Parallelization, Optimization, and Performance Analysis of Portfolio Choice Models

Ahmed Abdelkhalek and Angelos Bilas
Dept. of Elec. and Comp. Eng.
10 King's College Road
University of Toronto
Toronto, ON M5S 3G4, Canada
{abdel, bilas}@eecg.toronto.edu

Alexander Michaelides
Department of Economics
University of Cyprus
P. O. Box 20537
1678, Nicosia, Cyprus
alexm@ucy.ac.cy

Abstract

In this work we show how applications in computational economics can take advantage of modern parallel architectures to reduce the computation time in a wide array of models that have been, to date, computationally intractable. The specific application we use computes the optimal consumption and portfolio choice policy rules over the life–cycle of the individual. Our goal is two–fold: (i) To understand the behavior of a class of emerging applications and provide an efficient parallel implementation and (ii) to introduce a new benchmark for parallel computer architectures from an emerging and important class of applications.

We start from an existing sequential algorithm for solving a portfolio choice model. We present a number of optimizations that result in highly optimized sequential code. We then present a parallel version of the application. We find that: (i) Emerging applications in this area of computational economics exhibit adequate parallelism to achieve, after a number of optimization steps, almost linear speedup for system sizes up to 64 processors. (ii) The main challenges in dealing with applications in this area are computational imbalances introduced by algorithmic dependencies and the parallelization method and granularity. (iii) We present preliminary results for a problem that has not been, to the best of our knowledge, solved in the financial economics literature to date.

1. Introduction and Background

Household portfolio choice and asset management models are an emerging area in computational economics that have generated significant research interest. Household portfolio choice models determine the optimal amount of savings and its optimal allocation in alternative assets with the goal of maximizing the expected discounted sum of future utility. The optimal choices necessarily depend on the economic environment (preferences, endowments, investment opportunities and various market frictions like the inability to borrow) and the computational problem can become more or less difficult to solve depending on the assumptions that the researcher is willing to make. Asset management models determine the optimal asset allocation decision with the goal of maximizing the expected utility of wealth. These models are more appropriate for fund managers where real time decisions require quick computational algorithms to make the models operational.

Recent progress in computational economics has led to the desire of attacking these large scale problems using computational methods in a number of areas in economics and finance [2, 13, 15]. Generally, a common numerical method employed to solve these problems discretizes the endogenously evolving state variables and computes the optimal policy rules as functions of these variables [7, 20]. Working with multiple state variables results in complicated models that suffer from the "curse of dimensionality" [18], or exponential growth of the configuration space with the number of state variables. Moreover, limiting the configuration space, may have an impact on the accuracy of the results. In household portfolio choice models, for instance, the menu of assets is usually limited to a riskless asset and a risky stock market investment opportunity. In asset management models, the horizon is usually kept short to limit the level of uncertainty to (computationally) manageable levels and to a point that can offer real time advice to fund managers. Solution methods that would speed up the computational time would achieve both the introduction of more realistic economic assumptions in the models but also provide real time advice to professional fund managers.

In this work we explore whether parallel systems can enable researchers to study economic models for real–life problems that have not been feasible to date. We focus on a particular economic model that has recently received substantial attention in the financial economics literature [6–8, 11, 20]. The version

of the model we use [9] solves for the optimal consumption and portfolio choice policy rules over the life–cycle. It assumes the presence of undiversifiable labor income risk, one risky asset that earns the historically observed equity premium, one riskless asset, a precautionary savings motive, a no borrowing constraint, and a no short sales constraint. This model is known to have high computational demands.

Recent improvements in parallel architectures have resulted in efficient hardware distributed shared memory (DSM) multiprocessors that require modest effort in parallelizing existing sequential applications and they perform well for wide classes of applications [12]. We choose to work with a shared address space (SAS) abstraction for the following reasons: (i) The SAS abstraction is a high level abstraction that does not require explicit data movement. (ii) The shared address space abstraction is currently provided across a large number of architectures: small scale SMPs, tightly–coupled hardware DSM machines, and loosely–coupled clusters. (iii) One of our goals is to drive systems work in efficiently supporting a shared memory abstraction on modern architectures. For this reason we are interested in using this application as a benchmark representative of an emerging class of applications. The platform that we choose for our experiments is a 64–processor hardware distributed shared memory (DSM) system, an SGI Origin2000 [14] that is representative of today's aggressive hardware cache–coherent systems.

In this work we first port an existing sequential version of the application [9] to the environment we use and describe its behavior, computational, and memory access characteristics. Then we present a number of optimizations over the original sequential algorithm. In particular, we present a heuristic that improves the convergence speed and practically makes the computational time of the sequential version independent of the initial guesses used. Then, we provide an efficient parallel implementation for the shared address space abstraction. We present the major issues in parallelizing this class of applications, we propose a parallel algorithm, and describe its behavior and computational characteristics.

Our high–level contributions and conclusions are: (i) Using problem specific knowledge we improve the execution time of the sequential code by a factor of about 15. This gives us confidence that our parallelization effort is compared with a highly optimized sequential version. (ii) Emerging applications in this area of computational economics exhibit adequate parallelism to achieve, after a number of optimization steps, almost linear speedup for system sizes up to 64 processors, on today's tightly–coupled hardware–DSMs. (iii) The major challenges in this class of applications are imbalances in the computation and the induced synchronization. Data locality and coherency traffic is not a problem despite the producer–consumer memory access pattern to the main data structures, due to the small communication to computation ratio.

The rest of this paper is organized as follows. Section 2 describes at a high level the particular model we use. Section 3 describes our testbed. Section 4 describes the sequential version of the code and the optimizations we apply. Sections 5 and 6 discuss the parallelization of the application and our results. Section 7 presents related work. Finally, Section 8 draws our conclusions.

