HPCS Library Study Effort

Jack Dongarra[1,2,3], James Demmel[4], Parry Husbands[5], Piotr Luszczek[6]

[1]University of Tennessee

[2]Oak Ridge National Laboratory

[3]University of Manchester

[4]University of California Berkeley

[5]Lawrence Berkeley National Laboratory

[6]MathWorks

HPCS Library Study Effort

Jack Dongarra, James Demmel, Parry Husbands, Piotr Luszczek

1. Overview

In this report we present our research into the implementation of numerical libraries using the proposed HPCS languages. Faced with the fact that the community has very little application experience (the implementations are not yet mature) with these languages, we chose a somewhat atypical approach: perform a case study of parallel LU factorization and determine how this kernel can be implemented in the languages. As such we decided to gather various algorithmic techniques that have been successful and make connections to specific HPCS language features.

We settled on parallel LU factorization for a variety of reasons:

- It is a well known, understandable kernel
- Many implementations exist that span the performance spectrum
- Getting it to perform well in parallel on distributed memory machines reveals many programming issues, solutions to which aren't well represented in traditional languages.
-

In Section 2 we give a short description of the algorithm and outline some of the roadblocks to high performance. Section 3 presents some of the abstraction issues that arise when comparing the implementation of different versions of the algorithm in different languages. Section 4 contains our survey of the implementations. We detail our observations regarding implementing a high performance LU code in an HPCS language in Section 5 and conclude in Section 6.

2. LU Factorization and its Implementation Challenges

LU factorization attempts to decompose a general matrix A into a unit lower triangular (L) and upper triangular matrix (U). Row permutations are typically used for numerical stability and so a permutation matrix (P) is also generated such that $LU=PA$. The basic algorithm for this is shown below, assuming a square n x n matrix A:

for *i = 1* to *n-1*

- find maximum absolute element in column *i* below the diagonal
- swap the row of maximum element with row *i*
- scale column *i* below diagonal by *1/A(i,i)*
  L(i,i)=1
  for *j = i+1* to *n*
    L(j,i)=A(j,i)/A(i,i)
- Set row *i* of *U*
  for *j = i* to *n*
    U(i,j)=A(i,j)
- Perform a "trailing matrix update", i.e. update the part of the matrix below and to the right of *A(i,i)*
  for *j=i+1* to *n*
    for *k = i+1* to *n*
      *A(j,k) = A(j,k)-L(j,i)*U(i,k)*

  This step can equivalently be expressed as a "rank-one update":
  A(i+1:n,i+1:n) = A(i+1:n,i+1:n) -
  *L(i+1:n,i)*U(i,i+1:n)*

In order to achieve high performance through the use of BLAS-3 (matrix-matrix) operations, implementers usually express the algorithm in block form. Challenges to high performance in a parallel setting include management of the following:

- Communication for the row exchanges, updates to L and U, and the trailing matrix updates
- The dependencies in the algorithm

At this point it is interesting to note that sometimes the abstractions provided by a particular environment might inhibit optimization possibilities. A primary example of such inhibition is the set of design decisions that lead to the creation of the ScaLAPACK library.

The ScaLAPACK library implementers focused on two primary aspects of large scale parallel computing: scalability and portability. The former was addressed by the choice of appropriate parallel data organization and use of established parallel algorithms that could be proven to scale on distributed memory computers. However, the latter aspect reduced the available optimizations to a subset that can be implemented on major variants of parallel hardware. Consequently, the ScaLAPACK code employs a lock-step method that is characterized by heavy synchronization and lack of overlap of communication and computation in the temporal sense (in the spatial sense there exists some overlap as some of the processors are computing while others are communicating data between each other). As a result, ScaLAPACK is easily ported on any existing parallel platform, but its performance can be easily matched and often exceeded by codes targeted at a specific architecture.

