Comput Econ (2015) 45:261–284
DOI 10.1007/s10614-014-9419-x

Solving Dynamic Programming Problems on a Computational Grid

Yongyang Cai · Kenneth L. Judd · Greg Thain · Stephen J. Wright

Accepted: 20 January 2014 / Published online: 2 February 2014

Abstract We implement a dynamic programming algorithm on a computational grid consisting of loosely coupled processors, possibly including clusters and individual workstations. The grid changes dynamically during the computation, as processors enter and leave the pool of workstations. The algorithm is implemented using the Master–Worker library running on the HTCondor grid computing platform, which can be deployed on many networks. We implement value function iteration for large dynamic programming problems of two kinds: optimal growth problems and dynamic portfolio problems. We present examples that solve in hours on HTCondor but would take weeks if executed on a single workstation. The cost of using HTCondor is small because it uses CPU resources that otherwise would be idle. The use of HTCondor can increase a researcher's computational productivity by at least two orders of magnitude.

Keywords Numerical dynamic programming · Parallel computing · Grid computing · Value function iteration · Dynamic portfolio optimization · Multi-country optimal growth

Y. Cai (✉)
Hoover Institution, Stanford University, Stanford, CA, USA
e-mail: yycai@stanford.edu

Y. Cai
Becker Friedman Institute, University of Chicago, Chicago, IL, USA

K. L. Judd
Hoover Institution, Stanford University & NBER, Stanford, CA, USA
e-mail: kennethjudd@mac.com

G. Thain · S. J. Wright
Computer Science Department, University of Wisconsin, Madison, WI 53706, USA
e-mail: gthain@cs.wisc.edu

S. J. Wright
e-mail: swright@cs.wisc.edu

JEL Classification C61 · C63 · G11

1 Introduction: Motivation and Model

Many dynamic optimization problems in economics can be expressed as dynamic programming problems, but solving them numerically would require weeks or months of CPU time on personal computers. This constraint has limited the range of applications of dynamic programming in economics. Fortunately, many dynamic programming problems can be broken down into a large number of smaller problems that can be solved simultaneously. This structure allows one to use the massively parallel architectures that have been developed in the recent past. When applicable, parallelization can drastically reduce the "wall clock time", the time a user waits for the results. For example, if you have access to one hundred CPUs and can keep each one working on computations necessary for a solution, then the wall clock time can be reduced from 100 h to about 1 h.[1] Massive parallelism was initially used in supercomputers. Unfortunately, supercomputers are not used extensively in economics due to difficulties in access and the technical demands on users. HTCondor is a form of massive parallelism that avoids the access problems of supercomputers. This paper shows how economists can use HTCondor to solve large dynamic programming problems with their existing software, with only a few basic system commands, and at little cost in terms of money and bureaucratic processes.

Dynamic programming (DP) is an essential tool in solving problems of dynamic and stochastic controls in economic analysis, and often has a structure that can exploit parallelism. For example, in value function iteration, the period t iteration first solves the period t Bellman optimization problem at many distinct states, where each of these optimization problems use the period $t + 1$ value function. The results are then used to construct an approximation of the period t value function. If the computational burden of the approximation step is small relative to the collection of state-specific and independent Bellman optimization problems, then there is substantial potential for parallelization to reduce the wall clock time of DP computations.

In theory, this is obvious. However, achieving significant gains from parallelism requires the creation of sophisticated software to manage communications among the CPUs being used. Increasing the number of CPUs raises the potential gain from parallelism but increases the amount of communication that needs to be managed by the network software. Furthermore, it is difficult for economists to obtain time on supercomputers.[2] These barriers have made it impractical for the average economist to use massively parallel architectures.

Fortunately for economists, the economic principle of specialization has been at work and computer scientists have developed tools that make it easy for economists

[1] Parallelization does not reduce total CPU time. In fact, parallelization will create overhead costs that do not arise with serial computation. The benefit of parallelization is that the user will receive results sooner. The focus in parallel computing is to economize on human time and other resources, not on total CPU time.

[2] After this work was completed, the NSF launched the XSEDE project (https://www.xsede.org/), which makes it far easier to access high power computing resources. Some of the machines available through XSEDE use HTCondor, making the software we describe in this paper useful in XSEDE projects.

to harness the potential power of parallel computing. The first key concept is "high throughput computing", HTC, where software organizes a cluster so that when a computer is not being used for another purpose, it will be given tasks that are part of someone's parallel algorithm. In this paper, we use HTCondor.[3] The second key tool presented in this paper is Master–Worker (MW), a user-friendly parallelization tool deployed on HTCondor. We use these tools to show that dynamic programming problems can fully utilize the potential value of parallelism on existing networks of computers that currently exist in department and college networks. HTCondor acts as a management tool for identifying, allocating and managing available resources to solve large distributed computations. For example, if a workstation on a network is currently unused, HTCondor will detect that fact, and send it a task. HTCondor will continue to use that workstation until a higher-priority user (such as a student sitting at the keyboard) appears, at which time HTCondor ends its use of the workstation. This is called "cycle scavenging" and allows a system to take advantage of otherwise idle CPU time. In this paper, we show that HTCondor plus MW makes it possible to exploit massive parallelism to solve dynamic programming problems.

This paper is constructed as follows. Section 2 discusses the potential use of parallelism for dynamic programming. Section 3 gives an introduction of HTCondor-MW system. Section 4 describes numerical algorithms for solving DP problems. Section 5 introduces two types of parallel DP algorithms in the HTCondor-MW system. Sections 6 and 7, respectively, give computational results of the parallel DP algorithms in the HTCondor-MW system for solving multidimensional optimal growth problems and dynamic portfolio optimization problems. Section 8 concludes.

2 Dynamic Programming and Parallelism

Some economists have expressed substantial skepticism about the ability to solve interesting multidimensional dynamic programming problems, arguing that the curse of dimensionality creates unavoidable limits on what can be solved. Rust (1997) and Rust et al. (2002) prove that the curse of dimensionality is a problem for large classes of dynamic programming problems. However, a closer examination of these analyses shows that this pessimism is exaggerated. The curse of dimensionality is based on the worst-case analysis, saying nothing about the average cost of using an algorithm to solve a problem. The claims about the curse of dimensionality are made for arbitrary dynamic programming problems, and ignore mathematical properties that are often present in dynamic programming problems solved in economics. The mathematics literature has shown that many multidimensional problems do not display a curse of dimensionality. For example, Griebel and Wozniakowski (2006) show that as long as an unknown function has sufficient smoothness then there is no curse of dimensionality in approximation based on a finite number of samples of its values, nor on computing its integral. Therefore, problems with smooth payoffs, smooth transitions, and smooth value functions can avoid the curse of dimensionality. Many problems in economics can be formulated in ways that satisfy these requirements without losing the essential

[3] The previous name was Condor.

economics being studied. This paper uses the smooth function approximation in the dynamic programming algorithm so that there is no curse of dimensionality in the approximation method.

Even if one formulates a dynamic programming problem that can use effective approximation and quadrature methods to avoid the curse of dimensionality, multidimensional dynamic programming problems are still expensive to solve. Fortunately, the structure of DP problems allows one to use massive parallelism on many networks, clusters, and supercomputers. Many modern computer systems now offer researchers parallel computing tools. If parallelization can be used, it is the natural way to make otherwise intractable problems tractable. This paper shows that dynamic programming problems do have a structure that facilitates the use of parallelization on modern parallel systems, and in a flexible manner that allows for efficient use of even heterogeneous computing environments with high latency.

Parallel computing methods have been developed for some dynamic programming problems. This paper focuses on the synergies between efficient approximation methods and parallelism in high-latency systems not examined before.

Pflug and Swietanowski (2000) and Zenios (1999) use parallel stochastic programming methods to solve asset management problems. Stochastic programming methods have difficulty solving problems beyond ten periods (and often just assume a few periods), whereas the value function approach can handle arbitrary time horizons due to its use of function approximation methods.

Many have used discretization methods to solve DP problems. Chung et al. (1992) use a parallel dynamic programming algorithm based on finite difference methods to solve partial differential equations that arise in continuous-time stochastic dynamic programming. However, finite difference methods suffer from a curse of dimensionality problem as the dimension increases. Also, finite difference methods are suitable only for low latency data parallel structures. Abdelkhalek et al. (2001) uses a 64-processor low latency system to solve life-cycle problems, but discretizes the state space. Approximating continuous states with discrete-state problems will, like finite-difference methods, be limited by a curse of dimensionality.

Coleman (1992) approximates the consumption function in a growth problem, but uses local interpolation methods which are difficult to efficiently extend to higher dimensions. With local interpolation, the Euler equations are not smooth, which precludes the use of Newton's method and other fast solvers.

