Design of Interactive Environment for Numerically Intensive Parallel Linear Algebra Calculations*

Piotr Luszczek† Jack Dongarra

January 1, 2004

Abstract

Problem Solving Environments have a well established position as an essential tool for computational science. We focus our attention in this article on how to provide parallel computation capabilities to such environments that would allow for seamless access to hardware and software resources for numerical linear algebra. Instead of focusing on a particular implementation, we present an exploration of the design space of such an interactive environment and consequences of particular design choices. We also show tests of a prototype implementation of our ideas with emphasis on the performance perceived by the end user.

1 Introduction

Numerical linear algebra may well be regarded as the most basic and thus essential component of problem solving environments (PSE) that focus on numerical calculations. In this article, we intend not to focus on the user tool for accessing the parallel numerical capabilities we propose, but rather, on exploration of the design space available for such PSEs. To the user tool we refer as a *host environment* – there are plenty of such environments already in existence, whether commercial or freely available to the public. The challenge is, we believe, in seamlessly integrating parallel computing capabilities with these environments.

Even though this paper focuses on properly designed basic operations on matrices and vectors, the applicability of our arguments exceeds by far the scope of pure numerical linear algebra on dense matrices. Appropriate design of basic objects and their manipulations invites easy introduction of additional features such as sparse linear and eigenvalue solvers. This is particularly important when dealing with the complexity of parallel programming.

2 Related Work

Exhaustive survey of interactive environments for scientific computing deserves an article on its own. Therefore, we give only references to what we believe are the most relevant efforts that are related to numerical linear algebra. Python is a programming language by allows for very much interactive style of development and experimentation [1]. There exist numerous libraries that extend Python's numerical capabilities, the most popular include Numeric[2], Numarray[1], SciPy[2], MatPy[3], and ScientificPython[4]. Just for completeness' sake, we should also mention a similar framework for Perl called *The Perl Data Language*[5] with its shell for interactive work called `perldl`. Commonly known environments for interactive numerical calculations are Matlab [3], Octave[6], Scilab [4], Interactive Data Language [5], and Rlab (no longer maintained) [7]. Also, there exist environments

*This work is partially supported by the DOE LACSI – Subcontract #R71700J-29200099 from Rice University and by the NSF NPACI – P.O. 10181408-002 from University of California Board of Regents via Prime Contract #ASC-96-19020.

†Corresponding author's email: `luszczek@cs.utk.edu`

1 `http://www.stsci.edu/resources/software_hardware/numarray/`
2 `http://www.scipy.org/`
3 `http://matpy.sourceforge.net/`
4 `http://starship.python.net/~hinsen/ScientificPython/`
5 `http://pdl.perl.org/`
6 `http://www.octave.org/`
7 `http://rlab.sourceforge.net/`

that primarily focus on symbolic calculations but also allow for numerical computations – they are surveyed elsewhere [6]), here we only mention a few: Mathematica [7], Maple [8], Macsyma [9], Maxima [10]. Finally, there exist relatively many parallel extensions to Matlab [8] despite some scepticism dating back to 1995 about memory model (storage and parallel distribution of matrix data), granularity (most common Matlab tasks cannot be parallelized), and business situation (not enough market demand) [11]. Out of these extensions, Matlab*P [12, 13, 14] seems to be the most vigorous one riching its third major release version at the time of this writing.

3 LFC Overview in the Context of Parallel Solving Environments

LAPACK for Clusters (LFC) [15] is one of the projects of the Self-Adapting Numerical Software (SANS) framework [16]. It is intended to meet the challenge of developing next generation software by automated management of complex computing environments while delivering to the end user the full power of flexible compositions of the available algorithmic alternatives. LFC, in particular, automates the process of resource discovery and selection, data distribution, and execution of parallel numerical kernels for linear algebra calculations. As such we believe it is suitable for the interactive environment that we describe in this article.

4 Network Model

In our design, we consider primarily a client-server architecture as shown in Figure 1. An alternative would be to require the user to always operate on the computational server and rely on its capabilities – not only computational (which is the server's primary use) but also for example graphical (for data visualization). The latter (all-in-one) solution is inflexible as it limits the user to the functionality of the server. The limitation is not present in the scenario from Figure 1 because there exists clear separation of functionality and the server only needs to provide high performance computing capabilities (leaving other features to the client or possibly other servers).

