Grid-enabled estimation of structural economic models

Zhorin, Victor and Stef-Praun, Tiberiu

Computational Institute

4 November 2008

Grid-Enabled Estimation of Structural Economic Models

Tiberiu Stef-Praun and Victor Zhorin

November 4, 2008

Abstract

In this paper we present our experiences with the execution of structural economic models over the Grid using "cloud computing". We describe cases of distributed implementation and execution of occupational choice and financial deepening models of economic growth. We show how the application of Grid technology and resources naturally fits the studies of economic systems, by allowing us to capture effects of computationally challenging real-world characteristics such as heterogeneity of wealth, talent and access costs among economic agents.

1 Introduction

According to definition given by Ian Foster [1] a Grid is a system that:

1. coordinates resources that are not subject to centralized control;
2. using standard, open, general-purpose protocols and interfaces;
3. to deliver nontrivial qualities of service to meet complex demands.

One can not help but notice the conceptual similarity of the Grid with standard Walrasian economic world having Walrasian auctionereer in form of Grid-coordinating processes to deal with complex issues of inadequate resource supply, incomplete (by design) information available for each processing node, stochastic demand, heterogeneity of capabilities in available computing agents as well as intrinsically unavoidable failures happenning both due to the random shocks (hard drive crash, data segment missing in transmission, coding errors) and systemic factors (system is down for maintanaince, overwhelming demand that prevents certain tasks from executing).

We can envision the whole hierarchy of computing systems ranging from top-performing half-a-billion worth IBM BlueGene supercomputers to clusters based on charity-donated desktop computers as a structure reflecting real distribution of economic agents with different endowment in wealth and talent, different access to information datasets and varying setup costs.

> Algorithms encapsulated into the methods of a particular agent can only be implemented using the particular information, reasoning tools, time, and physical resources available to that agent. This encapsulation into agents is done in an attempt to achieve a more transparent and realistic representation of real-world systems involving multiple distributed entities with limited information and computational capabilities. [2]

Historically, the most powerful computing systems served the purpose of advancing the progress in natural sciences and applied engineering, while economics as a science developed using relatively parochial computational tools and techniques, arguably at the expense of human societies which were often reduced to poverty and misery resulting from untested economic and policy innovations.

The wedge between the desire of modern society to better understand and control financial and economic shocks rapidly propagating through all population strata and the possibility to discover true relationships between underlying economic parameters from existing economic theories is stunning.

While we are in a very early stage of using distributed computing systems to properly model the richness of economic agents behavior in real world, it is clear that lowering barriers to access grid computing is an essential step to scale economic models up to the level required.

This paper discusses techniques which we have employed to enable the development, deployment and execution of Matlab-based economic models in a distributed environment such as the academic Grids.

For a primer case, we concentrate on two non-trivial exaples: one involves testing Lloyd-Ellis & Bernhardt (LEB) occupational choice model using large spatially linked dataset covering $\geq 75\%$ of Thailand territory and the other involves constructing artificial economies from Greenwood-Jovanovich (GJ) model of economic growth with endogeneous financial deepening. Original LEB and GJ models were taken from qualitative to to quantitative estimation level by Robert Townsend, Xavier Gine, Hyeok Jeong, and Kenichi Ueda [3, 4, 5].

2 The Models of Economic Growth, a Primer and Examples

Understanding the failure of convergence in developing countries growth has been one of the key puzzles in economics. Solid rejection of factors of growth such as investments in physical capital, human capital in form of education, difficulty in proving causal link between economic growth and political institutions and weak empirical evidence of the hypothesis on "techological backwardation" lead to legitimate question "why hasn't the world improved like we thought it would?"[6]

As Abhijit Banerjee and Esther Duflo wrote in [7]:

> ...what we need to explain is less the overall technological backwardness and more why some firms do not adopt profitable technologies that are available to them (though perhaps not affordable).