Recent innovations in graphical processing units (GPU) hardware have opened up the possibility of applying GPUs to economics problems. For example, Aldrich et al. (2011) give numerical examples using value function iteration with a discretization method to solve a basic RBC model with two state variables and one control variable. Also Morozov and Mathur (2012) give three numerical examples of using GPUs to solve dynamic programming problems with three state variables. GPUs have substantial potential but are also low latency systems and communicate only with one CPU. Future work will exploit the synergies between parallelism on GPUs at the processor level with parallelism across CPUs in both high- and low-latency systems.

In this paper we show that our method can solve high-dimensional dynamic programming problems rapidly with high parallel efficiency across many comput-

ers/workstations, using fast Newton-type optimizers and efficient multi-dimensional approximation methods.

3 HTCondor: A Grid Platform

The idea to connect multiple computers to create a cluster for parallel computing is not new. For example, Creel (2005) uses distributed parallel computing on available computers to solve some econometric problems in parallel. However, as Creel notes

> While it is well-known that such speedups in computations are possible, and while the use of parallel computation within economics in general and econometrics in particular has a long history, it is also clear that only a small part of the computational work done in economics makes use of parallel computing. Two factors may explain this. Parallel programming has traditionally had a steep learning curve, at least compared to that of the popular high-level matrix programming languages. In the best of cases, programmers had to learn FORTRAN or C, and then how to use libraries of functions for parallelization. Next, parallel computing has traditionally been done on expensive mainframe computers that require generous budgets for their purchase, and skilled support personnel. More recently, clusters of commodity or workstation-class computers have become widely used. This solution is considerably less expensive in terms of hardware costs, but a dedicated cluster of computers for use by multiple users requires enough time and effort to construct and maintain that is out of the reach of most research groups that do not have a good budget for support personnel.

Many of these difficulties are solved by the HTCondor system. HTCondor is a high-throughput computing (HTC), open-source software framework for distributed parallelization of computationally intensive tasks on a cluster of computers. The HTCondor software is freely available to all; see http://research.cs.wisc.edu/htcondor/index.html for details. HTCondor can either be set up in advance by the cluster administrator, or, if you have another cluster system, it can be setup by an unprivileged user, by running the HTCondor processes as jobs in the underlying batch system. For those who do not want to set up HTCondor, XSEDE makes it possible for any researcher to access HTCondor systems around the US (see https://www.xsede.org/high-throughput-computing).

HTCondor MW systems can utilize code written in many languages. The interface between master and workers uses a C++ class, but the workers can use executables generated from other languages such as FORTRAN, MATLAB, R, etc. This flexibility provides a user with the ability to use that software that is best for his problem. For example, our dynamic programming algorithms generate vast numbers of smooth constrained optimization problems which we solve with NPSOL (Gill et al. 1994), an optimization package written in FORTRAN to solve those kinds of problems. In many econometric problems, people would like to call some R or MATLAB subroutines.

Supercomputing, often called high performance computing (HPC), uses MPI (message passing interface) for communication among processes and/or OpenMP for shared memory multiprocessing programming. The advantage of supercomputers is the specialized communication hardware that allows for rapid communication among

processors. However, on a supercomputer, a user is assigned a fixed number of processors. Algorithms that need different numbers of processors at different stages cannot be implemented efficiently on HPC architectures. Due to the necessity of having a block of processors, users must reserve time, and the lag time between requesting time and getting access increases with the number of desired processors and requested time. Moreover, economists face substantial bureaucratic hurdles in getting access to supercomputer time because the people who control supercomputers impose requirements that are met by few economists.

There are many developments in software and hardware that have created tools for parallelism.[4] HTCondor is a very flexible set of software tools, is open source, and can use existing state-of-the-art solvers and software without changes in the code. For these reasons, this paper focuses on implementing parallel dynamic programming methods in clusters with high latency.

The HTCondor team at the University of Wisconsin-Madison has developed several "flavors" of HTCondor, each fine-tuned for some specific type of parallel computing. Creel and Goffe (2008) describe a variety of approaches to parallel computing, including HTCondor. They expand on several of these points, and compare HTCondor to some alternatives. In this paper we use the HTCondor Master–Worker (MW) system for parallel algorithms to solve DP problems. The HTCondor MW system consists of two entities: a master process and a cluster of worker processes. The master process decomposes a problem into small tasks and puts those tasks in a queue. Each worker process first examines the queue, takes the "top" problem off the queue and solves it. The worker then sends the results to the master, examines the queue of unfinished tasks, and repeats this process until the queue is empty. The workers' execution is a simple cycle: take a task off master's queue, do the task, and then send the results to the master. While the workers are solving the tasks, the master collects the results and puts new tasks on the queue. This is a file-based, remote I/O scheme that serves as the message-passing mechanism between the master and the workers. See Thain et al. (2005) for more detailed discussion.

The MW paradigm helps the user circumvent the parallel computing challenges, such as load balancing, termination detection, and distribution of information across compute nodes. Moreover, computation in the MW paradigm is fault-tolerant: if a worker cannot complete a task, due to machine failure or interruption by another user, the master can detect this and put that task back on the queue for another worker to execute. Sometimes a task is very time-consuming, it will be worthy to let the another worker to continue the terminated job instead of restarting it. This can be supported by a checkpoint mechanism in HTCondor on a number of Unix platforms[5]: the state of a program will be snapshotted periodically so that the terminated job can be resumed later from the most recent snapshot when the scheduler allocates it a new machine.

[4] The ideas presented in this paper can be implemented using other tools. For example, Matlab has a Parallel Computing Toolbox. While this permits some parallel computing, its value is limited because most machines using this toolbox have a small number of cores. We will provide examples where we use up to 200 CPUs.

[5] Since checkpointing services have not been supported on Windows platforms, it will usually have better performance during the night than during the day.

The user can request any number of workers, independent of the number of tasks. HTCondor can make use of a heterogeneous collection of computers, where the fast computers will solve more tasks but slower computers can still contribute. You can also run a program using MPI on top of HTCondor, but we prefer not to, as MPI programs are less reliable in grid computing, because they require all nodes to be running at the same time—if one node crashes, the whole computation stops. With HTCondor and the MW paradigm, if one worker crashes, the whole computation can continue. This is very important if you want to scale to large computations.

HTCondor is a valuable alternative to HPC. In contrast to HPC, HTC is a paradigm with much greater flexibility and lower cost. The marginal cost of CPU time used in HTCondor is nearly zero (other than marginal electricity use) because HTCondor is using a computational resource that otherwise would go unused. HTCondor manages the number of processors being used in response to processor availability and the needs of the computational procedure. If HTCondor sees that a computation needs hundreds of processors, it will give the computation what it needs if the resources are available, but if it later sees that a computation needs only a dozen processors, it can free up unused processors and allocate them to other computations. HTC is opportunistic, utilizing any resource that becomes available and not forcing the user to make reservations. The disadvantage of HTC is that interprocessor communication will generally be slower. While this does limit the amount of parallelization that can be exploited, HTC environments can still efficiently use hundreds of processors for many problems, e.g., dynamic programming problems in this paper. For high-dimensional dynamic programming problems, each parallelized task can be computationally expensive and time-consuming on the order of minutes or more, so the network latency should not matter much.

There are other forms of parallel computing, such as grid computing which spreads work across computers connected only by the internet. These "clouds" are other examples of high latency parallel computing and could also implement the methods described below. We focus on HTCondor because it is a well-developed, user-friendly and free tool for high-latency parallelism.

4 Dynamic Programming

In economics and finance, we often encounter a finite horizon optimal decision-making problem that can be expressed in the following general model:

$$V_0(x_0, \theta_0) = \max_{a_t \in \mathcal{D}(x_t, \theta_t, t)} \mathbb{E}\left\{ \sum_{t=0}^{T-1} \beta^t u_t(x_t, a_t) + \beta^T V_T(x_T, \theta_T) \right\},$$

where x_t is a continuous state process with an initial state x_0, θ_t is a discrete state process with an initial state θ_0, and a_t is an action variable (x_t, θ_t and a_t can be vectors), $u_t(x, a)$ is a utility function at time $t < T$ and $V_T(x, \theta)$ is a given terminal value function, β is the discount factor ($0 < \beta \leq 1$), $\mathcal{D}(x_t, \theta_t, t)$ is a feasible set of a_t, and $\mathbb{E}\{\cdot\}$ is the expectation operator.

The DP model for the finite horizon problems is the basic Bellman equation,

$$V_t(x,\theta) = \max_{a \in D(x,\theta,t)} u_t(x,a) + \beta \mathbb{E}\{V_{t+1}(x^+,\theta^+)\},$$

for $t = 0, 1, \ldots, T-1$, where (x^+, θ^+) is the next-stage state conditional on the current-stage state (x, θ) and action a, and $V_t(x,\theta)$ is called the value function at stage t while the terminal value function $V_T(x,\theta)$ is given.

