A 7-Step Approach to the Design and Implementation of Parallel Algorithms

THOMAS F. STECK and GERARD G.L. MEYER
Electrical and Computer Engineering
Johns Hopkins University
3400 N. Charles Street, Baltimore, MD 21218
USA
http://www.ece.jhu.edu/~pcil

Abstract: This paper presents a 7-step, semi-systematic approach for designing and implementing parallel algorithms. In this paper, the target implementation uses MPI for message passing. The approach is applied to a family of matrix factorization algorithms– LU, QR, and Cholesky – which share a common structure, namely, that the second factor of each is upper right triangular. The efficacy of the approach is demonstrated by implementing, tuning, and timing execution on two commercially available multiprocessor computers.

KeyWords: parallel algorithms, matrix factorization, LU, QR, Cholesky

1 Introduction

This paper presents a semi-systematic approach to enable development of new numerical algorithms for parallel computers. The approach targets algorithmic (task-level) partitioning, as opposed to data partitioning. This work is the evolution of a design process used by researchers at the Johns Hopkins University Parallel Computing and Imaging Laboratory over the past ten years. It has proven effective when faced with a particular class of problems where parallel processing is required to reduce execution time, yet existing libraries and algorithms fail to achieve targeted performance.

The organization of the paper is as follows: design goals for the new parallel algorithms are presented, then a seven step approach is given. Rather than expanding on the approach in abstract, it is illustrated by application to matrix factorization. LU factorization is developed first. Briefly, it is shown how redefinition of computational tasks enables the same algorithms to be used for Cholesky and QR factorizations. The factorizations have been implemented on commercially available multiprocessors systems. Efficacy of the approach is demonstrated by comparing the execution time for a sample problem to a common vendor-tuned library available on two systems.

[1]This work was supported in part by a grant of HPC time from the DoD HPC Centers at Aberdeen Proving Ground, Maryland and Space and Naval Warfare Systems Center, San Diego, CA.

2 Goals

Two sets of goals are provided, the first are aimed at the 7-step approach introduced, whereas the second guide the development of the new parallel matrix factorization algorithms.

- Development Approach Goals
  - Provide a systematic approach that takes a solution method and develops parallel algorithms that execute faster than existing algorithms
  - Reduce the time and effort required to develop and implement new parallel algorithms
  - Capitalize on existing algorithms and code
  - Maintain sufficient flexibility to produce multiple "solution-equivalent" algorithms
  - Assist in the identification and inclusion of useful algorithm parameters
  - Capture sufficient information about the process and decisions to enable application to similar problems
- Parallel Algorithm Goals
  - **Execute faster than competing implementations**
  - Algorithm parameters that maximize performance on target machine when tuned properly
  - *Maintain numerical properties of selected solution method or algorithm*
  - *Integration of data distribution, computation, and result collection*

In the above, the items listed in *italics* may not be necessary or even desirable for other problems. It is still possible to use the following approach for developing parallel algorithms and implementations.

3 An Approach for Developing and Implementing Parallel Algorithms

Step 1: Select a solution method
- Textbook
- Numerical recipe
- Existing (preferably optimized) serial code

Step 2: Develop a visualization of the problem
- Identify computational tasks
- Generate a dependency graph

Step 3: Serial code development
- Matlab version– functional verification of tasks and dependencies
- C version– numerical verification and profiling on target system

Step 4: Identify possible distributions of data and computation to processors

Step 5: Determine how a chosen distribution specifies computational order and communication

Step 6: Describe the parallel algorithm(s)

Step 7: Parallel code development and implementation
- Matlab version (serial with parallel assignment and ordering)– functional verification, enables assignment of processor tasks prior to insertion of message passing
- C version with MPI– numerical verification, profiling, and tuning to target system

4 Application of Approach to LU Factorization

Problem statement

Given a nonsingular $m \times m$ matrix A, factor such that $A = PLU$ (L is $m \times m$ and unit lower triangular; U is $m \times m$ upper triangular; and P is an $m \times m$ permutation matrix).

