# Cost-Effective HPC Clustering For Computer Vision Applications

Julia Dietlmeier, Seán Begley and Paul F. Whelan
*Centre for Image Processing and Applications*
*RINCE, Dublin City University*
*Dublin, Ireland*
{*julia.dietlmeier, sean.begley, paul.whelan*}*@eeng.dcu.ie*

## Abstract

*We will present a cost-effective and flexible realization of High Performance Computing (HPC) clustering and its potential in solving computationally intensive problems in computer vision. The featured software foundation to support the parallel programming is the GNU Parallel Knoppix package with Message Passing Interface (MPI) based Octave, Python and C interface capabilities. The implementation is especially of interest in applications where the main objective is to reuse the existing hardware infrastructure and to maintain the overall budget cost. We will present the benchmark results and compare and contrast the performances of Octave and MATLAB.*

## 1. Introduction

Filtering, data-transforms, image segmentation, tracking and target recognition algorithms are among the applications in computer vision which are challenged by the size of the data sets, number of parameters to estimate and the need for real-time processing. The idea of replacing a single workstation by a cluster of machines, connected through a high-speed network, has been proven to be an efficient solution for accelerating time consuming batch processing, which could span over several hours, days or even weeks [1]. Many numerical algorithms can be equally implemented using either task or data parallelism.

The first category of task parallel problems includes applications such as multi-slice MRI scan processing and pattern search. A task parallel algorithm can easily be segmented into parts, each solved independently and synchronously on a separate compute node with very little communication in between. The second category embraces data parallel models which are best described by high-level vector and matrix operations on large data sets, shared among various compute nodes. Fourier transform and pixel-based image operations on the data array, therefore, can be performed separately for each element of the array. The HPC implementation of data parallelism results in additional communication overhead and requires an interconnect medium with high-bandwidth and low-latency properties.

In both cases, algorithm developers can clearly benefit from the emerging multicore processor market. The slowdown of Moore's Law and the growing demand for powerful computing have driven the hardware industry to develop processor chips that contain multiple computation cores [2,4]. Multicore CPU technology and the persistent price decline of personal computers facilitate the use of distributed and parallel computing on HPC clusters as an attractive cost option when compared to expensive integrated mainframe solutions. Many academic and research institutions can utilize the existing computing facilities in setting up an HPC cluster which can spread across different labs and buildings.

The distinct feature of HPC clustering is its excellent scalability that allows a user to resize a cluster by adding more compute nodes on demand. In addition, distributed HPC presents a robust and redundancy free solution to power supply failure. If not designed properly, however, the HPC performance can be limited by the CPU throughput, storage resources and the network efficiency. In addition, such factors as the structure of the algorithm, size of the data set, choice of the operating system and the reliable middleware will determine the degree of success in parallelizing an application [3]. In this paper, we present our experiences with building, configuring and testing an HPC cluster and discuss the performance results.

## 2. Software Survey

For our HPC project we identified the main design constraint as the reuse of the existing hardware infrastructure and approached the problem of selecting the appropriate software from the perspective of a vision researcher and therefore with the emphasis on the numerical data analy-

sis and processing. As a result we focused on the very high-level interpreted languages (VHLL), which also offer a more intuitive way of programming in comparison to the compiled languages. Ideally, an HPC software package should be an *all-in-one* product and provide not only job and resource management but also cluster configuration and administration options. Addressing our priorities in the selection of HPC software, we defined the additional criteria as: support for heterogenous hardware platforms, support for multiple VHLLs, user-friendly GUI and low cost maintenance.

Parallel programming tools, which use low-level constructs, such as a Message Passing Interface (MPI), a Parallel Virtual Machine (PVM) and parallel communication libraries (ScaLAPACK), form the class of back-end support products. An abstraction layer on top of the middleware is used by the front-end support tools to hide the technicalities of parallel programming from the user.

From the broad range of commercially available parallel programming tools we took a closer look at front-end products, such as Star-P from Interactive Supercomputing (ISC) and Distributed Computing Engine (DCE) and Distributed Computing Toolbox (DCT) from MathWorks. Both products offer a user-friendly way of parallel programming in the sense that the developer does not need to maintain master-slave communication between different cluster machines.

Star-P supports MATLAB, Python and R VHLLs and features the Software Development Kit (SDK) which enables a user to plug-in existing Java and C++ applications. ISC, a former start-up from the MIT Lincoln Laboratory, has developed and commercialized Star-P in response to MathWorks's initial reluctancy to invest in the field of parallel programming [10].

