{y - x}{2w}, & \tau &= \frac{sign(\alpha)}{|\alpha| + \sqrt{1 + \alpha^2}} \\ c &= \frac{1}{\sqrt{1 + \tau^2}}, & s &= \tau c. \end{aligned} \tag{14}$$

We set $c = 1$ and $s = 0$ if $w = 0$.

4.3 Our Extension to complex matrix

It is worth noting the above formulas only apply for real matrices. In the case of complex matrices, in order to make the two new columns orthogonal, we have to make $(\vec{u}')^H \vec{v}' = 0$, which still leads to Eq.[13], except that the inner products w, x and y are now defined differently:

$$w = \vec{u}^H \vec{v}, x = \vec{u}^H \vec{u}, y = \vec{v}^H \vec{v}. \tag{15}$$

Now the x and y are still real number, but w may be complex numbers, which makes the solution as shown in Eq. [14] not valid anymore.

Park [21] proposed a real algorithm for Hermitian Eigenvalue decomposition for complex matrices, Henrici [13] proposed a Jacobi method for computing the principal values of a complex matrix, both of them used two sided rotations. Inspired by their methods, we derived the following one sided Jacobi rotation method for complex matrices, we modify the rotation as follows:

$$(\vec{u}', \vec{v}') = (\vec{u}, \vec{v}) \begin{pmatrix} e^{j\beta} & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} c & s \\ -s & c \end{pmatrix} \begin{pmatrix} e^{-j\beta} & 0 \\ 0 & 1 \end{pmatrix}. \tag{16}$$

where we get the angel β from w: $w = |w|e^{j\beta}$, the formula to get c and s are as follows:

$$\begin{aligned} \alpha &= \frac{y-x}{2|w|}, & \tau &= \frac{sign(\alpha)}{|\alpha| + \sqrt{1+\alpha^2}} \\ c &= \frac{1}{\sqrt{1+\tau^2}}, & s &= \tau c. \end{aligned} \tag{17}$$

We set $c = 1$ and $s = 0$ if $|w| = 0$.

The idea is to first apply a complex rotation shown in Eq.[16], after this complex rotation, the inner product of the two updated columns becomes real number. It is easy to verify that the $(\vec{u}')^H \vec{v}' = 0$ is satisfied with our proposed rotation method.

If the matrix V is desired, the plane rotations can be accumulated. We compute

$$V_{k+1} = V_k Q_k \tag{18}$$

We can update the A and V simultaneously.

4.4 Parallelism of One sided Jacobi algorithm

The plane rotation has to be applied to all column pairs exactly once in any sequence (a sweep) of $n(n-1)/2$ rotations. Several sweeps are required so that the matrix converges. A simple sweep consists of a cyclic-by-rows ordering:

$$\begin{aligned} &(1,2),(1,3),\cdots,(1,n), \\ &(2,3),\cdots,(2,n),(3,4),\cdots,(n-1,n). \end{aligned} \tag{19}$$

Unfortunately, the cyclic-by-rows scheme is apparently not amenable to parallel processing. For instance, pairs $(1,2)$ and (1,3) can't be updated at the same time. However, it is easy to see some pairs are independent and could be executed in parallel if we change the order in the sequence. For instance, let us consider a matrix with 4 columns, with the cyclic-by-rows order, the sequence of a sweep is:

$$(1,2),(1,3),(1,4),(2,3),(2,4),(3,4). \tag{20}$$

Another possible sequence for a sweep groups independent pairs and executes them in parallel:

$$\{(1,2),(3,4)\},\{(1,4),(2,3)\},\{(1,3),(2,4)\}. \tag{21}$$

Where the pairs in curly brackets are independent. Generally speaking, two plane rotations $R(i, j)$ and $R(r, s)$ are independent if $i \neq r, i \neq s, j \neq r$ and $j \neq s$[23]. The feature has motivated the proposal of many parallel Jacobi ordering in which the $n(n-1)/2$ rotations required to complete a sweep are organized into groups of independent transformations. We call each of these groups a step. The parallel ordering can allow us to exploit the parallelism since the work associated to one step can be distributed among nodes in the multiprocessor system.