Step 1: Select a solution method

A high level text description of a solution method can be obtained by comparing either textbook descriptions, such as one would find in Golub and Van Loan's *Matrix Computations* [3], using a numerical recipe, such as those found in Numerical Recipes in C [5], or by analyzing the serial code of a numerical library like LAPACK [1]. The LAPACK routines are written in Fortran 77, but can be readily called from C. From the perspective of developing a parallel algorithm, the most attractive part of LAPACK is that it is built on BLAS [2] and LAPACK auxiliary routines that are written to extract improved performance from the processor registers and cache hierarchy. The actual LAPACK routine selected is *dgetrf*, which performs LU factorization with partial pivoting on double precision floating point data.

Step 2: Develop a visualization of the problem

A short text description of the algorithm aids in visualization. In the case of LU factorization, computation starts on the first row, with the upper left diagonal element and data for the first column being updated first. Next, the elements to the right are updated, and then updates are propagated through the remainder of the matrix. The process repeats for the next row, starting again with the diagonal element, followed by updates to the right and then down through the remainder of the matrix. The process terminates when the bottom right element is computed.

Further refinement of the problem visualization can be seen by examining figure 1, which illustrates a single stage of the update process and the division of the updates into block columns of width h.

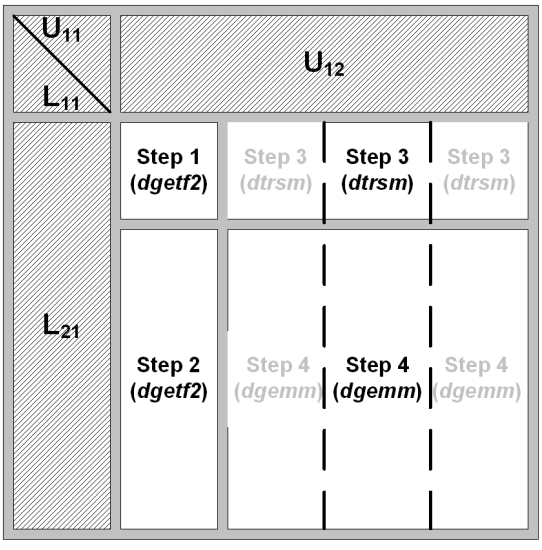

Figure 1: Intermediate LU update.

Each of the hashed areas represent the data from matrix A that has already been factored. The un-

shaded regions are currently being updated using BLAS or LAPACK subroutines shown in *italics*. *dgetf2* is the serial routine to determine the LU factorization with partial pivoting of the submatrix indicated. This is shown as two steps, but could be performed as a single step or as many smaller steps. Pivot information is also returned by *dgetf2* and must be applied to the remainder of the matrix. Using the factors determined in step 1, the remaining row is updated using backward substitution by calling the BLAS routine *dtrsm* in step 3. The updated top row and the updated column in step 2 are used to update the remaining submatrix in step 4, which performs a matrix-matrix multiply using the BLAS routine *dgemm*. The dashed lines divide steps 3 and 4 into block columns, providing finer computational granularity.

Using figure 1 as a model, two computational tasks are defined. The first task comprises steps 1 and 2, and the second task comprises steps 3 and 4. It should be noted that this is only one possible choice.

Two definitions assist in the development of the algorithm.

- h – Computational block size
- $M = \lceil m/h \rceil$ – Number of block rows
- $N = \lceil n/h \rceil$ – Number of block columns

The following notation is used to represent a submatrix of A. $A_{i,j}$ indicates the i'th block row and j'th block column of A. Explicitly, these are the elements of A, $\{a(r,s)|r \in (i-1)\cdot h+1, \ldots, min(m, i\cdot h);\ s \in (j-1)\cdot h+1, \ldots, min(n, j\cdot h)\}$. The colon symbol is used to represent a collection of elements, so $A_{i:M,j} \equiv A_{i,j}$, $A_{i+1,j}, \ldots, A_{M,j}$.