Published testimonials and case studies on the ISC website list numerous high-profile and well-funded research institutions such as the Air Force Research Lab (AFRL), NASA and the National Cancer Institute (NIC) among their customers [11]. Though acknowledging its capabilities we admit, that the only limitation of Star-P, when applied to our case, is its primary compatibility with expensive commercial platforms such as the SGI Altix 8-P server and therefore the associated costs. As a stand-alone product, Star-P does not explicitly support parallel computing on distributed heterogenous platforms.

To the contrary, DCT and DCE tools from MathWorks which were subsequently introduced in 2004 to fill-in the overlooked gap, are specifically tailored to support distributed computing. Despite its popularity and the vast availability of toolboxes, the downside of using MATLAB in HPC clustering is the MathWorks's licensing policy. The cost factor is directly proportional to the cluster size as each compute node requires an individual MATLAB license for the DCT toolbox.

For the reasons above we changed the direction towards the open source alternatives to MATLAB in search of a tradeoff between the cost and the flexibility.

### 2.1. ParallelKnoppix

There are two main open source alternatives to MATLAB: Scilab developed by INRIA with ENPC and GNU Octave developed by John W. Eaton at University of Texas. Both run on different platforms and support parallel programming through PVM-based ProActive Interface for Scilab and MPI Toolbox (MPITB) for Octave. To decide between both we have prioritized the asynchronous message passing and data-parallel capabilities of MPI over the dynamic process spawning feature of PVM and consequently arrived at the ParallelKnoppix (PK) software featuring MPITB for Octave [5,7,8].

ParallelKnoppix is a Debian based Linux distribution which runs from a Live-CD and supports *ad hoc* HPC clusters. Ad hoc implies that software does not access the hard disk and after the shutdown leaves the machines in their original state [5]. Developed by Michael Creel at the Universitat Autònoma de Barcelona, ParallelKnoppix is the *all-in-one* product which is aimed at the easy creation, configuration and maintenance of an HPC cluster. That is, the user of a cluster is also its system administrator. ParallelKnoppix offers a parallel programmer the choice of using Octave, Python and C environments to develop applications and supports up to 200 node clusters.

## 3. Cluster Architecture

The cluster can be used as well for production as for prototyping of a parallel application. This experimental setup is composed of one Master Node and seven Compute Nodes, connected by a high speed Gigabit network. Table 1 summarizes the details of the HPC hardware platform.

**Table 1. HPC Cluster Hardware Platform.**

| Component | Details |
|---|---|
| Machine | 8 x Dell 490 Precision workstation |
| Processor | Intel Xeon E5320, Quad Core 1.86 GHz 64-bit Technology Intel EM64T |
| RAM | 4GB |
| L2 Cache | 8MB |
| Default OS | Windows XP |
| PXE | Broadcom UNDI PXE-2.1 v8.3.8 |
| PCI | PCI 2.3, PCI Express 1.0A, PCI-X 2.0A |
| Interconnect | Gigabit Ethernet GigE |

When HPC operation is required, boot settings in BIOS menu can be changed to determine the location of Linux OS. The Master Node is set to boot a Linux kernel from the internal CD-ROM drive. Compute Nodes are set to boot the Linux OS from the boot server (Master Node) over the cluster network using the Preboot Execution Environment (PXE) protocol [5,13].

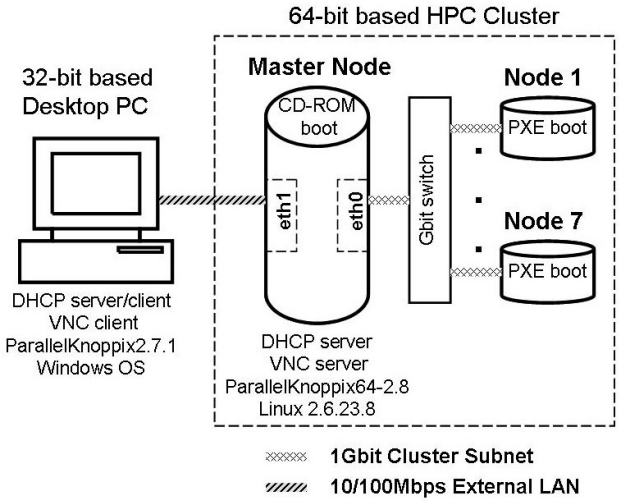


**Figure 1. HPC Cluster Configuration.**

We anticipated that the user can develop and prototype an application on the local desktop and use the HPC cluster mainly for production. Therefore, we defined two systems *Desktop* and *Cluster*, as illustrated in Fig. 1.