2. Economic Problem Statement and Model

We first summarize the most basic aspects of the consumption and portfolio choice model to be parallelized. More details can be found in [9]. The model computes the optimal consumption policy and portfolio choice over the life–cycle. There is one non-durable good, one riskless financial asset, and a risky time varying investment opportunity. The riskless asset yields a *constant* gross after tax real return, R_f, while the gross real return on the risky asset is denoted by $\widetilde{R}$. Time is discrete. At time t, the agent enters the period with invested wealth in the stock market $S(t-1)$ and the bond market $B(t-1)$ and receives $Y(t)$ units of the non-durable good. Following [8], cash on hand in period t is denoted by $X(t) = S(t-1)\widetilde{R(t)} + B(t-1)R_f + Y(t)$. The investor then chooses savings $B(t)$ in the bond market and $S(t)$ in the stock market to maximize welfare. Given these choices, the consumption policy function $C(t)$ is determined by the relation $C(t) = X(t) - S(t) - B(t)$.

The particular assumptions made about the economic environment are as follows:

(i) Preferences are of the constant relative risk aversion family: $U(C(t)) = \frac{C(t)^{1-\rho}}{1-\rho}$ when $\rho > 0$ and $\rho \neq 1$; if $\rho = 1$, then $U(C(t)) = \ln C(t)$.

(ii) The agent lives for a maximum of T periods ($0 \leq t < T$), and retirement occurs at time K, $K < T$. For simplicity K is assumed to be exogenous and deterministic. We allow for uncertainty in T in the manner of [11]. The probability that a consumer/investor is alive at time $t+1$ conditional on being alive at time t is denoted by $p(t)$ ($p(0) = 1$). Bequests are not left at the end of life: the numerical solution can accommodate a bequest motive, as part of future work.

(iii) No borrowing and no short sales of stocks are allowed; $B(t) \geq 0$ and $S(t) \geq 0$.

(iv) The exogenous stochastic process for individual earnings is given by $Y(t) = P(t)V(t)$ and $P(t) = GP(t-1)N(t)$, as described in [6]. The process $Y(t)$ is decomposed into a permanent component $P(t)$ with a shock $N(t)$ and a growth rate $g = \ln G$ and a transitory component $V(t)$. $\ln V(t)$ and $\ln N(t)$ are i.i.d. normal with mean $\mu_v = -.5 * \sigma_v^2$ and $\mu_n = -.5 * \sigma_n^2$, and variances σ_v^2 and σ_n^2, respectively.

(v) We assume that there is a single factor that can predict future excess returns. If we let $r_f, r(t), f(t)$ denote the net risk free rate, the net stock market return, and the factor that predicts future excess returns respectively, then $r(t+1) - r_f = f(t) + z(t+1)$. The autoregressive factor $f(t)$ predicting future returns (e.g. the dividend yield) follows a persistent process given by

$f(t+1) = \mu + \phi(f(t) - \mu) + \varepsilon(t+1)$. $z(t+1), \varepsilon(t+1)$ are innovations that can be contemporaneously correlated, μ is the unconditional mean of $f(t)$, and ϕ ($\phi > 0$) measures the persistence strength of $f(t)$.

In this economic environment, the individual's goal is to maximize

$$MAX_{\{S(t),B(t)\}_{t=1}^{T}} E(1) \sum_{t=1}^{T} \beta^{t-1} \{\Pi_{j=0}^{t-1} p(j)\} U(C(t)),$$

where $E(1)$ is the expectation conditional on information available at time $t = 1$, and $\beta = \frac{1}{1+\delta}$ is the constant discount factor. The two Euler equations at any given age t are given in [9]. Given the nonstationary process followed by labor income, we normalize by the permanent component of earnings $P(t)$ (see [6]). If we let $\lambda(C) = C^{-\rho}$ denote the marginal utility of consumption, define $Z(t+1) = \frac{P(t+1)}{P(t)}$, take advantage of the homogeneity of degree $-\rho$ of the marginal utility function, and label the m factor states $i, j = 1, ..., m$, then there are m bond and stock demand functions defined by the following two normalized Euler equations:

$$\lambda(c(x(t),t,i)) = MAX[\quad (1)$$
$$\lambda(x(t) - s(x(t),t,i)),$$
$$\frac{1+r}{1+\delta} E(t)\{\{Z(t+1)^{-\rho}\}\lambda(c(x(t+1),t+1,j))\}]$$
$$\lambda(c(x(t),t,i)) = MAX[\quad (2)$$
$$\lambda(x(t) - b(x(t),t,i)),$$
$$\frac{1}{1+\delta} E(t)\{\widetilde{R}(t+1)\{Z(t+1)\}^{-\rho}\lambda(c(x(t+1),t+1,j))\}]$$

In these equations the consumption policy function c and the normalized equivalent $x(t)$ of the endogenous state variable $X(t)$ are given by:

$$c(x(t+1),t+1,j) = x(t+1)$$
$$-s(x(t+1),t+1,j) - b(x(t+1),t+1,j)$$
$$x(t+1) = V(t+1)$$
$$+\{Z(t+1)\}^{-1}(s(x(t),t,i)\widetilde{R}(t+1) + b(x(t),t,i)R_f)$$

Finally, the variable j denotes the next period of factor value, $E(t)$ denotes the expectation conditional on information at time t, and lower case variables are normalized by $P(t)$.

2.1. Computational Method

Equations (1) and (2) form a system of two equations in two unknowns $s(x(t),t,i)$, $b(x(t),t,i)$. In this work we first solve the simplest problem where the next period stock return cannot be predicted by current variables and therefore the factor i does not enter as a separate state variable in the optimal control problem; the two unknown functions to be solved for are then given by $s(x(t),t)$, $b(x(t),t)$. In Section 6 we present preliminary results for the full problem.

The two unknown functions $s(x(t),t)$, $b(x(t),t)$ are computed from equations (1) and (2) using the following iterative algorithm. Starting with an assumption about the consumption policy function $c(x(t+1),t+1)$ in the terminal period of life $t = T-1$, we solve simultaneously this system of Euler equations using backward induction and a discrete grid over cash on hand, $0 \le x(t) < xmax$. For each year t, $s(x(t),t)$, $b(x(t),t)$ become functions of one state variable $x(t)$. The sequential implementation for the above model computes the consumption policy function $c(x(t),t)$ by computing $s(x(t),t)$, $b(x(t),t)$ from the set of equations (1) and (2).