3. Mapping to languages & Software Metrics

In this Section we discuss how we developed metrics that guide us through implementations in languages at differing levels of abstraction, the key criticism leveled against using source lines of

code (SLOC). In the survey to follow we augment traditional SLOC counts with an indication of the various helper abstractions that were used. These abstractions can either be serial or parallel. In the serial case we primarily have matrix abstractions: use of the familiar "triplet" notation for indexing, built-in matrix operators (\, for example, in Matlab), and "advanced" object oriented features. In addition, we assume that uniprocessor BLAS are provided. The parallel space is more diverse. Languages can provide some subset of any of the following:

- First class distributed arrays
- A global address space
- Data parallelism
- Multithreading
- Atomic transactions
- Advanced synchronization (single/sync variables, clocks, etc.)
- Parallel Matrix Abstractions such as the PBLAS and BLACS.
-

For those implementations that are concerned with high performance we also measure the best performance attained (absolute and % of peak), the number of processors on which this was measured (an indication of scalability) and, where available, uniprocessor performance (which tells us something about parallel overheads).

4. Survey of implementations

It is of course arguable how representative such codes are, but the fact that we can easily obtain versions of this algorithm for current and future languages are of interest to HPCS.

We present our findings in Table 1 below. Description of the columns of the table

1. **Language:** The main language used for the implementation
2. **Author**: the person who wrote the code
3. **Method**: method used to factorize
    a. Vectorized (calling BLAS 1)
    b. Blocked (calling BLAS 3)
    c. Recursive
    d. Parallel
    e. 1-D, 2-D
    f. Local factorization variants...
    g. Library-based (calling optimized library, perhaps written in a different language)
4. **Pivoting**: is partial pivoting done?
5. **Blocking**: are blocked calls to BLAS made?
6. **Driver**: is driver code included with matrix generation, etc?
7. **SLOC**: number of lines in editor (excluding large blocks of comments)
8. **Distribution**: parallel distribution type (or 0-D for sequential codes)
9. **Lookahead**: Can the code overlap panel factorizations with trailing matrix updates?
10. **Dist**. **Mem?**: Can this code run on distributed memory machines?

11. **Reuse L,U**: Can L and U be reused for further solves after the factorization is complete?
12. **Features**: Any other important features of the code. For example, examples suitable for teaching purposes are marked as "simple".

Language	**Author**	**Method**	**Pivot-ing**	**Block-ing**	**Driver**	**SLOC**	**Dist**	**Look-ahead**	**Dist. Mem?**	**Reuse L,U**	**Features**
MATLAB	Cleve Moler	Outer product, row-wise	Yes	No	No	37	0-D	No	No	Yes	Simple
Octave	Jason Riedy	Recursive	Yes	Yes	No	130	0-D	No	No	Yes	Algorithm by Sivan Toledo
Python	Piotr Luszczek	Outer product	Yes	No	No	40	0-D	No	No	Yes	Simple
Python	Piotr Luszczek	Outer product	Yes	Yes	No	95	0-D	No	No	Yes	Library
CAF	Robert Numrich	Outer product	Yes	No	Yes	1000	2-D	No	Yes	Yes	Simple, long
CAF	John Reid	Outer product	Yes	Yes	Yes	200	1-D	No	Yes	Yes	Simple
CAF	Robert Numrich	Outer product	Yes	Yes	Yes	120	2-D	No	Yes	Yes	CafLib, SLOC 9222
UPC	Parry Husbands	Outer product	Yes	Yes	Yes	5100	2-D	Yes (Dynamic)	Yes	U, not L	Fast
UPC	Calin Cascaval	Outer product	Yes	Yes	Yes	536	2-D	No	Yes		Simple
X10	Vivek Sarkar	Outer product	Yes	No	Yes	167	2-D	No (?)	Yes*	Yes	Simple
Chapel	Brad Chamberlain	Outer product, row-wise	Yes	No	No	40	0-D	No (?)	Yes*	Yes	Simple
Fortress	Guy Steele, Jan Willem-Massen	Outer-product, row-wise	Yes	No	Yes	100	0-D	No (?)	Yes*	Yes	Simple
HPF	M. Nakanishi	Outer product	Yes	No	No	70	1-D	No (?)	Yes	Yes	Simple
HPF	Anotine Petitet	Outer product	Yes	Yes	Yes	25	2-D	No (?)	Yes	Yes	Library
LINPACK	Cleve Moler	Outer product, vectorized	Yes	No	No	60	0-D	No	No	Yes	dgefa
LAPACK	LAPACK	Outer	Yes	Yes	No	100+	0-D	No	No	Yes	Dgetrf dgetf2