Or, more generally, the basic problem of economics (more specifically of mechanism design) of how to properly structure incentives so that people respond to them in optimal way, what is stopping people from choosing the kind of activities that could generate higher return (better life) is still largely unanswered.

Though the problem is most acute in developing countries its implications go beyond that and apply to developed countries as well. However, for empirical testing such possible growth factors as credit constraints, market failures to insure enterpreneurial risks can be more clearly collected from rich economic dynamics in transitional economies where different policies and changing conditions dramatically affect the welfare of population with directly observable survey data available.

Computational estimate of those factors is challenging due to intrinsic heterogeneity among the agents and non-trivial barriers in access to capital which do not allow to obtain easily computable closed-form solutions to test on empirical data. The underlying complexity of economic models is unavoidable to properly account for observable "anomalies", which in fact are so widespread that it is tempting to rather call perfectly clearing markets to be an exception.

2.1 Capital-Constrained Occupational Choice

We start with occupational choice model since it would allow to demonstrate the simplest way to transfer the computation process to large, publicly available academic computing resources.

The basis of this model is the household choice of occupation. A household at each time t has a choice to receive income at "safe" (subsistence) level γ, work as unskilled laborer at non-farm enterprise at wage w or start a firm with potential profit π determined by:

$$\begin{aligned} \pi(b_t, x_t, w_t) = \underset{k_t, l_t}{max} \; \{F(k_t, l_t) - w_t l_t - x_t\}; \\ s.t. \; k_t \in [0, b_t - x_t], l_t \geq 0 \end{aligned} \tag{1}$$

where production function has the most general quadratic form, which can be viwed as an approximation to large class of production functions possible

$$F(k_t, l_t) = \alpha k_t - \frac{1}{2}\beta k_t^2 + \sigma k_t l_t + \xi l_t - \frac{1}{2}\rho l_t^2 \tag{2}$$

l_t is supply of unskilled labor at market-cleared wage w_t, k_t is a capital utilized, b_t is initial wealth at start of the period t, x_t is a randomly drawn enterpreneurial cost from the following distribution

$$H(x, m(d)) = m(d)x^2 + (1 - m(d))x \tag{3}$$

The distribution of setup cost x for entrepreneurs is what adds diversity and complexity to the model. Those costs can vary both amongst different social groups and amongst different spatially separate entities as well. It is not directly observable but known to be distributed in the population according to density $H(x, m(d))$. Parameter of distribution m depends on spatial characteristics d and it is to be estimated from strcuctural modeling.

The end of period wealth available for consumption-saving choice would then depend on possible three types of occupational choice

$$W_{t+1}(b_t, x_t, w_t) = \begin{cases} \gamma + b_t, \text{ if stays at subsistence level} \\ w_t - \eta + b_t, \text{ if works for hire} \\ \pi(b_t, x_t, w_t) - x_t - \eta + b_t, \text{ if starts a business} \end{cases} \tag{4}$$

Parameter η is an additional cost of living outside subsistence sector.

This model also takes into account financial intermediation by using exogeneusly specified financial deepening.

Townsend and Gine [3] explain in details how some parameters are calibrated to match initial sample data and others are recovered from empirical data using maximum likelihood function estimations. This procedure, however, becomes computationally demanding as empirical dataset increases to include larger sets of economic agents participating with tambons (smallest agglomeration of villages) as representative economic agents. Since this optimization problem is highly-nonlinear in nature with unknown and non-convex structure of objective function it is hard to speed up computations just by using grid resources for each point estimate as such.

However, to avoid bottleneck problems we can use cross-sectional MLE parameters estimation by sorting the sample by some characteristic (for example, distance from major roads or cities). Once the sample is sorted into bins we can run estimation procedure in parallel for each subsample and recover parameter variations as dependent from characteristic chosen to vary across the sample. This procedure needs relatively few powerful computers running in parallel, exclusively dedicated to this task for several days. However, if the problem is opposite, that is, if we need for LEB to model for small samples requiring hours rather than days to converge and if we need to do estimation for a multitude of such samples, then we want to consider using larger number of grid nodes with less demand put on exclusive usage.