Two questions arise: (i) Do solutions for $s(x(t),t)$, $b(x(t),t)$ that satisfy (1) and (2) exist? (ii) Are these solutions unique? If we assume that $c(x(t+1),t+1)$ is given and is an increasing function of $x(t+1)$, then one can show that given $s(x(t),t)$ the right hand side of (1) is decreasing in $b(x(t),t)$ while the left hand side is increasing in $b(x(t),t)$. This guarantees existence and uniqueness of a solution for $b(x(t),t)$ from the bond Euler equation. The argument works in exactly the same fashion for $s(x(t),t)$ by symmetry given $b(x(t),t)$ for (2).

The iterative sequential algorithm [9] we use takes the following form:

```
for (t = T-2; t >= 0; t--) {
  for (v = 0; v <= GRID; v++) {
    s(x(t,v), t) = guess();
    repeat {
      s'(x(t,v), t) = s(x(t,v),t);
      b(x(t,v),t) = BisectEq1(s'(x(t,v),t));
      s(x(t,v),t) = BisectEq2(b(x(t,v),t));
    } until ( |s'(x(t,v),t)-s(x(t,v),t)| < e );
  }
}
```

(i) Given an initial guess for $s(x(t),t)$, equation (1) is a non–linear equation with one unknown $b(x(t),t)$. Thus, we can solve for $b(x(t),t)$ using a standard bisection algorithm [13].
(ii) Given $b(x(t),t)$ from the previous step, find with the same method $s(x(t),t)$ from equation (2).
(iii) If the maximum of the absolute differences between the initial $s(x(t),t)$ and its update from the previous step is less than a convergence condition, then a solution has been found and the algorithm proceeds to the next pair $s(x(t-1),t-1)$, $b(x(t-1),t-1)$. If not, the updated $s(x(t),t)$ becomes the new guess and we go back to the first step.

Alternative methods for solving this set of equations are discussed in Section 7.

3 Experimental Platform

For our work we use a 64–processor SGI Origin2000 with 16 GBytes of main memory and routers connected in a full hyper–cube topology [14]. The SGI Origin2000 is a system that is considered "aggressive" in terms of the communication architecture for today's standards. Table 1 compares the Origin2000 with other modern, hardware–DSM systems.

The nodes in the system are connected with a hardware cache–coherent interconnect that provides a shared memory abstraction to the programmer. Each pair of nodes share a router [14]. Routers are connected in a hypercube topology. Each network link has a peak bandwidth of 780 MBytes/s. The minimum (uncontented) latency for accessing remote memory (in clean state) is about 650ns.

Each node has two 300 MHz MIPS R12000 processors. Each processor has separate 32–KByte 2–way set–associative first–level instruction and data caches, and a unified, 8–MByte second–level cache with a 128–byte block size. The two processors in a node share a *hub*, memory, a communication controller (which sees all cache misses and incoming transactions), and a non–coherent memory bus. The memory bus in each node has a theoretical peak bandwidth of 780 MBytes/s. Coherency is provided within nodes in the same way as across nodes at cache–line granularity.

System	Local (ns)	Remote Clean (ns)	Remote Dirty in 3rd node (ns)	$\frac{Remote}{Local}$ Ratio (Clean)	$\frac{Remote}{Local}$ Ratio (Dirty)
Origin2000	338	656	892	2:1	3:1
Convex Exemplar X	450	1315	1955	3:1	5:1
Data General NUMALiiNE	240	2400	3400	10:1	14:1
Hal S1	240	1065	1365	5:1	6:1
Sequent NUMAQ	240	2500	N/A	10:1	N/A

Table 1. Latencies and remote–to–local latency ratios on different systems. The latencies are from processor request to the response coming back to the processor.

For our measurements we use the native SGI cc compiler (MIPSpro Compilers: Version 7.30) and the –O2 optimization level. The uniprocessor runs are done on a single node of the same system.

4. Sequential Implementation

To compute $s(x(t),t)$, $b(x(t),t)$ for each year t the code discretizes the state variable $x(t)$ in a 1–dimensional grid over the interval $[0, xmax]$ and approximates the functions by a set of values on the grid, solving the equations (1) and (2) with the method presented in Section 2.1.

The grid size may vary from a few tens of points to a few hundred points to provide sufficient accuracy in approximating the consumption policy function. In our work we set the grid size always to 64 or 128. These are relatively small granularities; we make these choices mainly because we are interested in comparing our work with existing results that have been obtained so far with the original version of the code. The total number of years in the life–cycle is not expected to vary much. The model computes the consumption policy function for 45 years of working life and 35 years of retirement. The rest of the input parameters to the model are explained in [9].

Conceptually, each function $c(x(t),t)$, $s(x(t),t)$, and $b(x(t),t)$ can be represented with two 2–dimensional arrays—one for the working life and one for retirement as shown in Figure 1. The major data structure of our sequential implementation follows this conceptual representation. Each row represents a year of the life–cycle and columns represent grid points on which $c(x(t),t)$, $s(x(t),t)$, and $b(x(t),t)$ are approximated for each year.

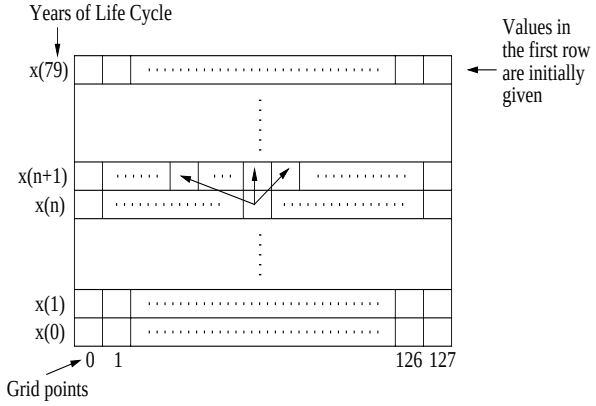

Figure 1. The two 2–dimensional arrays computed by the application, represented as a single larger array. Each element depends on a varying set of elements in the preceding row.