Along the same lines goes reasoning behind placing the object logic on the client rather the server (which only holds, presumably large, object data). It simplifies the design of the server and makes it possible to use it on a wider variety of platforms. The client, on the other hand, may leverage existing technologies (programming languages or libraries) for remote management of computational objects.

Lastly, we decide not to use any technology for transparent network access (such as remote procedure or object method invocation) due to, again, requirements (mostly software-related) that the server would have to meet in order to properly support such technology. Nevertheless, it is an interesting extension to consider, especially from the perspective of code developers (rather than the end users) that would want to add extra functionality to our system.

5 Object-Oriented Features

Regardless of the host environment, it is natural to regard parallel matrices and vectors as objects (rather than, for example, regions of memory) and adhere to object-based design principles. The benefits of such a choice range from a trivial encapsulation of matrix dimensions (they become object attributes and don't have to be specified explicitly) to more profound possibility of code reuse: the same script written and tested in a single-processor mode may be reused for parallel matrix objects – the calculations will be done in parallel. To simplify the exposition, (almost exclusively) only matrix objects are mentioned in the following passage, however, the arguments that we make equally apply to either matrix or vector objects.

The first decision to make is to choose either 0-based (first matrix entry is row 0 and column 0) or 1-based indexing scheme. There exist pros and cons for either of them and there is plenty of code in production use that requires us to implement both. The problem cannot be easily solved by following the convention of the host environment and let the server adapt to the currently used convention. Such a solution does not allow for:

- code migration between two host environments that use conflicting indexing schemes and

[8] http://supertech.lcs.mit.edu/~cly/survey.html

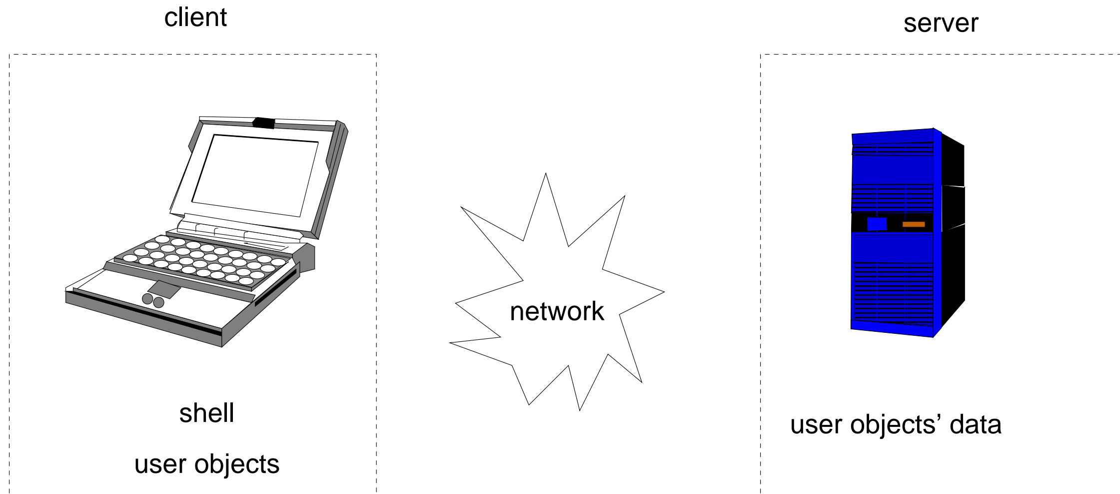

Figure 1: Network model of the computational environment for interactive calculations with parallel capabilities.

- porting of an existing code from a system of non-conformant indexing.

Therefore, we allow for both types of indexing to be used in whichever way it is possible within the constraints of the syntax of the host environment. While probably not as popular, a general form of indexing is used in some systems (for example Fortran 90) – the user can specify the range of allowed indices. More generally we allow indexing ranges to be specified as three values: first index, last index and the stride. This is already available in many interactive systems for numerical calculations.