This dichotomy leads us to the next case to consider where parallelization is essential and it naturally follows from the model structure.

2.2 Financial intermediaries and Risky Projects Insurance

Townsend-Ueda quantitative model of economic growth with endogeneous financial deepening [5] is based on the canonical model of Greenwood-Jovanovich (GJ). It is a stochastic dynamic programming model with forward-looking agents maximizing return of investment portfolio by choosing proportion of "risky" and "safe" assets. In terms of household s "safe" return is guaranteed by staying in low-return occupation and " risky" business is any enterpreneurial activity with uncertain outcome. The safe projects return is δ and enterpreneurial activity can result in $\theta_t + \epsilon_t^s$, where θ_t is aggregate stochastic shock and ϵ_t^s is idiosyncratic shock specific for particular household s.

An economic agent can also choose to use financial services to finance risky projects. Financial intermediaries provide two major services: 1) they provide insurance against idiosyncratic shocks by pooling funds from many enterpreneurs; 2) they select particular projects by having informational advantage over the agents in terms of better assessments of survivability of the projects under aggregate shocks. When choosing to apply to financial services the agents pay fixed entry cost $q > 0$ and variable per period cost $(1 - \gamma) \in [0, 1]$. A household s does not have to stay with the same project all the time and it can rebalance risky project portfolio allocation $\phi_t^s \in [0, 1]$ each time period Saving decisions are also endogeneous in this model and agents are heterogeneous in initial wealth.

Given parameters of the model (discount factor, entry cost, risk aversion, etc.) we can analytically estimate value functions for agents who join financial system and those who are never allowed to join due to restrictions. It is harder to obtain value function for agents who are not joining and those value functions are calculated numerically. There is no market clearance in this model, each agent return is independent of others but since return depends on wealth, the aggregate shocks will affect heterogeneously distributed agents differently.

With value and policy functions calculated based on pre-calibrated set of model parameters we can simulate the growth path of artificial economy subjected to a sequence of shocks lasting 17 years for which we have observable data. Then we perform Monte-Carlo simulations by randomly varying the sequence of shocks with *the actual* growth path of economy representing one of the possible Monte-Carlo realizations.

To deploy this model on the grid we have to clearly deliniate what each artificial economy needs in terms of inputs, how to colect the outputs and make final decision about choosing particular economy of interest out of thousands simulated.

We restructured the code to allow for different mode of operations with each artificial economy running on its own dedicated node with particular policy function supplied. The set of policy functions is generated by master nodes, each of which can correspond to a different set of underlying microeconomic parameters. Once the master nodes finish the computations of policy functions they send them to as many worker nodes as available and wait untill they finish time-path of development under those policies. In the final stage a collector checks results and compares them to observable time paths of GDP per capita growth, Theil inequality index and financial participation. This scheme allows flexible estimation of model parameters as well as trials with easily replacable objective function and quick selection of particular economy of interest.

3 High Performance Computing with The Grid

In an academic environment setting, High Performance Computing usually means high-throughput execution of the research models on Grid computing pools. We introduce in this section the concepts that define computational Grids, the interfaces they provide to the users and the ways to efficiently use such large computational resource pools.

3.1 Grid Infrastructure

Large scale computational resources can be classified in two main groups: tightly-coupled systems, which pack together in a highly controlled, compact, and efficient way a large number of identical hardware resources (e.g. IBM BlueGene supercomputers), or loosely coupled systems, which are generally built from commodity computers, connected through a local area network, and controlled by a job manager software that allocates the resources as needed.