The values for the last year of retirement are assumed to be known and are used to compute the rest of the years in a backward fashion. The first row in the working life array depends on the last row of the retirement array. The model computes the retirement period values first ($K \leq t < T$) before proceeding to the working life period ($0 \leq t < K$). The major data dependencies in the computation of each function are that each row in the array depends on the values of the previous row (Figure 1). Specifically, whenever an element needs a value from the previous year, this value is computed by linear interpolation from the two nearest values of the previous year that already exist. This computation may be repeated a number of times for each array element point, resulting in multiple dependencies.

4.1. Problem sizes

The model input parameters affect the amount of computation that is performed. For this reason we use in our evaluation a number of problem sizes. However, due to space reasons and the time required for complete benchmarking we present full results for two problem sizes. Table 2 shows the input parameters for the problem sizes we run and their requirements on memory. The authors in [9] provide a detailed explanation of the effect of each input parameter.

The memory required by the application is exponential on the grid sizes used to discretize each of the two state variables, age t and cash on hand $x(t)$. Age is discretized in 80 values and cash on hand in 128 resulting in the 2–dimensional array of

Figure 1. Thus, memory requirements are fairly small (in the order of a few MBytes for shared data). However, future extensions to the model that make use of more state variables will result in multi–dimensional grids and in much higher memory requirements. Although the current model configuration is not memory intensive, the dependencies between data and locality issues make parallelization challenging.

Problem Size	Grid size	$xmax$	δ	g	Memory (KBytes)	Uni Time (min)
PS1	128	3	0.10	0.03	≈ 200	≈ 8
PS2	128	9	0.09	0.02	≈ 400	≈ 126

Table 2. Input parameters for the two problem sizes we consider. The parameter ρ is set to 6 for both problem sizes. The time reported includes all optimizations of the sequential code.

4.2. Sequential Code Optimizations

The application uses mostly floating point (double) operations. The original application code is written in GAUSS [3]. GAUSS favors the use of arrays and array operations as the basic constructs in programs. Thus, the original program uses a 2–dimensional array as the general data structure for all variables. All computation is performed by manipulating arrays. This results in a number of inefficiencies.

As the first step in our work we translate the application code from GAUSS to C. Initially we use similar data structures as the GAUSS program and perform the computation in a similar way. The C version is about 1.5 times slower since it has all original inefficiencies and in addition uses dynamic memory management. Before parallelizing the code we provide an optimized C–based implementation of this sequential algorithm. We optimize the code in the following ways:

Eliminating dynamic memory management: The first step in optimizing the code was to eliminate dynamic memory management and multi–dimensional array operations. This resulted in code that performed comparably to the original GAUSS application.

Improving convergence and eliminating array operations: The computation for each year involves the convergence of the value for each of the grid points to a specified convergence condition. The original GAUSS implementation of the computation involved performing the iterations for each grid point until the maximum value among all grid points has converged. By using separate convergence conditions for each grid point we remove unnecessary iterations for grid points that converge faster than others. Moreover, the new code does not require multi–dimensional array operations. The resulting version of the code executes about one order of magnitude faster than the original version.

Loop optimizations: Computing each grid point involves a number of nested loops. Moving invariant loop computations outside of loops reduces the sequential execution time by approximately 10%. Moreover, the values of some of the variables in the loop–invariant code were leading to the identical repetition of loop iterations in different levels of the computation. Eliminating extra repetitions due to zero–valued variables led to an additional increase in performance of approximately 20%.

Reducing element dependencies: As explained earlier, each element in the 2–D array requires for its computation elements in the previous row. The original code would result in more dependencies than necessary. Eliminating the extra dependencies improved sequential execution time by about 5%. More importantly, it reduced the risk for higher wait times in the parallel version of the code (see Section 5). Furthermore, we explored the possibility of eliminating the linear interpolation to reduce the number of dependencies. However, this changes the accuracy of the computed results and interacts with other model parameters such as the grid size. Since the implications of these changes on the economic model under hand are not well understood we do not explore this direction any further.

Summary: Table 2 shows the time for running the optimized sequential code. The final version of the sequential code is highly optimized and executes about 15 times faster than the original version.

5. Parallel Implementation

As explained in Section 4, the computation performed by the application generates a set of values for each element of the two 2–dimensional arrays. We parallelize the code using the M4 macros and the threads model. The only form of synchronization among threads is barriers and point–to–point synchronization implemented with either locks or flags. The major data structures of the parallel version are similar to the sequential implementation (Figure 1).

To understand the computational behavior of this class of applications and to provide an efficient implementation we explore four major parameters of the configuration space:

(i) Granularity of parallelization: Given the data dependencies induced by the model we use, the most natural granularity of parallelization is the array element level; the computation for each array element is a single task assigned to one thread. This results in a producer–consumer memory access pattern with multiple consumers for each element.

(ii) Task size: The number of array elements that are assigned to each thread within a single row, or task size. The minimum task size is one array element per task. Small task sizes result in better load–balancing but have worse memory locality and may result in false sharing.

(iii) Task assignment: Assignment of row elements to threads can be either static or dynamic. Static assignment reduces bookkeeping overheads and does not require lock synchronizations, whereas dynamic assignment improves load–balancing, but introduces fine–grain lock synchronization.

(iv) Inter–row synchronization: The dependencies among array elements (Figure 1) result in the need for inter-row synchro-

nization. The simplest solution is to use barrier synchronization among all threads. However, this is unnecessary in many cases; Point–to–point synchronization that checks only for elements of the previous row that are needed for computing the current element can reduce wait time.

To represent the different versions of the code we use the notation {B,P}{S,D}G{0,1,4}, where:

{B,P} specifies whether barriers or point–to–point synchronization is used for inter–row synchronization.

{S,D} specifies whether static or dynamic allocation of tasks is performed within each row. Dynamic allocation implies the use of lock synchronization.

G{0,1,4} specifies the task size. A value of 0 means that the system automatically sets the task size to G/T, where G is the grid size and T the number of compute threads. A value of 1 or 4 means that the task size is either 1 or 4, respectively.

5.1. Results

In our work we explore all the above configurations. However, due to time and space constraints we present full results for all processor counts only for BSG0 and PDG1 that are the most representative of the configuration space.