Many Jacobi ordering have been proposed for different parallel computing platforms. Brent and Luk have examined various algorithm for multiprocessor arrays [3]and mesh-connected processors [4], Gao and Thomas [14] have investigated this problem using a recursive divide exchange communication pattern for hypercube multicomputers, D.Royo has extended the one-sided Jacobi method to 2D/3D mesh multicomputer[23]. Gao and Thomas's algorithm is optimal in terms of achieving both the maximum concurrency in computation and minimum overhead in communication.

4.5 Parallel scheme of our implementation

In this research, our target platform for parallel SVD is shared-memory architecture, so the communication may not be a big issue. We select to implement the Gao and Thomas algorithm. This algorithm examines the pairs of n elements on $n/2$ processors when n is power of 2. In each computation step, each processor examines one pair. During the communication stage, each processor exchange only one column with another processors. The total number of computation steps is $(n-1)$ and the network traffic, defined as the total number of messages set between processors, is $\frac{1}{2}n(n-1)$ messages[14]. The detailed recursive divide and exchange algorithm is beyond the scope of this paper, here we only give one example of parallel ordering for a matrix with 8 columns as in Table.1

step 1	(1,2)	(3, 4)	(5, 6)	(7,8)
step 2	(1,4)	(3, 2)	(5, 8)	(7,6)
step 3	(1,8)	(3, 6)	(5, 4)	(7,2)
step 4	(1,6)	(3, 8)	(5, 2)	(7,4)
step 5	(1,5)	(3, 7)	(6, 2)	(8,4)
step 6	(1,7)	(3, 5)	(6, 4)	(8,2)
step 7	(1, 3)	(7, 5)	(6, 8)	(4,2)

Table 1: Parallel Ordering of Gao and Thomas's algorithm

In our shared memory implementation, the number of slave threads p can be set equal to the number of available processors. All the column pairs in one step can be treated as a work pool, the works in this work pool will be distributed to the p slave threads, where $1 \leq p \leq \frac{n}{2}$. After each step, we implemented a barrier to make sure the step $k+1$ will always use the updated column pairs from step k. At the end of each sweep, we will check whether the convergence condition is satisfied. If not, then we start a new sweep again. Otherwise, the program will terminate.

The convergence behavior of different ordering may not be same. Hansen [17] discusses the convergence properties associated with various ordering. In order to enforce convergence, in our implementation, we have chosen to use a threshold approach [27]. We will omit any rotation if the inner product $(\vec{u})^H \vec{v}$ of the current column pair $\vec{u}$ and $\vec{v}$ is below the a certain threshold δ. The δ is defined as follows:

$$\delta = \epsilon \cdot \sum_{i=1}^{N} A[i]^H A[i], \tag{22}$$

Where the ϵ is the machine precision epsilon, $A[i]$ is the ith column of the initial A matrix. At the end of each sweep, if all the possible pairs in this sweep has converged according to above standard, then the problem has converged.

The speedup result of our implementation will be presented in Section 6.

5 Target Platform

In this research, our test platform includes a Sunfire SMP machine from Sun Inc, a Linux cluster and a Cellular computer architecture C64 underdevelopment, of which the C64 is our main interest. In this section, we will review the background and the main features of the Cellular computer architecture C64.

5.1 Background of cellular computer architecture Cyclops 64

The Cyclops64(C64) is a petaflop supercomputer project underdevelopment at IBM research Laboratory. We use it to benchmark our parallel SVD algorithm. The background of the Cyclops64 computer architecture is two fold. On one hand, current overall system performance is more and more limited by the performance of the memory subsystem, on the other hand, the technology development will produce chips with billions of transistors, enabling large quantities of logic and memory to be placed on a single chip. The Cyclops project is an renovative idea to explore the thread-level parallelism on a single chip multi-processor.

The main principle of the Cyclops architecture are[6]:(1) the integration of processing logic and memory in the same piece of silicon; (2) the use of massive intra-chip parallelism to tolerate latencies; (3) a cellular approach to building large systems. Besides, the lower latency inter-processor communication and synchronization will bring better performance.

There are other efforts to exploit the thread-level parallelism to achieve better performance on a chip. Some of them [19] are embedded systems for multimedia applications. However, these multiprocessors are application specific and not available for general-purpose use. The Cyclops system is a general purpose platform which can support a wide range of applications. Some possible kernel applications include FFT and other linear algebra such as BLAS 1 and 2 of LAPACK [1] package, Protein folding and other bioinformatic applications. In this research, we use it for solving the SVD linear algebra problem in the context of biomedical imaging.