### 3.1. Desktop

This is considered to be a 32-bit based desktop, which can be used to develop and prototype an application and to remotely access the cluster. A desktop with any operating system, with the exception of Linux, requires virtualization software such as VMware or QEMU for the deployment of ParallelKnoppix. We used the freely available VMware Server 1.0 and designed a desktop-based, shared-memory, completely virtual cluster which runs on the local Windows Host and is composed of multiple Linux-based virtual Compute Nodes. This configuration offers a virtual model to test the MPI communication but naturally does not yield any computational gain. A second option is to cluster several virtual machines across several physical hosts [13]. Therefore, the configuration of a desktop PC and a laptop provides a simple platform to test the real-time communication. For the cluster-desktop interconnect we used the Virtual Network Computing (VNC) service with a secure *ssh* link.

### 3.2. Cluster

We refrained from using virtualization software on the Master Node as it considerably limits the overall system throughput. The software platform is the 64-bit version, v2.8, of ParallelKnoppix featuring Linux kernel 2.6.23.8, KDE 3.5.8, openMPI 1.2.4, Octave 2.9.14, parallelpython 2.5rc, scipy 0.60 and numpi 1.0.3.1.

During the setup process of ParallelKnoppix, the Master node enables its own Dynamic Host Configuration Protocol (DHCP) service and supplies dynamic IP addresses to anyone present on the subnet. The presence of an additional DHCP server on the subnet will trigger a conflict in assigning IP addresses.

Failure-free cluster networking is therefore only possible with the physical presence of two network controllers on the Master Node. The integrated Broadcom NetExtreme 57xx controller with Tigon3 chip will be used for the Gigabit cluster subnet. The PCI v2.3 compliant RealTek RTL-8139 plug-in card will enable the external LAN connection. Once the DHCP enabled, the network structure will be a simple C-class network where all nodes included in the cluster will reside within this network. The Master Node will be the only workstation within the cluster connected to the external LAN [13].

## 4. Applications

Of the two types of parallel processing, we concentrated on task parallelism, where a user can repeatedly run an experiment in a *for*-loop or in nested *for*-loops. The overall loop can be subdivided into segments and sent to compute nodes for processing. Loop index variables enable the user to trace back the correct location of the processed segment to assemble the global result.

Data parallelism is a far more complex and time-consuming approach. The time investment required to write the code may outweigh the time savings in the parallel execution of an algorithm unless the parallel functions will be used extensively over a long period of time.

Our experimental setup aims to outline both, performance gain and bounds on HPC performance and involves different aspects of computer vision. To demonstrate a suitable application of task parallelism we selected the two dimensional Fast Fourier Transform (2D-FFT), the Continuous Wavelet Transform (CWT) and the Perspective Image Transform algorithms along with two types of matrix creation approaches, such as the random array generation and the vector-by-vector multiplication. Our design of experiments consists of running multiple $(n = 100)$ trials of a user function and recording the total execution time which contains a communication latency due to the message passing constructs. In the multi-node HPC configuration, we

divided the $n$ trials evenly among the cluster and calculated the execution time according to the arithmetic mean definition. For the best possible and objective comparison between Linux based Octave and Windows XP based MATLAB we did not utilize quad core capabilities of Intel Xeon E5320 CPU and let each operating system determine the available resources.

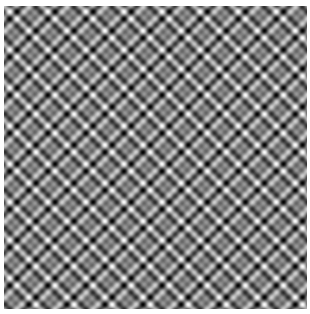

**Figure 2. Benchmark test pattern used for image transform tests.**

For the design of data transform experiments, we created a test image such as a synthesized texture composed of linear combination of pure sine waves of different random orientations as shown in Fig. 2.

## 4.1. Matrix Creation

In this experiment we generated a random square matrix with Gaussian $\{\mu = 0, \sigma = 1\}$ distributed entries and raised each of the elements to the power of 1000 [15]. This is a standard benchmark experiment that tests both, the speed at which matrices are created and the speed of the internal element by element processing.

