FAST SUBSPACE TRACKING USING COARSE GRAIN AND FINE GRAIN PARALLELISM

Daniel Rabideau
USAF Rome Laboratory
Griffiss AFB, NY 13441, USA

Allan Steinhardt
MIT Lincoln Laboratory
Lexington, MA 02173, USA

ABSTRACT

Subspace tracking is an integral part of many high resolution adaptive array methods. Unfortunately, the high computational complexity and non-parallel nature of traditional subspace tracking algorithms have deterred their use in real-time systems. In this paper, we discuss parallel mappings of the Fast Subspace Tracking algorithm. The serial complexity of this algorithm is already among the lowest { $O(Nr)$ for N channels and an r dimensional subspace}. In this paper, we show that even greater reductions in effective complexity can be achieved by mapping our algorithm onto multiple processors. Near linear speedup is obtained on machines spanning the range from fine grain systolic arrays to coarse grain commercially available MPPs.

1. INTRODUCTION

Although subspace-based signal processing methods are heralded for their superior performance, their transition into real-world systems has been slow because of their tremendous computational burden. Because of this, researchers have mounted a two-frontal attack on the problem. On the one hand, they have posed alternative matrix factorizations that are capable of providing (approximate) subspace information [1], [2], [3] at a lower cost than traditional methods. These algorithms typically have complexities in the range of $O(N^2 r)$, $O(N^2)$, or $O(Nr^2)$. On the other hand, they have used parallel processing to achieve even greater reductions in execution time [3], [4].

Previously [5], we introduced a subspace tracking algorithm called FST which required only O(Nr) flops per update. In this paper, we will show how FST can be mapped onto parallel machines with widely varying characteristics while achieving high efficiency and near linear speedup. It will be shown that the algorithm can be partitioned for both coarse and fine granularity, and thus should be capable of high performance on virtually all commercially available multiprocessors. Tests of our mappings are performed on an Intel Paragon, iPSC/860, iPSC/2 and iWarp.

2. PROBLEM STATEMENT AND FST ALGORITHM SUMMARY

The subspace tracking problem for sensor array processing can be formulated as follows. Let $\mathbf{X}$ denote the $(k-1)\times N$ matrix whose rows contain snapshots from an N channel array of sensors at time $k-1$. Let Γ denote the diagonal damping matrix used to exponentially window the data. Suppose that the windowed data matrix is initially factored as:

$$\Gamma\mathbf{X} \cong \mathbf{U}\cdot\mathbf{R}\cdot\mathbf{V}^H.$$

where $\mathbf{U}$ is a $(k-1)\times(k-1)$ orthonormal matrix, $\mathbf{R}$ is the $(k-1)\times N$ block diagonal matrix $diag\left(\left[\mathbf{R}_s \;\; \bar{\sigma}_n\cdot\mathbf{I}\right]\right)$ with $r\times r$ upper triangular $\mathbf{R}_s$ and average noise singular value $\bar{\sigma}_n$, and $\mathbf{V}=\left[\mathbf{V}_s \,\vdots\, \mathbf{V}_n\right]$ contains orthonormal bases for the signal and noise subspaces. Such a factorization is called a noise-averaged URV decomposition. Note that $\mathbf{U}$ and $\mathbf{V}_n$ are not needed by the standard superresolution methods and thus do not need to be tracked. The problem then reduces to one of updating $\mathbf{R}_s$, $\mathbf{V}_s$, and the average noise power $\bar{\sigma}_n^2$ as new snapshots arrive.