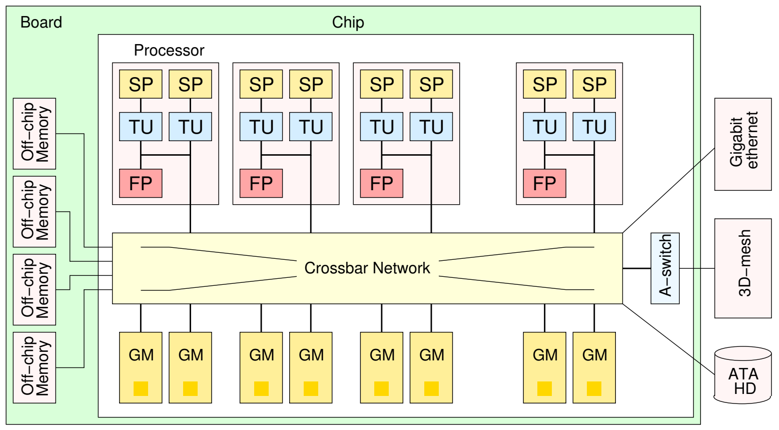

Figure 1: Cyclops64 Chip

5.2 Cyclops 64 Chip architecture

Figure.1 shows the hardware architecture of a C64 chip, the main component of a C64 node. Each C64 chip has 80 processors, each consisting of two thread units, a floating-point unit and two SRAM memory banks of 32KB each. A 32KB instruction cache, not shown in the figure, is shared among five processors. In a C64 chip architecture there is no data cache. Instead a portion of each SRAM bank can be configured as scratch-pad memory. Such a memory provides a fast temporary storage to exploit locality under software control. Processors are connected to a crossbar network that enables intra-chip communication, i.e. access to other processor's on-chip memory as well as off-chip DRAM, and inter-chip communication via input and output ports that connect each C64 chip to its nearest neighbors in the 3D-mesh. The intra-chip network also facilitates access to special hardware devices such as the Gigabit Ethernet port and the serial ATA disk drive attached to each C64 node.

5.3 Cyclops 64 software system

On the software side, one important part of the C64 system software is the C64 thread virtual machine. It is worth noting that OS is not developed for the C64 architecture since the OS would put considerable overhead on top of the machine which is aimed for simplicity from the bottom up. Instead, CThread is implemented directly on top of the hardware architecture as a micro-kernel/run-time system that fully takes advantage of the C64 hardware features.

C64 thread virtual machine includes a thread model, a memory model and a synchronization model. The details of those models are explained in [8]. Suffice it to say that, the C64 chip hardware supports a shared address space model: all on chip SRAM and off-chip DRAM banks are addressable from all thread units/processors on the same chip. That is all the threads see a single shared address space. Each thread unit has an associated 32KB SRAM bank. Each memory bank can be partitioned into two sections: one section is called "global" (or "interleaved") section, the other "local" (or "scratch-pad") section. All such "global sections" together form the (on chip) "global memory" in an interleaved fashion that are uniformly addressable from all thread units. All local (scratch-pad) sections are globally (but non-uniformly) addressable by all thread units. In conclusion, we can regard one cyclops chip as a single-chip SMP system with multiple thread units of execution.

CThread run-time system library provides the software/application developer with the minimum functionality to write multithreaded programs: thread management, support for mutual exclusion, synchronization among threads, inter-node communication (under evaluation), etc. In order to achieve high performance and scalability, the implementation of such functionality tries to match as close as possible the architecture underneath the microkernel/RTS.

In the thread synchronization model, the CThread mutex lock and unlock operations are directly implemented using C64 hardware atomic test-and-set operations, thus very efficient. Furthermore, a very efficient barrier synchronization primitive is provided. Barriers are implemented using the "Signal Bus" special purpose register. All the thread units on a chip are connected by an 8-bit bus, changes in the state of the bus propagates throughout the chip in a few cycles, providing a means for very fast global synchronization. The barrier function can be invoked by a group of threads. Threads will block until all participating threads in the operation has reached this routine.