**Table 2. Mean execution time required to generate a random matrix and raise each element to the power of $1000$ (seconds $\cdot\ 10^{-3}$).**

| Matrix Size | MATLAB | Octave with n Nodes | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| $200^2$ | 22.18 | 10.8 | 16.1 | 13.1 | 12.2 | 11.7 | 11.4 | 11.2 | 11.2 |
| $300^2$ | 49.53 | 23.4 | 22.7 | 17.6 | 15.8 | 14.5 | 13.8 | 13.3 | 12.8 |
| $400^2$ | 87.97 | 41.2 | 31.9 | 23.9 | 20.4 | 18.4 | 17.1 | 16.2 | 15.3 |
| $500^2$ | 137.2 | 64.5 | 44.2 | 32.4 | 27.2 | 23.9 | 21.9 | 20.2 | 19.2 |
| $600^2$ | 197.5 | 91.3 | 58.3 | 41.3 | 33.8 | 29.2 | 26.2 | 23.8 | 22.5 |
| $700^2$ | 268.6 | 125 | 76.9 | 53.5 | 46.4 | 37.1 | 35.0 | 30.2 | 27.8 |
| $800^2$ | 350.6 | 165 | 97.7 | 66.4 | 52.7 | 45.1 | 42.1 | 36.2 | 32.4 |

Data in Table 2 show that the single node Octave execution of the user function requires less time to perform a single calculation in comparison to the single node MATLAB execution. In addition, significant time saving is achieved for cluster configurations of more than one compute node.

## 4.2. 2D-FFT

The second experiment demonstrates the increase in computational speed achieved through the successive addition of compute nodes to the cluster. Although the implementation of the 2D-FFT differs between Octave and MATLAB, the accuracies are comparable down to an error of $1 \times 10^{-12}$, which for this work is considered negligible. Data in Table 3 show that Octave outperforms Matlab in the single node configuration.

**Table 3. Mean execution time of a 2D Fast Fourier Transform (seconds $\cdot\ 10^{-3}$).**

| Image Size | MATLAB | Octave with n Nodes | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| $16^2$ | 0.780 | .506 | 11.0 | 9.97 | 9.82 | 9.81 | 9.82 | 9.82 | 9.87 |
| $32^2$ | 1.100 | .559 | 11.3 | 9.90 | 9.83 | 9.80 | 9.79 | 9.81 | 9.87 |
| $64^2$ | 1.400 | .637 | 11.4 | 9.95 | 9.86 | 9.87 | 9.84 | 9.88 | 9.92 |
| $128^2$ | 2.500 | .971 | 11.6 | 10.1 | 10.0 | 9.96 | 9.92 | 9.93 | 9.98 |
| $256^2$ | 7.190 | 2.35 | 12.5 | 10.8 | 10.5 | 10.3 | 10.9 | 10.8 | 10.8 |
| $512^2$ | 37.19 | 15.1 | 19.0 | 15.0 | 13.7 | 13.6 | 13.2 | 12.9 | 12.6 |
| $1024^2$ | 465.5 | 85.2 | 54.4 | 38.4 | 31.6 | 27.5 | 24.9 | 23.0 | 21.3 |
| $2048^2$ | 2042 | 400 | 214 | 146 | 113 | 92.1 | 79.8 | 71.5 | 63.8 |

Adding more compute nodes to the cluster significantly reduces the total execution time, subject to the image size being larger than $512 \times 512$. The execution time of 2D-FFT of images smaller than $512 \times 512$ is comparable to the amount of time required to send messages.

## 4.3. Continuous Wavelet Transform

The Continuous Wavelet Transform (CWT) is commonly used for image compression and image filtering, and as such forms a good benchmark test for any system. The CWT produces a more accurate time-frequency spectrum but as an algorithm is also more demanding in both, computing power and memory resources when compared to the 2D-FFT. Therefore, given the same number of nodes and image sizes we expect the 2D-CWT processing to be slower than that of 2D-FFT. Our two-dimensional CWT algorithm is based on the 2D inverse FFT implementation of the two-dimensional convolution of the data sequence with the scaled and translated version of the mother wavelet. As the computational load for the 2D-CWT considerably increases with the number of scales, we limit the experiment to single-scale computation.