We present both speedups and execution time breakdowns for configurations up to 64 processors. Execution time is divided into the following components: (i) *Barriers* is the time spent in barrier synchronization. This time is measured by instrumenting the library code that is used for barrier synchronization. (ii) *Locks* is the time spent in lock synchronization. This time is also measured by instrumenting the library code. (iii) *Flag* synchronization is the time spent in point–to–point synchronization and is measured in the application itself. (iv) *Memory* time is the stall time for memory requests. (v) *Compute* time is the time the processor spends executing instructions. The sum of the memory stall and compute times is the exact time spent processing user instructions. However, the division to memory stall and compute time is approximate. We use pixie and prof to calculate the number of instructions the program executes and then conservatively translate this to compute cycles (MIPS R12000 is a quad issue processor). We attribute the rest of the time to memory stalls.

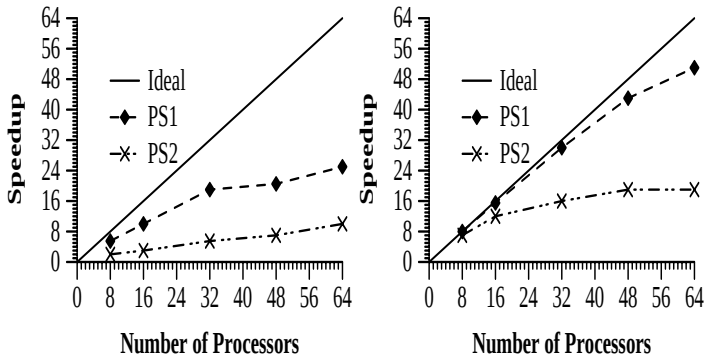

Figure 2. Speedups for BSG0 (left) and PDG1 (right).

Figure 2 presents speedups for both BSG0 and PDG1 for each problem size. We see that the small problem size (PS1) performs reasonably up to 32 processors with BSG0 but does not scale any further, whereas the larger problem size (PS2) does not perform well at all. Replacing global with point–to–point synchronization in PDG1 improves things dramatically for the small problem size. However, the larger problem size does not seem to scale beyond 16–32 processors. Thus, although fine–grain synchronization and dynamic task assignment help significantly they do not seem to address all problems. Figure 3 shows the execution time breakdowns for PDG1. We see that most of the time for the larger problem size is spent in flag synchronization. This is somewhat counter–intuitive, since usually increasing the problem size improves parallel performance. In the next section we proceed to better understand this problem and deal with the underlying issues.

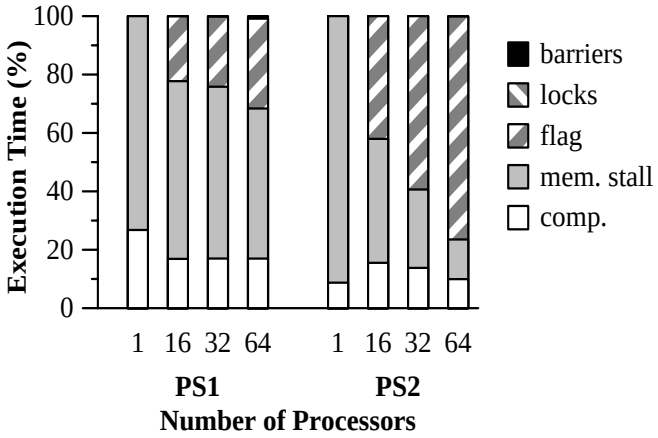

Figure 3. Average breakdowns of execution time for PS1 (left) and PS2 (right) for PDG1.

6. Addressing Computational Imbalances

Figure 4 shows the number of iterations required for each element of some rows to converge to the final value. We see that elements require very different numbers of iterations. Moreover, we find that the time for each element to converge is proportional to the number of iterations.

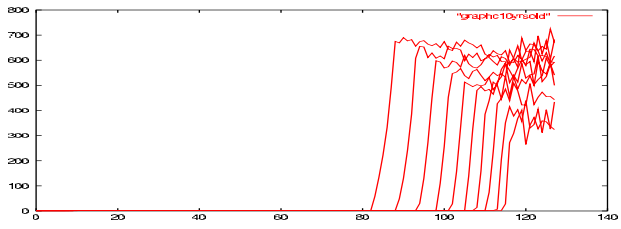

Figure 4. Number of iterations required for convergence for each element of $s(x(t), t)$ in years (rows) 35-44 for PS2. The x-axis represents the elements in one row of the array and each curve shows the number of iterations for the respective row.

The reason for the different number of iterations is that the computation for a particular element starts by making an initial guess for each control variable, e.g. $s(x(t), t)$. The algorithm then proceeds until it reaches a final value of $s(x(t), t)$ which satisfies the convergence criterion. Observing the way elements approach the final value of $s(x(t), t)$ we notice that values converge very slowly. Thus, the initial guess for each element affects the number of iterations. This suggests that for some elements the initial guess is closer to the final value than it is for other elements.

To address this issue we need to either improve the initial guess for each element or improve the speed of convergence.

Improving initial guesses: The original code uses as the initial guess for each row element the value of the same element in the previous row. This is based on the fact that the values of each function do not change in the model dramatically from year to year. Figure 5 shows the final values of $s(x(t), t)$ for all elements in the same rows as in Figure 4 for the large problem size. We see that values in the same year (row) follow approximately a linear pattern. Also, elements in successive years incur higher differences for larger values of $x(t)$. The diverging values of the functions for larger values of $x(t)$ and the very slow convergence result in very high computation imbalances.

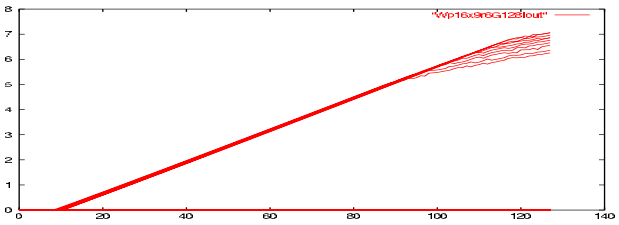

Figure 5. Final values for $s(x(t), t)$ for all elements in years 35-44 for PS2.

