CS 594 – Understanding Parallel Architectures:
From Theory To Practice

Projects

1 Reminder

You are welcome to suggest your own project as well.

2 Parallel matrix-matrix multiply

Matrix-matrix multiplication is a simple sequential kernel. However, its parallelization has motivated a great deal of research papers. Communication issues need to be addressed. Also, multiplying rectangular (non-square) matrices may lead to severe load imbalance problems. The objective of the work is to understand the best known algorithms and to make a nice synthesis of the existing literature.

3 Loop transformations

There are a great bunch of loop transformations in the literature: loop interchange, loop skewing, loop fusion, loop splitting, unimodular transformations, etc. The study will be based upon the book of Wolfe and the survey paper of Banerjee, Eigenmann, Nicolau and Padua.
The objective of the work is to come up with a nice survey of loop transformations. The scope, applicability and usefulness of the transformations should be made clear. Also, some synthesis efforts should be pursued; the report should not consist of a flat boring list of transformations !

4 Array redistribution

Run-time redistribution of arrays that are distributed in a block-cyclic fashion over a multidimensional processor grid is a difficult problem. The main motivation for the redistribution problem comes from the HPF programming style: scientific applications are decomposed into phases. At each phase, there is an optimal distribution of the data arrays onto the processor grid. Typically, arrays are distributed according to a `CYCLIC(r)` pattern along one or several dimensions of the grid. The best value of the distribution parameter r depends upon the characteristics of the algorithmic kernel as well as upon the communication-to-computation ratio of the target machine. Because the optimal value of r changes from phase to phase and from one machine to another (think of a heterogeneous environment), run-time redistribution turns out to be a critical operation.
There have been several papers dealing with the problem of efficient code generation for an HPF array assignment statement like

$$A[l_1 : u_1 : s_1] = B[l_2 : u_2 : s_2]$$

where both arrays A and B are distributed in a block cyclic fashion among a linear processor grid. The first papers have mostly dealt with arrays distributed using either a purely scattered or cyclic distribution (`CYCLIC(1)` in HPF) or a full block distribution (`CYCLIC(`$\lceil \frac{n}{p} \rceil$`)`, where n is the array size and p the number of processors). But recently, several algorithms have been published

to handle general block-cyclic `CYCLIC(k)` distributions. Sophisticated techniques involve finite-state machines, set-theoretic methods, diophantine equations, Hermite forms and lattices, or linear programming.
The objective of the work is to understand the algorithms and to make a theoretical and/or practical comparison of them.

5 Automatic blocking on shared memory machines

In class we emphasize the role of blocking on improving the performance of BLAS-2 and BLAS-3. This blocking is done at a higher level within the LAPACK software. For example, as we will discuss in class, Gaussian elimination can be rewritten so that the inner loop consists mostly of BLAS subroutine calls: solving an i by i triangular system with b right hand sides (i ranging from near 1 to nearly n), matrix-matrix multiplication ($n-i$ by i times i by b), as well as assorted BLAS-2 calls. In addition there are variations which call different BLAS on different size matrices. b is a block size to be chosen.
Thus, we have several degrees of freedom to optimize in designing a block version of Gaussian elimination. First, we need to decide which algorithm (i.e. which BLAS to call), and then we need to choose b (some algorithms have more than one block size to choose). There are at least two ways to try to do this.
The first way is static, or done before runtime. In this case, a setup program would run a few benchmarks, timing both BLAS for various matrix sizes as well as higher level algorithms (like Gaussian elimination) using various block sizes and combinations of BLAS. From these relatively few benchmarks, one could try to predict the best overall algorithm and blocksize, with that version being the final one installed. In the course of developing LAPACK, we have collected a great deal of this data, and so one could analyze it to see how well one could automatically optimize performance. The eventual goal would be an automatic LAPACK installation program for new architectures which would do far fewer benchmarks than we have done, and then choose the best algorithms and block sizes.
The second way is dynamic, or done at runtime. Here, the code would keep track of its performance as it goes, and possibly change the blocksize from step to step to optimize performance. This approach has advantages and disadvantages with respect to the static one. Its advantages are the extra flexibility of changing the block size (Bischof discusses this in several recent papers), and keeping code truly portable (typically numerical code is distributed in source form across machines, as with netlib, without any setup phase as needed in the static approach). Its disadvantage is in not being able to exploit truly different versions of the algorithm.
An interesting project would be to (1) analyze the existing data to produce performance predictors and optimal block sizes, (2) developing dynamic schemes (to date only QR decomposition has one), or (3) comparing the performance of both schemes.
There is literature available on this topic, as well as a great deal of on-line performance data.