**Table 4. Mean execution time of a Continuous Wavelet Transform at a single scale (seconds).**

| Image Size | MATLAB | Octave with n Nodes | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| $16^2$ | 0.142 | .022 | .029 | .023 | .021 | .020 | .019 | .019 | .018 |
| $32^2$ | 0.147 | .022 | .022 | .018 | .016 | .015 | .014 | .014 | .014 |
| $64^2$ | 0.142 | .026 | .025 | .019 | .017 | .016 | .015 | .015 | .014 |
| $128^2$ | 0.240 | .044 | .034 | .025 | .021 | .019 | .018 | .017 | .017 |
| $256^2$ | 0.262 | .140 | .082 | .057 | .046 | .039 | .035 | .032 | .029 |
| $512^2$ | 1.607 | .640 | .333 | .229 | .171 | .140 | .120 | .107 | .093 |
| $1024^2$ | 6.570 | 3.16 | 1.59 | 1.09 | .802 | .644 | .548 | .483 | .422 |

Data in Table 4 show that Octave outperforms MATLAB on a single node system and that the speed-up is enhanced with the addition of extra compute nodes. This is particularly true for images of larger sizes.

### 4.4. Perspective Image Transform

Perspective Image Transform, also called "image warp", is the standard algorithm used in ordnance image rectification or camera scene registration. For the parallelization, we wrote a user function which transforms each 2D pixel $p_I$ in the benchmark image $I$ by the $3 \times 3$ perspective homography matrix

$$H_{3\times 3} = \begin{pmatrix} 0.9 & 0.1 & 10 \\ -0.05 & 1.05 & -3 \\ 0.0001 & -0.00012 & 1 \end{pmatrix} \quad (1)$$

using the nearest neighbour backward transformation method to obtain the warped output pixel $p_W$ as:

$$p_W = H^{-1} p_I \quad (2)$$

Data in Table 5 indicate that in this experiment, MATLAB outperforms Octave on a one node system. Octave catches up in HPC configurations with 3 or more compute nodes.

## 5. Performance Limitations

This test implements vector-by-vector multiplication to create a matrix in the form $A = a^T * a$ and emphasizes the additional messaging overhead associated with sending and receiving MPI calls. The initial overhead linked to the setting up of our parallel platform amounts to about one second in total. This can be seen in Fig. 3 as a significant jump in the mean execution time when the tests are computed on two compute nodes rather than one.

**Table 5. Mean execution time of a function that perspectively warps the benchmark image (seconds).**

| Image Size | MATLAB | Octave with n Nodes | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| $64^2$ | 0.008 | .010 | .016 | .014 | .013 | .012 | .012 | .012 | .012 |
| $128^2$ | 0.016 | .027 | .025 | .019 | .017 | .016 | .015 | .014 | .014 |
| $256^2$ | 0.066 | .108 | .064 | .046 | .037 | .032 | .029 | .027 | .025 |
| $512^2$ | 0.316 | .466 | .244 | .167 | .127 | .104 | .091 | .081 | .072 |
| $1024^2$ | 1.350 | 2.04 | 1.04 | .708 | .525 | .421 | .359 | .317 | .276 |
| $2048^2$ | 5.431 | 9.71 | 4.86 | 3.30 | 2.44 | 1.95 | 1.66 | 1.47 | 1.27 |

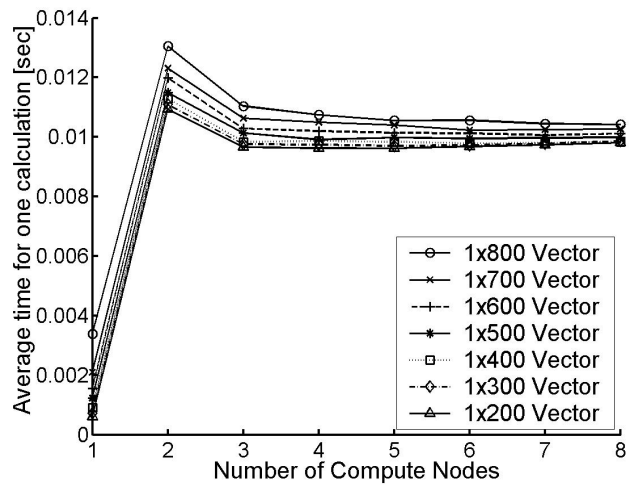


**Figure 3. Mean execution time of an $a^T * a$ operation (seconds).**

The time spent on MPI communication increases with the number of compute nodes and eventually can become larger than the time saved due to parallelization. This can be observed in Fig. 3 as the gradual increase in the mean execution time for the number of compute nodes larger than 3.