Task $T_1(i)$ - LU factorization of a block column $A_{i:M,i}$

- Given $A_{i:M,i}$, compute the lower unit trapezoidal matrix $L_{i:M,i}$ and the upper triangular matrix $U_{i,i}$ with permutation matrix $P_{i,i}$ such that $A_{i:M,i} = P_{i,i}L_{i:M,i}U_{i,i}$, where $P_{i,i}$ is represented by $IPIV(i)$ (*dgetf2*)
- Given $IPIV(i)$, perform pivots on $A_{i:M,1:i-1}$ (*dlaswp*)

Task $T_2(i,j)$ - Calculation of $U_{i,j}$ and update of $A_{i+1:M,j}$

- Given $IPIV(i)$, perform pivots on $A_{i:M,j}$ (*dlaswp*)
- Given $L_{i,i}$ and $A_{i,j}$, compute $U_{i,j}$ such that $L_{i,i}U_{i,j} = A_{i,j}$ (Backward Substitution - *dtrsm*)
- Given $L_{i+1:M,i}, U_{i,j}$, and $A_{i+1:M,j}$, compute $A_{i+1:M,j} = A_{i+1:M,j} - L_{i+1:M,i}U_{i,j}$ (*dgemm*)

The number of tasks varies depending on the size of the matrix and the computational block size. It is now possible to construct a dependency graph based on the tasks, or more accurately, it is possible to describe the form of the dependency graph. The actual dependency graph necessarily changes depending on the number of block rows and columns. Figure 2 is the dependency graph for a matrix with six block columns and rows.

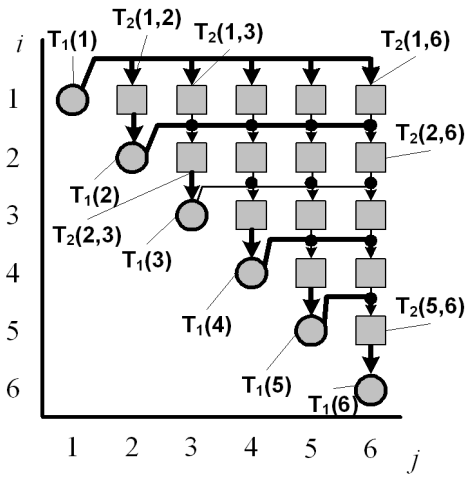

Figure 2: Dependency Graph

Step 3: Serial code development

Provided a dependency graph, the developer must then determine how to write the code. Usually, there are a number of potential paths, and the programmer must decide whether each path is captured in the code or not. Matlab is recommenced for initial development of serial code because the computation and sequencing can be developed without explicating data types and memory allocation. It is typically easier to verify the results using Matlab built-in functions than trying to perform verification directly in a high level language like C. Once the Matlab code is written, a functionally equivalent C code is developed on the target platform where verification against the Matlab code and performance profiling can take place. By obtaining execution times directly on the target platform, it may be possible to eliminate certain paths through the dependency graph or limit the granularity of tasks so that the process of assigning data and

tasks to processors becomes more manageable. The question to keep in the back of one's mind is when does the time spent sending data between processors exceed time that could be spent performing computation locally?

Step 4: Identify possible distributions of computation and data to processors

The assignment of tasks to processors chosen is diagonally oriented. In the chosen context, the first processor, P_0, initially has the entire matrix A in memory and performs the computations for all $T_1(j)$. To simplify programming, cyclic distribution is not permitted. Therefore, each processor is assigned a quantity of super-diagonals as shown in figure 3. Note that to compute each diagonal entry after $T_1(1)$, the updated data computed by $T_2(i-1,i)$ is needed. The consequence is that when the final $T_1(i)$ is computed, the results are entirely contained on processor P_0. Each task of the first row is assigned to one of the processors. The entire column of data is updated by the associated processor, establishing the initial data distribution, as shown in figure 4.