	team	product				100					
ScaLAPACK	Antoine Petitet	Outer product	Yes	Yes	No	180+ 140	2-D	No	Yes	Yes	PDGETRF PDGETF2
HPL	Antoine Petitet	Outer product	Yes	Yes	Yes	5000+	2-D	Yes (Static)	Yes	U, not L	
Titanium	Simon Yau	Outer product	No	Yes	Yes	388		No	Yes		
C	PLASMA team	Outer product	Yes	Yes	Yes	400	2-D	Yes (Dynamic)	No	Yes	Multithreaded
C	Panziera and Baron	Outer product	Yes	Yes	Yes		2-D	Yes (Dynamic)	Yes	U, not L	Multithreaded (up to 512p) + MPI
Cilk	Bradley Kuszmaul	Recursive	Yes	Yes	Yes	266	0-D		No		Multithreaded

Table 1. Findings

Because the level of abstraction varies wildly among the various languages, it is beneficial to say something about the services and abstractions that each language provides.

Language	**Services & Abstractions**
Matlab	triplet, BLAS as operators, data parallel abstraction
Python	triplet, BLAS as operators, data parallel abstraction
CAF	triplet, first class distributed arrays, global address space
UPC	first class distributed arrays, global address space
X10	first class distributed arrays, global address space, data parallel + multithreading, “clocks”, atomics, “advanced” OO
Chapel	first class distributed arrays, global address space, data parallel + multithreading, atomics, “advanced” OO
Fortress	first class distributed arrays, global address space, data parallel + multithreading, atomics, “advanced” OO
HPF	triplet, first class distributed arrays, data parallel
f77/f90	triplet, PBLAS, BLACS
Titanium	first class distributed arrays, global address space
Cilk	multithreading,

Table 2. Services & Abstractions of languages

Language	**Author**	**Best Performance GFlop/sec**	**p**	**Machine**	**% peak**	**Best 1p %peak**
CAF	Robert Numrich	509	60	Cray X1	71.0	92.1
UPC	Parry Husbands	2249	512	Itanium/Quadrics	78.4	91.8
UPC	Calin Cascaval	118	256	BG/L	16.4	52.5

HPL	Antoine Petitet	280600	131072	BG/L	76.4	80.1
C	PLASMA team	48.5	8	Intel Clovertown	57.0	70.3
C	Panziera and Baron	51870	10160	SGI Altix Cluster	85.1	90.1
ScaLAPACK	Antoine Petitet	44	64	Intel Pentium 4	14.3	47.0

Table 3. Performance of those codes that strive for high performance.

Taking LAPACK's code as an example, Table 4 provides a breakdown of line counts of various sections of the code:

	DGETRF	DGETF2	Total	Percentage
Leading comments	36	36	72	24.4%
Blank comments	50	43	93	31.5%
Other comments	19	13	32	10.8%
Total comments	105	92	197	67%
Declarations	11	11	22	7.5%
Argument checking	14	14	28	9.5%
Real work	30	18	48	16%
Total	160	135	295	

Table 4. Line counts.

Consequently, the total length can be thought of as anywhere from 48 SLOC (for "real work") up to 295 SLOC. And we ignore the code in the library calls to the Basic Linear Algebra Subroutines (BLAS): DGER, DSCAL, DSWAP, DGEMM,, DTRSM as well as LAPACK's auxiliary routines: DLASWP and ILAENV. Furthermore, this hardly captures the level of effort in the Parallel BLAS (PBLAS) or Basic Linear Algebra Communication Subroutines (BLACS), which were designed with a lot more generality and complexity in mind than needed for ScaLAPACK's PDGETRF subroutine alone. In comparison, the UPC version sacrifices the generality and builds the complexity from scratch and so comes in last in the SLOC metric (if SLOC could be considered as a metric).