6 Eigenproblems and singular value problems on distributed memory machines

The hope of the LAPACK 2 project is that BLAS based algorithms will again be sufficient to extract most of the parallelism of distributed memory machines, including those of widely differing architectures like the SP-2 the Intel paragon, and networks of workstations. Work is underway

elsewhere to attempt porting pieces of LAPACK to these machines, but little work has been done on the codes for eigenvalues/vectors and singular value/vectors. There are two kinds of projects available here: seeing how well existing LAPACK algorithms work on the machines, and exploring algorithms quite different from those used in LAPACK. The Intel and SP-2 are quite different, and a project could attempt to use both machines or just one.
The project deals with computing the eigenvalues of an upper Hessenberg matrix A. This is the hardest problem in LAPACK to parallelize. The existing algorithm (SHSEQR, based on EISPACK's HQR) only uses the BLAS on rather small matrices, since using larger matrices changes the numerical properties of the algorithm and seems to slow it down considerably. Porting this algorithm to a distributed memory machine and optimizing its performance would be quite useful. Quite recently three quite different approaches have been proposed, all of which are apparently much more parallelizable than SHSEQR. However, they have quite different numerical properties, not all of which are attractive:

1. Homotopy algorithm: The idea is to start with a different matrix A_0 whose eigenvalues and eigenvectors are known, and then gradually deform A_0 into A, deforming the eigenvectors and eigenvalues at the same time. In other words, if $A(t)$ is a "curve" connecting $A_0 = A(0)$ and $A = A(1)$, then the algorithm follows the curves of the eigenvalues $\lambda(t)$ of $A(t)$ from the known values $\lambda(0)$ to the desired values $\lambda(1)$. Each curve for each eigenvalue can be followed in parallel with all the other curves, so this algorithm appears quite parallelizable. Unfortunately, there is no guarantee the algorithm will not "mix up" these curves if they get close together, possibly computing multiple copies of some eigenvalues and no copies of others. One project would be to implement this, see how much faster than SHSEQR it can be made to run, and empirically study when it gets the right answers.

2. Arnoldi's method: This algorithm accesses the original matrix A (which does not even have to be Hessenberg) just by multiplying it by a vector (or several vectors), and so it is quite parallelizable. However, it is designed just to get small subset of the eigenvalues of A (in practice, one sometimes just wants a certain subset, such as the eigenvalues with largest absolute values or largest real part). Here, one would do a parallel implementation and compare its speed computing a subset of the eigenvalues with SHSEQR which must compute all of them.

3. Matrix-multiply based: This is a variation on the power method for finding eigenvalues in which almost all the work is done by matrix-multiplication. This makes it potentially quite fast, but the convergence rate may be quite slow. Again, one would compare an implementation with SHSEQR.

7 Extending the scope of LAPACK

There are a number of useful algorithms which we would like to include in LAPACK but have not yet done so. These include fast algorithms for the Sylvester and Lyapunov equations, algorithms to analyze singular pencils of matrices (these arise in control theory), and Jacobi's method for the symmetric eigenproblem. One project would be to do careful, fast implementations of some subset of these algorithms on one or more of the high performance machines available this semester.

8 Sparse linear equation solver

Solving the linear system $Ax = b$ where A is large, sparse, symmetric and positive definite is an important problem in many applications. Here we consider Cholesky, a variation on Gaussian elimination. This involves factoring $A = LL^T$ where L is lower triangular and then solving $Ly = b$ and $L^T x = y$. It is more difficult to exploit parallelism than for dense matrices because one must simultaneously exploit the sparsity structure of A, i.e. take advantage of all the zeros in A and (if one is clever) in L. One can represent the sparsity structure of A by its graph (which has nodes 1 through n and an edge from i to j if $A_{ij} \neq 0$), and the order in which one can do the elimination operations to compute L as a tree. Parallelism arises in processing leaves of the tree in parallel, and the challenge is to choose the order of the rows and columns of A to make the resulting tree as "bushy" as possible, so there is as much parallelism as possible. This part of the algorithm is graph theoretic in nature, and it is interesting to try to parallelize this graph algorithm as well. An interesting project would involve either parallelizing an existing code or trying to write all or part of such a code from scratch.