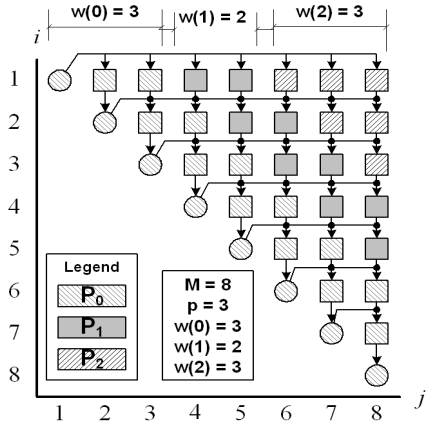

Figure 3: Computation Distribution

Step 5: Determine how a chosen distribution specifies computational order and communication

Observe how the work distribution shown in figure 3 defines a set of possible computation orders and communication requirements. First, P_0 must compute $T_1(1)$. After this, $T_2(1,j)$ can be computed in any order, but the results from $T_1(1)$ must be locally available first. This implies that P_0 sends results of $T_1(1)$ to P_1 and P_2 as soon as possible. The task $T_1(2)$ can begin only after $T_1(1)$ and $T_2(1,2)$ are complete and the results available. The remainder of

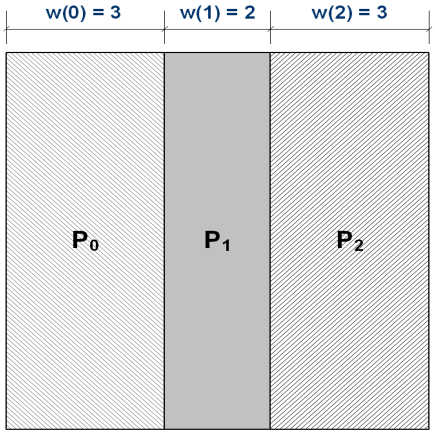

Figure 4: Initial Data Distribution

the tasks in the row, $T_2(2,j)$ depend on $T_1(2)$ and $T_2(i-1,j)$. If either dependency is performed on another processor, communication must be performed.

Step 6: Describe the parallel algorithm

In the algorithm that follows, p processors are assigned to compute the factorization. Assignment of tasks to processors is guided by a set of work distribution parameters $w(j)$, where j is the processor index. A few additional parameters and definitions will aid in writing the new parallel algorithm.

- $R(i), i \in 1,2,\ldots,M$ – Data of the i'th block row.
  - $R(i) = \{a(j,k) | \forall j \in (i-1)\cdot h+1,\ldots,\min(i\cdot h, m); \forall k \in 1,\ldots,n\}$
- $C(j), j \in 1,2,\ldots,N$ – Data of the j'th block column.
  - $C(j) = \{a(i,k) | \forall i \in 1,\ldots,m; \forall k \in (j-1)\cdot h+1,\ldots,\min(j\cdot h, n)\}$
- $IPIV(i), i \in 1,2,\ldots,M$ – Pivot information of the i'th block row.

Parallel LU Factorization Algorithms

Given an $m \times n$ matrix A stored in the memory of a processor P_0, the following steps perform LU factorization with partial pivoting, concluding with the results completely contained on processor P_0.

Setup

1. Choose a block size h.
   (a) Define tasks $T_1(i), \forall i \in 1,\ldots,M$.
   (b) Define tasks $T_2(i,j), \forall i \in 1,\ldots,M, \forall j \in i+1,\ldots,N$.
   See Step 2 for task definitions.
2. Choose the number of processors p
3. Choose width parameters $w(0), w(1),\ldots,w(p-1)$ such that $\Sigma w(j) = N$

(a) Processor P_0 is assigned tasks $T_1(i)$ for $i \in 1, \ldots, M$

(b) Processor P_r is assigned task $T_2(i,j)$ if $\sum_{s=0}^{r-1} w(s) + i \leq j < \sum_{s=0}^{r} w(s) + i$

Data Distribution and Computation

Note: The following steps are the actions taken on each processor $P_0, \ldots, P_{p-1}$. The local processor index is r.

