# UTILIZATION OF PARALLELIZATION ALGORITHMS IN INSAR/PS-INSAR PROCESSING

**Petar S. Marinkovic[1], Ramon F. Hanssen[1], and Bert M. Kampes[1,2]**

[1] *DEOS - Delft Institute of Earth Observation and Space Systems, Delft University of Technology, Kluyverweg 1, 2629HS Delft, The Netherlands, Email: {p.s.marinkovic ; r.f.hanssen}@lr.tudelft.nl*

[2] *German Aerospace Center (DLR), Oberpfaffenhofen, 82234 Wessling, Germany, Email: bert.kampes@dlr.de*

## ABSTRACT

This study describes a method for InSAR/PS-InSAR processing using parallelization algorithms. The InSAR/PS-InSAR processing chain is analyzed from the aspect of the computational load, which results in a specific parallelization model. The theoretical concept for the porting of Delft Object-oriented Radar Interferometric Software (Doris) into a parallel computational environment is described. The features of the presented parallelization model include efficient interprocessor communications utilized by MPI (Multi Passage Interface) and a "chained" data distribution capability. An example of the mapping process is illustrated for the case of simple InSAR processing.

Key words: InSAR, PS-InSAR, parallelization, Doris software.

## 1. INTRODUCTION

Classical InSAR processing is known to be computationally laborious. Although many aspects of the data processing chain are already well known, there is a need for more efficient methods that provide results in lesser time. Especially the advent of radar time-series analysis, [1], needed for precise pin-point deformation measurements, is a driving force for improvements in processing technology, due to the very large quantities of data and intermediate products. Improvements in processing speed can be obtained by a strict initial selection of potentially useful reflections, but this will also limit potential alternative applications. A-posteriori implementation of small corrections to the processing chain, as well as storage of variance-covariance information requires keeping as much information at hand as possible.

This paper discusses possible improvements in the processing chain by means of a combined parallelization and distribution algorithm. The processing chain is analyzed from the aspect of the computational load. Consequently, the analysis resulted with the specific parallelization model of the InSAR application space.

The developed model effectively makes use of the inherently spatially localized nature of most processing steps of the interferogram production algorithm to formulate the division of labor, both spatially across the input data and temporally along the pipeline of algorithm tasks. This formulation can be used to split the load among all available processing units, by either using multiple threads of execution on a multiprocessor configuration, or by a message passing interface technique (MPI), [2], to coordinate multiple processes distributed across a cluster of computers. Subsequently, the concept for the parallelization of Doris (Delft Object-oriented Radar Interferometric Software), [3], is developed and presented.

As a case study, the Matlab/Octave development environment is used for the preliminary evaluation of the performance of the presented parallelization model.

The paper proceeds in Section 2 with a discussion on the InSAR/PS-InSAR processing algorithms and introduces the processing and performance cubes as the mean for the performance evaluation of the production chain within the single/multi-processor environment. Section 3 introduces the concept for the parallelization of the Doris software together with basics of implemented interprocessor communication model (MPI - Message Passing Interface), [2], and data distribution model. Section 4 discusses the results of numerical experiments performed using the previously discussed model within the Matlab environment. Finally, section 5 outlines the perspectives of future work and presents the conclusions.

*Proc. of the 2004 Envisat & ERS Symposium, Salzburg, Austria 6-10 September 2004 (ESA SP-572, April 2005)*

## 2. PARALLELIZATION OF INSAR/PS-INSAR PROCESSING ALGORITHM

### 2.1. Strategy for parallelization

In order to effectively parallelize the InSAR processing algorithm, it is important to discuss all parameters that influence the performance. The analysis of the processing chain is performed in the form that instead of a detailed analysis and evaluation of each processing step, which would be beyond the scope of the paper, the "similar" steps are identified, grouped in processing layers, and evaluated.

The introduced criteria on similarity are established by the order of the processing step within the processing chain, data handling, complexity of the particularly implemented algorithm, used libraries, and mathematical nature.