Given the significant popularity of the loosely coupled parallel computing environments, illustrated by the large computing Grids of the academic world (TeraGrid, OSG, EGEE, etc) and by the similar infrastructure successes from the commercial world (Amazon Elastic Computing Cloud, Google search engine infrastructure, commercial Grid resource rentals from Sun, IBM, etc), we will focus our discussion on these models. However similar principles and discussions apply to tightly-coupled systems as well.

The structure of a Grid consists of a large pool of computers (or nodes), all dedicated to running applications (or *jobs*) on behalf of the users, and the pool has one special node in the pool, the *head node*, which is the place where the user submits jobs to the pool, through the job manager interface. An important observation for the application developer: the computing resource pool can be abstracted through the interface it exports to the user: the user submits an application together with its inputs to the computing cluster, then the job manager sends all these to one of its available resources (nodes), and after the execution ends, it returns the results to the user. This *job submission* abstraction is useful in two ways: First, sending jobs to the Grid is conceptually similar to making a function call, so the application developers should feel comfortable with making their applications Grid-ready. We discuss this in detail in the following section. Second, the universal nature of the application invocation interface for submitting tasks to computational resources has been designed to hide away the heterogeneity and the complexities of using many machines in parallel, and it has allowed for the development of the Globus [8] middleware, a standardized universal layer for accessing multiple Grid sites.

3.1.1 Using Computational Grid sites

The central idea has already been introduced: as long as the application can be described as a set of tasks, and even better, if those tasks can be run in parallel, independent of each other, the procedure of running an application on the Grid is reduced to submitting the component tasks with the proper inputs to the job manager in the right order.

Sending a single job to the Grid usually involves making sure that the machines belonging to that Grid can actually run the task that the user submits: the hardware, the operating system and all required libraries for the application should be suitable for desired application. This step is usually referred to as “installing the application into the Grid”. Once the application is installed, the multiple invocations of can be easily instantiated through the job manager. In different grid installations there are various job managers software solutions: Condor, PBS, LSF, Sun Grid Engine, etc. The more general alternative is to use Globus-enabled grids, which provide a universal layer on top of the heterogeneous Grid sites, and hides away the specifics of each resource pool through a standard job description interface.

Often, when the application is too big and requires very long time to complete, or when reuse of the various application components is needed, the model will need to be divided in several blocks, some of which are interdependent. In such cases, the way to run the decomposed model on the grid is to construct the dependency graph for the execution order of the blocks, usually by determining the files that link outputs from one component to the inputs of other components. Then that graph is passed to a *workflow engine* that knows how (in what order) to send the components to be executed on the grid. This procedure is similar to scripting, where the script interpreter is a special piece of software that a) enforces the dependencies between the pieces and b) takes care of the execution of the components.

There are several Grid-focused workflow-expressing tools, such as Condor's DagMan, Swift (from Globus), and we will introduce here our experiences with the Swift workflow/scripting system.

3.1.2 Distributed execution overhead

The benefits of Grids (mainly simplified/standardized access to large computer pools) do not come without a cost. Given that the resources are distributed across a network, there are latencies associated with transferring the data to the machine allocated to the task. Also, especially in the academic Grids, one has to keep in mind that these are *shared resources*, and therefore the user might have to wait its turn to be serviced by some nodes in the cluster. This wait in the *job queue* can some times be significant, as the general rule is to serve jobs on a *first-come-first served basis*, with priority given to smaller taks. A rule of the thumb is to have tasks that are no shorter than 3-5 minutes and no longer than 2 hours, otherwise the overhead of using Grids might become significant. There are reservation mechanisms or batching facilities to solve these issues, but they also require some overhead to be set up.

3.2 Setting up a Computational Environment with Matlab

The academic Grid sites have been set up to support scientific research. However, the heterogeneous requirements of each individual research model leads in practice to the requirement that Grid users *set up their own environment*: they need to make sure that their specific application can be run on that specific Grid's nodes.