## 6. Discussion

The experiment in Section 4.4, which was designed to assess the Perspective Image Transformation, did not demonstrate MATLAB's full potential as the written image transform function did not contain any nested *for*-loops. Generally, an implementation of nested *for*-loops is slow for all interpreted languages. MATLAB features a Just-In-Time (JIT) compiler to speed-up computations [12]. Octave, at present, doesn't support JIT compilation, and so runs much slower when *for* and nested *for*-loops are encountered.

## 7. Conclusion

Based on experimental results we verify that especially for the case of processing large data sets our Octave based HPC cluster yields significant time savings.

Moreover, 8-node Octave outperforms single node MATLAB as illustrated in Table 6.

**Table 6. Computational gain of the 8-node ParallelKnoppix and Octave based HPC, compared to the single node MATLAB.**

| | |
|---|---|
| Matrix creation | $t_{OCTAVE} = t_{MATLAB} \setminus 10$ |
| 2D-FFT | $t_{OCTAVE} = t_{MATLAB} \setminus 32$ |
| CWT | $t_{OCTAVE} = t_{MATLAB} \setminus 15$ |
| Image warp | $t_{OCTAVE} = t_{MATLAB} \setminus 4$ |

A prerequisite for writing parallel code under ParallelKnoppix is the knowledge of low-level MPI programming. The developer tells the cluster machines how to communicate, which part of the code to execute and how to assemble the end result. It is a challenging but, according to our experience, also a straight forward and an intuitive approach to parallelizing an application.

We prioritize the cost factor above the user-friendliness of the GUI and conclude that a ParallelKnoppix based HPC cluster provides a cost-effective and computationally efficient platform to solve large-scale numerical problems.

The reader is referred to our Computing Cluster User's Guide [13] for a detailed description on configuration and maintenance of the presented HPC ParallelKnoppix Cluster and for a short tutorial on parallel Octave.

We also would like to advise the reader of the next generation PelicanHPC project [14], which is aimed to substitute ParallelKnoppix in the future.

## Acknowledgment

This work was funded as part of the HEA-PRTLI IV National Biophotonics and Imaging Platform Ireland (NBIPI).

## References

[1] C. S. Yeo, R. Buyya, H. Pourreza, R. Eskicioglu, P. Graham, and F. Sommers. *Handbook of Nature-Inspired and Innovative Computing: Integrating Classical Models with Emerging Technologies*. Springer Science+Business Media Inc., New York, USA, 2006.

[2] N. Bliss. Addressing the Multicore Trend with Authomatic Parallelization. *Lincoln Laboratory Journal*, 17(1):187-198, 2007.

[3] Y. -C. Fang, S. Iqbal and A. Saify. Designing High-Performance Computing Clusters. *Dell Power Solutions*, May 2005.

[4] K. Asanovic, R. Bodic et. al. The Landscape of Parallel Computing Research: A View from Berkley. *http://www.eecs.berkley.edu/Pubs/TechRpts/2006/EECS-2006-183.html*, 2006.

[5] M. Creel. ParallelKnoppix Tutorial. *http://pareto.uab.es/mcreel/ParallelKnoppix/Tutorial/*.

[6] R. Choy and A. Edelman. Parallel MATLAB: Doing It Right. *Proc. of the IEEE*, 93(2):331-341, 2005.

[7] M. Creel. User-Friendly Parallel Computations with Econometric Examples. *Computational Economics*, 26:107-128, 2005.

[8] M. Creel. I Ran Four Million Probits Last Night: HPC Clustering With ParallelKnoppix. *Journal of Applied Econometrics*, 22:215-223, 2007.

[9] S. Raghunathan. Making a Supercomputer Do What You Want: High-Level Tools for Parallel Programming. *Computing in Science and Engineering*, 8(5):70-80, 2006.

[10] C. Moler. Parallel Matlab: Multiple Processors and Multiple Cores. *The MathWorks News and Notes*, June 2007.

[11] Star-P Success Stories. *http://www.interactivesupercomputing.com/success/*.

[12] S. Clanton. Speeding Up the Scientific Process. *Linux Journal*, 2003, *http://www.linuxjournal.com/node/6722/print*.

[13] J. Dietlmeier and S. Begley. S129 Computing Cluster User's Guide. February 2008, available online at: *http://www.vsg.dcu.ie/code/IMVIP08_CIPA_Cluster_User's_Guide.pdf*.

[14] PelicanHPC Project resources. *http://pareto.uab.es/mcreel/PelicanHPC/*.

[15] S. Steinhaus. Comparison of mathematical programs for data analysis. Edition 5.03. 2008, online at: *http://www.scientificweb.de/ncrunch/*.