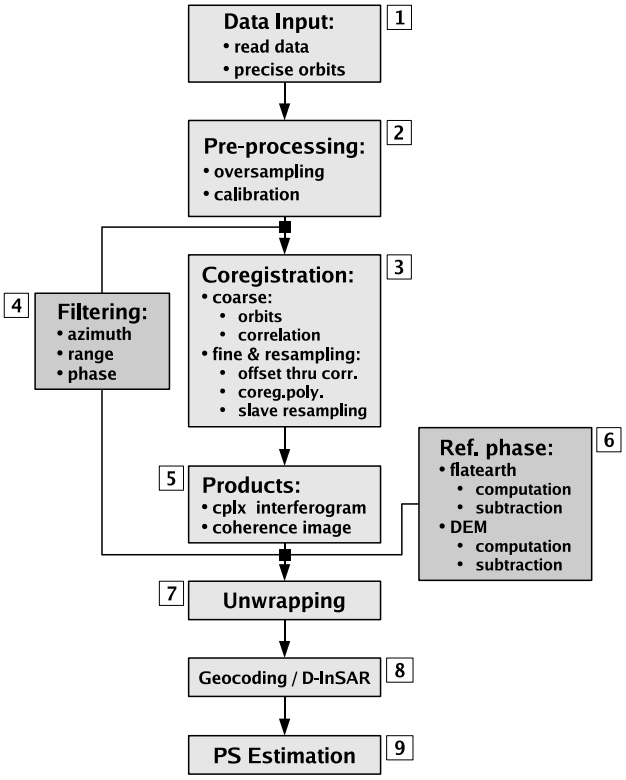


*Figure 1. InSAR processing flowchart.*

The reason behind this analysis is that almost each production step, see Fig. 1, can be implemented in a different manner with a different algorithm, which in turn would result in varying processing times even on the same hardware configuration. For example, resampling of the slave image to the master grid can be performed by using different algorithms, [5], where each of the comparable results requires a different computational time.

Additionally, and more importantly, the reasoning for such a distinction of the processing chain is that different computer languages are usually extremely sensitive to different code structures, e.g., Matlab code with many loops is significantly slower from other longer but more structured codes. Consequently, InSAR processing is identified in the structure of the processing layers, which represent the group of processing steps. These steps would in any combination of algorithm, programming language and computer system result in relatively similar computational times.

In this manner the algorithm performance is measured in terms of "time-to-solution", and the possible trap of equating the steps performance in "Gigaflops" or "Megaflops" is avoided. The algorithm or implementation that delivers the highest Giga/Megaflops rate for each processing step is not necessarily the one which provides the fastest solution [4].

As an extreme example: one could use high-megaflops dense-matrix methods to solve a problem where sparse methods would work just as well. The $O(N^3)$ dense methods deliver outstanding Megaflops numbers due to the regular structure of the algorithms and the considerable efforts that have gone into designing optimized libraries. Sparse methods, on the other hand, deliver significantly fewer Megaflops, but only require $O(N^2)$ operations. The sparse methods would obtain results more quickly, but would report lower Megaflops rates.

### 2.2. InSAR/PS-InSAR processing chain: the analysis

The analysis is performed using Delft Object-oriented Interferometric Software (Doris), [3], chosen because it is a fully functional interferometric processing software in the public domain. Doris follows the classic UNIX philosophy that each tool should perform a single, well-defined function, and complex functions should be built by connecting a series of simple tools into a "pipeline." In a nutshell, Doris consists of series of programs (modules) that perform different interferometric tasks. More details on Doris are provided in [3].

It is important to note, ahead of a discussion on the structure of the InSAR processing algorithm, that the algorithm input in the performed analysis is Single Look Complex (SLC) data. The pre-processing of the raw data, both radar and orbit, is not evaluated in this work.

#### 2.2.1. Analysis

Following the constraints from the parallelization strategy, the analysis in this section is presented from

the computational load/time aspect. The starting point is a realization of the processing summary from Fig. 1. In the following list the main processing steps are identified, without a detailed description. For a more detail review of the processing chain the reader is referred to [5] and more information on InSAR processing using Doris software can be found in [3].

Note that for the PS-InSAR processing steps 4, 6, 7 and 8 are not required.