Cilk

Category	**SLOC**
Serial Kernels	82
LU	34
Backsolve	51
Trailing Matrix	22

UPC

Category	**SLOC**
Threading Package	215
Panel Factorization	1002
Update to U	110
Trailing Matrix Update	454
Back Substitution	368

PLASMA

Category	**SLOC**
Scheduler	190
Panel Factorization	10
Trailing Matrix Updates	70
Driver	100
Comments	30

5. Writing in an HPCS Language

From our survey, we can conclude that while pure data parallel approaches to writing LU factorization can produce compact code, they do not perform particularly well. This leads us to consider alternative approaches. Because all of the HPCS languages include task parallel facilities and bearing in mind that the simple alternative of simulating an SPMD code such as HPL is always available, we consider the issues involved in writing task parallel LU factorization codes.

We restrict our attention here to multithreaded implementations which have enjoyed a resurgence in recent years. Because our results indicate that blocking and look-ahead are required for performance, we also focus on these two aspects. Blocking is primarily provided by the matrix abstraction while support for look-ahead is dependent on the parallel control flow and synchronization primitives in the language.

Multithreaded approaches have some potential advantages on distributed memory machines:

- Better communication latency tolerance
- Look-ahead (algorithmic latency tolerance) is dynamic leading to improved machine utilization

There are, however, some costs:

- User control over the schedule is needed in order to minimize parallel execution time.
- User (or system) control over the amount of buffering required in distributed memory machines.

The scheduling issue is paramount for performance. It essentially comes down to scheduling a directed acyclic graph (DAG) of tasks on each of the processors. These tasks correspond to the major operations of the algorithm, and edges between them represent dependencies that must be satisfied before the task can run. In the dense linear algebra case, the tasks and dependencies are statically determined by the matrix size and block size. In more complex algorithms, the tasks and edges may be dynamically determined by the data.

Ultimately the scheduler (either a global or many local ones) must decide, for each processor/core, the "best" task to run at any given time, knowing which dependencies have already been met and some information (flops, running time) about the task pool. The difficulty lies in the definition of "best". There are many, possibly competing requirements:

- The task must advance the parallel execution of the algorithm. The scheduler's decision should delay other tasks as little as possible. This is also known as the "critical path" issue.

- The sequence of tasks run on any given processor/core should incur as few cache misses as possible (this may compete with the previous requirement). Because of the dominance of BLAS-3 operations in LU factorization, this is less of an issue here.
- The tasks must be chosen so that buffer memory is not exceeded.

The definition and implementation of protocols for interacting with schedulers is, however, still a research topic (and so have been excluded from the HPCS languages). As such, schedulers have traditionally been built in an application specific manner using parallel control flow features (spawns and waits) combined with various data structures, such as scoreboards for keeping track of dependencies. Thread priorities are also another way of influencing the scheduler, but to our knowledge this hasn't been widely used in scientific computing codes. We anticipate the use of similar techniques in X10, Fortress, and Chapel. Features in these languages for task control include single and sync variables (for producer consumer relationships), spawns with locality directives, guarded statements (that fire when a condition is satisfied), and atomic regions. These are the basic tools that will be used for constructing schedulers.

6. Conclusions

Even with its perceived simplicity, parallel LU factorization presents unique challenges to language designers and library writers. We have shown that scaling up the available hardware resources has to be accompanied by programming language tools. If the tools are not provided, then firstly the scaling of the code quickly deteriorates, and secondly the fraction of the peak performance observed in a sequential environment can never be achieved in a parallel setup. But performance is only one part of HPCS' productivity goal. The other important part is programmer effort in delivering a well performing code. Both the programming language features and a rich set of third party libraries are required to achieve this goal.

7. Acknowledgment

This material is based on research sponsored by DARPA under agreement number FA8750-06-1-0240. The US Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation therein.

8. References

[1] Allen E., Chase D., Hallett J., Luchangco V., Maessen J-W., Ryu S., Steele G. L., and Tobin-Hochstadt S. *The Fortress Language Specification*. Available at http://research.sun.com/projects/plrg/Publications/index.html, 2007

[2] Blumofe R. and Leiserson C. "Space-Efficient Scheduling of Multithreaded Computations," *SIAM J. on Computing, 27*, 1 (1998), 202-229.

[3] A. Buttari, J. Dongarra, P. Husbands, J. Kurzak and K. Yelick. "Multithreading for Synchronization Tolerance in Matrix Factorization," To Appear in *Proceedings of the 2007 SciDAC Conference*, Boston, MA, July 2007.