4.1 Numerical DP Algorithms

In DP problems, if state variables and control variables are continuous, then value functions must be approximated in some computationally tractable manner. It is common to approximate value functions with a finitely parameterized collection of functions; that is, $V(x,\theta) \approx \hat{V}(x,\theta;\mathbf{b})$, where $\mathbf{b}$ is a vector of parameters. The functional form $\hat{V}$ may be a linear combination of polynomials, or it may represent a rational function or neural network representation, or it may be some other parameterization specially designed for the problem. After the functional form is fixed, we focus on finding the vector of parameters, $\mathbf{b}$, such that $\hat{V}(x,\theta;\mathbf{b})$ approximately satisfies the Bellman equation (Bellman 1957). Algorithm 1 is the parametric DP method with value function iteration for finite horizon problems with both multidimensional continuous and discrete states. More detailed discussion of numerical DP can be found in Cai (2010), Judd (1998), Cai and Judd (2010), and Rust (2008). In the algorithm, n is the dimension for the continuous states x, and d is the dimension for discrete states $\theta \in \Theta = \{\theta^j : 1 \le j \le D\} \subset \mathbb{R}^d$, where D is the number of different discrete state vectors. The transition probabilities from θ^j to $\theta^{j'}$ for $1 \le j, j' \le D$ are given.

4.2 Approximation

An approximation scheme has two ingredients: basis functions and approximation nodes. Approximation nodes can be chosen as uniformly spaced nodes, Chebyshev nodes, or some other specified nodes. From the viewpoint of basis functions, approximation methods can be classified as either spectral methods or finite element methods. A spectral method uses globally nonzero basis functions $\phi_j(x)$ such that $\hat{V}(x;\mathbf{b}) = \sum_{j=0}^{m} b_j \phi_j(x)$. Examples of spectral methods include ordinary polynomial approximation, ordinary Chebyshev polynomial approximation, shape-preserving Chebyshev polynomial approximation (Cai and Judd 2013), and Chebyshev–Hermite approximation (Cai and Judd 2012b). In contrast, a finite element method uses local basis functions $\phi_j(x)$ that are nonzero over sub-domains of the approximation domain. Examples of finite element methods include piecewise linear interpolation, shape-preserving rational function spline interpolation (Cai and Judd 2012a), cubic splines, and B-splines.

4.2.1 Chebyshev Polynomial Approximation

Chebyshev polynomials on $[-1, 1]$ are defined as $\mathcal{T}_j(x) = \cos(j \cos^{-1}(x))$, while general Chebyshev polynomials on $[x_{\min}, x_{\max}]$ are defined as $\mathcal{T}_j((2x - x_{\min} -$

Algorithm 1 Parametric dynamic programming with value function iteration for problems with multidimensional continuous and discrete states

Initialization. Given a finite set of $\theta \in \Theta = \{\theta^j : 1 \le j \le D\} \subset \mathbb{R}^d$ and the probability transition matrix $P = \left(p_{j,j'}\right)_{D\times D}$ where $p_{j,j'}$ is the transition probability from $\theta^j \in \Theta$ to $\theta^{j'} \in \Theta$ for $1 \le j, j' \le D$. Choose a functional form for $\hat{V}(x, \theta; \mathbf{b})$ for all $\theta \in \Theta$, and choose the approximation grid, $\mathbb{X}_t = \{x_t^i : 1 \le i \le N_t\} \subset \mathbb{R}^n$. Let $\hat{V}(x, \theta; \mathbf{b}^T) = V_T(x, \theta)$. Then for $t = T-1, T-2, \ldots, 0$, iterate through steps 1 and 2.

Step 1. Maximization step. Compute

$$v_{i,j} = \max_{a \in \mathcal{D}(x^i, \theta^j, t)} u_t(x^i, \theta^j, a) + \beta \mathbb{E}\{\hat{V}(x^+, \theta^+; \mathbf{b}^{t+1})\}, \tag{1}$$

for each $x^i \in \mathbb{X}_t$ and $\theta^j \in \Theta$, $1 \le i \le N_t$, $1 \le j \le D$, where the next-stage discrete state θ^+ is random with probability mass function $\Pr(\theta^+ = \theta^{j'} \mid \theta^j) = p_{j,j'}$ for each $\theta^{j'} \in \Theta$, and x^+ is the next-stage state transition from x^i and may be also random.

Step 2. Fitting step. Using an appropriate approximation method, for each $1 \le j \le D$, compute $\mathbf{b}_j^t$, such that $\hat{V}(x, \theta^j; \mathbf{b}_j^t)$ approximates $\{(x^i, v_{i,j}) : 1 \le i \le N_t\}$ data, i.e., $v_{i,j} \approx \hat{V}(x^i, \theta^j; \mathbf{b}_j^t)$ for all $x^i \in \mathbb{X}_t$. Let $\mathbf{b}^t = \left\{\mathbf{b}_j^t : 1 \le j \le D\right\}$.

$x_{\max})/(x_{\max} - x_{\min}))$ for $j = 0, 1, 2, \ldots$ These polynomials are orthogonal under the weighted inner product: $\langle f, g\rangle = \int_{x_{\min}}^{x_{\max}} f(x)g(x)w(x)dx$ with the weighting function $w(x) = \left(1 - ((2x - x_{\min} - x_{\max})/(x_{\max} - x_{\min}))^2\right)^{-1/2}$. A degree m Chebyshev polynomial approximation for $V(x)$ on $[x_{\min}, x_{\max}]$ is

$$\hat{V}(x; \mathbf{b}) = \sum_{j=0}^{m} b_j \mathcal{T}_j\left(\frac{2x - x_{\min} - x_{\max}}{x_{\max} - x_{\min}}\right), \tag{2}$$

where $\mathbf{b} = \{b_j\}$ are the Chebyshev coefficients.

If we choose the Chebyshev nodes on $[x_{\min}, x_{\max}]$: $x^i = (z_i + 1)(x_{\max} - x_{\min})/2 + x_{\min}$ with $z_i = -\cos\left((2i-1)\pi/(2m')\right)$ for $i = 1, \ldots, m'$ with $m' > m$, and Lagrange data $\{(x^i, v_i) : i = 1, \ldots, m'\}$ are given (where $v_i = V(x^i)$), then the coefficients b_j in (2) can be easily computed by the Chebyshev regression algorithm (see Judd 1998).

4.2.2 Multidimensional Complete Chebyshev Approximation

In an n-dimensional approximation problem, let the domain of the value function be

$$\left\{x = (x_1, \ldots, x_n) : x_j^{\min} \le x_j \le x_j^{\max},\ j = 1, \ldots, n\right\},$$

for some real numbers $x_j^{\min}$ and $x_j^{\max}$ with $x_j^{\max} > x_j^{\min}$ for $j = 1, \ldots, n$. Let $x^{\min} = (x_1^{\min}, \ldots, x_n^{\min})$ and $x^{\max} = (x_1^{\max}, \ldots, x_n^{\max})$. Then we denote $[x^{\min}, x^{\max}]$ as the

domain. Let $\alpha = (\alpha_1, \ldots, \alpha_n)$ be a vector of nonnegative integers. Let $\mathcal{T}_\alpha(z)$ denote the product $\mathcal{T}_{\alpha_1}(z_1) \cdots \mathcal{T}_{\alpha_n}(z_n)$ for $z = (z_1, \ldots, z_n) \in [-1, 1]^n$. Let

$$Z(x) = \left(\frac{2x_1 - x_1^{\min} - x_1^{\max}}{x_1^{\max} - x_1^{\min}}, \ldots, \frac{2x_n - x_n^{\min} - x_n^{\max}}{x_n^{\max} - x_n^{\min}} \right)$$

for any $x = (x_1, \ldots, x_n) \in [x^{\min}, x^{\max}]$.

Using these notations, the degree-m complete Chebyshev approximation for $V(x)$ is

$$\hat{V}_m(x; \mathbf{b}) = \sum_{0 \le |\alpha| \le m} b_\alpha \mathcal{T}_\alpha \left(Z(x) \right), \tag{3}$$

where $|\alpha| = \sum_{j=1}^n \alpha_j$ for the nonnegative integer vector $\alpha = (\alpha_1, \ldots, \alpha_n)$. So the number of terms with $0 \le |\alpha| \le m$ is $\binom{m+n}{n}$ for the degree-m complete Chebyshev approximation in $\mathbb{R}^n$.

4.3 Numerical Integration

In the objective function of the Bellman equation, we often need to compute the conditional expectation of $V(x^+)$. When the random variable is continuous, we can use numerical integration to compute the expectation. Gaussian quadrature rules are often applied in computing the integration.

4.3.1 Gauss–Hermite Quadrature

In the expectation operator of the objective function of the Bellman equation, if the random variable has a normal distribution, then it will be good to apply the Gauss–Hermite quadrature formula to compute the numerical integration. That is, if we want to compute $\mathbb{E}\{f(Y)\}$ where Y has a distribution $\mathcal{N}(\mu, \sigma^2)$, then

$$\begin{aligned}
\mathbb{E}\{f(Y)\} &= (2\pi\sigma^2)^{-1/2} \int_{-\infty}^{\infty} f(y) e^{-(y-\mu)^2/(2\sigma^2)} dy \\
&= (2\pi\sigma^2)^{-1/2} \int_{-\infty}^{\infty} f(\sqrt{2}\,\sigma\, x + \mu) e^{-x^2} \sqrt{2}\sigma\, dx \\
&\doteq \pi^{-\frac{1}{2}} \sum_{i=1}^{m} \omega_i f(\sqrt{2}\sigma x_i + \mu),
\end{aligned}$$

where ω_i and x_i are the Gauss–Hermite quadrature with m weights and nodes over $(-\infty, \infty)$. See Cai (2010), Judd (1998), Stroud and Secrest (1966) for more details.