1. Data input: Single Look Complex and orbit data.
2. Pre-processing: Oversampling of data, usually by factor of two in range and azimuth direction, and amplitude calibration. The calibration module is are implemented as independent functions and are not part of the standard version of Doris. Note that oversampling in range is implemented in Doris.
3. Coregistration and resampling: The determination of the coregistration polynomial that describes the transformation of the slave to master image, which is subsequently used for the resampling of slave image to the master grid.
4. Data filtering: Optional spectral filtering and phase filtering. The spectral filtering in the range and azimuth is performed to increase signal-to-noise ratio. The phase filtering is performed either with a simple pre-defined spatial averaging kernel, user-defined 2D convolution kernels, or with the Goldstein filter.
5. Products: Computation of the interferometric products, i.e., a complex interferogram and coherence image.
6. Reference phase: Computation and subtraction of the interferometric phase correction of a reference body, i.e., ellipsoid and external Digital Elevation Model.
7. Phase unwrapping: The reconstruction of the original phase from the "wrapped" phase representation, These modules are implemented as independent functions and are not part of the standard distribution of Doris. The SNAPHU phase unwrapping software is usually used [6].
8. Geocoding and D-InSAR: The conversion of the unwrapped phase to height, and the (azimuth, range) coordinates are geo-referenced products are computed.
9. Persistent scatterer processing: selection, construction of a network, deformation parameter estimation, phase correction, unwrapping, correction for the atmosphere, interpolation, etc. These modules are implemented as independent functions and are not part of the standard version of Doris.

From the algorithm study it can be concluded that most processing steps of the interferogram production are inherently local. A simple example of this statement would be the processing of two different crops of the same master/slave combination. As long as the master grid in the coregistration step is fixed, i.e., the same input master subset for different slave subset, the resulted interferograms are comparable with the interferogram achieved by the full scene processing. The computational load is another important aspect that has to be taken into account. Due to the reasons given in the previous section, it is very hard to make a general statement on this. However, there is a strong correlation between the computational load and data distribution. As shown in the previous example, the smaller the input data set, the faster the computation of the end result is performed, even though the processing steps defined in the task pool remain the same.

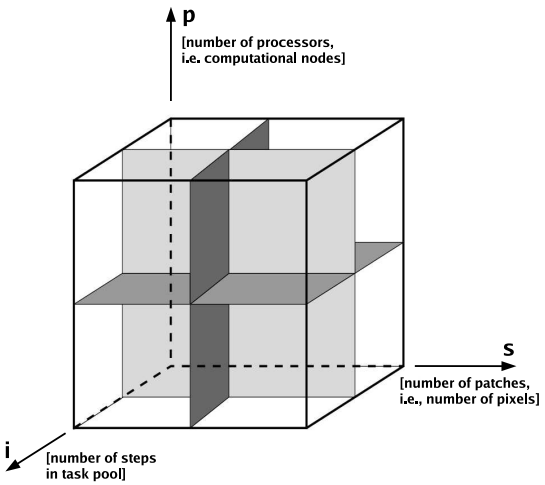


*Figure 2. Performance cube. Each point represents an execution time that depends on the number of processors, the number of iterations and the size of the data set.*

This concept can be visualized by a processing cube, Fig. 2. Each point within the performance cube represents an execution time that depends on the number of steps in the task pool ($i$), the size of the input data set ($s$) and on the number of processing nodes ($p$). In other words, the execution time is presented as a function of $(p, s, i)$, i.e., each parameter of $(p, s, i)$ triple has a direct influence on the execution time. The distinct points $(p, s, i)$ in the processing cube can be used as a means for preliminary algorithm evaluation of the processing chain.

In order to visualize the performance, the cube can be cut perpendicularly to each axis. This results in three plots, each illustrated by a gray plane in Fig. 2. Subsequently, each of these plots can be drawn two-dimensionally, treating the second axis as a parameter, Eq. (1). The resulting graphs would illustrate the following equation set:

$$
\begin{aligned}
T &= f_{i=const}(s/p), \\
T &= f_{s=const}(i/p), \\
T &= f_{p=const}(i \cdot s)
\end{aligned} \tag{1}
$$

The main focus of this preliminary parallelization study of Doris software is to answer the following questions.

- How to successfully decompose and distribute the input data over the processing nodes?
- How to manage the processes defined in the task pool on the distributed data?
- How to successfully utilize the multiprocessor environment into the processing chain?

Figure 3 illustrates the method used for parallelizing the processing algorithm. Data decomposition is combined according to the master/slave (manger/worker) principle.

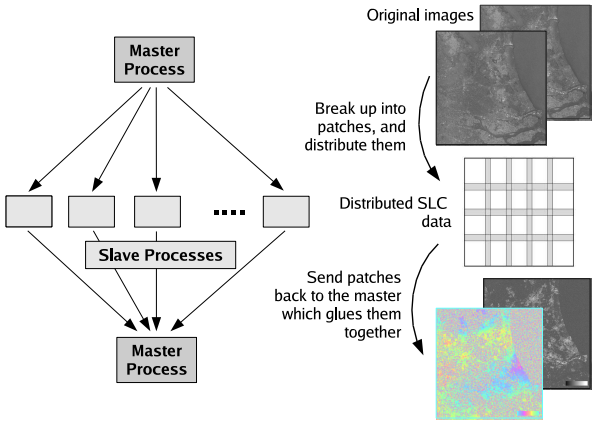


*Figure 3. Parallelization strategy: Master-slave method (left) combined with data decomposition (right). The input data set is divided into equally large, partially overlapping patches which are the processed with minima interprocessor communication.*