To address this issue we compute the initial guess by linearly extrapolating the two previous values of the same year. This is based on the observation that the output function is almost linear. Figure 6 shows the number of iterations for each element for one problem size. We see that almost all elements converge much faster. The only exception is elements where the computed function changes form, in which case linear extrapolation from the previous values does not help. Although other forms of extrapolation may address this issue, we do not explore this direction further since linear extrapolation gives good results and the exact form of the computed functions is not known.

Improving the initial guesses makes the computation of each element more balanced and reduces execution time of the sequential code by about one order of magnitude. Although this approach addresses the problem of computational imbalance, it introduces more dependencies and cannot be used in the parallel version of the code since all elements of a row are computed in parallel by different processors.

Improving the speed of convergence: In the process of converging from the initial guess to the final value, new values can

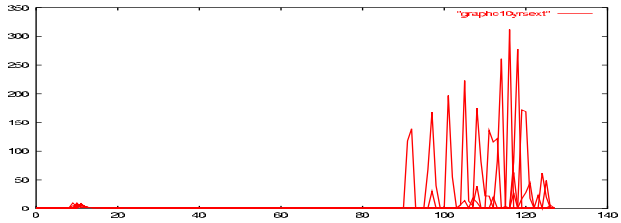

Figure 6. Number of iterations for elements of $s(x(t), t)$ in years 35-44 for PS2 using linear extrapolation.

fluctuate and they do not always move towards the final value. However, we notice that in practice new values always move towards the final value (at very small steps as noted above). We use this observation to introduce a heuristic, which we call the min–max heuristic, to improve the speed of convergence.

We implement a bisection–like algorithm to search for the final value. We first attempt to find a range in which the final value resides. We conservatively pick a fairly large interval as follows. For the lower bound of the interval we use the final value of the corresponding element in the previous year (row). To get the upper bound we add to the lower bound the difference of the final values of the elements of the two previous rows.

The obtained values for the minimum and maximum of the range are still only guesses at this stage. We then verify that these values are reasonable by making sure that if we use the lower bound as an initial guess the next value computed is above the lower bound and similarly if we use the upper bound as an initial guess the new value is less than the upper bound. As a fail–safe mechanism, if the chosen interval does not seem to contain the final value we pick another interval. This process continues a small number of times and if we are unable to find a reasonable initial interval we revert to using the original method for solving the set of equations.

Once the initial interval is obtained, a bisection method is used to find the final value. We pick the mid–point of the current interval as the initial guess and solve the system of equations to obtain a new value. If the new value is less than the value at the mid–point then we assume that the final value will be less than the value at the mid–point as well and we make the mid-point the upper bound of the current interval. If the new value is above the value at the mid–point then we make the mid–point the lower bound. Thus, each iteration reduces the size of the interval by half. Again, as a fail–safe mechanism, if the next value computed happens to be outside the range, then this element is declared to be an irregular one and we use the original method for solving the set of equations.

This method works very well and dramatically reduces the number of iterations for the expensive elements. Figure 7 shows the number of iterations required for each array element for the large problem size. Moreover, the min–max heuristic reduces the variation in execution time between different problem sizes. A side–effect is that, besides reducing the number of iterations for elements that converge slowly, it also increases the num-

ber of iterations for elements that would converge after one or two iterations in the original code; in the min–max heuristic we need to compute two initial values—one for the lower and one for the upper bound. We believe that we can improve this by using the min–max method only when necessary. However, for the purpose of this work this would result in second–order effects and we do not explore this direction any further. Finally, it is important to note that the min–max heuristic does not introduce any new dependencies and can be used in the parallel version of the code.

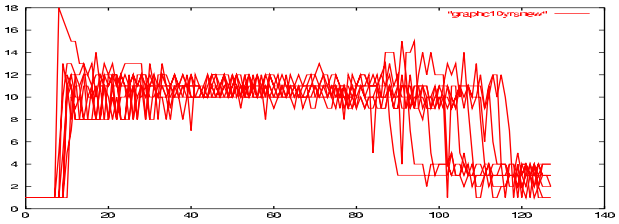

Figure 7. Number of iterations for elements of $s(x(t), t)$ in years 35-44 for PS2 using the min–max heuristic.

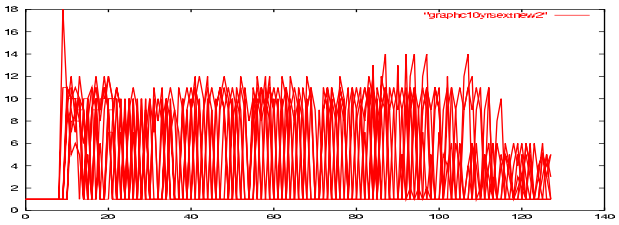

Figure 8. Number of iterations for elements of $s(x(t), t)$ in years 35-44 for PS2 using both the min–max heuristic and linear extrapolation.

Input	Original	Linear Extr.	Min–max	Both
PS1	8min	7min	29min	20min
	1, 406, 10.1	1, 319, 3.5	1, 19, 5.7	1, 103, 4.0
PS2	126min	15min	31min	22min
	1, 841, 100.2	1, 447, 5.5	1, 18, 5.8	1, 103, 4.1

Table 3. For each problem size, the first row presents the uniprocessor execution time in minutes, whereas the second presents the minimum, maximum, and average number of iterations across all array elements.

Although it is possible to combine the two heuristics, the additional improvement is not significant as shown in Figure 8. Table 3 shows the execution time and the minimum, maximum, and average number of iterations across all elements for each heuristic. In summary, although finding better initial guesses by using linear extrapolation (or a combination of linear extrapolation and the min–max heuristic) seems to perform slightly better, the min–max heuristic for improving the speed of convergence is easier to parallelize and also reduces the dependence of the convergence speed on the initial values, overall. Thus, we next incorporate the min–max heuristic in the parallel version of our code.