Matlab is a very popular environment to build up economic models with an optimization library (either internal to Matlab, such as *fmincon* or some external one (*knitro*, *ipopt*, etc) plugged in. Hence, setting up the scientific environment for computational economists translates into making available both Matlab and solvers on the Grid.

The case of setting up a Matlab environment on the Grid is very illustrative, especially since one has to deal with restrictions due to the fact that Matlab is commercial software and it needs a license in order to be run. In a Grid environment, that means that it needs licenses on all the machines where the pieces of Matlab codes are sent for execution, which could be very costly. Fortunately, Matlab allows for the distribution of their run-time libraries (MCR) under non-restrictive license, and as long as the researcher compiles his Matlab model (using the procedure described below) into an executable, setting up the code for the Grid simply consists of the compilation and installation of the MCR libraries and of the executable onto the Grid site.

Here is how one compiles Matlab code: Assume that the code consists of a main program "main.m" and two dependencies "dep1.m" and "dep2.m". Below is included a snapshot from the Makefile that builds the executable "mainapp" from the codes above:

```
compile-master:
    /soft/matlab-7.5-r1/bin/mcc -o mainapp  \
    -W main -T link:exe -d  . -v  main.m  -a dep1.m -a dep2.m
```

The result is a set of files in the bin folder, and the most important ones are "mainapp" and "run_mainapp.sh", and these, together with the MCR libraries are the ones that need to be installed on the Grid site. The run_mainapp.sh then needs to be invoked with the first argument the path to the MCR libraries' Grid installation location, e.g.:

```
/home/ba01/u102/stef/local/run_mainapp.sh \
    /home/ba01/u102/stef/local/MCR-Purdue-X64/v77
```

It is worth mentioning that other alternatives are also available for running Matlab applications on the Grid. One relevant example is using Octave, a free, fully Matlab-compatible programming environment, in conjunction with invoking external libraries for solvers or other specialized codes. A good reference is our previous work done to run a Moral Hazard economic model on the Grid [9].

The working environment for the Grid can be considered as being set up when it is ready to execute the application given proper inputs. This is a one-time effort, after which, the general strategy consists of feeding proper inputs to the same grid enabled application, as required by the economic model. We detail below how this can be automated, the final goal being to obtain the expected, correct results from the economic model.

3.3 Executing decomposed applications on the Grid with Swift

What does it take to run an application consisting of several self-sufficient blocks ? In the general example above we considered the Matlab code to be the block that describes the model, and the numerical solver to be the block that actually produces the results of the model. One has to note that the link between the two blocks consists of the optimization problem description (be it a set of equations or some numerical expression of those equations) which *is output the Matlab block* and which is fed *as an input* to the optimization software. For generality reasons, and to accommodate the possibility that the Matlab block and the optimization block can each reside and execute on different machines in the Grid, we choose to pass the outputs of upstream blocks as inputs to downstream blocks as *files*. The steps for running the case above consists of executing the (compiled) Matlab code on some Grid site, retrieving the output (the representation of the optimization problem to be solved) and passing it as a parameter to the solver block (which can take place on some other node in a completely different Grid site). This simple example also reiterates the usefulness of the simple interfaces made available by Grids: the execute interface, and the file transfer interface.

In section four, we extend the discussion with significantly more complex economic models, and we illustrate how we have decomposed those models into blocks. In this section, we continue with the presentation of Swift, a Grid scripting and workflow execution tool that enables users to express and execute applications composed of modules with relatively more complex dependencies amongst each other.

3.4 Swift Grid scripting tool

Swift is both a modeling language and an execution engine for the Grid. The modeling part of Swift allows one to define the application blocks as *atomic procedures*. Each of these atomic procedures are mapped by Swift into one or mode installations of the corresponding specific code block on some Grid site.

Having multiple installations on different Grids for the same atomic block means that Swift will be able to send in parallel all the invocations of that atomic block to all the defined Grid endpoints. These atomic procedures usually produce files as outputs (to allow for the transfer of their results between execution nodes).