The software tool chain of C64 platform currently provides a compiler, linker and simulator for users. A number of optimization level are supported by the compiler. A multi-chip multi-threading functional accurate simulator (FAST) is also provided. The main features are: (1) FAST can generate the execution trace and/or an instruction statistics report to help a software/application developer tuning and optimizing a program; (2) It can also generate timing result at cycle level of a program simulated; (3) Detailed system simulation are slow. For an application to be simulated, sometimes the code has to be slightly modified.

6 Experiments and Results

In this section, we will present the experimental result of both loop level parallelization and the fine level parallelization of SVD. The fine level experiments described in this paper are carried out by using Sunfire machine and the cellular computer architecture C64, the speedup performance of the two platforms are compared.

6.1 Loop level parallelization

The loop level parallelization is carried out on the Sunfire SMP machine and Linux cluster. The SMP machine used is the DBI-RNA1 at Delaware Biotechnology Institute. DBI-RNA1 is a Sun Sunfire 4800 Server with 12 SPARC 750MHz CPUs, and 24 gigabyte memory. The code has also been ported to the Cellular computer architecture C64, however, due to the fact that the Simulator at this stage is slow to carry out the loop level parallelization experiment, in this section, we will only present the Loop level parallelization result on Sunfire SMP machine and Linux cluster.

In the data used in this experiment, the number of coils is 4, the image size is 128 by 128, the number of phase encodes is 38.

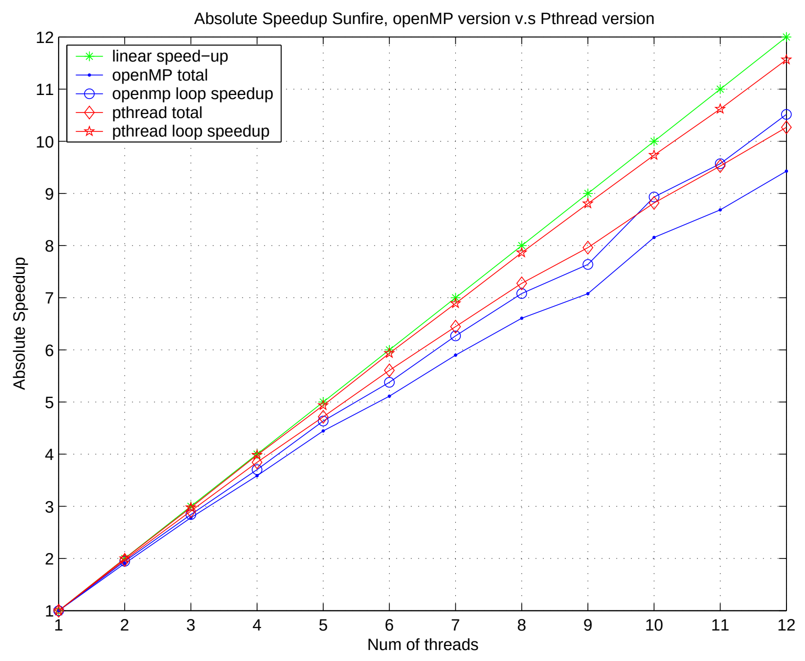

Figure 2: Loop level parallelization result on Sunfire

Figure.2 presents the result of both Pthread and OpenMP. The speedup of the both the total execution time and the loop only are presented. From the figure, it can be seen that both Pthread and OpenMP achieved near linear performance up to 12 threads. This is due to the fact that the tasks (iterations) of the loop are totally independent of each other.

According to the very well known "Amdahl's" law, if a program can be express as two portions: the serial (nonparallelizable) portion S and the parallel portion P, then the time $T(n)$ required to complete a task on n parallel processors can be approximated as:

$$T(n) = S + \frac{P}{n} \tag{23}$$

and the speedup for n CPUs can be expressed as:

$$sp = \frac{T(1)}{T(n)} = \frac{S+P}{S+\frac{P}{n}}. \tag{24}$$

From the above equation, it can be seen that the speedup of a parallel program can not continue to grow forever. Instead, there is a theoretical limit for the speedup:

$$sp_\infty = \lim_{n\to\infty} sp = \frac{S+P}{S} \tag{25}$$

According to our timing experiment, the total execution time of loop body occupies about 98.79 % of the total execution of the sequential program, which means the limit of the speedup is around 82.64. We assume for bigger data set, the loop body time percentage will be even bigger, and in real application, the loop body may be used to handle real-time stream data acquired. So we will only focus on the