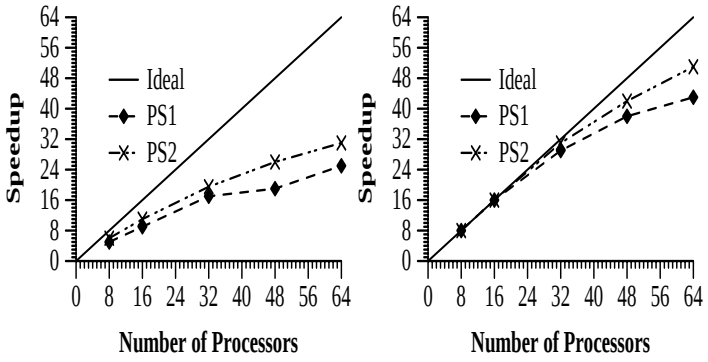

Figure 9. Speedups for BSG0 (left) and PDG1 (right) with the min–max heuristic.

6.1. Results

Figure 9 shows the speedups for both BSG0 and PDG1. We see that BSG0 performs and scales comparably for both problem sizes, but results in sub-linear speedups. When point–to–point synchronization and dynamic task assignment are used (PDG1), performance and scalability improve dramatically for both problem sizes for all processor counts.

Most of the remaining overhead in the best version of our code (PDG1) is due to wait time in flag synchronization. Figure 10 shows the execution time breakdowns for PDG1 for each problem size. The execution time breakdowns show that the ratio of memory stall time to compute cycles is about the same between the sequential and the parallel implementations or better in the parallel implementation. This shows that the parallel implementation exhibits very good locality, that the per–processors caches are helpful, and that there is no or very little false sharing.

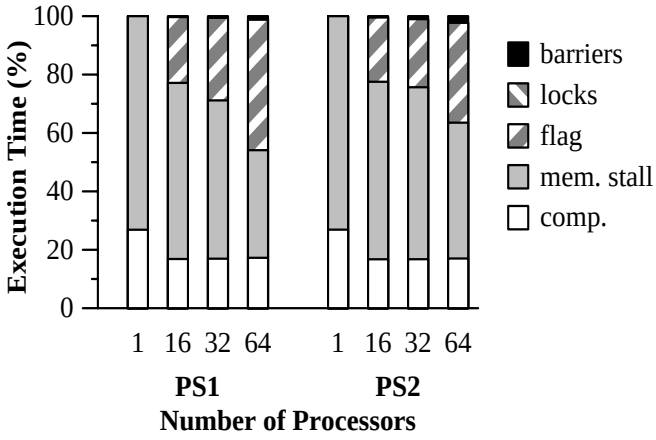

Figure 10. Average breakdowns for execution time for PS1 (left) and PS2 (right) for PDG1 with the min–max heuristic.

We should note here that the breakdown of the execution time in compute cycles and memory stall time is approximate, so it can only be used as a hint of what the actual costs are. The

less–than–ideal speedup in the parallel version for larger processor counts, e.g. speedup of about 48 for PS2 on 64 processors, is due to the increased flag synchronization time. Further reduction in synchronization time to improve overall performance would require finer–grain parallelization. Although at the processor scales we examine this would most likely result in second–order improvements, it may be important to explore this direction for systems with larger processor counts.

6.2. Increasing the number of state variables

The min–max heuristic we present to improve the speed of convergence results in uniform convergence times for all combinations of input parameters. Thus, the computational demands of this class of applications arises now only from the inherent complexity of the model that is used. The model that we started with, has now a relatively short execution time—about one half hour on a single processor and a few minutes on the 64–processor SGI Origin2000.

To demonstrate the effectiveness of our approach we now present preliminary results for a more complex model that, to our knowledge, has not been presented to date in the economics literature. In this model the next period stock return can be predicted by current variables. Therefore, a factor is introduced as an extra state variable, as described in Section 2. Table 4 shows the execution time of this model on a system with up to 24 processors and all our code optimizations (PDG1 with the min–max heuristic). Since for these runs we are not interested in detailed timing measurements, we do not obtain exclusive access to the system. Although the system was not heavily loaded at the time of the runs, speedups may not be accurate. For this reason we report only the absolute execution times.

Processors	1	4	8	16	24
Time(min)	N/A	62	32	17	14

Table 4. Time and speedup for the more complex model with 5 control and state variables. The input parameters are $xmax$=10.0, ρ=3.0, δ=0.1, g=0.03, with a grid of 50 elements.

7. Related Work

The proliferation of computational methods in economics and their use in making models in economics and finance more realistic is illustrated in [2, 13, 15, 16, 18, 23].

Examples of recent work in the area of household portfolio choice models include [9, 10] that solve for the optimal savings/stock market allocation decision in the presence of undiversifiable labor income risk, an infinite planning horizon, and liquidity constraints. Also, [7, 20] solve similar models over the life–cycle determining the optimal savings and asset allocation decision as a household ages. Interesting issues that are addressed involve scientific advice on optimal asset allocation over the life–cycle, the determination of asset prices in an economy with changing demographics, and the optimality of the currently operating social security system. Examples of strategic asset allocation models are presented in [4, 5].

The stochastic optimal control method we use in this work may be contrasted with the stochastic linear programming approach that has been followed by, for instance, [16] for solving similar asset management problems. Currently both methodologies are being used to address problems in the area. Stochastic linear programming has made progress with large scale applications in asset management precisely due to parallelization [23]. To our knowledge, our work is the first effort to parallelize a portfolio choice stochastic optimal control model over the life–cycle with undiversifiable labor income risk.

There are alternative methods for solving equations (1) and (2). The authors in [7], for instance, use grid search [13] to find the optimal policy functions. Grid search involves evaluating the function being maximized at a fine grid over the different control variables and then picking the points that maximize the function. We chose the bisection method for its simplicity but future work in this area should explore alternative methods for solving these systems. In [15] the authors present an extensive survey of alternative methods to solve non–linear systems of equations, such as weighted residual methods (chapter 6), the parameterized expectations approach (chapter 7) and finite difference methods (chapter 8). As the authors in [2] point out, however, there is no consensus as to what constitutes the *best* method to use at this point.

In the parallel computer architecture community, there has been work on architectures that efficiently support a distributed shared address space [1, 14, 21]. Another body of work [12, 17, 22] has focused on understanding the behavior of different classes of applications on shared address space systems. Finally, a number of studies, such as [19], have compared shared memory and message passing architectures for different classes of applications. Our work is complementary to all these efforts providing a new application that can be used to evaluate parallel architectures.