Suppose at time k we obtain a new snapshot, $\mathbf{x}_k$. The FST algorithm performs the update in nine steps. They are as follows:

1) Compute the coefficients of the new snapshot in the signal subspace.

$$\mathbf{x}_s = \mathbf{V}_s^H\cdot\mathbf{x}_k$$

2) Project $\mathbf{x}_k$ into the noise subspace:

$$\mathbf{v}_n = \frac{\mathbf{x}_k - \mathbf{V}_s\cdot\mathbf{x}_s}{\beta}$$

$$\beta = \left\|\mathbf{x}_k - \mathbf{V}_s\cdot\mathbf{x}_s\right\|_2$$

3) Append the coefficients onto $\mathbf{R}$ and window:

$$\begin{bmatrix}\gamma(\Gamma\mathbf{X}) \\ \hline \mathbf{x}_k\end{bmatrix} = \left[\begin{array}{c|c}\mathbf{U} & \mathbf{0} \\ \hline \mathbf{0} & 1\end{array}\right]\cdot\left[\begin{array}{c|c|c}\gamma\cdot\mathbf{R} & & \\ \hline \mathbf{x}_s^H & \beta & \mathbf{0}\end{array}\right]\cdot\left[\mathbf{V}_s \,\vdots\, \mathbf{v}_n \,\vdots\, \mathbf{V}_n^{\perp}\right]$$

0-7803-2431-5/95 $4.00 © 1995 IEEE

Also, define:

$$\mathbf{R}_{append} \equiv \begin{bmatrix} \gamma \cdot \mathbf{R}_s & 0 \\ & \vdots \\ & 0 \\ 0 \ \cdots \ 0 & \gamma \cdot \bar{\sigma}_n \\ \mathbf{x}_s^H & \beta \end{bmatrix}$$

4) Use plane rotations to perform a QR-like update:

$$\mathbf{R}_0 = \mathbf{Q} \cdot \mathbf{R}_{append}$$

where $\mathbf{R}_0$ is upper triangular.

5) Compute an estimate, **w**, of the smallest singular vector of $\mathbf{R}_0$ (use a condition estimation algorithm).

6) Compute a sequence of rotations such that

$$\mathbf{Q}_1^H \cdot \mathbf{w} = [0 \ \cdots \ 0 \ \ 1]^T$$

Apply these rotations to $\mathbf{R}_0$ while rotating ($\mathbf{Q}_2$) so that:

$$\mathbf{R}_1 = \mathbf{Q}_2 \cdot \mathbf{R}_0 \cdot \mathbf{Q}_1$$

is upper triangular.

7) Choose rotations ($\mathbf{Q}_{1a}$) to zero the super diagonal elements of column $r+1$ of $\mathbf{R}_1$ (causing fill only in the last row). Then, perform another QR-like update to return the matrix to upper triangular form, i.e.,

$$\mathbf{R}_2 = \mathbf{Q}_{2a} \cdot \mathbf{R}_1 \cdot \mathbf{Q}_{1a}$$

8) Accumulate rotations in **V**:

$$[\mathbf{V}_s \vdots \mathbf{v}_n] \leftarrow [\mathbf{V}_s \vdots \mathbf{v}_n] \cdot \mathbf{Q}_1 \cdot \mathbf{Q}_{1a}$$

9) Sphericalize the noise subspace by zeroing the small super-diagonal entries of column $r+1$ of $\mathbf{R}_2$ and forming a weighted average of the resulting noise singular values.

The result is again a noise-averaged URV decomposition of a matrix which is close to the windowed data matrix. The performance of this algorithm is quite good when compared to other subspace tracking algorithms[5]. The complexity of this algorithm is only $O(Nr) + O(r^2)$.

3. PARALLEL MAPPINGS

In developing a parallel mapping of any algorithm, one must consider the issue of *granularity*. In a "fine" grain mapping, many small groups of tasks are individually assigned to the nodes of the parallel processor. This allows many independent tasks to execute concurrently, at the expense of potentially higher communication costs. In a "coarse" grain mapping, tasks are first gathered into a few large groups and then these groups are mapped onto the nodes of the parallel processor. Such an approach reduces the amount of communication required, but also may sacrifice some concurrency.