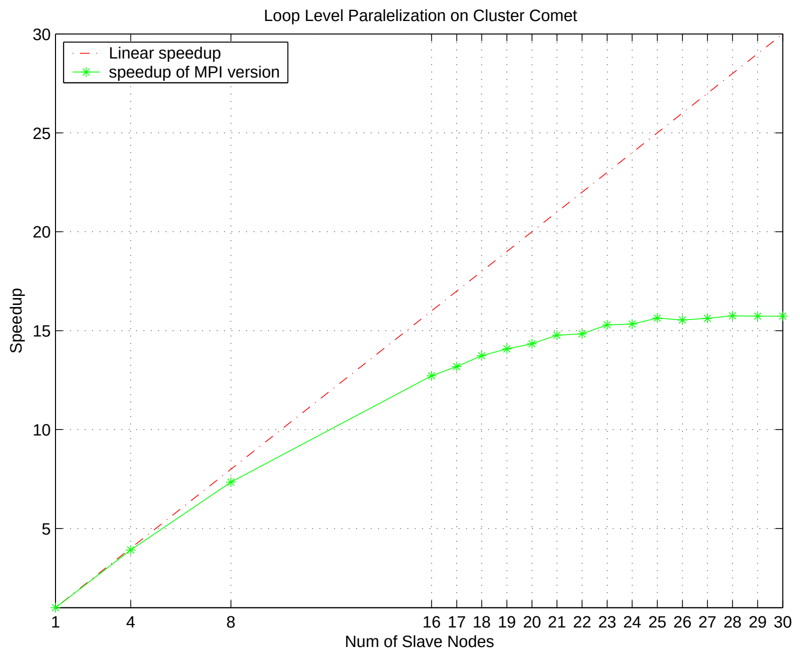

Figure 3: Loop level parallelization on Linux cluster

speedup of the loop itself in the following discussion. For instance, in the fine level parallelization part, only the speedup of the loop is shown.

A MPI version is also implemented and tested on a Linux Cluster. The Linux cluster ”Comet” consists of 18 nodes, each containing two 1.4 GHz AMD Athlon processors – total of 36 processors – and 512MB of DDR SDRAM memory. The interconnection network for the nodes is a switched 100Mbps ethernet. From the figure, it can be seen that the MPI speedup can achieve good speedup till the number of slave nodes reaches around 20. After that, the speedup gradually top to around 16. It is because when the number of slave increases, the work load distributed to each slave become smaller, which can not justify the communication overhead at the initialization stage.

6.2 Fine level parallelization: parallel SVD on SMP machine Sunfire

In this section, the speedup result of the one sided Jacobi SVD on Cyclops 64 for complex matrix is reported. Figure.4 shows the speedup for the matrix size 128 by 128 through the size 1024 by 1024 (Pthread version). The number in the matrix are uniformly random double precision number. From the figure, it can be seen that for small problem size such as 128 by 128, the speedup is limited: the speedup tops at around 4. This is because that the task grain is not big enough to justify the overhead associated with the thread creation and synchronization such as barrier and mutex. In order to achieve good speedup for small problem size, small thread synchronization overhead is necessary, which is a good feature of Cyclops 64 architecture.