If Y is log normal, i.e., $\log(Y)$ has a distribution $\mathcal{N}(\mu, \sigma^2)$, then we can assume that $Y = e^X$ where $X \sim \mathcal{N}(\mu, \sigma^2)$, thus

$$\mathbb{E}\{f(Y)\} = \mathbb{E}\{f(e^X)\} \doteq \pi^{-\frac{1}{2}} \sum_{i=1}^{m} \omega_i f\left(e^{\sqrt{2}\sigma x_i + \mu}\right).$$

4.3.2 Multidimensional Integration

If we want to compute a multidimensional integration, we could apply the product rule. For example, suppose that we want to compute $\mathbb{E}\{f(X)\}$, where X is a random vector with multivariate normal distribution $\mathcal{N}(\mu, \Sigma)$ over $\mathbb{R}^n$, where μ is the mean column vector and Σ is the covariance matrix, then we could do the Cholesky factorization first, i.e., find a lower triangular matrix L such that $\Sigma = LL^\top$. This is feasible as Σ must be a positive semi-definite matrix from the covariance property. Thus,

$$\begin{aligned}
\mathbb{E}\{f(X)\} &= \left((2\pi)^n \det(\Sigma)\right)^{-1/2} \int_{\mathbb{R}^n} f(y) e^{-(y-\mu)^\top \Sigma^{-1}(y-\mu)/2} dy \\
&= \left((2\pi)^n \det(L)^2\right)^{-1/2} \int_{\mathbb{R}^n} f\left(\sqrt{2}Lx + \mu\right) e^{-x^\top x} 2^{n/2} \det(L) dx \\
&\doteq \pi^{-\frac{n}{2}} \sum_{i_1=1}^{m} \cdots \sum_{i_n=1}^{m} \omega_{i_1} \cdots \omega_{i_d} f\Bigg(\sqrt{2} l_{1,1} x_{i_1} + \mu_1, \\
&\qquad \sqrt{2}(l_{2,1} x_{i_1} + l_{2,2} x_{i_2}) + \mu_2, \cdots, \sqrt{2}\left(\sum_{j=1}^{n} l_{n,j} x_{i_j}\right) + \mu_n\Bigg), \qquad (4)
\end{aligned}$$

where ω_i and x_i are weights and nodes over $(-\infty, \infty)$, $l_{i,j}$ is the (i, j)-element of L, and $\det(\cdot)$ means the matrix determinant operator.

5 Parallel Dynamic Programming

The numerical DP algorithms can be applied easily in the HTCondor MW system for DP problems with multidimensional continuous and discrete states. To solve these problems, numerical DP algorithms with value function iteration have the maximization step that is mostly time-consuming in numerical DP. Equation (1) in Algorithm 1 computes $v_{i,j}$ for each approximation point x^i in the finite set $\mathbb{X}_t \subset \mathbb{R}^n$ and each discrete state vector $\theta^j \in \Theta$, where N_t is the number of points of $\mathbb{X}_t$ and D is the number of points of Θ, so there are $N_t \times D$ small-size maximization problems. In high-dimensional problems, $N_t \times D$ will be large, then it will take a huge amount of time to do the DP maximization step. However, these $N_t \times D$ small-size maximization problems can be naturally parallelized, in which one or several maximization problem(s) could be treated as one task. Since these maximization problems are independent, both serial and parallel dynamic programming algorithms have these computation costs and

will give the same solution. All the parallelization scalability comes through parallelism across discrete state space points and/or approximation nodes of continuous state variables.

5.1 Type-I Parallelization

When D is large, we could separate the $N_t \times D$ maximization problems into D tasks, where each task corresponds to a discrete state vector θ^j and all continuous state nodes in $\mathbb{X}_t$. Algorithm 2 is the architecture for the master processor, and Algorithm 3 is the corresponding architecture for the workers.

Algorithm 2 Type-I parallel dynamic programming with value function iteration for the master

Initialization. Given a finite set of $\theta \in \Theta = \{\theta^j : 1 \le j \le D\} \subset \mathbb{R}^d$. Set $\mathbf{b}^T$ as the parameters of the terminal value function. For $t = T-1, T-2, \ldots, 0$, iterate through steps 1 and 2.

Step 1. Separate the maximization step into D tasks, one task per $\theta \in \Theta$. Each task contains parameters $\mathbf{b}^{t+1}$, stage number t and the corresponding task identity for some θ^j. Then send these tasks to the workers.

Step 2. [Step. 2] Wait until all tasks are done by the workers. Then collect parameters $\mathbf{b}_j^t$ from the workers, for all $1 \le j \le D$, and let $\mathbf{b}^t = \left\{\mathbf{b}_j^t : 1 \le j \le D\right\}$.

Algorithm 3 Type-I parallel dynamic programming with value function iteration for the workers

Initialization. Given a finite set of $\theta \in \Theta = \{\theta^j : 1 \le j \le D\} \subset \mathbb{R}^d$ and the probability transition matrix $P = \left(p_{j,j'}\right)_{D\times D}$ where $p_{j,j'}$ is the transition probability from $\theta^j \in \Theta$ to $\theta^{j'} \in \Theta$ for $1 \le j, j' \le D$. Choose a functional form for $\hat{V}(x, \theta; \mathbf{b})$ for all $\theta \in \Theta$.

Step 1. Get parameters $\mathbf{b}^{t+1}$, stage number t and the corresponding task identity for one $\theta^j \in \Theta$ from the master, and then choose the approximation grid, $\mathbb{X}_t = \{x_t^i : 1 \le i \le N_t\} \subset \mathbb{R}^n$.

Step 2. For this given θ^j, compute

$$v_{i,j} = \max_{a \in \mathcal{D}(x^i, \theta^j, t)} u(x^i, \theta^j, a) + \beta \mathbb{E}\{\hat{V}(x^+, \theta^+; \mathbf{b}^{t+1})\},$$

for each $x^i \in \mathbb{X}_t, 1 \le i \le N_t$, where the next-stage discrete state $\theta^+ \in \Theta$ is random with probability mass function $\mathbb{P}(\theta^+ = \theta^{j'} \mid \theta^j) = p_{j,j'}$ for each $\theta_{j'} \in \Theta$, and x^+ is the next-stage state transition from x^i and may be also random.

Step 3. Using an appropriate approximation method, compute $\mathbf{b}_j^t$ such that $\hat{V}(x, \theta^j; \mathbf{b}_j^t)$ approximates $\{(x^i, v_{i,j}) : 1 \le i \le N_t\}$, i.e., $v_{i,j} \approx \hat{V}(x^i, \theta^j; \mathbf{b}_j^t)$ for all $x^i \in \mathbb{X}_t$.

Step 4. Send $\mathbf{b}_j^t$ and the corresponding task identity for θ^j to the master.

Algorithm 2 describes the master's function. Suppose that the value function for time $t+1$ is known, and the master wants to solve for the value function at period t. For each point $\theta \in \Theta$, the master gathers all the Bellman optimization problems

associated with that θ, together with the solution for the next period's value function, and sends that package of problems to a worker processor. It does this until all workers are working on some such package. When the master receives the solutions from a worker, it records those results and sends that worker another package of problems not yet solved. This continues until all θ specific packages have been solved, at which point the master repeats this for period $t-1$.

Algorithm 3 describes the typical worker task. It takes the θ^j package from the master, solves the Bellman optimization problem for each node in $\mathbb{X}_t$, and computes the new value for $\mathbf{b}_j^t$, the coefficients for the time-t value function in the θ^j dimension, and sends the coefficients to the master. In the algorithm, the discrete states are assumed to be exogenously evolving, but this assumption is not required. For example, if the probability transition matrix P is dependent on control variables, a, the workers can still do the same job, while the only difference is that the expectation operator in the objective function uses a controlled probability mass function, $p_{j,j'}(a)$, for the transition probability from θ^j to $\theta^{j'}$.

5.2 Type-II Parallelization

The case where we parallelize only across the discrete dimensions is easy to implement, and is adequate if the number of available workers is small relative to the number of points in Θ. If we have access to more workers, then we will also parallelize across points in $\mathbb{X}_t$. That is, if the number of nodes for continuous states, N_t, is large, or the maximization step for each node is time-consuming, then it will be possible to break the task for one θ^j into subtasks and maintain parallel efficiency. If the fitting method requires all points $\{(x^i, v_{i,j}): 1 \le i \le N_t\}$ to construct the approximation, then each worker cannot do step 3 and 4 along with step 1 and 2 in Algorithm 3, as it has only an incomplete set of approximation nodes x^i for one given θ^j. Therefore, the fitting step is executed by the master. Thus we have Algorithm 4 for the master process and Algorithm 5 for the workers.