For a given computer, the choice between coarse and fine granularity depends on the relative speed at which data can be communicated between processors. On most commercial MPPs (e.g. Intel's Paragon, and iPSC/860, iPSC/2) communication delays for small messages are relatively high. Thus, coarse grain mappings work best. However, some machines (e.g. transputer arrays, VLSI systolic arrays, and Intel's iWarp) have been designed specifically to implement communication efficiently and are well suited to fine grain mappings.

Given that a certain computer operates best on mappings of a specific granularity, one must also choose an algorithm that partitions easily at that granularity. As with computers, algorithms too are often better suited to one granularity. Below, we consider ways to partition FST efficiently for both coarse and fine granularity.

3.1. Coarse Grain FST

Here, we seek a way to partition the FST algorithm so that large groups of operations may be performed concurrently, with a minimum of interprocessor communication. Furthermore, when interprocessor communication is to be used, we favor a few large messages rather than many small ones in order to reduce the cost of startup latency.

With this in mind, let us consider the cost of FST: $O(Nr) + O(r^2)$. The first term arises from operations performed on the $r+1$ dimensional subspace (steps 1, 2, 8), $[\mathbf{V}_s \vdots \mathbf{v}_n]$, whereas the second term arises from operations on the $(r+1)\times(r+1)$ principal submatrix of **R** (steps 3 - 7, 9). In typical sensor array processing applications, N is much greater than r. Thus, partitioning **V** across processors will result in a large reduction in execution time.

Next, observe that the operations involving **V** can be expressed as a sequence of column-oriented dot products, caxpys and rotations. Thus, we may achieve nearly ideal load balancing by assigning N/p rows of **V** (and their associated flops) to each processing element. Furthermore, since most microprocessor's FPUs operate best on unit stride data, the rows assigned to a processor

should be consecutive. This leads to the block-by-row mapping of **V** to the processing elements.

The remaining operations (on the smaller **R** matrix) are much more tightly coupled and are not well suited to parallelization on a coarse grain machine. Instead, these operations are assigned to a separate "host" processor. Fig. 1 illustrates the coarse grain mapping of FST.

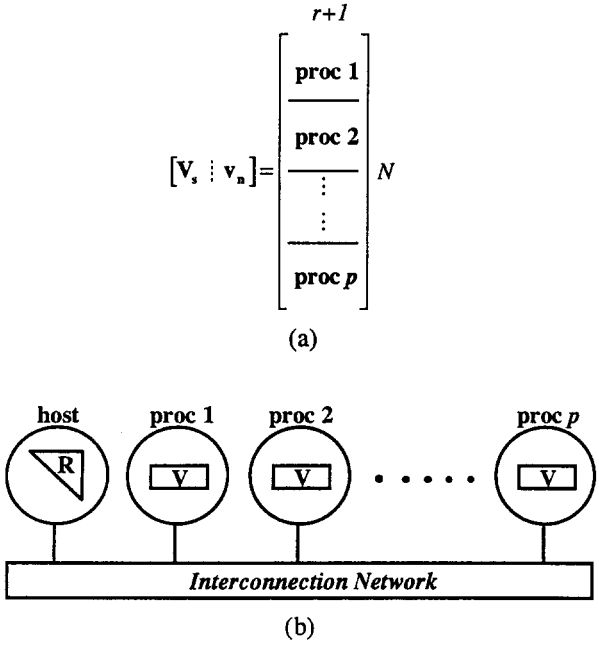


Fig. 1. Coarse Grain Mapping of FST.
(a) Partitioning of V. (b) Mapping of Data to Nodes