#### 2.2.2. Data decomposition

This method was applied to divide the input images into smaller patches. The right part of Fig. 3 shows the partitioning scheme. The sub-images overlap each other, necessary because of possible effects at the edges of the patches. To obtain satisfying results in conjunction with an acceptable amount of overhead calculation, a preliminary "trial and error" analysis shows that the overlapping region should be about one-eighth of the patch size.

The decomposition influences, as expected, the algorithmic behavior of all steps of the task pool. All the algorithms became very suitable for parallelization, except for the coregistration, where a special care is required with the coarse coregistration offset.

For example, the impact of this decomposition on all algorithms within a pool that uses FFTs would be that the overall execution time of that particular step is reduced to approximately $O(n^2)$, because the FFT is applied to equally sized, small patches, regardless of the total problem size. Normally, FFT has a time complexity of $O(n^2 * log(n))$, as shown in [7].

#### 2.2.3. Master / Slave method

This is a centralized approach, which is used to distribute the patches to all available processors. One processor, the master (manager), does the I/O and controls all other processors, i.e., the slaves (workers). This scheme is illustrated in the left part of Fig. 3. The master reads the whole data set, partitions it and sends sub-images to the slaves. The slaves perform calculations and send back the data. The server collects the results, post-processes them, and stores the processed and fitted patches. Since the master node performs no "real" computation, it can control many slave nodes simultaneously. In some special cases, the master node can act both as a master and as a slave node.

## 3. IMPLEMENTATION OF THE PARALLELIZATION CONCEPT

By optimizing and analyzing the sequential version of the InSAR processing algorithm, as presented in the previous section, the parallelization concept is defined. This section deals with the practical application of the concept, which is realized in the form of a parallelization kernel. Figure 4 depicts the kernel structure.

The kernel is fine-tuned for Doris software, and its implementation is an ongoing project within DEOS.

1. **Client manager:** responsible for communication with the client. It provides functions for reading commands and arguments from the client and sending the results back to the client. It is only used in the master server process.

2. **Server (connection) manager:** takes care of communications between master/slave node processes. It mainly controls broadcasting of commands and arguments from the master process to the slave processes, and collection of results and error codes. It also provides rank and size information to the processes.

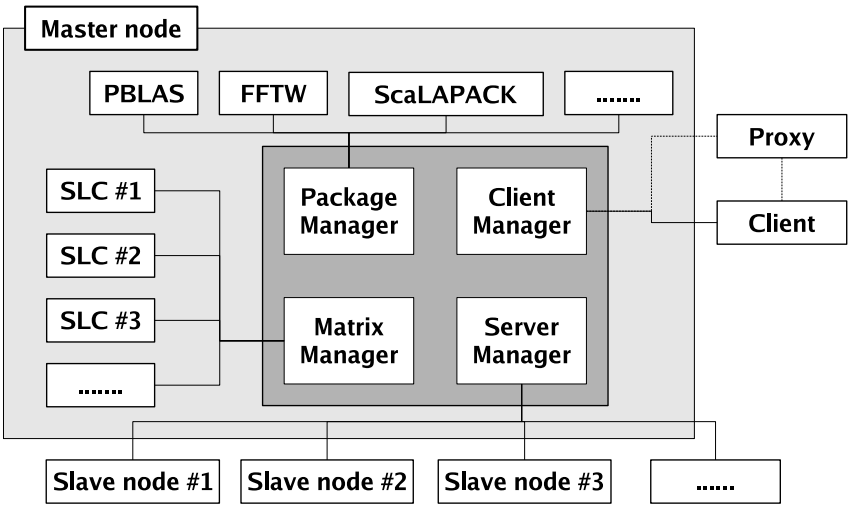


*Figure 4. Parallelization kernel.*

3. **Package (library functions) manager:** responsible for maintaining a list of available libraries and functions provided by them. When initiated by the master process, the package manager will also perform the actual call to the library functions.
4. **Matrix manager:** all the functions needed to create, delete, and change the matrices on the server processes. It maintains mapping from client-side matrix identifiers to actual matrices on the server. It is also responsible for performing "garbage" collection. The input data (complex matrices) are partitioned in a way that vector over a set of MPI processes, so that each process was responsible for a contiguous block of data. The important feature of this style of data partitioning is the minimization of the communication between master/slave or slave/slave nodes. The only data that need to be shared between processors is the end-points of every data block.