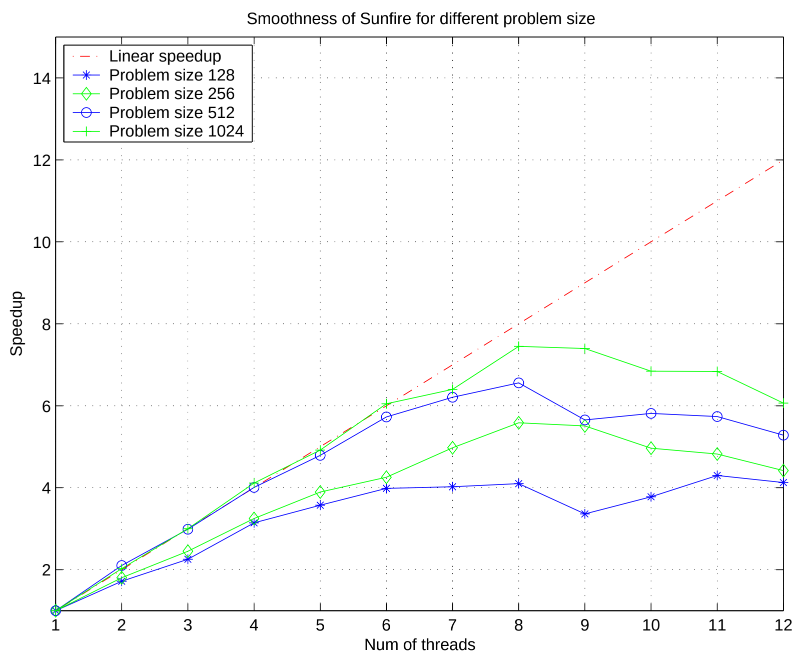

Figure 4: Speedup of parallel one sided Jacobi complex on Sunfire

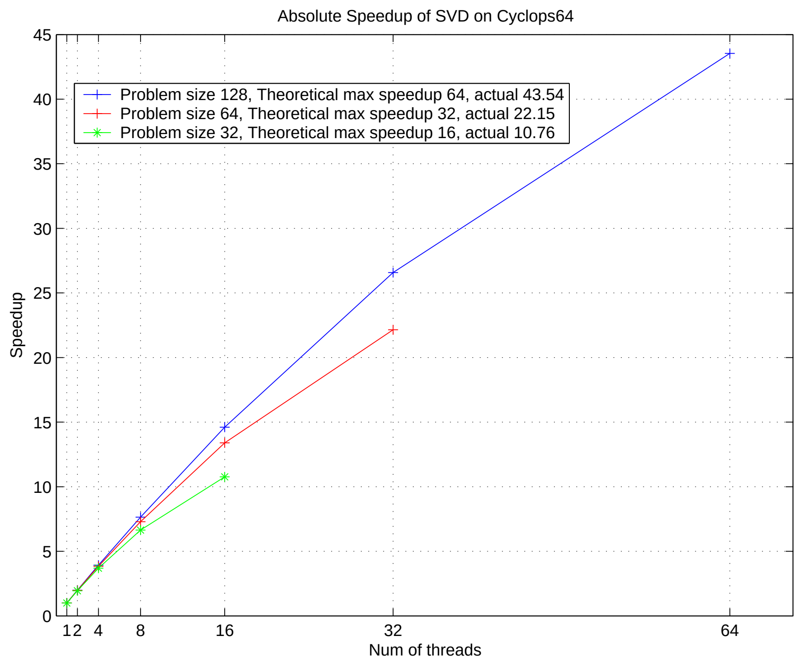

Figure 5: Speedup of parallel one sided Jacobi complex on C64

6.3 Fine level parallelization: parallel SVD on Cyclops64

In this section, the speedup result of the one sided Jacobi SVD on Cyclops 64 for complex matrix is reported. Figure.5 shows the speedup for the matrix size 128 by 128, 64 by 64 and 32 by 32. The

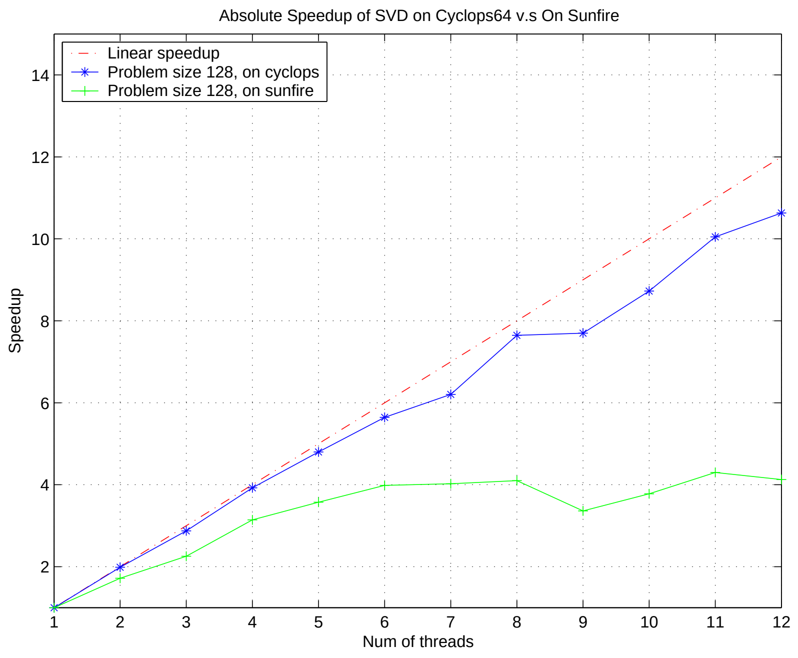

Figure 6: parallel SVD C64 v.s Sunfire

number in the matrix are uniformly random double precision number. According to GaoThomas parallel ordering, the maximum speedup for matrix n by n is $\frac{n}{2}$. In our experiment, for matrix size 128 by 128, we have measured the actual speedup of 43 which is around 68 %.