A related problem is how the end of a range is specified. This may be illustrated with an array declaration (of size `N`) in Fortran:

```
REAL A(N)
```

and in C:

```
float A[N];
```

While both declarations use `N` as the upper bound specifier, Fortran uses it inclusively (the allowed indices are `1`, `2`, ..., `N`) and C uses it exclusively (allowed indices are `0`, `1`, ..., `N-1`). Similarly, Matlab uses inclusive convention and Python uses exclusive one. Since, there is no single scheme used across different host environments we need to provide for both.

An important decision to make is to decide whether matrix objects should operate with copy or view semantics. The most common situation when this decision has to be made is during submatrix operations. Consider an m by n matrix A partitioned as follows:

$$A = [A_1\, A_2]$$

where A_1 and A_2 are m by n_1 and m by n_2 matrices, respectively, with $n_1 + n_2 = n$. A common way (across different host environments) to refer to A_1 is `A[:,:n1]`. The question is whether such a reference should create a copy of the appropriate portion of A or, instead, only produce an alias (a view). There is no good answer to this question because there exist situations where either the former or the latter solution is preferable. Different systems solve this problem differently: Matlab and Python use the copy semantics while Fortran 90 uses the view semantics (with the caveat that some compilers make a copy upon calling a function if an array alias is used and when the function returns, the result is copied back to the original location). It is possible to have a copy semantics with performance of view semantics using the copy-on-write technique: initially, only an alias is created, and when subsequent code attempts to write data to the alias, a copy is made (this technique has a shortcoming, though: when only a small portion is aliased and the original is no longer needed, all the data of the original need be kept rather than just the

aliased portion). Most likely, end users will opt for copy semantics, while developers will prefer the view semantics. Therefore, we choose to allow both in our system.

While some environments have only one numeric data type for calculations, it is much more desirable to have many of them for customization of memory requirements, arithmetic semantics (fixed-point versus floating-point), and interfaces to external data sources or sinks. The flexibility of multiple data types comes at the price of resolving issues with mixed-type operations. Automatic variable casting is a feature of almost any programming language in wide spread use. An expression like `x + y` is handled correctly even if `x` and `y` are variables of different numerical type. The most common behavior in such a case is to *promote* (type-cast to the larger type) one of the values and then perform calculations. The promotion rule works well for statically typed languages where it is always possible to specify the type of the result and even types of arguments (by type-casting them explicitly). Most PSEs use some form of dynamic typing and therefore it is harder to ensure correct type for the result. Another issue is the fact that variables in PSEs refer to more complex objects (matrices and vectors in the case of our system). That complexity allows for wider range of possibilities in dealing with mixed-type operations. The two major issues to consider are the memory allocation (promotion could potentially require a few times more space to be used) and tensor-rank change (an outer product of two vectors produces a matrix: $A = xx^T$ – a different data type all together). Another example to consider is a situation when single-precision floating-point arithmetic is desired (may be due to storage constraints) and by accident (a bug or just a bad language design) a double-precision result is produced in the midst of long calculations. If the promotion rule was used then any subsequent operation with this double-precision value would introduce double-precision results which in turn would create even more double-precision values – clearly not what was desired in the first place. Making *downcasting* (type-casting to the smaller type) the default behavior is not an option either as it would, among other arguments, be counter-intuitive for many novice users. Any solution in between those two extremes might be more desirable but would still not work properly in some situations. Therefore, we opt for providing means for ensuring appropriate kind of automatic casting.

The type of PSE that we are describing deals with tensors of different ranks:

- 0 – numerical values,
- 1 – vectors of numerical values, and
- 2 – matrices of numerical values.

Such environments add a unique aspect to the type-casting problem described above: reduction of tensor rank (an object property, as opposed to matrix rank – a numerical property). Consider a common operation that reduces tensor rank – a vector dot-product of the form:

$$\alpha = y^T x$$

It is trivial to claim that the result (α) should be a tensor of rank 0 if x, and y are tensors of rank 1. In general, when vectors have at least two rows and matrices have at least two rows and columns then predicting the rank of the result is easy. However, if those assumptions do not hold true than the rank cannot be inferred. Let's consider a general case of matrix-matrix multiply:

$$C = AB$$

where: A is m by k, B is k by n, and C is m by n. If either m or n is 1 then the multiply reduces the tensor rank by 1. If k is 1 than the reduction is by 2. However, the type of the result cannot be changed even if potential tensor rank reduction occurs. The reason being common usage scenarios. In particular, if a matrix algorithm (such as an iterative method or a dense linear solver) is formulated in terms of submatrices (so called *block algorithm*) then it is expected to work even if the submitraces degenerate to single values (*block size* is 1) – 1 by 1 matrices. There is no general way of detecting when type change should follow tensor rank reduction. Therefore, we choose not to perform the type change by default since this facilities interactive work. However, an option should be included to allow type change during debugging sessions.