[4] Buttari A., Dongarra J., Kurzak J., Langou J., Luszczek P., and Tomov S. "The Impact of Multicore on Math Software," In *Proceedings of PARA 2006*, Umeå, Sweden, June 2006.

[5] Buttari A., Langou J., Kurzak J., and Dongarra J. Parallel *Tiled QR Factorization for Multicore Architectures*. Technical Report UT-CS-07-598, University of Tennessee, Computer Science Department, July 2007. Also published as LAPACK Working Note 190.

[6] Callahan D., Chamberlain B. L., and Zima, H,P. "The Cascade High Productivity Language," In *Proceedings of the 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2004)*, pages 52-60. IEEE Computer Society, 2004.

[7] Choi J., Dongarra J., Ostrouchov S., Petitet A., Walker D., and Whaley, R.C. "The Design and Implementation of the ScaLAPACK LU, QR, and Cholesky Factorization Routines," *Scientific Programming, 5*, (1996), 173-184.

[8] Cicotti P. and Baden S. "Asynchronous programming with Tarragon," In *Proceedings of the 15th IEEE International Symposium on High Performance Distributed Computing*, June 19-23 2006.

[9] Ebcioglu K., Saraswat V., and Sarkar, V. "X10: an Experimental Language for High Productivity Programming of Scalable Systems," In *Proceedings of the P-PHEC 2005 Workshop*, held in conjunction with HPCA 2005, 2005.

[10] El-Ghazawi T., Carlson W., Sterling T., and Yelick K. *UPC: Distributed Shared-Memory Programming*. Wiley-Interscience, 2005.

[11] Husbands P. and Yelick K. "Multi-Threading and One-Sided Communication in Parallel LU Factorization," To Appear in *Proceedings of SC 07*, November 2007

[12] Kurzak J. and Dongarra J. *Implementing Linear Algebra Routines on Multi-Core Processors with Pipelining and a Look Ahead.* Technical Report UT-CS-06-581, University of Tennessee, Computer Science Department, 2006. Also published as LAPACK Working Note 178.

[13] Panziera J.-P. and Baron J. "A Highly Efficient Linpack Implementation Based on Shared-Memory Parallelism," In *Proceedings of the 2005 International Supercomputer Conference*, 2005.

[14] Snir M., Otto S., Huss-Lederman S., Walker D., and Dongarra J. *MPI: The Complete Reference - 2nd Edition: Volume 1*. The MIT Press. ISBN 0-262-57123-4, 1998.

[15] The Top 500 Supercomputer Sites. Available at: http://www.top500.org, 2007.

[16] UPC Consortium. UPC Language Specification, v1.2. Available at: http://upc.lbl.gov/docs/user/upc_spec_1.2.pdf, 2005.

Acronyms

Explanation of acronyms used in this report:

• BLAS – Basic Linear Algebra Subprograms
• BLACS - Basic Linear Algebra Communication Subroutines
• CAF – Co-array Fortran
• DAG - Directed Acyclic Graph
• DARPA – The Defense Advanced Research Projects Agency
• DGEMM – Double-precision General Matrix-Matrix multiply
• DGER – Double-precision General Rank 1 Update (BLAS)
• DLASWP – Double-precision LAPACK Auxiliary Swap
• DSCAL – Double-precision Scale (BLAS)
• DSWAP - Double-precision Swap (BLAS)
• DTRSM – Double-precision Triangular Matrix Solve Matrix (BLAS)
• HPCS – High Productivity Computing Systems
• HPF – High Performance Fortran
• HPL – High Performance Linpack benchmark
• ILAENV – Integer LAPACK Auxiliary Environment
• LAPACK – Linear Algebra PACKage
• Linpack – LINear PACKage: a set of Fortran subroutines for numerical linear algebra; also a benchmark based on one of the Linpack subroutines
• LU – Lower Upper
• MPI – Message Passing Interface
• PBLAS – Parallel BLAS
• PDGETRF – Parallel Double-precision General Triangular Factorization (ScaLAPACK)
• PLASMA – Parallel Linear Algebra for Scalable Multi-core Architectures
• ScaLAPACK – Scalable LAPACK
• SLOC – Source Line of Code
• SPMD – Single Program Multiple Data
• UPC – Unified Parallel C