1. Initialize counters
   (a) Set block row counter $i = 0$
   (b) If not P_0, set block column counter $j = \sum_{s=0}^{r-1} w(s)$
   (c) Set last processor index $\bar{r} = p - 1$
2. Perform initial data distribution
   (a) If P_0, send $C(\sum_{s=0}^{r-1} w(s) + 1), \ldots, C(\sum_{s=0}^{r} w(s))$ to P_r
   (b) Else P_r, for $r \in 1, \ldots, p-1$, receives $C(\sum_{s=0}^{r-1} w(s) + 1), \ldots, C(\sum_{s=0}^{r} w(s))$ from P_0
3. Update counters
   (a) Update block row counter $i = i + 1$
   (b) If $i > \sum_{s=\bar{r}}^{p-1} w(s)$, set $\bar{r} = \bar{r} - 1$.
   (c) If P_r, for $r \in 1, \ldots, \bar{r}$, update block column counter $j = j + 1$
4. Perform computations for block row i

   If P_0
   (a) If $i \neq 1$ and $i \leq N - w(0)$, receive $C(i + w(0) - 1)$ from P_1
   (b) Compute $T_1(i)$
   (c) If $\bar{r} \neq 0$, send $C(i)$ and $IPIV(i)$ to $P_1, \ldots, P_{\bar{r}}$
   (d) If $w(0) > 1$, compute $T_2(i, i+1), \ldots, T_2(i, i + w(0) - 1)$

   If $P_r, r \in 1, \ldots, \bar{r}$
   (a) Receive $C(i)$ and $IPIV(i)$ from P_0
   (b) If $i \neq 1$ and $P_r \neq P_{\bar{r}}$, receive $C(j + w(r) - 1)$ from P_{r+1}
   (c) Compute $T_2(i,j)$
   (d) Send $C(j)$ to P_{r-1}
   (e) If $P_r \neq P_{\bar{r}}$, compute $T_2(i, j+1), \ldots, T_2(i, j + w(r) - 1)$
   (f) Else if $P_r = P_{\bar{r}}$ and $j + 1 \leq N$, compute $T_2(i, j+1), \ldots, T_2(i, N)$
5. If $i < M$, go to step 3.

Step 7: Parallel code development and implementation

It is now possible to implement the parallel algorithm on the target platform. As in the serial case, the authors prefer to first use Matlab to polish the details of the parallel algorithm. It is worth noting that Jeremy Kepner of MIT Lincoln Laboratories has developed a parallel version of Matlab that uses file I/O as a means of transferring messages using MPI-like instructions [4]. The use of Matlab at this juncture is for functional verification prior to message passing. A standard serial Matlab program is written, but separate memory variables are created for each processor.

The Matlab code must now include data distribution (based on the width parameters $w(j)$) and generation of indices for processor task and data assignment. The parallel processing is performed by looping through each processor's computations row by row. Processing starts with P_0 completing task $T_1(1)$. The updated data must be sent to the rest of the processors. In Matlab, this is just a copy of data between separate memory variables. Using the initial data assignment, P_0 computes assigned type 2 tasks within the current row. Next, P_2 receives the updated data from P_0, and then it computes the type 2 tasks assigned to it in the first row. This continues for each of the processors until all of the type 2 tasks in the row have been computed. The process repeats for each block row of the matrix.

The C code developed in step 3 serves as a starting point for the parallel implementation. MPI initialization is added to the code so that it can be executed on multiple processors. Wherever there was memory copied between processors in the Matlab code, an MPI send and receive pair are needed. At this point, blocking MPI communications is appropriate; however, non-blocking MPI calls can be added after functional verification. Non-blocking MPI communications should follow two general rules: 1) initiate communications as early as possible *(MPI_Isend / MPI_Irecv)* , and 2) complete communications as late as possible– just in time for computation *(MPI_Wait)*.

The implementation can now be tuned to the target system. In this case, for a given $m \times n$ matrix, the parameters that require tuning are h, p, and $w(0), w(1), \ldots, w(p-1)$.

5 Extension to QR and Cholesky

Most of the previous application of the approach remains the same for QR and Cholesky factorization.

The key is in the following task definitions that appear in step 2 of the approach. They are based on the major subroutines in LAPACK's *dgeqrf* and *dpotrf* for QR and Cholesky, respectively.