This constellation of parallelization kernel offers great advantages. First of all, debugging is made easier because bugs are localized and thus are much easier to identify. This approach also allows easier extension of all the kernel modules. For example, as a standard the *basic server connection manager* makes use of MPI (Message Passing Interface) as the means of communication between server processes. However, one could write a *server connection manager* that uses PVM (Parallel Virtual Machine) instead. As long as the new version implements all the public functions in the class, no change is necessary in any other part of the core code. Similar extensions can be made to client connection, see Section 4, or matrix manager as well, with for example a different method of distributing the data.

Since the presented parallelization model strongly depends on the MPI technique the following section gives a brief introduction to it, in regard to this particular application.

### 3.1. Message Passing Interface (MPI)

In parallel computing the Message Passing Interface (MPI) is the de-facto standard for implementing programs on multiple processors. MPI defines C, Fortran and in the latest version C++ language functions for executing point-to-point communication in a parallel program. MPI has proved to be an effective model for implementing parallel programs and is used by many of the world's most demanding applications (weather modeling, weapons simulation, aircraft design, etc.) [2].

In the message passing model, a process is executed on each node of the system. Operations are performed primarily on data local to its node, i.e., stored within its memory. Therefore partitioning or decomposition of the data, so that individual computational objects are assigned to individual processors, has to be performed as well. This decomposition can be either static, i.e., fixed, or dynamic, i.e., changing in response to its running process. Please note that, in the discussed parallelization kernel, the case of static data decomposition model is used. Frequently, for the node process to continue, data from other parts of the processing system are required. In that case messages are transmitted containing the required information. Both data passing and synchronization among concurrent processes is accomplished in this manner. Information is not shared between processes or processors except by explicit interaction through messages.

Several message passing interfaces for concurrent programs have been developed. Parallel Virtual Ma-

chine (PVM) and MPI are the most widely ones. MPI is the product of a broad effort to provide a standardized programming interface and derived much from the earlier PVM. The basic set of functions incorporated by MPI provide an unprecedented degree of portability across platforms. Although it contains over one hundred functions, MPI programs can be written using only six basic primitives. A case study presented in the following section is based on the MPI realization in Matlab with only 7 basic functions.

## 4. A CASE STUDY

The purpose of this section is to illustrate the ideas behind the analysis and design implementation and tuning a parallel program using MPI. There is obviously no way to survey all possible approaches to the design and implementation of parallel programs. A simple but non-trivial interferometric problem has been chosen, where it's implementation on a particular node and in a parallel environment is explored. The ideas behind these processes and, to some extent, the tools used are common in interferometric processing problems.

For the evaluation purposes the subset of basic MPI routines is implemented in Matlab. The basis for this implementation is MatlabMPI, [8], and Matlab*P, [9], implementations which are further developed for this specific purpose. The toolbox-like set of MatlabMPI scripts allow a Matlab program, with a certain modifications, to be run in a parallel computer environment. Hence, in this way, message passing routines are passed between Matlab processes. More specifically, Matlab scripts are written in a fashion similar to writing programs with a "compiled language" using MPI. This approach has the advantage of flexibility, i.e., theoretically it is enabled to build and parallelize the system in Matlab, as they are build in "compiled languages" with MPI. In this way a parallelization model similar to the one presented in Section 3 is achieved.

Precisely, this section compares the performance of the presented parallelization model in a non-trivial interferometric processing problem, i.e., the harmonic interpolation (oversampling) of complex data, using Matlab. For both, local and parallel FFT computations a conventional high-performance publicly available FFTW package is used, [10]. Note that in Matlab the FFTW library is implemented after version 6.5.

Even though mathematics behind the problem are relatively simple, the implementation is rather complex due to large interferometric data sets. The full scene size is approximately 1.5 billion pixels, and oversampling of such a scene by, for example a factor of 4 (in both range and azimuth) results in 6 billion pixels.