6 Host Environment Integration

In order to integrate our system with existing PSEs we consider the following features to be the most relevant:

1. networking capabilities,
2. name spaces, and

3. object-orientation.

In the following we give a review of how these features are supported in some of the existing environments and what can be done if they are missing.

Networking capabilities are the most essential for our system since we envision communication between the client and the server to be done over some kind of network. Even though this requirement may be loosened a bit by using other means of inter-process communication we do not consider it at the moment. Out of the host environments that we initially target, networking is fully supported in Python with fairly complete implementation of the Unix socket API (Application Programming Interface). Maple, Mathematica, Matlab, and Octave require an extension written in a native language (most commonly C) to be able to use sockets – this creates a portability problem since the native part of the interface has to be written and tested for every computing platform we wish to support. Another issues is that each host environment has its own way for writing extensions. Luckily, most of the aforementioned environments support Java so this is the way to write just one code and use it in many environments. Finally, since Octave does not support Java as of this writing, an extension can be written using system calls such as: `system()`, `popen()`, `popen2()`, `fork()`, `exec()`, `pipe()`, `dup2()`, and `waitpid()`. These calls are all available from the Octave command line or scripts. The external program that is called (with, say, `system()`) may be written in Java so that we can reuse the code from other environments. Matlab also has a similar capability (namely: `system()`, `dos()`, `unix()`, and `perl()`).

Support of name spaces is an important but not essential feature that we would like to use. The most common way of dealing with lack of name spaces is by prefixing each function name with a common string indicative of function origin. This prevents name clashes between global names and allows many contributors to introduce new capabilities. Python offers more sophisticated way of dealing with this problem – it has a hierarchical module system comparable to that of ISO C++ and Java. Matlab comes close to it by implementing functions only relevant in the context of one particular class of objects (they are commonly referred to object methods but in Matlab have invocation syntax just like regular functions). For all other environments we need to use the prefixing trick.

Object-orientation is an important feature as it allows, among others, for a simple statement like `a+b` to be interpreted differently depending on what `a` and `b` are: it might result in adding two numbers together or it might (as it is in our case) result in addition of two matrices on a remote server. Most of the host environments that we know are object-based – they provide ability to create and manipulate built-in objects (such as matrices or vectors) but do not allow to create new ones. Matlab is somewhat more advanced in that respect as it allows for creation of new objects and overloading of functions and operators for them. However, it does not support the full life cycle of an object – in particular it does not notify user-defined objects when they are about to be garbage-collected. This is an important capability in the presence overloaded operators since they tend to produce anonymous temporary objects – there is no way to reclaim their storage even manually. This problem can be somewhat alleviated by using Java from within Matlab since JVM (Java Virtual Machine) implements garbage collection with cyclic dependency detection and allows for objects to execute code when they are about to be garbage collected. Python, aside from being a good interactive environment, is an object-oriented language so this is probably the best target for our environment. In other environments we need to resort to function syntax – it takes a lot from expressiveness but still allows to use the functionality that we offer.

7 Parallel Execution

Creating an interactive environment for parallel computations has many design challenges even if limited to numerical linear algebra. The first issue to resolve is the fact that vectors and matrices most often have different requirements for data layout: vector computations are likely to benefit from 1-D layout while for matrices 2-D distribution is preferable. We believe that a novice user of our environment would not know (or care to know) the difference between the two and therefore would benefit from some form of automation in this respect so that vectors and matrices may be used without worrying about their parallel properties. A good solution for such a scenario seems to be distributing vectors in 1-D fashion and matrices in 2-D. In case when a matrix and vector are to be used together (e.g. in matrix-vector multiplication), the

vector needs to be made conformant to the matrix' layout to perform the operation efficiently. Such a solution involves relatively small communication penalty. For more advanced users, full control of data distribution is the way to go as these users know the consequences of data distribution for parallel computation.