Algorithm 4 Type-II parallel dynamic programming with value function iteration for the master

Initialization. Given a finite set of $\theta \in \Theta = \{\theta^j : 1 \le j \le D\} \subset \mathbb{R}^d$. Choose a functional form for $\hat{V}(x, \theta; \mathbf{b})$ for all $\theta \in \Theta$, and choose the approximation grid, $\mathbb{X}_t = \{x_t^i : 1 \le i \le N_t\} \subset \mathbb{R}^n$. Set $\mathbf{b}^T$ as the parameters of the terminal value function. For $t = T-1, T-2, \ldots, 0$, iterate through steps 1 and 2.

Step 1. Separate $\mathbb{X}_t$ into M disjoint subsets with almost equal sizes: $\mathbb{X}_{t,1}, \ldots, \mathbb{X}_{t,M}$, and separate the maximization step into $M \times D$ tasks, one task per $(\mathbb{X}_{t,m}, \theta^j)$ with $\theta^j \in \Theta$, for $m = 1, \ldots, M$ and $j = 1, \ldots, D$. Each task contains the parameters $\mathbf{b}^{t+1}$, the stage number t and the corresponding task identity for $(\mathbb{X}_{t,m}, \theta^j)$. Then send these tasks to the workers.

Step 2. Wait until all tasks are done by the workers. Then collect all $v_{i,j}$ from the workers, for $1 \le i \le N_t$, $1 \le j \le D$.

Step 3. Using an appropriate approximation method, for each $\theta^j \in \Theta$, compute $\mathbf{b}_j^t$ such that $\hat{V}(x, \theta^j; \mathbf{b}_j^t)$ approximates $\{(x^i, v_{i,j}): 1 \le i \le N_t\}$, i.e., $v_{i,j} \approx \hat{V}(x^i, \theta^j; \mathbf{b}_j^t)$ for all $x^i \in \mathbb{X}_t$. Let $\mathbf{b}^t = \left\{\mathbf{b}_j^t : 1 \le j \le D\right\}$.

Algorithm 5 Type-II parallel dynamic programming with value function iteration for the workers

Initialization. Given a finite set of $\theta \in \Theta = \{\theta^j : 1 \le j \le D\} \subset \mathbb{R}^d$ and the probability transition matrix $P = \left(p_{j,j'}\right)_{D\times D}$ where $p_{j,j'}$ is the transition probability from $\theta^j \in \Theta$ to $\theta^{j'} \in \Theta$ for $1 \le j, j' \le D$. Choose the approximation grid, $\mathbb{X}_t = \{x_t^i : 1 \le i \le N_t\} \subset \mathbb{R}^n$, which is the same with the set $\mathbb{X}_t$ in the master.

Step 1. Get the parameters $\mathbf{b}^{t+1}$, stage number t and the corresponding task identity for one $(\mathbb{X}_{t,m}, \theta^j)$ with $\theta^j \in \Theta$ from the master.

Step 2. For this given θ^j, compute

$$v_{i,j} = \max_{a\in\mathcal{D}(x^i,\theta^j,t)} u(x^i, \theta^j, a) + \beta\mathbb{E}\{\hat{V}(x^+, \theta^+; \mathbf{b}^{t+1})\},$$

for all $x^i \in \mathbb{X}_{t,m}$, where the next-stage discrete state $\theta^+ \in \Theta$ is random with probability mass function $\mathbb{P}(\theta^+ = \theta^{j'} \mid \theta^j) = p_{j,j'}$ for each $\theta^{j'} \in \Theta$, and x^+ is the next-stage state transition from x^i and may be also random.

Step 3. Send $v_{i,j}$ for these given $x^i \in \mathbb{X}_{t,m}$ and θ^j, to the master process.

If it is quick to compute $\mathbf{b}_j^t$ in the fitting step (e.g., Chebyshev polynomial approximation using Chebyshev regression algorithm), then we can just let the master do the fitting step like the type-II parallel DP algorithm. However, if the fitting step is time-consuming, then the master could send these fitting jobs for each discrete state θ^j to the workers, and then collect the new approximation parameters.

Our parallel algorithms have used only the most basic techniques for coordinating computation among processors. There are many other places where parallelization might be useful. For example, if the Bellman optimization problem corresponding to a single point (x^i, θ^j) in the state space were itself a large problem, and we had a large number of processors, then it might be useful to use a parallel algorithm to solve each such state-specific problem. There are many possible ways to decompose the big problem into smaller ones and exploit the available processors. We have discussed only the first two layers of parallelization that can be used in dynamic programming. How fine we go depends on the number of processors at our disposal and the communication times across computational units.

5.3 Sparsity

In many cases, the probability transition matrix is sparse and this fact can be exploited to reduce communication cost. For example, suppose that a worker is given the task to compute the value function for θ^j. When it computes the expectation in the objective function of the maximization problems, it only needs access to the value functions for those $\theta^{j'}$ which can be reached from θ^j in one period. That is,

$$\mathbb{E}\{\hat{V}(x^+, \theta^+; \mathbf{b}^{t+1})\} = \sum_{1\le j'\le D,\, p_{j,j'}\neq 0} p_{j,j'}\mathbb{E}\{\hat{V}(x^+, \theta^{j'}; \mathbf{b}_{j'}^{t+1})\}.$$

Therefore, when the master forms the description of a task for a worker, it only needs to include those $\mathbf{b}_{j'}^{t+1}$ with nonzero transition probability $p_{j,j'}$ (instead of the whole set of parameters, $\mathbf{b}^{t+1}$) in the tasks corresponding to θ^j, i.e., $\left\{\mathbf{b}_{j'}^{t+1} : p_{j,j'} > 0,\ 1 \le j' \le D\right\}$ where $p_{j,j'} = \mathbb{P}(\theta^+ = \theta^{j'} \mid \theta^j)$, and then send this subset of $\mathbf{b}^{t+1}$ to the workers in Step 1 of Algorithm 2 or 4. This saves on master-worker communication costs.

6 Application to Stochastic Optimal Growth Models

We consider a multi-dimensional stochastic optimal growth problem. We assume that there are d sectors, and let $k_t = (k_{t,1}, \ldots, k_{t,d})$ denote the capital stocks of these sectors which is a d-dimensional continuous state vector at time t. Let $\theta_t = (\theta_{t,1}, \ldots, \theta_{t,d}) \in \Theta = \{\theta_t^j : 1 \le j \le D\} \subset \mathbb{R}^d$ denote current productivity levels of the sectors which is a d-dimensional discrete state vector at time t, and assume that θ_t follows a Markov process with a stable probability transition matrix, denoted as $\theta_{t+1} = g(\theta_t, \xi_t)$ where ξ_t are i.i.d. disturbances. Let $l_t = (l_{t,1}, \ldots, l_{t,d})$ denote elastic labor supply levels of the sectors which is a d-dimensional continuous control vector variable at time t. Assume that the net production function of sector i at time t is $f(k_{t,i}, l_{t,i}, \theta_{t,i})$, for $i = 1, \ldots, d$. Let $c_t = (c_{t,1}, \ldots, c_{t,d})$ and $I_t = (I_{t,1}, \ldots, I_{t,d})$ denote, respectively, consumption and investment of the sectors at time t. We want to find an optimal consumption and labor supply decisions such that expected total utility over a finite-horizon time is maximized, i.e.,

$$
\begin{aligned}
V_0(k_0, \theta_0) = \max_{k_t, I_t, c_t, l_t} \ & \mathbb{E}\left\{\sum_{t=0}^{T-1} \beta^t u(c_t, l_t) + \beta^T V_T(k_T, \theta_T)\right\}, \\
& \text{s.t. } k_{t+1,j} = (1-\delta)k_{t,j} + I_{t,j} + \epsilon_{t,j}, \quad j = 1, \ldots, d, \\
& \Gamma_{t,j} = \frac{\zeta}{2} k_{t,j} \left(\frac{I_{t,j}}{k_{t,j}} - \delta\right)^2, \quad j = 1, \ldots, d, \\
& \sum_{j=1}^{d} \left(c_{t,j} + I_{t,j} - \delta k_{t,j}\right) = \sum_{j=1}^{d} \left(f(k_{t,j}, l_{t,j}, \theta_{t,j}) - \Gamma_{t,j}\right), \\
& \theta_{t+1} = g(\theta_t, \xi_t),
\end{aligned}
$$

where k_0 and θ_0 are given, δ is the depreciation rate of capital, $\Gamma_{t,j}$ is the investment adjustment cost of sector j, and ζ governs the intensity of the friction, $\epsilon_t = (\epsilon_{t,1}, \ldots, \epsilon_{t,d})$ are serially uncorrelated i.i.d. disturbances with $\mathbb{E}\{\epsilon_{t,i}\} = 0$, and $V_T(k, \theta)$ is a given terminal value function. For this finite-horizon model, Cai and Judd (2012b) solve some of its simplified problem. An infinite-horizon version of this model is introduced in Haan et al. (2011), Juillard and Villemot (2011), and a nonlinear programming method for dynamic programming is introduced in Cai et al. (2013a) to solve the multi-country growth model with infinite horizon.