Fig. 2 shows the measured speedup using this coarse grain mapping on an Intel iPSC/860 (with unoptimized C code). Consider Fig. 2a. Comparing the one processor (host only) and two processor (host + single node) curves, we see that computations involving the **R** matrix have only a small effect on the overall execution time. Next, as we increase the number of processors from two to four we divide the operations on **V** among three processors. This results in a speedup of about three for all values of r -- ideal linear speedup. Next, with eight processors, the operations on **V** are partitioned among seven processors leading to a speedup of close to seven -- nearly linear. Now, as we continue to increase the number of processors, we start to observe the end of the linear speedup region. This is because as $r \rightarrow N/p$ the host will start to dominate the execution time. To prevent this degradation in performance, one must partition **R** across a small number of processors. However, in applications such as Space-Time Processing, N is often much larger than r (e.g. r approx. equal to $\sqrt{N}$) so this is not much of a problem for a reasonable number of processors.

Figs. 3 shows the performance of coarse grain FST on the Intel Paragon and iPSC/2. Note that the speedup curves are much alike despite the great variation in processing speed and I/O bandwidth. This demonstrates that the speedup and efficiency of our coarse grain mapping is relatively insensitive to the compute-to-I/O ratio of the target machine. Thus, it should map well onto any coarse grain machine.

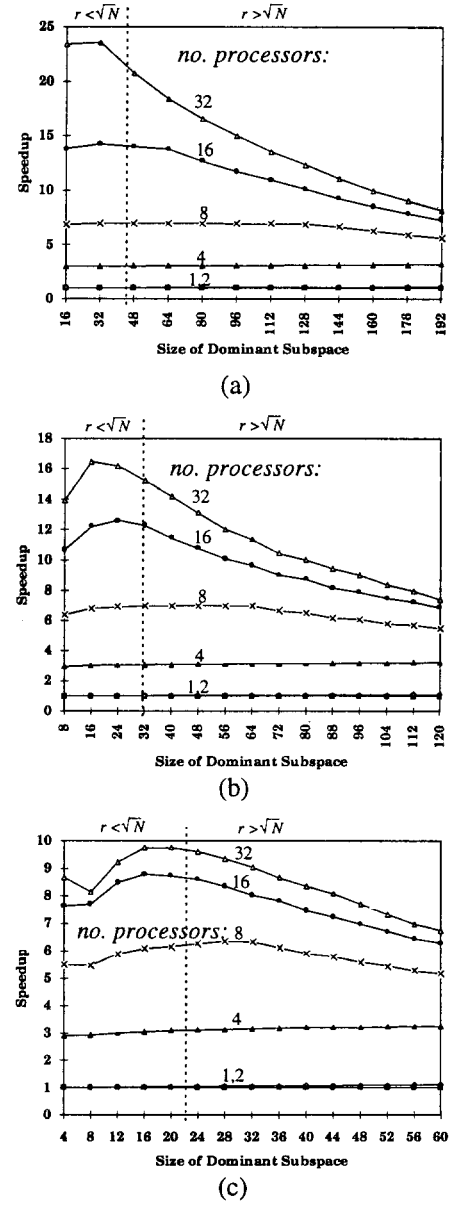
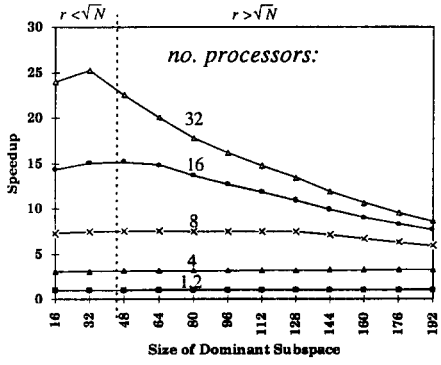
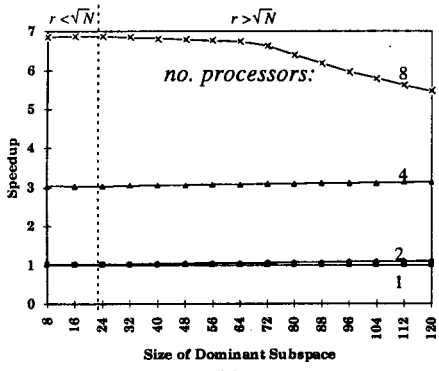