In Fig.6, we compare the performance of the complex SVD on Sunfire and on cyclops. From the figure, it can be seen that Cyclops64 shows much better performance even for this the small matrix size as 128. The actual biomedical data shows similar result and is not plotted due to the space limitation.

It is worth noting that Jacobi SVD is slower than other SVD algorithms, for the actual data with a matrix size 152 by 128,our implementation is about 2 times slower than ZGESVD in the CLAPACK package, which means, with at least 2 processors, the parallel SVD will be faster than ZGESVD.

7 Conclusions

The SPACE RIP technique uses multiple receiver coil and utilizes the sensitivity profile information from a number of receiver coils in order to minimize the acquisition time. In this paper, we focused on the parallel reconstruction of SPACE RIP. Firstly We analyzed the algorithm and identified one major loop as the program bottleneck to be parallelized. The loop level parallelization is implemented with Pthread , OpenMP and MPI and archived near linear speedup on Sunfire 12 CPUs SMP machine. Secondly, we analyzed the one sided Jacobi algorithm of SVD in the context of biomedical field and proposed a rotation method for complex matrix. One sided Jacobi algorithm for parallel complex SVD is implemented using the GaoThomas parallel ordering [14]. Thirdly, we ported the code to the new Cellular computer architecture C64, which makes SPACE RIP one of the first biomedical applications on C64. The speedup of the parallel SVD on Cyclops is shown to have achieved 43 for parallel SVD problem with matrix size 128 by 128. The performance comparison of C64 and Sunfire showed that the CThread of C64 has smaller overhead and is suitable for fine grain parallel application. Detailed time profiling of both Pthread and CThread primitives will be further explored in our future work. Lastly, The combination of loop level and fine level will generate even more speedup given sufficient number of processing unit, which will be further explored on the Cyclops64 platforms.

Further research directions include: (1) The utilization of the scratch pad memory and software caching strategy to further optimize the code on the C64. The performance effect of different compiler optimization level may also be explored. (2) We will continue on larger data set when faster version of the simulator becomes available. (3) We will explore whether different block Jacobi SVD algorithm may allow us to obtain better data locality and also better performance on larger data size.

8 Acknowledgments

We would like to thank Dr. Ziang Hu, Clement Leung for many advices and suggestions. We also thanks Fei Chen, Weirong Zhu, Juan B. del Cuvillo with the Cyclops experiment platform. They patiently answered all my questions regarding the Cyclops software tool chain. Juan B. del Cuvillo 's work on the Cyclops architecture helps me to write the section 5.2 and 5.3 . Many thanks to Dr. Walid Kyriakos and his student Jennifer Fan from Harvard Medical School for the sequential code of SPACE RIP and many good discussions about the SPACE RIP algorithm. Some formulas are adopted from [20], in order to keep consistency, similar notations are used. We also acknowledge the support from our sponsors. Also the email suggestions about complex SVD from Dr. Zlatko Drmac at University of Zagreb is highly appreciated.

References

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[2] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.

[3] Richard P. Brent and Franklin T. Luk. The solution of singular-value and symmetric eigenvalue problems on multiprocessor arrays. *SIAM Journal on Scientific and Statistical Computing*, 6(1):69–84, January 1985.

[4] Richard P. Brent, Franklin T. Luk, and Charles F. Van Loan. Computation of the singular value decomposition using mesh-connected processors. *Journal of VLSI and Computer Systems*, 1(3):242–260, 1985.

[5] David R. Butenhof. *Programming with POSIX(R) Threads*. Addison-Wesley Pub. Co., 1997.