6.1 Dynamic Programming Model

The DP formulation of the multi-dimensional stochastic optimal growth problem is

$$
\begin{aligned}
V_t(k,\theta) = \max_{c,l,I}\ & u(c,l) + \beta \mathbb{E}\left\{ V_{t+1}(k^+,\theta^+) \mid \theta \right\}, \\
& \text{s.t. } k_j^+ = (1-\delta)k_j + I_j + \epsilon_j, \quad j = 1,\ldots,d, \\
& \Gamma_j = \frac{\zeta}{2} k_j \left(\frac{I_j}{k_j} - \delta \right)^2, \quad j = 1,\ldots,d, \\
& \sum_{j=1}^{d} \left(c_j + I_j - \delta k_j \right) = \sum_{j=1}^{d} \left(f(k_j, l_j, \theta_j) - \Gamma_j \right), \\
& \theta^+ = g(\theta, \xi_t),
\end{aligned}
$$

for $t = 0,\ldots,T-1$, where $k = (k_1,\ldots,k_d)$ is the continuous state vector and $\theta = (\theta_1,\ldots,\theta_d) \in \Theta = \{(\vartheta_{j,1},\ldots,\vartheta_{j,d}) : 1 \le j \le D\}$ is the discrete state vector, $c = (c_1,\ldots,c_d)$, $l = (l_1,\ldots,l_d)$, and $I = (I_1,\ldots,I_d)$ are control variables, $\epsilon = (\epsilon_1,\ldots,\epsilon_d)$ are i.i.d. disturbance with mean 0, and $k^+ = (k_1^+,\ldots,k_d^+)$ and $\theta^+ = \left(\theta_1^+,\ldots,\theta_d^+\right) \in \Theta$ are the next-stage state vectors. Numerically, $V(k,\theta)$ is approximated with given values at finite nodes, so the approximation is only good at a finite range. That is, the state variable must be in a finite range $[\underline{k}, \bar{k}]$, then we should have the restriction $k^+ \in [\underline{k}, \bar{k}]$. Here $\underline{k} = (\underline{k}_1,\ldots,\underline{k}_d)$, $\bar{k} = (\bar{k}_1,\ldots,\bar{k}_d)$, and $k^+ \in [\underline{k}, \bar{k}]$ denotes that $k_i^+ \in [\underline{k}_i, \bar{k}_i]$ for all $1 \le i \le d$. Like most computational methods, our Chebyshev polynomial approach to approximation is defined on a compact domain. However, the solution to the economic problem does not have binding constraints. Therefore, the range is chosen to be wide so that the restriction $k^+ \in [\underline{k}, \bar{k}]$ will not be binding for all $t = 0,\ldots,T-1$.

6.2 Numerical Example

In the following numerical example, we see the application of parallelization of numerical DP algorithms for the DP model of the multi-dimensional stochastic optimal growth problem. We let $T = 3, \beta = 0.8, \delta = 0.025, \zeta = 0.5, [\underline{k}, \bar{k}] = [0.2, 3.0]^d$, $f(k_i, l_i, \theta_i) = \theta_i A k_i^{\psi} l_i^{1-\psi}$ with $\psi = 0.36$ and $A = (1-\beta)/(\psi\beta)$, for $i = 1,\ldots,d$, and

$$
u(c,l) = \sum_{i=1}^{d} \left[\frac{(c_i/A)^{1-\gamma} - 1}{1-\gamma} - (1-\psi) \frac{l_i^{1+\eta} - 1}{1+\eta} \right],
$$

with $\gamma = 2$ and $\eta = 1$.

In this example, we let $d = 4$. So this is a DP example with 4-dimensional continuous states and 4-dimensional discrete states. Here we assume that the possible values of θ_i and θ_i^+ are

$$\vartheta_1 = 0.85, \quad \vartheta_2 = 0.9, \ \vartheta_3 = 0.95, \ \vartheta_4 = 1.0, \ \vartheta_5 = 1.05, \ \vartheta_6 = 1.1, \ \vartheta_7 = 1.15,$$

and the probability transition matrix from θ_i to θ_i^+ is a 7×7 tridiagonal matrix:

$$P = \begin{bmatrix} 0.75 & 0.25 & & & & & \\ 0.25 & 0.50 & 0.25 & & & & \\ & 0.25 & 0.50 & 0.25 & & & \\ & & 0.25 & \ddots & \ddots & & \\ & & & \ddots & 0.50 & 0.25 & \\ & & & & 0.25 & 0.75 & \end{bmatrix},$$

for each $i = 1, \ldots, 4$, and we assume that $\theta_1^+, \ldots, \theta_d^+$ are independent of each other. That is,

$$\Pr[\theta^+ = (\vartheta_{i_1}, \ldots, \vartheta_{i_4}) \mid \theta = (\vartheta_{j_1}, \ldots, \vartheta_{j_4})] = P_{i_1,j_1} P_{i_2,j_2} P_{i_3,j_3} P_{i_4,j_4},$$

where P_{i_α, j_α} is the (i_α, j_α) element of P, for any $i_\alpha, j_\alpha = 1, \ldots, 7, \alpha = 1, \ldots, 4$.

In addition, we assume that $\epsilon_1, \ldots, \epsilon_4$ are i.i.d., and each ϵ_i has 3 discrete values:

$$\delta_1 = -0.01, \quad \delta_2 = 0.0, \ \delta_3 = 0.01,$$

while their probabilities are $q_1 = 0.25$, $q_2 = 0.5$ and $q_3 = 0.25$, respectively. That is,

$$\Pr[\epsilon = (\delta_{n_1}, \ldots, \delta_{n_4})] = q_{n_1} q_{n_2} q_{n_3} q_{n_4},$$

for any $n_\alpha = 1, 2, 3, \alpha = 1, \ldots, 4$. Moreover, $\epsilon_1, \ldots, \epsilon_4$ are assumed to be independent of $\theta_1^+, \ldots, \theta_4^+$.

Therefore,

$$\begin{aligned} \mathbb{E}\{V(k^+, \theta^+) \mid \theta = (\vartheta_{j_1}, \ldots, \vartheta_{j_4})\} = & \sum_{n_1, n_2, n_3, n_4 = 1}^{3} q_{n_1} q_{n_2} q_{n_3} q_{n_4} \\ & \sum_{i_1, i_2, i_3, i_4 = 1}^{7} P_{i_1,j_1} P_{i_2,j_2} P_{i_3,j_3} P_{i_4,j_4} \\ & \times V(\hat{k}_1^+ + \delta_{n_1}, \ldots, \hat{k}_4^+ + \delta_{n_4}, \vartheta_{i_1}, \ldots, \vartheta_{i_4}), \end{aligned} \tag{5}$$

where $\hat{k}_\alpha^+ = (1 - \delta) k_\alpha + I_\alpha$, for any $\alpha = 1, \ldots, 4$.

From the formula (5), it seems that we should compute the value function V at a large number of points up to $3^4 \times 7^4 = 194{,}481$ in order to evaluate the expectation. But in fact, we can take advantage of the sparsity of the probability transition matrix P. After canceling the zero probability terms, the evaluation of the expectation will need to compute the value function at a number of points ranging from $3^4 \times 2^4 = 1{,}296$ to

Table 1 Statistics of parallel DP under HTCondor-MW for the growth problem

Wall clock time for all 3 VFIs	8.28 h
Wall clock time for 1st VFI	0.34 h
Wall clock time for 2nd VFI	3.92 h
Wall clock time for 3rd VFI	4.01 h
Total time workers were up (alive)	16.9 days
Total cpu time used by all workers	16.5 days
Number of (different) workers	50
Average number present workers	49
Overall parallel performance	98.6 %

$3^4 \times 3^4 = 6{,}561$, which is far less than the case without using the sparsity. Moreover, the communication cost between the master and workers is also far less than the case without using the sparsity.

The continuous value function approximation is the complete degree-6 Chebyshev polynomial approximation method (3) with $7^4 = 2{,}401$ Chebyshev nodes for continuous state variables, the optimizer is NPSOL, and the terminal value function is chosen as

$$V_T(k, \theta) = u(f(k, \mathbf{e}, \mathbf{e}), \mathbf{e})/(1 - \beta),$$

where $\mathbf{e}$ is the vector with 1's everywhere. Here $\mathbf{e}$ is chosen because it is the steady state labor supply for the corresponding infinite-horizon problem and is also the average value of θ.