Fig. 2. Speedup obtained using the coarse grain mapping of FST on an iPSC/860 hypercube (a) N = 2000. (b) N = 1000 (c) N = 500



(a)



(b)

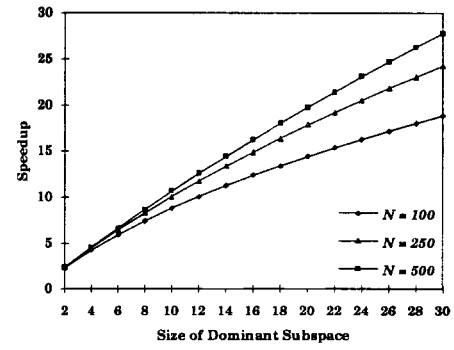
Fig. 3. Speedup obtained using coarse grain FST. (a) Paragon mesh, N = 2000 (b) iPSC/2, N = 500.

3.2. Fine Grain Mapping

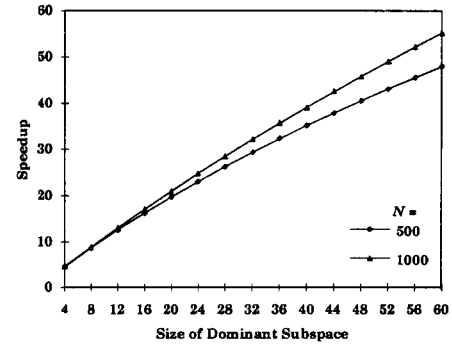
A small grain size allows more independent tasks to execute concurrently on separate processors, at the expense of greater communication delays. As such, fine grain mappings are appropriate for machines with low I/O-to-compute ratios. With regard to matrix-based signal processing algorithms, a single "fine grain" might consist of a small number of floating point operations, e.g., a single plane rotation or multiply-add.

The FST algorithm may be mapped onto a fine-grain, linear array of r+1 processors by assigning the i^{th} column of **R** (and **V**) to the i^{th} processor. In this way, the algorithm is implemented as sequences of plane rotations or CAXPY operations which are pipelined across the array.

Fig. 4 shows the speedup obtained using a fine grain mapping on an iWarp system. In this figure, note that the number of processors increases with the subspace dimension (i.e., scaled speedup). Thus, ideal linear speedup would consist of a straight line with slope one. We observe that speedup is nearly linear and improves as the problem size increases.



(a)



(b)

Fig. 4. Speedup obtained using fine grain FST on an iWarp system. N = 100, 250, 500 and 1000. ($p = r$+1).

4. SUMMARY

In this paper, we show that (for $N >> r$) the FST subspace tracking algorithm maps efficiently onto parallel processors having widely varying compute-to-I/O characteristics. Low computational complexity combined with efficient parallel mappings make FST a good candidate for implementation in real-time systems.

[1] P. Comon, and G. H. Golub, "Tracking a few extreme singular values and vectors in signal processing," Proc. IEEE, vol. 78, pp. 1327-1343, 1990.

[2] G. W. Stewart, "An updating algorithm for subspace tracking," *IEEE Trans. SP*, vol. 40, pp. 1535-1541, 1992.

[3] E. M. Dowling, L. P. Ammann, and R. D. DeGroat, "A TQR-iteration based adaptive SVD for real time angle and frequency tracking," *IEEE Trans. SP*, vol. 42, pp. 914-926, 1994.

[4] M. Moonen, P. Van Dooren, and J. Vandewalle, "Updating Singular Value Decompositions. A parallel implementation," *SPIE Advanced Algorithms and Archectures for Signal Processing IV*, pp. 80-91, 1989.

[5] D. J. Rabideau, and A. O. Steinhardt, "Fast subspace tracking," *Proc. 7th SP Workshop on SSAP*, June 1994.