Another aspect to consider is execution synchronization between the client and the server. Some of the literature on the subject uses the term *lazy evaluation* to refer to one of the possible scenarios [17]. This term was loosely borrowed from functional languages where it refers to a way of evaluating (potentially infinite) expression lists. In the context of parallel PSEs it was used to mean that only every other remote request is blocking the client until the server's completion. In the client-server and parallel literature, generalization of this way of communication is simply referred to as asynchronous: the client submits a request to the server and doesn't block while the server is processing the request. Asynchronous mode, in our opinion, is not good for an interactive environment since it splits the call process into two phases: submission and completion requests. It is not the way existing sequential environments operate – their behavior is equivalent to a synchronous mode (each request is blocked on the client side until the server fulfills the request). This mode is also more intuitive for most of the users. A mid-way solution is transactional processing: the user starts a transaction by making a special call, then all the transaction requests are submitted, and then the call finalizing the transaction is made which blocks until all the transaction requests are served. It differs from the asynchronous mode by not executing all the transaction requests when they are submitted, but rather, sending them all in one batch – the server has more information to order them for better performance. Finally, we do not consider using threads in the host environment as a separate case – in a sense it can be regarded as a type of asynchronous system when a synchronous request blocks only one thread and the polling for completion takes place by inter-thread signaling.

8 Miscellaneous Issues

An important aspect of any numerical system is compliance with the IEEE 794 standard [18]. While the standard is commonly accepted by many hardware vendors it is still rare to find fully (or at least mostly) compliant product. While the full standard compliance is hard to achieve, there exist a few aspects of the standard that are often quoted as essential. Amongst those, the most contentious issues include raising an exception when calculation produces or involves infinities or NaNs (Not-a-Number). While the standard is clear about usage and behavior of QNaNs (Quiet NaN) and SNaNs (Signaling NaNs) – the end users seem to be divided as to which should be chosen as the default. We are bound here by what is the typical behavior of the host environment and what is available on the server (to what extend it complies with the standard). Some environments have a way of dealing with non-conformant hardware or system libraries, e.g. in Python, floating-point exceptions are caught by a Unix signal handler. However, the POSIX standard leaves the behavior of the `SIGFPE` signal (floating point exception) not fully defined and therefore, in general, the signal handling function cannot return. To return the control back to the offending code the `setjmp()` and `longjmp()` functions are used. Of course, this trick won't help if no signal is raised upon illegal calculations.

There exist a few options for data storage and transfer that we consider useful. Certainly, users will have some data sets stored locally on their client machines. These local data need to be transferred to the server for manipulation. However, data transfers are, in general, quite slow and therefore only explicit requests will move the data between the server and the client. During calculation, the best place for data would be the server while at the end, the results need to be transfered back to the client (in case the server does not provide reliable storage capabilities). In the meantime, the data is prone to be lost due to hardware or software crashes so at some point fault-tolerance should be considered. Another likely scenario is downloading data from an external source. A very helpful extension is support for scientific data formats.

Security is an important asset of any software piece, especially the kind we are considering – that provides server-like capabilities. In this area, we only intend to leverage existing solutions with initial focus on the port-forwarding feature of `ssh(1)`. It seems relevant in the presence of firewalls and NATs (Network Address Translation) that prevent connections to all but few selected ports.

High level interface to complex parallel software libraries requires flexible and yet convenient way of denoting algorithms. Most relevant are the number and type of

keystrokes that the user has to enter in order to be able to express his intentions. And while simplicity encourages experimentation, flexibility allows for more functionality to be added easily (for example as user contributions) after the PSE has been created. This becomes particularly important in the context of changing the behavior of a computational environment; two main configuration types need to be considered:

1. Global and
2. Local.

The global type includes: configuration files (*dot-files* in Unix parlance), command line options, and global program variables. In a sense, all of them provide a similar type of customization with different timing and scoping. However, since a PSE may be regarded as a language, it is important to maintain its semantic consistency. Therefore, global configuration is a valid solution when there is only one default setting mandated as a standard. For example, it might be possible to specify whether integer division rounds down or up, but the round-down behavior should be the default one and as such it should guarantee the proper behavior of all programs. With this assumption at hand, the user could conveniently change the rounding setting for one particular code while maintaining default setting for all others. Similarly, library developers would not have to be bothered by including support for both types of rounding; only the default (round-down) could be considered, and it is the user's responsibility to use the library with the default setting or otherwise bear the consequences. Of course, when using high quality libraries, a warning would be issued or the non-default case would be handled by a different algorithm.