6.3 HTCondor-MW Results

We use the master Algorithm 2 and the worker Algorithm 3 to solve the optimal growth problem. There are seven possible values of θ_i for each $i = 1, \ldots, 4$, and each task consists of updating the value function at one specific θ^j; therefore, the total number of HTCondor-MW tasks for one value function iteration is $7^4 = 2{,}401$. Furthermore, we use seven approximation nodes in each continuous dimension to construct a degree six complete polynomial; therefore, each task computes 2,401 small-size maximization problems as there are 2,401 Chebyshev nodes.

Under HTCondor, we assign 50 workers to do this parallel work. Table 1 lists some statistics of our parallel DP algorithm under HTCondor-MW system for the growth problem after running 3 value function iterations (VFI). The last line of Table 1 shows that the parallel efficiency of our parallel numerical DP method is very high (up to 98.6 %) for this example. We see that the total cpu time used by all workers to solve the optimal growth problem is nearly 17 days, i.e., it will take nearly 17 wall clock days to solve the problem without using parallelism. However, it takes only 8.28 wall clock hours to solve the problem if we use the parallel algorithm and 50 worker processors.[6]

[6] The pool has machines of different characteristics, but a typical machine has eight cores and 8 GB memory, uses the Intel(R) Xeon(R) CPU E5345 @ 2.33GHz with 1 Gbps (Gigabit per second) Ethernet.

Table 2 Parallel efficiency for various numbers of worker processors

# Worker processors	Parallel efficiency (%)	Average task wall clock time (s)	Total wall clock time (h)
50	98.6	199	8.28
100	97	185	3.89
200	91.8	186	2.26

Table 2 gives the parallel efficiency with various number of worker processors for this optimal growth model. We see that it has an almost linear speed-up when we add the number of worker processors from 50 to 200. We see that the wall clock time to solve the problem is only 2.26 h now if the number of worker processors increases to 200.

Parallel efficiency drops from 99 to 92 % when we move from 100 processors to 200. This is not the critical fact for a user. The most important fact is that requesting 200 processors reduced the waiting time from submission to final output by 1.6 h. Focussing on the user's waiting time is one of the values of the HTC approach to parallelization. Since the average task takes about 3 min, the network latency does not matter much. Moreover, our tasks are computationally expensive but not data intensive (as we use smooth functional approximation and fast Newton-type optimization solvers instead of discretization and grid search method), our memory requirement is small.

7 Application to Dynamic Portfolio Problems with Transaction Costs

We consider a dynamic portfolio problem with transaction costs. We assume that an investor begins with some initial wealth W_0, invests it in several assets, and manages it at every time t so as to maximize the expected utility of wealth at a terminal time T. We assume a power utility function for terminal wealth, $u(W) = W^{1-\gamma}/(1-\gamma)$ where $\gamma > 0$ and $\gamma \neq 1$. Let $R = (R_1, \ldots, R_n)^\top$ be the random one-period return of n risky assets, and R_f be the return of the riskless asset. The portfolio share for asset i at the beginning of period t is denoted $x_{t,i}$, and let $x_t = (x_{t,1}, \ldots, x_{t,n})^\top$. The difference between wealth and the wealth invested in stocks is invested in bonds. At the beginning of every period, the investor has a chance to re-balance the portfolio with a proportional transaction cost rate τ for buying or selling stocks. Let $\delta^+_{t,i} W$ denote the amount of asset i purchased, expressed as a fraction of wealth, and let $\delta^-_{t,i} W$ denote the amount sold, where $\delta^+_{t,i}, \delta^-_{t,i} \geq 0$, for periods $t = 0, \ldots, T-1$.

We assume that the riskless return R_f and the risky assets' return R may be dependent on a discrete time stochastic process θ_t (could be a vector), denoted by $R_f(\theta_t)$ and $R(\theta_t)$ respectively, for $t = 0, \ldots, T-1$. Then the dynamic portfolio problem becomes

$$V_0(W_0, x_0, \theta_0) = \max_{\delta^+, \delta^- \geq 0} \mathbb{E}\{u(W_T)\}, \tag{6}$$
$$\text{s.t. } W_{t+1} = \mathbf{e}^\top X_{t+1} + R_f(\theta_t)(1 - \mathbf{e}^\top x_t - y_t) W_t,$$

$$
\begin{aligned}
&X_{t+1,i} = R_i(\theta_t)(x_{t,i} + \delta^+_{t,i} - \delta^-_{t,i})W_t,\\
&y_t = \mathbf{e}^\top(\boldsymbol{\delta}^+_t - \boldsymbol{\delta}^-_t + \tau(\boldsymbol{\delta}^+_t + \boldsymbol{\delta}^-_t)),\\
&x_{t+1,i} = X_{t+1,i}/W_{t+1},\\
&\theta_{t+1} = g(\theta_t, \xi_t),\\
&t = 0, \ldots, T-1; \quad i = 1, \ldots, n,
\end{aligned}
$$

where $\mathbf{e}$ is the column vector with 1's everywhere, $X_{t+1} = (X_{t+1,1}, \ldots, X_{t+1,n})^\top$, $\boldsymbol{\delta}^+_t = (\delta^+_{t,1}, \ldots, \delta^+_{t,n})^\top$, and $\boldsymbol{\delta}^-_t = (\delta^-_{t,1}, \ldots, \delta^-_{t,n})^\top$. Here, W_{t+1} is time $t+1$ wealth, $X_{t+1,i}$ is time $t+1$ wealth in asset i, $y_t W_t$ is the change in bond holding, and $x_{t+1,i}$ is the allocation of risky asset i.

7.1 Dynamic Programming Model

The DP model of the multi-stage portfolio optimization problem (6) is

$$
V_t(W, x, \theta) = \max_{\delta^+, \delta^- \geq 0} \quad \mathbb{E}\left\{V_{t+1}(W^+, x^+, \theta^+)\right\},
$$

for $t = 0, 1, \ldots, T-1$, while the terminal value function is $V_T(W, x, \theta) = W^{1-\gamma}/(1-\gamma)$. Given the isoelasticity of V_T, we know that the value function can be rewritten as

$$
V_t(W_t, x_t, \theta_t) = W_t^{1-\gamma} \cdot H_t(x_t, \theta_t),
$$

for some functions $H_t(x_t, \theta_t)$, where W_t and x_t are respectively wealth and allocation fractions of stocks right before re-balancing at stage $t = 0, 1, \ldots, T$, and

$$
\begin{aligned}
H_t(x, \theta) = \max_{\delta^+, \delta^-} \ & \mathbb{E}\left\{\Pi^{1-\gamma} \cdot H_{t+1}(x^+, \theta^+)\right\}, \\
& \text{s.t. } \delta^+ \geq 0, \delta^- \geq 0, \\
& x + \delta^+ - \delta^- \geq 0, \\
& y \leq 1 - \mathbf{e}^\top x, \\
& \theta^+ = g(\theta, \xi_t),
\end{aligned}
\tag{7}
$$

where $H_T(x, \theta) = 1/(1-\gamma)$, and

$$
\begin{aligned}
y &\equiv \mathbf{e}^\top(\delta^+ - \delta^- + \tau(\delta^+ + \delta^-)),\\
s_i &\equiv R_i(\theta)(x_i + \delta^+_i - \delta^-_i),\\
\Pi &\equiv \mathbf{e}^\top s + R_f(\theta)(1 - \mathbf{e}^\top x - y),\\
x^+_i &\equiv s_i/\Pi,
\end{aligned}
$$

for $i = 1, \ldots, n$ and $t = 0, 1, \ldots, T-1$. See Cai et al. (2013b) for a detailed discussion of this dynamic portfolio optimization problem.

Since W_t and x_t are separable, we can just assume that $W_t = 1$ dollar for simplicity. Thus, at time t, δ^+ and δ^- are the amounts for buying and selling stocks respectively, y is the change in bond holding, s is the next-stage amount vector of dollars on the stocks, Π is the total wealth at the next stage, and x^+ is the new fraction vector of the stocks at the next stage. In this model, the state variables, x and x^+, are continuous in $[0, 1]^n$.