[6] Calin Cascaval, Jos G. Casta nos, Luis Ceze, Monty Denneau, Manish Gupta, Derek Lieber, José E. Moreira, Karin Strauss, and Henry S. Warren Jr. Evaluation of a multithreaded architecture for cellular computing. In *HPCA*, pages 311–322, 2002.

[7] Rohit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan, and Jeff McDonald. *Parallel Programming in OpenMP*. Morgan Kaufmann, 2000.

[8] Juan B. del Cuvillo, Ziang Hu, Weirong Zhu, Fei Chen, and Guang R. Gao. CAPSL memo 55: Toward a software infrastructure for the cyclops64 cellular architecture. Technical report, CAPSL Group, Department of ECE, University of Delaware, 2004.

[9] J. Demmel. *Applied Numerical Linear Algebra*. SIAM, 1997.

[10] J. Demmel. Accurate SVDs of structured matrices. *SIAM J. Matrix Anal. Appl.*, 21(3):562–580, 2000.

[11] J. Demmel, M. Gu, S. Eisenstat, I. Slapnicar, K. Veselic, and Z. Drmac. Computing the singular value decomposition with high relative accuracy. *Linear Algebra Appl.*, 299:21–80, 1999.

[12] James Demmel and Kresimir Veselic. Jacobi's method is more accurate than QR. October 1989. Lapack Working Note 15 (LAWN-15), Available from netlib, http://www.netlib.org/lapack/.

[13] George E. Forsythe and Peter Henrici. The cyclic jacobi method for computing the principal values of a complex matrix. *Transactions of the American Methematical Society*, 94(1):1–23, 1960.

[14] G. R. Gao and S. J. Thomas. An optimal parallel jacobi-like solution method for the singular value decomposition. In *Proc. Internat. Conf. Parallel Proc.*, pages 47–53, 1988.

[15] G. Golub and C. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, 1996.

[16] M. Gu, J. Demmel, and I. Dhillon. Efficient computation of the singular value decomposition with applications to least squares problems. Technical Report CS-94-257, Computer Science Dept., University of Tennessee, Knoxville, 1994. LAPACK Working Note 88, http://www.netlib.org/lapack/lawns/lawn88.ps.

[17] E. R. Hansen. On cyclic jacoib methods. *Journal of Soc. Indust. Appl. Math.*, 11:448–459, 1963.

[18] M. R. Hestenes. Inversion of matrices of biorthogonalization and related results. *J. Soc. Induct. Appl. Math.*, 6:51–90, 1958.

[19] T. Koyama, K. Inoue, H. Hanaki, M. Yasue, and E. Iwata. Single-chip multiprocessor for audio and video signal processing. *IEEE J. Solid-State Circuits*, 36:17681774, Nov. 2001.

[20] Kyriakos WE, Panych LP, Kacher DF, Westin C-F, Bao SM, Mulkern RV, and Jolesz FA. Sensitivity profiles from an array of coils for encoding and reconstruction in parallel (SPACE RIP). *Magn Reson Med*, 44:(2):301–308, 2000.

[21] Haesun Park. A real algorithm for the hermitian eigenvalue decomposition. *BIT*, 33:158–171, 1993.

[22] Pruessmann KP, Weiger M, Boernert P, and Boesiger P. SENSE, sensitivity encoding for fast MRI. *Magn Reson Med*, 42:952–962, 1999.

[23] Dolors Royo, Miguel Valero-García, and Antonio González. Implementing the one-sided jacobi method on a 2d/3d mesh multicomputer. *Parallel Computing*, 27(9):1253–1271, August 2001.

[24] H. Rutishauser. The jacobi method for real symmetric matrices. In *J. H. Wilkinson and C. Reinsch, editors, Linear Algebra, Volumn II of Handbook for Automatic Computations, chapter II/1*, volume II, pages 202–211, 1971.

[25] Sodickson DK and Manning WJ. Simultaneous acquistion of spatial harmonics(SMASH): fast imaging with radiofrequency coil arrays. *Magn Reson Med*, 38:591–693, 1977.

[26] T. Sterling, D. Becker, and D. Savarese. BEOWULF: A parallel workstation for scientific computation. *Proceedings of the 1995 International Conference on Parallel Processing (ICPP)*, pages 11–14, 1995.

[27] J. H. Wilkinson. *The Algebraic Eigenvalue Problem, pp. 277-278*. Clarendon Press, Oxford, 1965.