Swift also allows for using standard programming constructs such as loops, conditional expressions and also for defining and manipulating complex data structures (arrays, etc). All these are generally used to build *compound procedures* which are generally used to express parallelism in invoking the atomic procedures with different parameters (e.g. in parameter sweeps).

Handling of the inputs and outputs happens through a mapping mechanism which maps (usually) file objects within Swift to files names on the local (as an input file) or remote machine (as a result file).

Once the problem *workflow* is defined (by expressing all the dependencies - i.e. input and output files - between all the atomic procedures), invoking Swift results in the execution of all those application blocks, in parallel if possible, as long as they have the inputs defined (either as files that have been provided locally by the user or as long as the upstream blocks which generate inputs for the current block has been run successfully).

Swift will interface directly with the Grid sites, and hide the details of file transfers and application block execution. It also provides its own scheduler which balances the load of submitting applications to the Grid sites, and it also handles errors and task resubmissions.

4 Grid Mapping and Execution of the economic models

Coming back to the models mentioned in section two, we detail in this section the decomposition of those models, the coding in Swift, mention issues with preparing and running the models, and display and comment the results of their execution on the Grid.

4.1 Occupational choice model

For the case of the Occupational Choice model, the main "driver" for the parallel execution is the input data set.

```
file modelOut, file paramOut) compiledLEB (file sampleIn, file paramIn, file dependencies[]){
app{
compiledLEB @filename(sampleIn) @filename(paramIn) @filename(modelOut) @filename(paramOut);
}}
```

We are driving the execution of the whole model by providing the sample data *sampleIn* from the various spatial regions of the economy (at tambon level, as mentioned above) in parallel as inputs, and by having the model run on each of these data sets in parallel. The speed advantage from the Grid comes from being able to logically disaggregate the Thailand input data into administrative sub-regions, and from having those sub-regions-driven models run independently of each other.

This version of running the model is more "lightweight" because the model already has access to proper pre-computed structural parameters (in the *paramIn* file) that resulted from fitting the model to the data. For an actual structural estimation run, we vary the *paramIn* file (which contains the combinations of the model's parameters), and run the model again as a single block in the the loops that vary the parameter values. The power of Grid really comes in when your model is amenable to be run in loops (see the foreach block below).

```
//having defined some ranges for the parameters ...
foreach mIndex in mRange {
foreach omegaIndex in omegaRange{
foreach gIndex in gRange{
//define output files to receive results
file outParam<single_file_mapper; file=@strcat("param-",mIndex,"-",omegaIndex,"-",gIndex,".out")>;
file outModel<single_file_mapper; file=@strcat("model-",mIndex,"-",omegaIndex,"-",gIndex,".out")>;
//invoke the LEB model
(outModel, outParam) = paramCompiledLEB (sampleIn, m+mStep*mIndex, omega+omegaIndex*omegaStep, \
beta, alpha, rho, sigma, gamma, xi, g+gIndex*gStep, nyu, dependencies);
} } }
```

4.2 Financial Intermediation model

The Financial Intermediation model is a better case for our discussion because it can be decomposed into computational blocks. Reviewing the description of this model from section two, one can observe that there exists a "logical decomposition" of the model: one can isolate the part that computes the policy and value function for the agents; also the part that simulates the effect of the shocks on the economies can be run as a stand-alone component (and all-together in parallel, replicated in N instances, as determined by the Monte-Carlo procedure); and also the last part where the results of the parallel runs are collected and compared to the real data from the economic surveys.

It is important to remember that each of the block introduced above is very likely to be executed on a different machine in the resource pool. This means that the process of splitting up the code into pieces also requires a *global variables export* stage at the end of the block, together with a *global variables import* stage at the beginning of the following block, to ensure the “continuity” of the code across different computing resources. This is equivalent to a user-chosen checkpointing step in the code, where the relevant variables in the memory are save to the disk, then the code is shipped off to a new machine, the variables are re-loaded into memory, and the process continues. For the case illustrated above, all the variables determined by the value function and policy function computation, together with the variables provided by the user as inputs are saved into the $policy_file$ and passed on to all the parallel simulation blocks following the main procedure.