7.2 Numerical Examples

We choose a portfolio with $n = 6$ stocks and one riskless bond. The investor wants to maximize the expected terminal utility after $T = 6$ years with the terminal utility, $u(W) = W^{1-\gamma}/(1-\gamma)$, with $\gamma = 4$. At the beginning of each year $t = 0, 1, \ldots T-1$, the investor has a chance to rebalance the portfolio with a proportional transaction cost rate $\tau = 0.002$ for buying or selling stocks. We assume that the stock returns are independent each other, and stock i has a log-normal annual return, i.e., $\log(R_i) \sim \mathcal{N}(\mu_i - \sigma_i^2/2,\ \sigma_i^2)$ with $\mu_i = 0.07$ and $\sigma_i = 0.25$, for $i = 1, \ldots, n$. We assume that the bond has a riskless annual return $\exp(r_t)$, while the interest rate r_t is a discrete Markov chain, with $r_t = 0.01, 0.02, 0.03, 0.04$ or 0.05, and its transition probability matrix is

$$P = \begin{bmatrix} 0.7 & 0.3 & & & & \\ 0.3 & 0.4 & 0.3 & & & \\ & 0.3 & 0.4 & 0.3 & & \\ & & 0.3 & 0.4 & 0.3 & \\ & & & 0.3 & 0.7 & \end{bmatrix}.$$

We use the degree-4 complete Chebyshev polynomials (3) as the approximation method, and choose 5 Chebyshev nodes on each dimension, so that we can apply the Chebyshev regression algorithm to compute the approximation coefficients in the fitting step of numerical DP algorithms. Thus, the number of approximation nodes is $5^6 = 15{,}625$ for each discrete state, so the total number of small-size maximization problems for one value function iteration is $5 \times 5^6 = 78{,}125$. We use the product Gauss–Hermite quadrature formula (4) with 5 nodes for each dimension, so the number of quadrature nodes is $5^6 = 15{,}625$ for each discrete state. Therefore, after using the sparsity of the probability transition matrix, the computation of the expectation in the objective function of the maximization problem (7) includes $2 \times 5^6 = 31{,}250$ or $3 \times 5^6 = 46{,}875$ evaluations of the approximated value function at stage $t + 1$ for each approximation node. We use NPSOL as our optimization solver for solving the maximization problem (7) .

7.3 HTCondor-MW Results

We apply Algorithm 4 and 5 to solve the high-dimensional dynamic portfolio problem. Each HTCondor-MW task solves 25 small-size maximization problems, implying

Table 3 Statistics of parallel DP under HTCondor-MW for the 7-asset portfolio problem with stochastic interest rate

Wall clock time for all 6 VFIs	3.6 h
Wall clock time for 1st VFI	4.8 min
Wall clock time for 2nd VFI	43.4 min
Wall clock time for 3rd VFI	40.6 min
Wall clock time for 4th VFI	41.5 min
Wall clock time for 5th VFI	42.9 min
Wall clock time for 6th VFI	43.7 min
Total time workers were up (alive)	29.3 days
Total cpu time used by all workers	27.4 days
Number of (different) workers	200
Average number present workers	194
Overall parallel performance	94.2 %

that each value function iteration is broken into 3,125 MTCondor-MW tasks. Our HTCondor program requested 200 workers, and was given 194 processors on average.

Table 3 lists some statistics of our parallel DP algorithm under HTCondor-MW system for the portfolio problem with six stocks and one bond with stochastic interest rates. The parallel efficiency of our parallel numerical DP method is 94.2 % for this example, even when we use 200 workers. Moreover, the total cpu time used by all workers to solve the dynamic portfolio optimization problem is more than 27 days, i.e., it will take more than 27 days to solve the problem using a single core. However, it takes only about 3.6 wall clock hours to solve the problem if we use the type-II parallel DP algorithm and 200 worker processors. This reduction in "waiting time" cost to a researcher makes it possible to solve problems that essentially cannot be solved on a laptop.

8 Conclusion

This paper presented an implementation of parallel dynamic programming methods in HTCondor Master–Worker system, and demonstrated its ability to solve demanding high-dimensional dynamic programming problems efficiently. We have focused on computational details of the algorithm, and its integration with the Master–Worker tool in HTCondor. This is just one example of how HTCondor makes it possible for a user to adapt serial code for high-latency massively parallel systems with minimal effort. The programmer can use exactly the same code for the key numerical tasks. The only requirement is for the user to set up input–output commands to facilitate communications between the Master and Workers, and tell the HTCondor system how many processors you would like to use. An example of this code is given in https://sites.google.com/site/dpinhtcondor/.

While we only used DP examples, it is clear that these tools can be used for many related economics problems. For example, the nested fixed point method for solving structural models (see Rust 1987) requires solutions to dynamic programming prob-

lems in its "inner loop". Our Master could manage both the solution of specific DP problems within the search for the econometric estimates. Public finance analyses often need to compute the response of economic actors to alternative tax codes. Given the central role of dynamic programming in dynamic economic analyses, it is clear that HTCondor and similar tools have wide potential use in economics.

Acknowledgments We are grateful to the editor and anonymous reviewers for their insightful comments and suggestions. Cai and Judd gratefully acknowledge National Science Foundation support (SES-0951576). We also thank Miron Livny for his generous support and access to the HTCondor cluster at the University of Wisconsin-Madison.

References

Abdelkhalek, A., Bilas, A., & Michaelides, A. (2001). Parallelization, optimization and performance analysis of portfolio choice models. In *Proceedings of the 2001 international conference on parallel processing (ICPP01)* (pp. 277–286).

Aldrich, E. M., Fernandez-Villaverde, J., Gallant, A. R., & Rubio-Ramrez, J. F. (2011). Tapping the supercomputer under your desk: Solving dynamic equilibrium models with graphics processors. *Journal of Economic Dynamics and Control*, *35*, 386–393.

Bellman, R. (1957). *Dynamic programming*. Princeton: Princeton University Press.

Cai, Y. (2010). *Dynamic programming and its application in economics and finance*. PhD thesis, Stanford University.

Cai, Y., & Judd, K. L. (2010). Stable and efficient computational methods for dynamic programming. *Journal of the European Economic Association*, *8*(2–3), 626–634.

Cai, Y., & Judd, K. L. (2012a). Dynamic programming with shape-preserving rational spline Hermite interpolation. *Economics Letters*, *117*(1), 161–164.

Cai, Y., & Judd, K. L. (2012b). Dynamic programming with Hermite approximation. NBER Working Paper No. w18540.

Cai, Y., & Judd, K. L. (2013). Shape-preserving dynamic programming. *Mathematical Methods of Operations Research*, *77*(3), 407–421.

Cai, Y., Judd, K. L., Lontzek, T. S., Michelangeli, V., & Su, C.-L. (2013a). Nonlinear programming method for dynamic programming. NBER Working Paper No. w19034.

Cai, Y., Judd, K. L., & Xu, R. (2013b). Numerical solutions of dynamic portfolio optimization with transaction costs. NBER Working Paper No. w18709.

Chung, S. L., Hanson, F. B., & Xu, H. H. (1992). Parallel stochastic dynamic programming: Finite element methods. *Linear Algebra and Its Applications*, *172*, 197–218.

Coleman, W. J. (1992). Solving nonlinear dynamic models on parallel computers. Discussion Paper 66, Institute for Empirical Macroeconomics, Federal Reserve Bank of Minneapolis.

Creel, M. (2005). User-friendly parallel computations with econometric examples. *Computational Economics*, *26*(2), 107–128.

Creel, M., & Goffe, W. L. (2008). Multi-core CPUs, clusters, and grid computing: A tutorial. *Computational Economics*, *32*(4), 353–382.

Den Haan, W. J., Judd, K. L., & Juillard, M. (2011). Computational suite of models with heterogeneous agents II: Multi-country real business cycle models. *Journal of Economic Dynamics & Control*, *35*, 175–177.

Gill, P., Murray, W., Saunders, M. A., & Wright, M. H. (1994). User's guide for NPSOL 5.0: A Fortran package for nonlinear programming. Technical Report, SOL, Stanford University.

Griebel, M., & Wozniakowski, H. (2006). On the optimal convergence rate of universal and nonuniversal algorithms for multivariate integration and approximation. *Mathematics of Computation*, *75*(255), 1259–1286.

Judd, K. L. (1998). *Numerical methods in economics*. Cambridge, MA: The MIT Press.

Juillard, M., & Villemot, S. (2011). Multi-country real business cycle models: Accuracy tests and test bench. *Journal of Economic Dynamics & Control*, *35*, 178–185.

Morozov, S., & Mathur, S. (2012). Massively parallel computation using graphics processors with application to optimal experimentation in dynamic control. *Computational Economics*, *40*(2), 151–182.

Pflug, G. C., & Swietanowski, A. (2000). Selected parallel optimization methods for financial management under uncertainty. *Parallel Computing*, *26*(1), 3–25.
Rust, J. (1987). Optimal replacement of GMC bus engines: An empirical model of Harold Zurcher. *Econometrica*, *55*(5), 999–1033.
Rust, J. (1997). Using randomization to break the curse of dimensionality. *Econometrica*, *65*(3), 487–516.
Rust, J. (2008). Dynamic programming. In S. N. Durlauf & E. Blume (Eds.), *New Palgrave dictionary of economics* (2nd ed.). Basingstoke: Palgrave Macmillan.
Rust, J., Traub, J. F., & Wozniakowski, H. (2002). Is there a curse of dimensionality for contraction fixed points in the worst case? *Econometrica*, *70*(1), 285–329.
Stroud, A., & Secrest, D. (1966). *Gaussian quadrature formulas*. Englewood Cliffs, NJ: Prentice Hall.
Thain, D., Tannenbaum, T., & Livny, M. (2005). Distributed computing in practice: The condor experience. *Concurrency and Computation: Practice and Experience*, *17*(2–4), 323–356.
Zenios, S. A. (1999). High-performance computing in finance: The last 10 years and the next. *Parallel Computing*, *25*(13–14), 2149–2175.