QR factorization

Task $T_1(i)$ - QR factorization of a block column $A_{i:M,i}$ and determination of block reflectors $\mathrm{T_i}$

- Given $A_{i:M,i}$, compute the lower unit trapezoidal matrix $Y_{i:M,i}$ and the upper triangular matrix $R_{i,i}$, and the column vector β_i such that $Q_{i:M,i}R_{i,i} = A_{i:M,i}$. Formulation of Q from β and Y is found in the references [1]. (*dgeqr2*)
- Given $Y_{i:M,i}$ and β_i, compute the block reflectors $\mathrm{T_i}$. (*dlarft*)

Task $T_2(i,j)$ - Update of $A_{i+1:M,j}$

- Given $Y_{i+1:M,i}$, $\mathrm{T_i}$, and $A_{i:M,j}$, apply block reflectors $\mathrm{T_i}$ to update $A_{i:M,j}$. (*dlarfb*)

Cholesky factorization

Task $T_1(i)$ - Cholesky factorization of a block column $A_{i:M,i}$

- Given $A_{i,i}$ and $U_{1:i-1,i}$, compute $A_{i,i} = A_{i,i} - U_{1:i-1,i}^T U_{1:i-1,i}$ (*dsyrk*)
- Given $A_{i,i}$, compute the upper triangular matrix $U_{i,i}$ such that $A_{i,i} = U_{i,i}^T U_{i,i}$. (*dpotf2*)

Task $T_2(i,j)$ - Calculation of $U_{i,j}$ and update of $A_{i+1:M,j}$

- Given $A_{i,j}, U_{1:i-1,i}$, and $U_{1:i-1,j}$, compute $A_{i,j} = A_{i,j} - U_{1:i-1,i}^T U_{1:i-1,j}$ (*dgemm*)
- Given $A_{i,j}$ and $U_{i,i}$, compute $U_{i,j}$ such that $U_{i,i}^T U_{i,j} = A_{i,j}$ (Backward Substitution - *dtrsm*)

These definitions are directly related to the BLAS and LAPACK auxiliary routines listed above, and so credit again needs to go to the people that developed the routines [2, 1]. The common dependency graph implies the same task and data assignment, as well as the same computational order and communication.

6 Experimental Results

The algorithms were implemented and tuned on two commercially available target systems available from IBM and HP. Since the intent is to demonstrate the approach and not compare the systems, they are labeled as machines X and Y in the chart below. For the test, a 1500×1500 matrix was factored using newly developed algorithms for LU, QR, and Cholesky. The best times obtained using ScaLAPACK equivalent routines on these systems are reported. All times are obtained using *MPI_Wtime*.

Table 1: New Algorithm Performance (Times in milliseconds)

	Machine X	Machine Y
JHU LU	372	174
ScaLAPACK LU	366	180
JHU QR	266	233
ScaLAPACK QR	570	382
JHU Cholesky	192	142
ScaLAPACK Cholesky	480	157

The new algorithms perform favorably, providing a measure of merit for the approach.

7 Conclusions

This paper presented a semi-systematic 7-step approach to developing and implementing numerical algorithms for multiprocessor computers. While not automatic, the approach reduces time and effort to develop and implement new parallel algorithms, exploiting existing serial code and capturing sufficient information to enable application to similar problems.

References:

[1] Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., Croz, J. D., Greenbaum, A., Hammarling, S., McKenney, A., Ostrouchov, S., and Sorensen, D. *LAPACK Users' Guide, Second Edition*. SIAM Publications, 1995.

[2] Gallivan, K., Jalby, W., and Meier, U. The use of BLAS3 in linear algebra on a parallel processor with a hierarchical memory. *SIAM J. Sci. Comput. 8*, 6 (1987), 1079–1083.

[3] Golub, G. H., and Van Loan, C. F. *Matrix Computations*, third ed. The Johns Hopkins University Press, Baltimore, M.D., 1996.

[4] Kepner, J. Parallel programming with MatlabMPI. In *Proceedings of the High Performance Embedded Computing (HPEC 2001) workshop, MIT Lincoln Laboratory* (Lexington, MA, September 2001).

[5] Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. B. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, New York, NY, 1992.