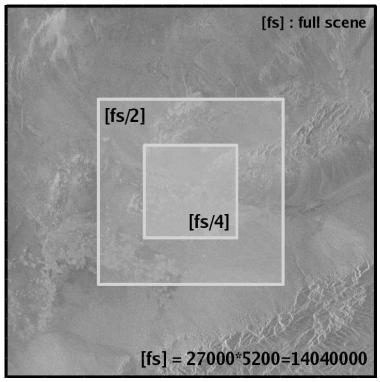


*Figure 5. Example of input data set for a case study.*

The performance is evaluated on the subsets of the real data, like shown on Fig. 5. All computations are performed via the parallelization kernel as shown in Fig. 4. The kernel divides the complex matrix among the processes, on which subsequently computations are performed, and in the last step the parallelization kernel gathers the results. The data buffering, both splitting and gathering, is utilized through the *matrix manager*, the *package manager* is responsible for calls to the specific library, while the *server manager* preforms i/o communication with nodes of the cluster system. Note that in the Matlab implementation of the discussed parallelization concept the *client manager* is omitted. In this implementation, the client manager is Matlab itself. Slave nodes are realized with the standard 100Mb network environment through Network File Sharing (NFS) and SSH protocol.

It is not an easy task to evaluate precisely the improvements achieved on the parallelization. Naturally, there is always a tendency to report near linear "speedup" or near unity "parallel efficiency", but as shown in Section 2, it proves to be difficult to reach such results.

The results of the case study are the following. The full scene oversampling on the single-unit processing environment took 1 unit of time. The simple cluster system of 3 standard Linux based cluster stations, without any system optimization, finished the task in approximately 0.40 of time unit, which means the end result is achieved 2.5 times faster. Similar results are obtained in the processing of smaller data sets and in different hardware configurations.

It can be concluded from this case study that computational time, even in the case of the relatively simple computer cluster configuration, is significantly reduced. The model is proved to be effective, despite some minor complications with implementation, e.g., data distribution and buffering over the network. Additionally, the parallelization model is not directly influenced by data manipulation which

was intended.

## 5. CONCLUSIONS AND FUTURE WORK

The final goal is an end-to-end application for the operational processing of radar remote sensing data. Components of such a system should manage every aspect of the production, including phases such as the automatic update of intermediate products following a change, auditing of historical results and the ability to trap exceptional conditions and redirect the approach to overcome these conditions. The finally developed software is currently implemented on a Linux Beowulf cluster system.

Moreover, the method for distributing small image pieces (patches) amongst slave nodes proved to be a simple but powerful model for parallel radar image processing. Accordingly, as the result of the case study showed, this and similar implementations appear to be scalable on a wide range of processors and computer languages. Since almost no communication between the slave nodes is necessary, the parallel version can be executed on nearly any configuration.

A further study and more advanced development in algorithms profiling and analysis is still required to enable a system to be presented for a generic purpose in the most demanding user contexts.

## REFERENCES

[1] A. Ferretti, C. Prati, and F. Rocca. Permanent scatterers in SAR interferometry. *IEEE Transactions on Geoscience and Remote Sensing*, vol. 38, 5:2202–2212, September 2000.

[2] W. Gropp, S. Huss-Lederman, A. Lumsdaine, et. al. *MPI: The Complete Reference (Vol. 2)* The MIT Press, September 1998.

[3] B. Kampes, R. Hanssen, and Z. Perski. Public domain tools in radar interferometry. In *Third International Workshop on ERS SAR Interferometry, 'FRINGE03', Frascati, Italy, 3-5 Dec 2003*, page cdrom, 9 pages, 2003

[4] W. Gropp, E. Lusk, and T. Sterling, editors. *Beowulf Cluster Computing with Linux, 2nd ed.* The MIT Press, 2003.

[5] R. Hanssen. *Radar Interferometry: Data Interpretation and Error Analysis.* Kluwer Academic Publishers, 2001.

[6] C.W Chen and H.A. Zebker. Phase unwrapping for large SAR interferograms: Statistical segmentation and generalized network models. *IEEE Transactions on Geoscience and Remote Sensing*, vol. 40, 8:1709–1719, August 2002.

[7] J.S. Lim. *Two-Dimensional Signal and Image processing.* Prentice-Hall International, London, 1983.

[8] J. Kepner and S. Ahalt. MatlabMPI. Accepted for publication in *Journal of Parallel and Distributed Computing*, 2004

[9] R. Choy and A. Edelman. Parallel Matlab: Doing it right. Available on-line at *http://www-math.mit.edu/~edelman/*

[10] M. Frigo and S.G. Johnson. FFTW: An Adaptive Software Architecture for the FFT. In *ICASSP conference proceedings*, vol. 3, pp. 1381-1384, 1998.