Relevant local configuration types include: object attributes, shadow objects or explicit syntax. The first two are somewhat similar as shadow objects are just aliases of their originals with some of the attributes changed. For example if `A` is a square matrix, `A.I` (a shadow object of `A`) could indicate inverse of `A` but creating `A.I` would not immediately produce a numerical inverse of `A` but rather, LU decomposition would be used whenever `A.I` is multiplied by a matrix or vector. Compared to object attributes, shadow objects are more explicit. From clarity standpoint, object attributes (and to a lesser extend shadow objects) are not as good as explicit syntax (e.g. function call) but are far more succinct and more suitable for interactive and high level environments.

Hardware specifications	
CPU type	Pentium 4
	Intel Xeon
CPU clock rate	2.4 GHz
System bus clock rate	0.4 GHz
L1 data cache	8 Kbytes
L1 instruction cache	12 Kbytes
L2 unified cache	512 Kbytes
Main memory	1 GBytes
Network	Gigabit Ethernet
	NIC: Intel e1000

Table 1: Parameters of the Dell Precision 530s machine used in the tests.

9 Implementation

At the moment, the basic infrastructure of our design has been implemented and successfully applied to a dense matrix factorization and iterative solution method in Matlab and Python environments. We expect to reach more mature stage of the software upon publication of this article since the work on the project is on-going. Our preliminary tests show that the overhead of remote execution can be offset when problem sizes become prohibitive for a sequential environment and it is possible to reap the benefits of parallel computation.

Table 1 shows hardware configuration used in our tests. MPICH 1.2.4 was used as the MPI implementation. Figure 2 shows the timing results for our tests that were performed on a non-dedicated system. The objective was to solve in double precision floating-point arithmetic a system of linear equations of the following form:

$$Ax = b, \quad (1)$$

where A is n by n real matrix ($A \in \mathbf{R}^{n\times n}$), and x and b are n-dimensional real vectors ($b, x \in \mathbf{R}^n$). The values of A and b are known and the task is to find x satisfying (1). LU decomposition was first used to factor A:

$$LU = PA, \quad (2)$$

where:

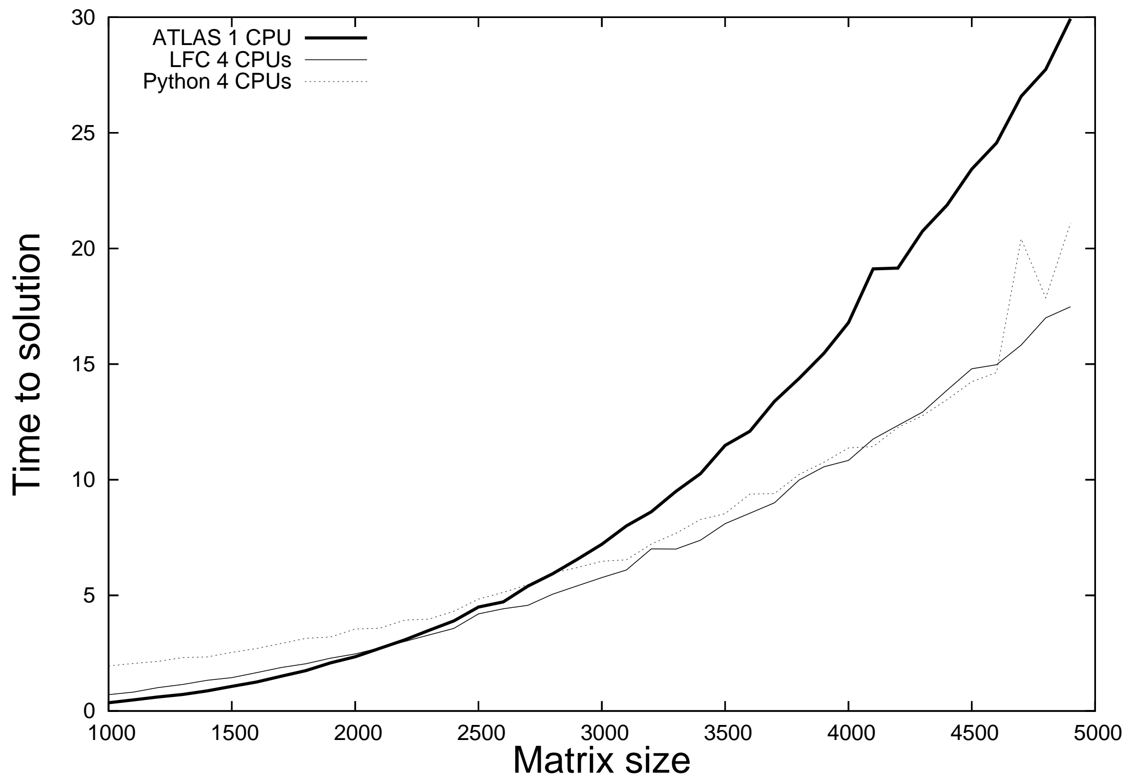

Figure 2: Comparison of time to solution of a system of linear equations of varying size with different methods.

- L is a lower triangular matrix with unitary diagonal,
- U is an upper triangular matrix with arbitrary diagonal,
- P is a row permutation matrix.

Solution to (1) was then obtained in two steps:

$$y = L^{-1}Pb \quad (3)$$
$$x = U^{-1}y \quad (4)$$

Three scenarios were used to obtain a solution:

- sequential computation using an optimized programming library,
- parallel computation using an optimized programming library,
- remotely controlled parallel computation with synchronous calls.

For the first scenario, ATLAS [19, 20] library was used on a single CPU. In particular, the functional equivalent of LAPACK's [21] `DGESV()` routine was used that performs LU decomposition *in-situ* – no data copying is involved. The second scenario utilized two dual-CPU machines that performed computations on a 2 by 2 process grid with LFC's equivalent of ScaLAPACK's [22] `PDGESV()` routine. Again, no data copying was involved. Finally, the third scenario used the same hardware and software as the second one but the execution initiation and timing was done on a remote computer running Python interpreter. The round-trip time between the client and one of the nodes of the computational server grid (as measured by the `ping` program) was about 82 milliseconds – a value representing a 16-hop connection (as measured by the `tracepath` program) through wireless access point and an ADSL line. In this scenario, a copy was made of the system matrix to store its LU factors computed by `PDGESV()`: $x = A^{-1}b$ was written as `x = A.I * b` but the inverse of A was not calculated explicitly but rather the LU decomposition of a copy of A was used. It's a trade-off between convenience and optimality (the optimal notation being for example `pgesv(A, x, b)`) and we intended for our tests to reveal how much this convenience costs.

Analysis of Figure 2 show two important matrix sizes for our testing environment: the size for which parallel execution is faster than sequential execution (3000 in our case) and the size for which the matrix copy overhead (discussed above) is negligible (4000 in our case). The graph shows counter-intuitive effect of copy-free solve being slower than the solve with copy overhead – this is to be expected on a non-dedicated system and is more likely to occur the longer the solution time is. Another effect worth noting for matrices larger than 4500 is the unexpected increase of time to solution for the remote execution. Very likely explanation is a sudden surge in the load of the network that connects the client and server. As mentioned above, in our tests the network was a part of the Internet and in general is not reliable and cannot be dedicated for the time of experiment.

10 Future Work

It is conceivable that the implementation might exhibit itself as a OGSA-compliant service. For maximum flexibility however, such a service would not be running on the server. The service would be running on a proxy server capable of OGSA interaction. This proxy server would interact with the actual computational server through a simplified protocol. This is much like NetSolve's three-tier approach [23].