New to the monolithic-application Matlab developer is also the procedure that collects all the outputs of the individual simulations into a single data structure in memory, for further processing (the comparison of the computed paths in our example). This scenario is known as a *fan-in* procedure, and has two main aspects to it. The first aspect is the management of the parallel execution processes, and of their outputs. Swift, as the parallel execution engine, ensures (read: retries) on behalf of the user that all the blocks are run and all the outputs are copied back from the remote resources, and that no subsequent blocks are run before *all* the outputs have become available on the local machine (the one which runs the application workflow through Swift). This introduces the second aspect: the naming of the output files needs to be such that when all the files are copied back they do not have conflicting (identical) names, and also the names should be descriptive enough for the subsequent block to be able to extract and store the results from each of those files into its proper location in memory, in its corresponding variable.

5 Conclusion

We have reported our experiences with two complex economic models, and we gave the reader an idea about the thought process and about the computational tools that we have employed to map computationally constrained problems onto large computing pools. We would like to emphasize that designing and implementing models in a modular fashion, and keeping in mind the possibility of running those codes in parallel on Grid computing resources, is often a significant way of improving one's capacity of solving bigger and more realistic models.

The main result that we have achieved with decomposing and parallelizing these codes is a significant (in some cases) speedup. For instance, with the Financial Intermediation model, parallelizing the simulation of the shocks on the economy has produced a 20x speedup as compared with the same run on a single machine. These improvements are highly dependent on the structure (internal dependencies) of the decomposed model and on the computational resources that we were able to acquire at the time of running the models.

Regardless of the application specifics, the advantages which make Swift a useful tool are the *automated management of the execution* of the models' blocks and the obscuring of the details on interacting with and using of the Grid computing resource pools.

We would like to acknowledge Robert Townsend and Ian Foster for initiating and supporting this work

References

[1] Ian Foster, 2002. *What is the Grid? A Three Point Checklist.*, http://www-fp.mcs.anl.gov/ foster/Articles/WhatIsTheGrid.pdf.

[2] Leigh Tesfatsion, Agent-based Computational Economics: A Constructive Approach to Economic Theory, 2006. *Handbook of Computational Economics*, Volume 2., Elsevier.

[3] Xavier Gine and Robert Townsend (2004). Evaluation of Financial Liberalization: a General Equilibrium Model with Constrained Occupation Choice. *Journal of Development Economics* 74,269-307.

[4] Hyeok Jeong and Robert Townsend (2008). Growth and Inequality: Model Evaluation Based on an Estimation-Calibration Strategy. *forthcoming, Macroeconomic Dynamics.*

[5] Robert Townsend and Kenichi Ueda, Financial Deepening, Inequality, and Growth: A Model-Based Quantitative Evaluation. *Review of Economic Studies* (2006) 73, 251-293.

[6] William Easterly (2002), *The Elusive Quest for Growth: Economists' Adventures and Misadventures in the Tropics*. MIT Press. 356 pp.

[7] Abhijit Banerjee and Esther Duflo (2005). Growth Theory through the Lens of Development Economics. In P. Aghion and S. N. Durlauf (Eds.), *Handbook of Development Economics*, pp. 473-552. Elsevier B.V.

[8] Globus: A Metacomputing Infrastructure Toolkit. I. Foster, C. Kesselman. *Intl J. Supercomputer Applications*, 11(2):115-128, 1997.

[9] Stef-Praun, T., Madeira, G., Foster, I., and Townsend, R. (2007) Accelerating solution of a moral hazard problem with Swift *e-Social Science Conference, Ann Arbor, MI*