8. Conclusions

In this paper we study an emerging and important class of applications in computational economics and finance. We show how parallelism can be exploited to reduce the computational time in an important class of applications in computational economics. This work is (to our knowledge) among the first that examines the parallelization and behavior of portfolio choice models over the life–cycle. We start with an existing sequential implementation of a realistic computational model. We first optimize the sequential implementation using problem specific knowledge. Then we provide an efficient and scalable parallel implementation.

We find that the major challenges in this class of applications are computational imbalances and the induced synchronization due to algorithmic dependencies. Improving the speed of con-

vergence in the sequential version of the code, replacing global with point–to–point synchronization, and using dynamic task scheduling result in a parallel implementation that performs and scales well to large numbers of processors. Moreover, memory requirements are not an important issue in the particular models we use. We find that the producer–consumer relationship dominates shared memory accesses. Despite this access pattern and the fine–grained data accesses, we find that data placement and false sharing are not an issue even at large processor counts. This is due to the small computation to communication ratio. Future extensions that are now computationally possible to these models may result in much higher memory requirements. However, we do not expect that this would affect the fundamental memory access patterns. The final version of our code performs and scales very well up to 64 processors.

In summary, in this work we have substantially improved the execution time of the sequential model implementation and we have provided an efficient and scalable parallel implementation for realistic financial economics problems. Our work makes feasible the computation of complex, real–life models with multiple states and multiple control variables that have not been analyzed by financial economists to date due to their computational requirements. Thus, parallelism can indeed provide one solution to computationally intensive problems that arise in a wide array of economic models, and future work should concentrate on using parallel architectures in computing solutions to more realistic models in economics and finance.

9. Acknowledgments

We would like to thank Shezad K. Okhai and Rajit Jhaver for helping with implementing intermediate versions of the code, William G. Wichser and Courtney Gibson for their help with runs on the Origin2000 and Christina Christara, Kostas Plataniotis, and Stavros A. Zenios for useful comments on aspects of this work. The authors thankfully acknowledge the support of Natural Sciences and Engineering Research Council of Canada, Nortel Institute of Technology, Communications and Information Technology Ontario, and University of Cyprus and European Union through the HERMES Center of Excellence in Computational Finance and Economics at the University of Cyprus.

References

[1] G. A. Abandah and E. S. Davidson. A comparative study of cache-coherent nonuniform memory access systems. In *Proceedings of the 12th Ann. Int'l Symp. on High Performance Computing Systems and Applications*, May 1998.

[2] H. M. Amman, D. A. Kendrick, and J. Rust. *Handbook of Computational Economics*, volume 1. Elsevier, The Netherlands, 1996.

[3] I. Aptech Systems. Gauss. http://www.aptech.com.

[4] N. Barberis. Investing for the long run when returns are predictable. In *Journal of Finance*, volume 55:1, 2000.

[5] M. Brennan, E. Schwartz, and R. Lagnado. Strategic asset allocation. In *Journal of Economic Dynamics and Control*, volume 21, pages 1377–1403, 1997.

[6] C. D. Carroll. Buffer stock saving and the life cycle / permanent income hypothesis. In *Quarterly Journal of Economics*, volume CXII:1, pages 3–55, 1997.

[7] J. Cocco, F. Gomes, and P. Maenhout. Portfolio choice over the life cycle. In *Working Paper*, 1999.

[8] A. Deaton. Saving and liquidity constraints. In *Econometrica*, volume 59:5, pages 1221–48, 1991.

[9] M. Haliassos and A. Michaelides. Computation and calibration of household portfolio models. In *Household Portfolios. Editors: L. Guiso, M. Haliassos, and T. Japelli*. MIT Press, 2001.

[10] J. Heaton and D. Lucas. Portfolio choice in the presence of background risk. In *The Economic Journal*, volume 110, pages 1–26, 2000.

[11] G. Hubbard, J. Skinner, and S. Zeldes. The importance of precautionary motives for explaining individual and aggregate saving. In *The Carnegie Rochester Conference Series on Public Policy. Editors: A. Meltzer and C. I. Plosser*, volume XL, Amsterdam, North Holland, 1994.

[12] D. Jiang and J. P. Singh. Does application performance scale on cache-coherent multiprocessors: A snapshot. In *Proceedings of the 26th International Symposium on Computer Architecture (ISCA)*, May 1999.

[13] K. Judd. *Numerical Methods in Economics*. MIT Press, Cambridge, MA, 1998.

[14] J. P. Laudon and D. Lenoski. The SGI Origin2000: a scalable cc-numa server. In *Proceedings of the 24rd Annual International Symposium on Computer Architecture*, June 1997.

[15] R. Marimon and A. Scott. *Computational Methods for the Study of Dynamic Economies*. Oxford University Press, Oxford, UK, 1998.

[16] J. Mulvey and H. Vladimirou. Stochastic network programming for financial planning problems. In *Management Science*, volume 38, pages 1642–1664, 1992.

[17] T. A. Ngo and L. Snyder. On the influence of programming models on shared memory computer performance. In *Scalable High Performance Computing Conference*, Apr. 1992.

[18] J. Rust. Numerical dynamic programming in economics. In *Handbook of Computational Economics. Editors: H. M. Amman and D. A. Kendrick and J. Rust*, Amsterdam, The Netherlands, 1996. Elsevier.

[19] H. Shan and J. P. Singh. Comparison of message passing, SHMEM and cache-coherent shared address space programming models on the SGI Origin 2000. In *International Conference on Supercomputing*, June 1999.

[20] K. Storesletten, C. Telmer, and A. Yaron. Persistent idiosyncratic shocks and incomplete markets. In *Working Paper*, 2001.

[21] H. J. Wasserman, O. M. Lubeck, Y. Luo, and F. Bassetti. Performance evaluation of the SGI Origin2000: A memory-centric characterization of LANL ASCI applications. In *Supercomputing '97*, Nov 1997.

[22] S. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. Methodological considerations and characterization of the SPLASH-2 parallel application suite. In *Proceedings of the 23rd International Symposium on Computer Architecture (ISCA)*, May 1995.

[23] S. Zenios. High-performance computing in finance: The last ten years and the next. In *Parallel Computing*, volume 25, pages 2149–2175, 1999.