An interesting direction to pursue is creation of compilation system so that it is possible to translate existing scripts to a stand-alone executable. Such capability provides opportunity to have a client-server environment for experimentation and debugging while the compiled executable could be used on systems with only batch queue access where setting up a server is not possible.

References

[1] Bill Venners. Programming at Python speed: A conversation with Guido van Rossum, 27 January 2003. Available at http://www.artima.com/intv/speed.html.

[2] Paul F. Dubois, Konrad Hinsen, and J. Hugunin. Numerical python. *Computers in Physics*, 10(3), May-June 1996.

[3] Mathworks Inc. *MATLAB 6 User's Guide*, 2001.

[4] Claude Gomez, editor. *Engineering and Scientific Computing with Scilab*. Birkhäuser, Boston, 1999.

[5] Liam E. Gumley. *Practical IDL Programming*. Morgan Kaufmann Publishers, 1 edition, July 2001.

[6] E. Schrüfer. EXCALC – a package for calculations in modern differential geometry. In D. V. Shirkov, V. A. Rostovtsev, and V. P. Gerdt, editors, *Proc. IV Int. Conf.*, Computer Algebra in Physical Research, pages 71–80, Dubna, U.S.S.R., 1990. World Scientific, Singapore, 1990.

[7] S. Wolfram. *Mathematica: A System for Doing Mathematics by Computer*. Addison-Wesley, Reading, Mass., 1988.

[8] B. W. Char et al. *Maple V, Language Reference Manual*. Springer, 1991.

[9] R. H. (Richard H.) Rand. *Computer algebra in applied mathematics: an introduction to MACSYMA*. Number 94 in Research notes in mathematics. Pitman Publishing Ltd., London, UK, 1984.

[10] Paulo Ney de Souza, Richard J. Fateman, Joel Moses, and Cliff Yapp. The Maxima book. See `http://maxima.sourceforge.net/`, 2003.

[11] Cleve Moler. Why there isn't parallel Matlab. *Mathworks Newsletter*, 1995. Cleve's corner.

[12] Long Yin Choy and Alan Edelman. Matlab*p 2.0: A unified parallel matlab. Technical report, Massachusetts Institute of Technology, January 2003. URI: `http://libraries.mit.edu/dspace-mit/`.

[13] Long Yin Choy. MATLAB*P 2.0: Interactive supercomputing made practical. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, September 2002.

[14] Parry Jones Reginald Husbands. *Interactive Supercomputing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, February 1999.

[15] Zizhong Chen, Jack Dongarra, Piotr Luszczek, and Kenneth Roche. Self-adapting software for numerical linear algebra and LAPACK for clusters. *Parallel Computing*, 29(11-12):1723–1743, November-December 2003.

[16] Jack J. Dongarra and Victor Eijkhout. Self adapting numerical algorithms for next generation applications. *International Journal of High Performance Computing Applications*, 17(2):125–132, 2003. ISSN 1094-3420.

[17] Boyana Radenska Norris. An environment for interactive parallel numerical computing. Technical Report UIUCDCS-R-99-2123, University of Illinois, Urbana, Illinois, November 1999.

[18] ANSI/IEEE Standard 754-1985. Standard for binary floating point arithmetic. Technical report, Institute of Electrical and Electronics Engineers, 1985.

[19] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2001.

[20] Jack J. Dongarra and Clint R. Whaley. Automatically tuned linear algebra software (ATLAS). In *Proceedings of SC'98 Conference*. IEEE, 1998.

[21] E. Anderson, Z. Bai, C. Bischof, Suzan L. Blackford, James W. Demmel, Jack J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and Danny C. Sorensen. *LAPACK User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, Third edition, 1999.

[22] L. Suzan Blackford, J. Choi, Andy Cleary, Eduardo F. D'Azevedo, James W. Demmel, Inderjit S. Dhillon, Jack J. Dongarra, Sven Hammarling, Greg Henry, Antoine Petitet, Ken Stanley, David W. Walker, and R. Clint Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, 1997.

[23] Sudesh Agrawal, Jack Dongarra, Keith Seymour, and Sathish Vadhiyar. NetSolve: Past, present, and future – a look at a grid enabled server. In F. Berman, G. Fox, and A. Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*. Wiley Publisher, 2003.